Buzibachstrasse 31
6023 Rothenburg
Switzerland

Phone +41(0)41 541 50 40
info@zub.ch
www.zub.ch

AposslDE

Help

Version 7.00.102

AposslDE
Help
mzub | Edition 2025 - 9 | AposslDE 7.00.102

Table of Contents

1. Introduction to the ApossIDE 26
   1.1. Contact and Legal Information 26-27
2. Getting Started and Safety Tips 28-29
   2.1. System Architecture 29-33
   2.2. Installing the ApossIDE 33-34
   2.3. Wiring the Controller 34-35
   2.4. Starting and Exiting the ApossIDE 35-37
   2.5. Connecting to the Controller 37
      2.5.1. V24 (Serial) Connections 37-39
      2.5.2. CAN-LPT Connections 39-42
      2.5.3. CAN-USB Connections 42-45
      2.5.4. USB Connections 45-47
      2.5.5. EtherCAT Connections (Beckhoff) 47-48
      2.5.6. EtherCAT Connections (zub) 48-50
      2.5.7. Network Connections 50-53
      2.5.8. Simulator Connections 53-55
   2.6. Executing a Simple Test Program 55-58
   2.7. Setting Basic Parameters 58-61
   2.8. Checking the Encoder 61-63
   2.9. Checking the Motor 63-65
3. ApossIDE User Interface 66
   3.1. ApossIDE Window 66-67
   3.2. Toolbar 67-69
   3.3. Sidebar 69-71
   3.4. Findbar 71
   3.5. Communication Window 71-74
   3.6. Keyboard and Mouse Functions 74
      3.6.1. Function Keys 74
      3.6.2. Esc key 74-75
      3.6.3. Mouse functions 75-76
      3.6.4. Popup Menus 76
      3.6.5. Keyboard functions 76-78

3.6.6. Undo Function 78
3.6.7. Tabs 78
3.7. File Menu 78
3.7.1. New 78
3.7.2. Sample 78-79
3.7.3. Exit Program 79
3.8. Edit Menu 79-80
3.8.1. Paste Array Assignment 80
3.8.2. Find and Replace 80-82
3.8.3. Find in Files 82-84
3.8.4. Go to 84-85
3.8.5. Bookmarks 85
3.8.6. Folding 85
3.8.7. SDO Macro Selector 85-86
3.9. Development Menu 87
3.9.1. Execute 87-88
3.9.2. Break [Esc] and Continue 88
3.9.3. Messages -> Log file 88
3.9.4. Debugging Commands 88-89
3.9.5. Watch Add / Start / Stop 89-91
3.9.6. Syntax Check 91
3.9.7. Compile to file 91
3.9.8. Compare to Controller 91-93
3.9.9. Break all 93
3.9.10. Start all 93-94
3.9.11. Download all 94
3.9.12. Delete Programs all 94
3.9.13. Select Controller 94-95
3.9.14. Close Interface / Close All Interfaces 95
3.10. Controller Menu 95-97
3.10.1. Programs 97-100
3.10.1.1. Autostart 100-102
3.10.2. Parameters 102

3.10.2.1. Global / Axis 102-103
3.10.2.2. Save to file 103-104
3.10.2.3. Restore from file 104-106
3.10.2.4. Save ALL to files 106
3.10.3. Reset 106
3.10.4. Error History 106-108
3.10.5. Diagnostics 108-109
3.10.6. Read/Write SDO 109-110
3.10.7. Download Firmware 110-111
3.10.8. State Machine 111
3.11. Tools Menu 111
3.12. Settings Menu 112
3.12.1. Compiler 112-117
3.12.2. Interface Parameters 117-118
3.12.2.1. V24 Parameters 118-119
3.12.2.2. CAN-LPT Parameters 119-120
3.12.2.3. CAN-USB Parameters 120-122
3.12.2.4. USB Parameters 122
3.12.2.5. EtherCAT Parameters (Beckhoff) 122-123
3.12.2.6. EtherCAT Parameters (zub) 123-125
3.12.2.7. Network Parameters 125
3.12.2.8. Simulator Parameters 125-126
3.12.3. Editor Settings 126-128
3.12.4. Options 128-130
3.13. Windows Menu 130-131
3.14. Help Menu 131
3.15. Download Menu 131
3.15.1. Download Firmware 131-133
3.15.2. Download Programs 133-135
4. ApossIDE Programming 136
4.1. ApossC Programming Language 136-137
4.1.1. Differences Between Standard C and ApossC 137-141
4.1.2. Differences Between APOSS and ApossC 142-153

4.2. Special Arrays 153-154
4.2.1. ANSI Arrays 154
4.2.2. DIM Arrays 154-156
4.3. Interrupt Handling 156
4.3.1. Interrupt Behaviour 156-158
4.3.2. Interrupt Routines 158-160
4.3.3. Interrupts 160
4.3.3.1. CANINPUT 160-161
4.3.3.2. ERROR 161-162
4.3.3.3. ETHERNET 162
4.3.3.4. DIGINPUT 162-163
4.3.3.5. PARAM 163-165
4.3.3.6. PERIOD 165-166
4.3.3.7. Position 166-169
4.3.3.8. SERIALCHAR 169
4.3.3.9. SETOUT 169-171
4.3.3.10. STATBIT 171-172
4.3.3.11. TIME 172-173
4.4. Error Handling 173-176
4.5. Preprocessor 176-177
4.5.1. Continuation lines 177-178
4.5.2. #include 178
4.5.3. #define, #undef 178-180
4.5.4. #ifdef, #ifndef, #if 180-181
4.5.5. #config 181-183
4.6. Program Samples 183-186
4.7. Debugging Programs 186-188
4.7.1. Starting the Debugger 188-189
4.7.2. Stopping the Debugger 189
4.7.3. Single-Stepping 189-190
4.7.4. Using Breakpoints 190-191
4.7.5. Displaying and Modifying Variables 191-192
4.7.6. Displaying the Executing Line 192

5. ApossIDE Tools 193-194
5.1. CAM Editor 194-195
5.1.1. Starting the CAM Editor 195-196
5.1.2. CAM Editor Toolbar 196-198
5.1.3. Editing CAM Curves 198-199
5.1.3.1. CAM Curve Diagram 199-202
5.1.3.2. CAM Curve Tab 202-208
5.1.3.3. Display Tab 208-210
5.1.3.4. Axis Data Tab 210-212
5.1.3.5. Fix Points Tab 212-215
5.1.3.6. Start Stop Points Tab 215-217
5.1.4. Editing Path Curves 217-219
5.1.4.1. Path Curve Diagram 219-222
5.1.4.2. Path Curve Tab 222-225
5.1.4.3. Display Tab 225-227
5.1.4.4. Fix Points Tab 227-231
5.1.4.5. Background Images 231-232
5.1.4.6. Overlay Files 232-235
5.1.4.7. Multiple Paths 235-236
5.1.4.8. Viewing Individual Axes 236-237
5.1.5. Editing Data Arrays 237-238
5.2. Array Editor 238
5.2.1. Starting the Array Editor 238-240
5.2.2. Array Editor List 240-242
5.2.3. Array Editor Functions 242-244
5.2.4. User mode WITHOUT an Array Definition File 244-245
5.2.5. Expert mode WITHOUT an Array Definition File 245-246
5.2.6. User mode WITH an Array Definition File 247-248
5.2.7. Expert mode WITH an Array Definition File 248-249
5.2.8. Creating an Array Definition File 249-251
5.3. Oscilloscope 251-252
5.3.1. Oscilloscope Versions 252
5.3.1.1. Free Run Oscilloscope 252-253

5.3.1.2. Single Shot Oscilloscope 253
5.3.1.3. Record Oscilloscope 253-254
5.3.2. Oscilloscope Window 254-255
5.3.2.1. Chart Window 255-258
5.3.2.2. Navigation Window 258-260
5.3.3. Curve Selection List 260
5.3.3.1. Name Column 260-261
5.3.3.2. Grid Column 261-262
5.3.3.3. Status Column 262-263
5.3.3.4. Active Column 263-264
5.3.3.5. Mouse Position Column 264
5.3.3.6. Last, Minimum, and Maximum Columns 264
5.3.3.7. Cursor A/B Columns 264-265
5.3.3.8. Units Column 265
5.3.3.9. Controller and Interface Columns 265-266
5.3.3.10. Modifying a Curve's Properties 266
5.3.3.11. Curve Properties: Display 266-271
5.3.3.12. Curve Properties: Axis 271-272
5.3.3.13. Curve Properties: Source 272-273
5.3.4. Starting the Oscilloscope 273-276
5.3.4.1. Usage Notes for the Single Shot Oscilloscope 276-277
5.3.4.2. Usage Notes for the Record Oscilloscope 277-279
5.3.5. Saving and Opening Oscilloscope Files 279
5.3.6. Updating from Record Files 279
5.3.7. Reading .csv Files 279-281
5.3.8. Reading Old Testrun Files 281-282
5.3.9. Adding Oscilloscope Curves 282-283
5.3.9.1. Usage Notes for Polling Mode 283-285
5.3.9.2. SDO Curve 285-286
5.3.9.3. PDO Curve 286-288
5.3.9.4. Variable Curve 288-290
5.3.9.5. Derived Curve 290-292
5.3.9.6. Group Curve 292-294

5.3.9.7. Record Curve 294-295

5.3.9.8. User Curves 295-297

5.3.9.9. Adding Curves from other Oscilloscope Files 297-298

5.3.10. Starting and Ending the Test 298-300

5.3.10.1. Usage Notes for the Single Shot Oscilloscope 300-301

5.3.11. Analyzing the Results 301-302

5.3.12. Repeating the Test 302

5.3.12.1. Usage Notes for the Record Oscilloscope 302

5.3.13. Oscilloscope Settings 302-305

5.4. Module Editor 305-306

5.4.1. Module Tree View 306-308

5.4.2. Module Graphics View 308-309

5.4.3. Module Window 309-312

5.4.4. Module Editor Toolbar 312-313

5.4.5. Module Editor Settings 313

5.4.6. Module Linking 313-315

5.5. State Machines 315-316

5.5.1. Defining Signals and Event Types 316-318

5.5.2. Defining the State Hierarchy 318-319

5.5.3. Specifying the Event Types to be Handled by a State 319-320

5.5.4. Defining State Machine Variables 320-321

5.5.5. Specifying the State Machine Configuration 321-324

5.5.6. Defining Handlers for Events 324

5.5.6.1. Predefined Handlers 324-325

5.5.6.2. State Transitions 325-328

5.5.6.3. State Functions 328-330

5.5.7. Starting and Stopping State Machines 330-331

5.5.8. State Machine Execution 331

5.5.9. Posting Events to State Machines 331-332

5.5.10. Subscribing to Events and Publishing Events 332-334

5.5.11. Clearing Pending Events 334-335

5.5.12. Handling General Controller Errors 335-336

5.5.13. State Machine Timers 336-338

5.5.14. Detecting General Operation Events 338-339
5.5.14.1. RS485/RS232 Events 339-340
5.5.14.2. Serial Interface Events 340-341
5.5.14.3. PDO Message Events 341-342
5.5.14.4. General CAN Message Events 342-343
5.5.14.5. Profile Generator Events 343-346
5.5.14.6. Ethernet events 346-347
5.5.15. Detecting Changes to Parameters 347-349
5.5.16. Detecting Digital Input Changes 349-350
5.5.17. Detecting Motor Positions 350-352
5.5.18. Detecting Axis Status Changes 352-353
5.5.19. State Machine Errors 353-356
5.5.20. Debugging State Machines 356-357
5.5.20.1. User Configuration of History 357-358
5.5.20.2. Program Configuration of History 358-359
5.5.20.3. Viewing State Machine History 359-360
5.5.21. Sample Programs 360
5.5.21.1. Path Curve Sample 360-371
5.6. Simulator 371-374
5.6.1. Simulator Files 374
5.6.2. Required Program Adaptations 374-375
5.6.3. Simulator User Interface 375-377
5.6.4. Advanced Simulator Topics 377-378
5.7. Monitor 378-379
5.7.1. Control Statements 379
5.7.1.1. WindowWidth and WindowHeight 379
5.7.1.2. WindowTitle 380
5.7.1.3. WindowFlags 380
5.7.1.4. OpFlags 380-381
5.7.1.5. PollInterval 381
5.7.1.6. Fade 381
5.7.2. Layout Element Statements 381-382
5.7.2.1. Array 382-383

5.7.2.2. Bit 383-384
5.7.2.3. Bitmap 385
5.7.2.4. Box 385-386
5.7.2.5. Button 386-387
5.7.2.5.1. Controller 387-388
5.7.2.5.2. Download 388
5.7.2.5.3. Get 388-389
5.7.2.5.4. Group 389
5.7.2.5.5. PDO 389-390
5.7.2.5.6. Set 390
5.7.2.6. Checkbox 390-391
5.7.2.7. Child 391-392
5.7.2.8. Constant 392
5.7.2.9. Controller 392-393
5.7.2.10. Font 393-394
5.7.2.11. List 394
5.7.2.11.1. Axis 394-395
5.7.2.11.2. Lookup 395
5.7.2.11.3. Table 395
5.7.2.12. Message 395-396
5.7.2.12.1. Controller 396-397
5.7.2.12.2. Load 397
5.7.2.12.3. Lookup 397-398
5.7.2.12.4. Print 398
5.7.2.12.5. Value 398-399
5.7.2.13. PDO 399
5.7.2.14. SDO 399-400
5.7.2.15. Slider 400-401
5.7.2.16. Table 401
5.7.2.17. Text 401-402
5.7.2.18. Value 402-404
5.7.3. Example Monitor Files 404
5.7.3.1. Fonts and Text Strings 404-406

5.7.3.2. Simple Polling 406-408
5.7.3.3. Background Fading 408-409
5.7.3.4. Displaying Status Bits 409-410
5.7.3.5. Polling SDO Values on Demand 410-412
5.7.3.6. Receiving PDO Messages 412-414
5.7.3.7. Reading and Writing SDO Values 414-416
5.7.3.8. Writing Specific SDO Values and Writing Multiple SDO Values 416-418
5.7.3.9. Setting Bits 418-419
5.7.3.10. Listboxes and SDO Index Offsets 419-420
5.7.3.11. Showing and Hiding Dialog Fields 420-424
5.7.3.12. Multiple Controllers, Polling Load, and Dynamically Selecting Controllers 424-426
5.7.3.13. Specified Controllers and Interface Types 426-428
5.7.3.14. Child Dialogs 428-430
5.7.3.15. Displaying Analog Input 430-432
5.7.3.16. Displaying and Setting Regulation Parameters 432-435
5.7.3.17. Displaying and Setting User Parameters 435-438
5.8. Batch Mode 438-439
5.8.1. Configuring and Starting Batch Mode 439-440
5.8.2. Using ".zip" Files 440
5.8.3. Controller ID Numbers 440
5.8.4. Command File Structure 440-445
5.8.5. Answer File Structure 445-446
5.8.6. Error Codes 446
5.8.7. Batch Mode Examples 446-448
5.9. Log Mode 448-450
5.9.1. Configuring and Starting Log Mode 450-451
5.9.2. Command File Structure 451-454
5.9.3. Answer File Structure 454-456
5.9.4. Log File Structure 456-458
5.9.5. Oscilloscope 458
5.10. Recorder 458
5.10.1. Setting Up the ApossIDE Recorder 458-459

5.10.2. Preparing the Controller 459
5.10.3. Starting the Recorder on the PC 459-460
5.10.4. Recorder Settings 460-461
5.10.5. microSD Card Files 461-462
5.10.6. PC Output Files 462
5.10.7. Customizing the Application 462-463
5.10.8. Recorder Limitations 463
5.11. Download Mode 463
5.11.1. Download Mode 463-465
5.12. Current Regulator Calculator 465
5.12.1. Current Regulator Calculator 465-467
6. Motion Control Topics 468
6.1. Scaling 468
6.1.1. Master Units [MU] 468
6.1.2. How to Set Up Resolution and Gear Parameters 468-471
6.1.3. Encoder Position Values 471
6.2. Synchronization 471
6.2.1. Setting Synchronization Parameters 471-472
6.2.2. Synchronization Commands 472-473
6.3. Optimizing the PIDController 473
6.3.1. How the Control Process Works 473-474
6.3.2. Position Control Loop Design 474-475
6.3.3. PID Factors 475-477
6.3.4. Optimizing your Controller Settings Step-by-step 477
6.3.4.1. Setting Controller Behavior 477-478
6.3.4.2. The quickest route to optimal control 478
6.3.4.3. Ten steps for optimum control for MACS controllers 478-479
6.3.4.4. What to do, if ...? 479-480
6.4. Limited Jerk 480-483
6.4.1. Examples of Limited-Jerk Movements 483-488
6.5. CAM Curves 488-490
6.5.1. CAM Curve Behaviour 490-492
6.5.2. CAM Curve Scaling and Positioning 492-496

6.5.3. Dynamic CAM Curves 496-498

6.5.3.1. Dynamic Connecting Curves 498-500

6.5.3.2. Immediate Curves 500

6.5.3.3. Special Cases 501-502

6.5.4. Modifying the CAM Curve Array 502-503

6.5.5. Quick Start Tutorial for Users 503

6.5.6. Example 1: Basic Synchronization 503-516

6.5.6.1. Example 1: Using the Simulator 516-520

6.5.7. Example 2: Synchronization with Marker Correction 520-528

6.5.7.1. Example 2: Using the Simulator 528

6.5.7.2. Example 2: Large Sensor Distances 528-529

6.5.7.3. Example 2: Problematic Situations 529-531

6.5.8. Example 3: Engaging and Disengaging 531-540

6.5.9. Example 4: Slave Synchronization with Marker 540-544

6.5.10. Example 5: CAM Box 544-545

6.6. Working with Kinematics 545

6.6.1. Multi-axis Kinematics 545-546

6.6.1.1. Cartesian Kinematics 546-548

6.6.1.2. H-Bot/CoreXY Kinematics 548-550

6.6.1.3. SCARA Kinematics 550-555

6.6.1.4. Dual SCARA Kinematics 555-557

6.6.1.5. Delta Kinematics 557-561

6.6.1.6. Stewart Kinematics 561-563

6.6.2. Basic Coordinate Systems 563-566

6.6.3. Transformations 566-570

6.6.4. Working with Points 570-573

6.6.5. Understanding Point Storage Structure 573-575

6.6.6. Path Curves 575-577

6.6.6.1. Additional Coordinate Systems 577-579

6.6.6.2. Path Fixed Points 579-581

6.6.6.3. Start and End Directions and Velocities 581-582

6.6.6.4. Thinning Fixed Points 582-584

6.6.6.5. Arc Length CAM Curves 584-587

6.6.6.6. Kinematics synchronization 588-589
6.6.7. Working with Paths 589-590
6.6.7.1. Program Access to Paths 590-591
6.6.8. Dynamic Systems 591-593
6.6.9. Paths and State Machines 593
6.7. Internal Amplifier 593-594
6.7.1. EC Motor 594
6.7.1.1. Sinusoidal Commutation 594-595
6.7.1.2. Block Commutation 595
6.7.2. DC Motor 595-596
6.7.3. Stepper Motor 596
6.7.3.1. Stepper Motor - Open Loop 596-597
6.7.3.2. Stepper Motor - Closed Loop 597-598
6.7.4. Twin Mode 598
6.7.5. Brake Chopper 598-601
6.7.6. General 601
6.7.6.1. Overcurrent Protection (OVC) 601-604
6.7.6.2. Amplifier Control Loop Design 604-606
6.7.6.3. Current Ripple and Motor Chokes 606
6.8. Internal Encoder 606-607
6.8.1. Incremental Encoder 607
6.8.2. Signal Generator 607
6.8.3. Sin Cos Encoder 607
6.8.4. SSI Encoder 607-610
6.9. Homing 610-611
6.9.1. Homing Methods 611
6.9.1.1. Homing Method 1 (Negative Limit Switch & Index) 611-612
6.9.1.2. Homing Method 2 (Positive Limit Switch & Index) 612-613
6.9.1.3. Homing Method 7 (Home Switch Positive Speed & Index) 613-614
6.9.1.4. Homing Method 11 (Home Switch Negative Speed & Index) 614-615
6.9.1.5. Homing Method 17 (Negative Limit Switch) 615-616
6.9.1.6. Homing Method 18 (Positive Limit Switch) 616-617
6.9.1.7. Homing Method 23 (Home Switch Positive Speed) 617-618

6.9.1.8. Homing Method 27 (Home Switch Negative Speed) 618-619
6.9.1.9. Homing Method 33 (Index Negative Speed) 619-620
6.9.1.10. Homing Method 34 (Index Positive Speed) 620-621
6.9.1.11. Homing Method 37 (Actual Position) 621
6.9.1.12. Homing Method -1 (Current Threshold Positive Speed & Index) 621-622
6.9.1.13. Homing Method -2 (Current Threshold Negative Speed & Index) 622-623
6.9.1.14. Homing Method -3 (Current Threshold Positive Speed) 623-624
6.9.1.15. Homing Method -4 (Current Threshold Negative Speed) 624-625
7. Controller Reference 626
7.1. Modules Description 626-630
8. Function Reference 631-633
8.1. ansicpy 633-634
8.2. ansilen 634
8.3. arraycat 634-636
8.4. arraycpy 636-637
8.5. arraylen 637-638
8.6. arrayset 638
9. Command Reference 639-640
9.1. Axis and Information Commands 640-642
9.2. Amplifier Commands 642
9.3. Input/Output Commands 642
9.4. Interrupt Functions 642
9.5. Communication CANopen Commands 642-643
9.6. Communication CAN Basic Commands 643
9.7. Communication EtherCAT Master Commands 643
9.8. Ethernet Commands 643-645
9.9. Synchronization Commands 645
9.10. Kinematics Commands 645-646
9.11. Memory Commands 646-647
9.12. Communication Serial Commands 647
9.13. Data Recording Commands 647
9.14. State Machine Commands 647-648

9.15. System Commands 648-649
9.16. Acc 649
9.17. AmpErrorClear 649-650
9.18. AnalogInput 650
9.19. AnalogOutput 650
9.20. Apos 650-652
9.21. AposDiff 652
9.22. Avel 652
9.23. AxisControl 652-655
9.24. AxisCvelStart 655
9.25. AxisCvelStop 655-656
9.26. AxisHomeStart 656
9.27. AxisHomeStatus 656-657
9.28. AxisHomeStop 657
9.29. AxisLinAbsStart 657-658
9.30. AxisLinRelStart 658
9.31. AxisMoving 658-659
9.32. AxisPosAbsStart 659
9.33. AxisPosAbsStartTvel 659-660
9.34. AxisPosRelStart 660-661
9.35. AxisStatus 661-662
9.36. AxisStatusClear 662
9.37. AxisStop 662-663
9.38. AxisTargetReached 663
9.39. AxisWaitReached 663
9.40. CamArrayIndex 663-664
9.41. CamArrayLen 664-665
9.42. CanDelete 665
9.43. CanIn 665-666
9.44. CanInit 666-667
9.45. CanOpenRestart 667
9.46. CanOut 667-668
9.47. Cpos 668

9.48. CposDiff 668-669
9.49. CurvePos 669-670
9.50. Cvel 670
9.51. Dec 670-671
9.52. DefCanIn 671
9.53. DefCanOut 671-672
9.54. DefCposOrigin 672
9.55. DefMasterCpos 672-673
9.56. DefMasterOrigin 673
9.57. DefOrigin 673-674
9.58. Delay 674
9.59. Delayus 674
9.60. Delete 674-675
9.61. DigInput 675
9.62. DigInputByte 675-676
9.63. DigInputWait 676
9.64. DigOutput 676-677
9.65. DigOutputByte 677
9.66. DimArray 677-678
9.67. DimArrayNum 678
9.68. ECatMasterCommand 678-680
9.69. ECatMasterConfig 680
9.70. ECatMasterInfo 680-683
9.71. ErrorAxis 683
9.72. ErrorClear 683-684
9.73. ErrorInfo 684
9.74. ErrorNo 684
9.75. EthernetClose 684
9.76. EthernetGetConnectionStatus 684-685
9.77. EthernetOpenClient 685
9.78. EthernetOpenServer 685-686
9.79. EthernetReceiveTelegram 686
9.80. EthernetSendTelegram 686-687

9.81. Exit 687
9.82. HallDetectStart 687-688
9.83. HallDetectStatus 688-690
9.84. InterruptDelete 690
9.85. InterruptDisable 690-691
9.86. InterruptEnable 691-692
9.87. InterruptSetup 692-693
9.88. Ipos 693-694
9.89. IposDiff 694
9.90. JerkFinVel 694
9.91. JerkStopDist 694-695
9.92. KinematicsCpoint 695
9.93. KinematicsWorkToMc 695-697
9.94. KinematicsSetup 698-699
9.95. LinkPdo 699-701
9.96. LinkSdo 701-703
9.97. Mapos 703
9.98. MaposDiff 703
9.99. Mavel 703-704
9.100. MemoryDump 704-706
9.101. MemoryDumpStatus 706
9.102. Mipos 706-707
9.103. MiposDiff 707-708
9.104. MotorAlignStart 708-709
9.105. MotorAlignStatus 709
9.106. PathAcc 709-710
9.107. PathArrayIndex 710-711
9.108. PathContinue 711-712
9.109. PathDec 712-713
9.110. PathJerk 713-714
9.111. PathMcToUu 714
9.112. PathMove 714-716
9.113. PathNext 716-717

9.114. PathStart 717-720
9.115. PathStop 720-721
9.116. PathUuToMc 721-722
9.117. PathVel 722
9.118. Pdo 723-724
9.119. #pragma 724-726
9.120. print 726-727
9.121. printcontrol 727
9.122. printf 728-729
9.123. RecordDest 729-730
9.124. RecordIndex 730-731
9.125. RecordStart 731-732
9.126. RecordStop 732-733
9.127. RecordTime 733-734
9.128. RecordTimeBase 734-735
9.129. RecordType 735-736
9.130. ResetOrigin 736
9.131. Save 736
9.132. SdoRead 736-737
9.133. SdoReadSeg 737-738
9.134. SdoReadSegp 738-740
9.135. SdoStatus 740-741
9.136. SdoWrite 741-742
9.137. SdoWriten 742-743
9.138. SdoWriteSeg 743
9.139. SerialSetup 744
9.140. SerialReadChar 744-745
9.141. SerialReadComm 745
9.142. SerialWriteComm 745-746
9.143. SetMasterOrigin 746
9.144. SetOrigin 746-747
9.145. SmClear 747
9.146. SmConfig 747

9.147. SmEvent 747
9.148. SmInput 747
9.149. SmMachine 747-748
9.150. SmParam 748
9.151. SmPeriod 748
9.152. SmPosition 748
9.153. SmPost 748
9.154. SmPublish 748
9.155. SmRun 748-749
9.156. SmState 749
9.157. SmStop 749
9.158. SmSubscribe 749
9.159. SmSystem 749
9.160. SmTimer 749
9.161. SmUnsubscribe 749
9.162. SmUrgent 749
9.163. sprint 749-750
9.164. sprintf 750
9.165. SyncCam 750-752
9.166. SyncCamMarkerMaster 752-753
9.167. SyncCamMarkerSlave 753
9.168. SyncCamStart 753-754
9.169. SyncCamStop 754-755
9.170. SyncDefOrigin 755-758
9.171. SyncError 758
9.172. SyncMarker 758-759
9.173. SyncMarkerRestart 759-760
9.174. SyncPos 760-761
9.175. SyncSetCurve 762-763
9.176. SyncShiftOrigin 763
9.177. SyncStatus 763-764
9.178. SyncStatusClear 764
9.179. SyncVel 764-766

9.180. Sysvar 766-767
9.181. Time 767
9.182. TimeDiffus 767
9.183. TimeHw 767-768
9.184. TrackError 768-769
9.185. TransIdent 769
9.186. TransInverse 769
9.187. TransPoint 769-770
9.188. TransReflect 770
9.189. TransRotateXY 770-771
9.190. TransRotateYZ 771-772
9.191. TransRotateZX 772-773
9.192. TransScale 773
9.193. TransStretch 773-774
9.194. TransTranslate 774-775
9.195. Tvel 775
9.196. UserStatus 775-776
9.197. Vel 776
9.198. WdActivate 776-777
9.199. WdReset 777
10. Parameter Reference 778-779
10.1. Initializing Temporary Parameters 779-780
10.2. User Parameters 780
10.3. Global Parameters 780-781
10.4. Axis Parameters 781-785
11. Error Reference 786
11.1. Error Messages 786-789
12. Technical Reference 790
12.1. Special Features, product specific 790
12.1.1. External Storage Device (microSD Card/ USB Memory Stick) 790
12.1.1.1. Writing Process (microSD Card/ USB Memory Stick) 790
12.1.1.1.1. Logging Service 790-791
12.1.1.1.2. Memory Dump Process 791

12.1.1.2. Reading Process (microSD Card/ USB Memory Stick) 791-792
12.1.1.2.1. Command File 792-793
12.1.1.2.2. Report File 793-794
12.1.1.2.3. System Update Without ApossIDE 794
12.1.1.2.4. User SDO Download 794-795
12.1.1.2.5. Array SDO Download 795-796
12.1.1.3. Other Functionalities (microSD Card/ USB Memory Stick) 796
12.1.1.3.1. Access with FTP 796
12.1.1.3.2. Webserver 796-797
12.1.2. Powerdown Save Functions 797-798
12.1.3. Webserver 798-799
12.1.4. Real Time Clock 799
12.1.5. Emergency Reset 799-800
12.1.6. FoE Firmware Update 800
12.1.7. Ethernet protocol 800
12.1.8. Watchdog 801-802
12.1.9. FTDI Driver 802-803
12.2. CAM Array Structure 803-810
12.2.1. CAM Array Diagram 810-812
12.3. "Record" Array Structure 812-813
12.4. Illustrations 813
12.4.1. BinFileMap 814-816
12.4.2. Marker Correction 816
12.5. SysDef Defines 816-868
12.6. Serial PDO 868-869
12.7. Version History 869
12.7.1. ApossIDE 869
12.7.1.1. What's New in Version 7.00 869-870
12.7.2. Firmware 870
12.7.2.1. What's New in Version 8.0.00 870-871
12.7.2.2. What's New in Version 9.0.00 871-872
12.8. Controller Types 872-873
12.8.1. MACS5 873

12.8.2. MACS5-Compact 873-874
12.8.3. MiniMACS 874
12.8.4. MasterMACS 874-875
12.8.5. MiniMACS6-AMP-4 875
12.8.6. Controller Dependent Feature List 875-877
12.8.7. MicroMACS6 877
12.9. Developer Tools 877
12.9.1. ZbMoc Library 877-879
12.9.1.1. Function Groups 879-884
12.9.1.2. ZbMoc Structures 884
12.9.1.2.1. ZbMocMoconIdentS 884-885
12.9.1.2.2. ZbMocMoconInfoS 885-886
12.9.1.2.3. ZbMocParamS 886-887
12.9.1.2.4. ZbMocAxisParamS 887-889
12.9.1.3. ZbMoc Error Codes 889-898
12.9.1.4. Function Descriptions 898
12.9.1.4.1. ZbMocAutoClear 898
12.9.1.4.2. ZbMocAutoSet 898
12.9.1.4.3. ZbMocAxisParamRead(DEPRECATED) 898-899
12.9.1.4.4. ZbMocAxisParamWrite(DEPRECATED) 899
12.9.1.4.5. ZbMocBlobInfoRequest 899-900
12.9.1.4.6. ZbMocBlobRead 900
12.9.1.4.7. ZbMocBlobWrite 900-901
12.9.1.4.8. ZbMocBreak 901-902
12.9.1.4.9. ZbMocBreakAll 902
12.9.1.4.10. ZbMocCanOpenDownloadSDO 902
12.9.1.4.11. ZbMocCanOpenInhibitPDO 902-903
12.9.1.4.12. ZbMocCanOpenPollPDO 903-904
12.9.1.4.13. ZbMocCanOpenQueuePDO 904
12.9.1.4.14. ZbMocCanOpenReadSegmentedSDO 904
12.9.1.4.15. ZbMocCanOpenReadSDO 904-905
12.9.1.4.16. ZbMocCanOpenStart 905
12.9.1.4.17. ZbMocCanOpenUploadSDO 905-906

12.9.1.4.18. ZbMocCanOpenWritePDO 906
12.9.1.4.19. ZbMocCanOpenWriteSegmentedSDO 906
12.9.1.4.20. ZbMocCanOpenWriteSDO 906-907
12.9.1.4.21. ZbMocCanReadRaw 907
12.9.1.4.22. ZbMocCanWriteRaw 907-908
12.9.1.4.23. ZbMocClearError 908
12.9.1.4.24. ZbMocClose 908
12.9.1.4.25. ZbMocCloseAll 908
12.9.1.4.26. ZbMocCNFRestoreFromFile 908-909
12.9.1.4.27. ZbMocCNFSaveToFile 909
12.9.1.4.28. ZbMocConnect 909-910
12.9.1.4.29. ZbMocContinue 910
12.9.1.4.30. ZbMocDebugPrint 910
12.9.1.4.31. ZbMocDebugPrintw 910
12.9.1.4.32. ZbMocDisconnect 910-911
12.9.1.4.33. ZbMocExec 911
12.9.1.4.34. ZbMocExecTemp 911
12.9.1.4.35. ZbMocGetAxisPosition 911-912
12.9.1.4.36. ZbMocGetError 912
12.9.1.4.37. ZbMocGetIFDriverID 912
12.9.1.4.38. ZbMocGetNumber 912-913
12.9.1.4.39. ZbMocGetZbMocVersion 913
12.9.1.4.40. ZbMocMemoryDump 913
12.9.1.4.41. ZbMocMemoryLock 913-914
12.9.1.4.42. ZbMocMoconBusyWait 914
12.9.1.4.43. ZbMocMoconClearFactory 914
12.9.1.4.44. ZbMocMoconClearParameters 914-915
12.9.1.4.45. ZbMocMoconClearPrograms 915
12.9.1.4.46. ZbMocMoconDeleteEeprom 915
12.9.1.4.47. ZbMocMoconExecutionStatus 915-916
12.9.1.4.48. ZbMocMoconFind 916-917
12.9.1.4.49. ZbMocMoconIdent 917
12.9.1.4.50. ZbMocMoconInfo 917

12.9.1.4.51. ZbMocMoconInfoIdx 917-918
12.9.1.4.52. ZbMocMoconNameSet 918
12.9.1.4.53. ZbMocMoconSaveRam 918
12.9.1.4.54. ZbMocMoconSaveRamSection 918-919
12.9.1.4.55. ZbMocMoconSetCanParameters 919-920
12.9.1.4.56. ZbMocMoconState 920
12.9.1.4.57. ZbMocMoconWaitForString 920
12.9.1.4.58. ZbMocOpenCanLpt 920-922
12.9.1.4.59. ZbMocOpenCanUsb 922-923
12.9.1.4.60. ZbMocOpenEtherCat 923-924
12.9.1.4.61. ZbMocOpenSerial 924-925
12.9.1.4.62. ZbMocOpenTcp 925-926
12.9.1.4.63. ZbMocOpenUsb 926-927
12.9.1.4.64. ZbMocOpenV24 927
12.9.1.4.65. ZbMocOpenVlt 928-929
12.9.1.4.66. ZbMocOpenSim 929-930
12.9.1.4.67. ZbMocParamRead 930
12.9.1.4.68. ZbMocParamWrite 930
12.9.1.4.69. ZbMocPollMessage 930-931
12.9.1.4.70. ZbMocProbe 931
12.9.1.4.71. ZbMocProgramDelete 931
12.9.1.4.72. ZbMocProgramList 932
12.9.1.4.73. ZbMocProgramLoad 932-933
12.9.1.4.74. ZbMocProgramLoadBinary 933
12.9.1.4.75. ZbMocProgramReadBinary 934
12.9.1.4.76. ZbMocProgramSave 934
12.9.1.4.77. ZbMocProgramSaveAs 934-935
12.9.1.4.78. ZbMocReadMemPage 935
12.9.1.4.79. ZbMocReadUserArray 935-936
12.9.1.4.80. ZbMocReadUserVar 936
12.9.1.4.81. ZbMocSDOGetUsage 936-937
12.9.1.4.82. ZbMocSDOGetUsageDetails 937-938
12.9.1.4.83. ZbMocSetCallbackPDO 938

12.9.1.4.84. ZbMocSetCallbackProgress 938-939
12.9.1.4.85. ZbMocSetDebugFile 939
12.9.1.4.86. ZbMocSetDebugFlags 939-940
12.9.1.4.87. ZbMocSetDebugHistory 940
12.9.1.4.88. ZbMocSetDebugImmediate 940
12.9.1.4.89. ZbMocSetDebugLevel 940-941
12.9.1.4.90. ZbMocSetDebugMaxLen 941
12.9.1.4.91. ZbMocSetDebugRetain 941
12.9.1.4.92. ZbMocSourceDeleteFromController 941-942
12.9.1.4.93. ZbMocSourceRestoreFromController 942
12.9.1.4.94. ZbMocSourceSaveInController 942-943
12.9.1.4.95. ZbMocUserParamReadRaw 943
12.9.1.4.96. ZbMocUserParamWriteRaw 943-944
12.9.1.4.97. ZbMocV24ReadRawChar 944
12.9.1.4.98. ZbMocV24ReadRawString 944-945
12.9.1.4.99. ZbMocV24WriteRaw 945
12.9.1.4.100. ZbMocVersionCounts 945-946
12.9.1.4.101. ZbMocWriteUserArray 946
12.9.1.4.102. ZbMocWriteUserVar 946-947
12.9.2. ZbComp Library 947-948
12.9.2.1. Function Descriptions 948
12.9.2.1.1. ZbCompFreecode 948
12.9.2.1.2. ZbCompFromFile 948
12.9.2.1.3. ZbCompApossCFromFile 948-949
12.9.2.1.4. ZbCompGetCodesize 949
12.9.2.1.5. ZbCompGetError 949-950
12.9.2.1.6. ZbCompGetErrorCount 950
12.9.2.1.7. ZbCompGetMinFirmware 950
12.9.2.1.8. ZbCompGetVersion 950-951
12.9.2.1.9. ZbCompOptions 951
12.9.2.1.10. ZbCompReadcode 951-952
13. Glossary 953-955
14. Index 956-979

1 Introduction to the ApossIDE

The **Aposs Integrated Development Environment (ApossIDE)** is a comprehensive software development tool for programming, testing, and debugging motion control solutions by zub machine control AG. The ApossIDE provides two programming languages that may be used to develop user applications that run on zub controllers: ApossC and APOSS.

- **ApossC** is the *preferred* programming language. It is a subset of the popular C programming language and will be very familiar to developers that have previously used the C programming language. It is a more powerful language to use than APOSS.
- **APOSS** is a legacy programming language. It was originally developed to have a simplified syntax structure that is easier to learn. However, this results in some restrictions that are not present in ApossC.

Users who are new to the ApossIDE, should choose to use ApossC rather than APOSS.

APOSS *must* be used with controllers that have firmware versions older than 9.0.00. ApossC may only be used on controllers with firmware version 9.0.00 or newer. All MACS5 and newer controllers may be upgraded to firmware version 9.0.00. Please contact zub Machine Control AG for information.

Help for the APOSS Programming Language is available by using the **Help → APOSS Help (Legacy)** menu command in the ApossIDE or by following this link: APOSS Help.

Both languages offer support for effective programming of drive solutions. They have been extended with high performance motion control commands. Even very complex positioning and synchronization functions can be initiated using simple commands and will run completely autonomously as background tasks. Unlike typical PLC systems, interrupt functions allow fast, real-time reactions that are independent from the program main loop cycle time.

The ApossIDE combines the following components in a high-performance integrated environment:

- ApossC Programming Language
- APOSS Programming Language
- Editor with Syntax-Highlighting
- Context-sensitive Online-Help
- Program and Parameter Handling
- Online-Debugging of Process data and Variables
- Smart-Oscilloscope with real-time support
- Graphical curve editor

The ApossIDE is supported on PC systems running Microsoft Windows 7, Windows 10 or Windows 11. Previous versions of the ApossIDE for older Microsoft Windows operating systems are available with limited support.

The ApossIDE can be freely downloaded by visiting **www.zub.ch**.

1.1 Contact and Legal Information

Contact Information

© zub machine control AG, 2022

All rights reserved. Reproducing or disassembling the computer algorithms in this program is strictly forbidden. All information contained in this manual is subject to technical changes.

ApossIDE Version 7.00.xx

zub machine control AG
Buzibachstrasse 31
CH-6023 Rothenburg
Switzerland

Phone +41 - (0)41 - 54150-40
E-mail info.zub@maxongroup.com

support.maxongroup.com
www.maxongroup.ch

End User License Agreement

This End User License Agreement ("EULA") is entered into between zub machine control AG, Buzibachstrasse 31, CH-6023 Rothenburg ("maxon entity") and customer that accepted this agreement either by manual or electronic signature, or by any type of electronic declaration of intent, such as by clicking a download button or similar. Acceptance of this agreement or using the software indicates that customer has read, understood and accepted these terms. If customer does not accept this EULA, customer must refrain from using the software.

Disclaimer of Warranty

Every effort has been made to make this information and technical data as complete and as accurate as possible, but no warranty or fitness is implied. The information and technical data are provided on as "as is" basis. zub machine control AG shall have neither liability nor responsibility to any person or entity with respect to any loss or damage arising from the information or technical data or from the use thereof.

Trademarks

- APOSS, APOSS OS, and the zub logo are registered trademarks of zub machine control AG.
- Microsoft, Windows 7, and Windows 10 are either registered trademarks or trademarks of the Microsoft Corporation in the USA and other countries.
- VLT is a registered trademark of Danfoss A/S.
- EtherCAT and the EtherCAT logo are registered trademarks of Beckhoff Automation GmbH.

Acknowledgements

zub machine control AG gratefully acknowledges the open source community for generously providing the following software packages:

Scintilla	www.scintilla.org	Text editor functionality. License information
SOEM	openethercatsociety.github.io	EtherCAT Master functionality. License information
zlib	www.zlib.net	File compression library. License information
JPEG	www.ijg.org	JPEG compression library. License information

2 Getting Started and Safety Tips

Safety Tips

Please read these operating instructions in full. In particular, in order to be able to work with the system safely and professionally, please observe the hints and cautionary remarks highlighted with STOP, ⚠, or "!!!" in this manual and in the hardware manuals.

It is absolutely essential to be familiar with, and observe, the safety tips in the hardware manual.

> STOP At all times, it must be possible to switch off the controller and the motor with an EMERGENCY STOP button.

> STOP The motor must be able to turn completely freely so that a sudden jolt cannot cause damage.

Requirements

The ApossIDE is supported on standard PC computers running the Microsoft Windows 7, Windows 10 or Windows 11 operating system. Previous versions of the ApossIDE for older Microsoft Windows operating systems are available with limited support.

You should be familiar with the basic functions and terminology of the Microsoft Windows interface since this manual does not explain these basics. The manual covers the specifics of the ApossIDE only.

Conventions

The information in this manual follows the typographical conventions described below:

> STOP STOP: Instructions to avoid personal injury and equipment damage.

> ⚠ CAUTION: Important information for avoiding problems.

> NOTES: Notes and other information.

> HINTS: Tips and suggestions.

Keyboard and **Function keys** are printed in bold text, for example the **Ctrl** key, the **Esc** key, or the **F1** key.

Buttons in dialog windows are also printed in bold text, for example **OK**.

Menu commands are printed in bold text and include "→" to indicate the menu path, for example **Controller** → **Parameters**.

Keywords for commands and parameters are written in capitals, for example AXEND and KPROP.

Names of parameters are printed in bold faced italics, for example ***Proportional Factor***.

Cross References to other parts of the text are marked in color, for example: System Architecture. Click on the colored link and the help will jump to the desired section. Click on **Back** in the menu bar to return. Cross references to other documents are printed between quotes, for example: See "Hardware reference".

Glossary words are printed in green underlined text, for example: MLONG. Placing the mouse over the colored text will cause the glossary explanation to appear in a popup window.

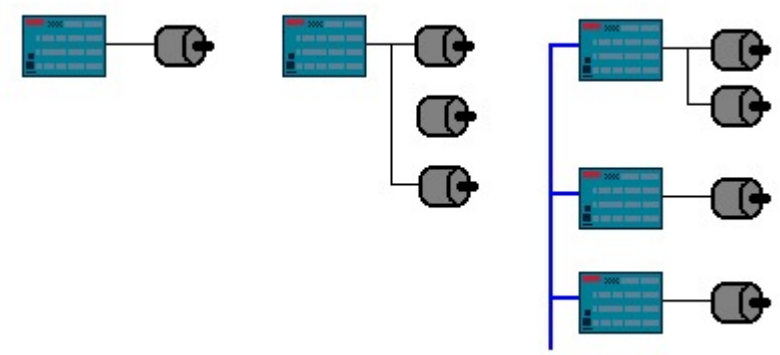
Illustrations in this document showing screen images were created using the ApossIDE running on

Windows 7, Windows 10, and Windows 11 computers. If the AposslDE is running on a different Windows operating system or with different Windows display "Personalization" settings, then the screens will be similar but may not be exactly the same.

2.1 System Architecture

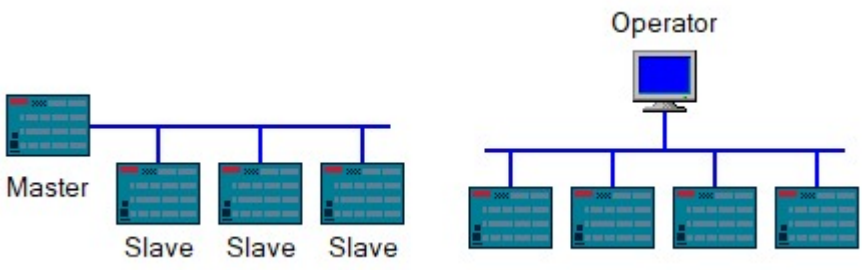
System Configuration

zub motor controllers are designed to be used in a wide variety of system configurations. These systems may be as simple as a single controller and a single motor. Or, depending on which controller is being used, there may be multiple motors connected to a single controller. Even more complex systems can be created by networking multiple controllers together. This can be done, for example, using a Controller Area Network (CAN).



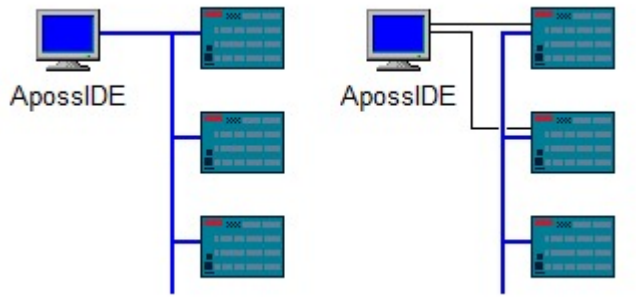
Systems are frequently designed as turnkey systems, operating on their own without an attached PC computer. These systems function by reacting to external “events” or “triggers” that are signalled to the controller by means of digital input pins. Limit switches, optical sensors, pressure sensors, etc., are examples.

Systems may also be configured in a master-slave arrangement, with a single master controller overseeing multiple slave controllers. Communication between the master and slaves can be done using the network connection, digital input and output pins, or both. Another system configuration will include an “operator console” of some sort, often a PC computer running a custom application program.



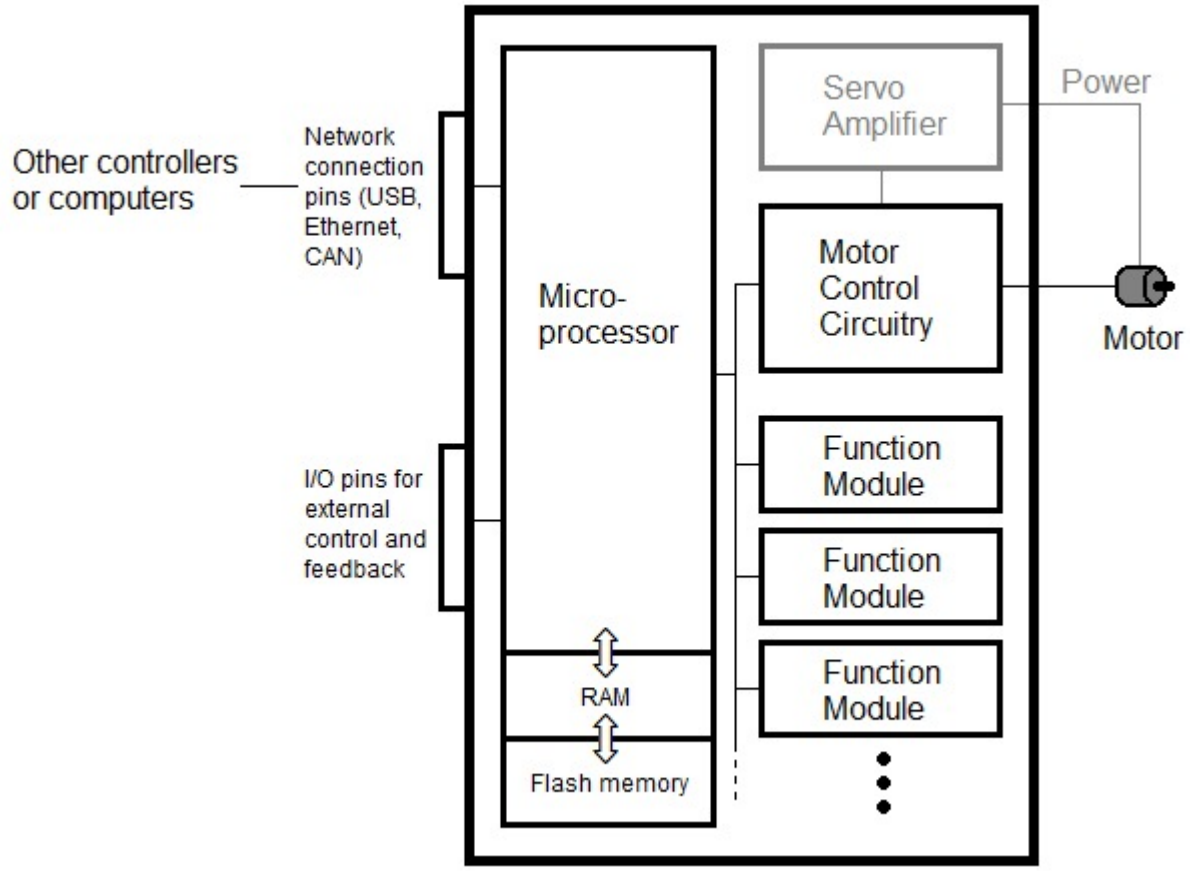
zub motor controllers will handle all of these different configurations with ease.

Each controller in a system will run its own custom application program. These programs will then tailor the behavior of the individual controllers to the exact requirements of the overall system. These programs, written in either the ApossC or APOSS programming language, can be created by the end user himself for his own specific application. This is done using the AposslDE which has been installed on a PC computer. The PC computer is then connected to the controllers either with a private connection or with the shared network. All application programs are developed, downloaded to the controllers, tested, and debugged, using the AposslDE.



Inside the Controller

Like any modern computer, the controller contains a microprocessor, memory, and network communication interfaces. However, the controller also contains specialized hardware tailored specifically for motion control tasks. This includes digital I/O interfaces and other specialized external interfaces that can be used to connect encoders, sensors, and actuators (e.g. signal lamps, lifting magnets, valves, etc.). Some models of controller also include integrated servo amplifiers which allow direct connection to the motor's power terminals. In addition, there may be one or more "function modules" that provide application programs with access to hardware and/or software functions such as encoders, counters, latches, pulse generators, etc.



The operating system (firmware) built into the controller's microprocessor, is an enhanced operating system that provides application programs with access to all internal and external devices, interfaces, and motor control functions. The operating system provides "high-level" programming interfaces to complex functions like motor positioning, synchronization, CAN-communication, etc. These functions are then executed by the operating system

as autonomous background tasks, freeing the application program to focus on more important tasks. For example, position information from absolute or incremental encoders is processed and the speed, positioning, and synchronization profiles are calculated and tracked automatically.

Typical industrial and PC network interfaces (e.g. CAN, USB, and EtherCAT) allow data exchange with other systems and peripherals. Digital and analog I/O interfaces provide an additional means for application programs to control external hardware, receive feedback from external hardware, and communicate with other controllers and computers. These network interfaces and I/O interfaces offer the possibility to use the controller, not only as a standalone unit, but also as part of a powerful and complex control system consisting of multiple interconnected devices such as PCs, PLCs, servo amplifiers, frequency converters, and motors. The system can be as simple or as complex as the application requires.

Function Modules

The firmware provides special functionality with help of function modules. There are modules for digital/analoge inputs and outputs, axis modules, bus modules for bus communication and more. There are also virtual modules for each group to provide the access to the modules out of the application program. The virtual modules are linked by default to the most used hardware modules or other function modules for an easy start. However the modules can be re-linked freely to configure the system that fits best to the user's demands. This is done by writing to specific parameter(s) (process data source) as available by the module. The re-linking can be done at any time while running but in most applications it should be done once at the beginning.

For more information on function modules, please see Controller Reference.

Microprocessor Memory

The controller contains a block of volatile RAM memory, which keeps its data only while the controller is in power-on state, and a block of non-volatile "flash" memory, which is able to maintain its data (e.g. programs, user data) even in power-off state. All data in the flash memory is "permanent" and is preserved across power off/on cycles. *The RAM memory is always cleared and re-initialized each time the controller is reset (e.g. powered off/on)*. This re-initialization is done by copying the appropriate information from the flash memory to the RAM memory. The controller executes programs and processes data *only* in the RAM memory.

Various functions in the ApossIDE and in the application programming languages allow the user to explicitly save information from RAM memory back into flash memory. This provides the means for the user to pre-configure the controller so that the next time the controller is powered on, appropriate control settings are already preset. This includes the ability to automatically start a saved application program.

> ⚠ The EPROM used for flash memory has limited lifetime. However, it can be reprogrammed approximately 10000 times.

User Application Programs

User applocation programs are written in either the ApossC or APOSS programming language, compiled using the ApossIDE, and then downloaded to the controller. The controller simultaneously supports a single "temporary" program saved in RAM and up to 128 different "permanent" programs saved in flash memory. The number of permanent programs depends on the amount of available memory in the controller, the size of the programs, and the firmware version of the controller. Controllers with firmware versions prior to 7.0.00 can only save up to 91 permanent programs. The temporary program saved in RAM will always be lost when the controller is reset (e.g. powered off/on). Permanent programs will persist through power off/on cycles.

Programs are *always* executed as temporary programs in RAM. If a permanent program is executed, then it is first automatically copied from flash memory to RAM and then executed in RAM. Since only one temporary program at a time is supported, executing a subsequent program will overwrite any previous temporary program.

Each time the **Development** → **Execute** command is executed, the current application program being edited in the ApossIDE is compiled and downloaded as a temporary program to RAM. This will overwrite any previous temporary program. It will not alter any of the existing permanent programs.

Programs can be saved permanently by using **Controller** → **Programs**. This will download and save the program in both RAM *and* flash memory. Program source can also be optionally downloaded and saved.
Controller → **Programs** can be used to view and manage all the permanent programs saved on the controller. It can also be used to manually start any of the permanent programs.

The controller is designed to be able to Autostart any of the permanent programs during power up. As well as starting a predefined program, the controller also has the ability to select a program to be started based on signals connected to the controller's input pins. These pins could be set by a PLC, for example. The configuration of the auto start program is done by setting various Global Parameters to appropriate values. For details on how to set parameters, see **Controller** → **Parameters** → **Global / Axis**.

System Parameters

The controller supports a significant number of operational "parameters". These can be set by the user to control all aspects of the operation of the controller. For example, maximum velocities, maximum accelerations, maximum allowable tracking errors, etc., can be specified. See Parameter Reference for a complete list of the available parameters.

The controller supports a "temporary" set of all parameters in RAM only (a "permanent" copy of all parameters in flash memory is no longer supported). The parameters located in RAM will keep their values only during a *single* execution of an application program. At power-on and *each* time a program starts, the parameters are restored with default values. This has the advantage that application programs always start with the same known parameter values. While user application programs are executing the parameters in the controller can be changed at any time by the application program or through SDO access.

Parameter values may be changed by the user either with **Controller** → **Parameters** → **Global / Axis** or with the Module Editor or Array Editor. They may also be changed by an application program using the Sysvar array. However, parameter values that are modified by a user application program are updated *only* in RAM and must be explicitly copied to an array and saved in flash memory (using the Save command) if that is the user's intent. Otherwise, after a controller reset or the next time a program is started, the modified temporary parameter settings are discarded and the settings will be restored to the default parameter values. Parameters in arrays and user parameters can be reset with help of the ApossIDE. **Controller** → **Reset** → **Parameter**.

The ApossIDE can be used to back up the user parameter and array values to a file on the PC hard drive. This is done using **Controller** → **Parameters** → **Save to file**. These values can then be restored by the user at a later time with **Controller** → **Parameters** → **Restore from file**. As a result, **Restore from file** cannot generally be used to initialize parameter values for application programs. To handle the issue of initializing temporary parameters, users should design their programs either to use #config statements or to use "Parameter Include Files" (see Initializing Temporary Parameters).

For more information on parameters, please see Parameter Reference.

User Data Arrays

The controller also provides space within RAM and flash memory for "temporary" and "permanent" user data arrays. As with system parameters, the permanent user data arrays are copied from flash memory to RAM at power-on and *each* time a program starts. Executing user programs use the copies in RAM.

User data arrays are accessed by user programs using DIM Arrays. They are intended to be used to save large blocks of application-specific data that must persist across program starts or be exchanged between different programs. For examples, see CAM Curves or Path Curves. There may be an arbitrary number of DIM arrays and each may have an arbitrary length (limited by the amount of available memory on the controller). Internally, DIM arrays are numbered consecutively starting at 0. All programs in the controller will share access to the *same* user data arrays.

The number of DIM arrays in use may be defined differently in different programs and not all programs need to define or use all arrays. However, the size of each DIM array in use *must* correspond for all programs. The naming of the DIM arrays inside a program has no meaning outside of that program. DIM arrays are internally identified

by their array number which is determined by their position within the DIM definition section of the program. Hence, it is possible to assign different names to the same DIM array in different programs, but it is not possible to assign different array sizes.

DIM arrays can be viewed and managed using the Array Editor or the CAM Editor. DIM arrays modified by the user with the ApossIDE are automatically updated in *both* RAM and flash so they will become permanent. DIM arrays will also be created automatically if they do not already exist when an application program that uses them is executed. DIM arrays created by application programs will be created *only* in RAM and will therefore be temporary. Also, all modifications made to DIM arrays by programs will be temporary. Programs can use the Save command to save DIM arrays permanently in flash memory.

All DIM arrays can be backed up using **Controller** → **Parameters** → **Save to file** and restored using **Controller** → **Parameters** → **Restore from file**. The arrays will be restored to *both* RAM and flash memory. The user can delete all DIM arrays with **Controller** → **Reset** → **Arrays**.

2.2 Installing the ApossIDE

The ApossIDE software may be freely downloaded from the **maxon online shop**. It may also be requested free of charge by e-mail, telephone, or fax (see Contact Information).

Download and install the software as follows:

1. Visit the **maxon online shop**, select a MACS Controller, and then select the **Downloads** page.
2. Navigate to **Software/Firmware.**
3. Download the ApossIDE install program to your hard drive.
4. Execute the install program by double-clicking on it.
5. Follow the instructions displayed by the install program.

See Requirements for details of supported operating systems.

The ApossIDE can be installed over top of a previously installed version of the ApossIDE. It is not necessary to uninstall the old version before installing the new version.

> If you will be using a CAN-USB stick to connect to controllers or using a USB connection to controllers, then you must install the ApossIDE *before* plugging in either the CAN-USB stick or the controller for the first time. Otherwise, Windows will install the wrong device driver software and it will not be possible to connect to the controller.

> The version of the currently installed ApossIDE can be found in the Windows Registry here:
> HKEY_CURRENT_USER/Software/zub machine control AG/Aposs/Version

Uninstalling the ApossIDE

The ApossIDE software can be uninstalled in the normal way using the Windows Control Panel. Do this as follows:

1. Ensure that all CAN-USB sticks and controllers have been unplugged. Otherwise, Windows will not unload the device drivers and the drivers will not be uninstalled.
2. Start the Windows **Control Panel** and select **Programs and Features** (for Windows 7). Other operating systems such as Windows 10 or Windows 11 will be similar but may not be exactly the same.
3. Look for all of the following in the list. If found, then select them and remove them.
   - ApossIDE #.#.#
   - APOSS Software
   - APOSS Version #.#
   - Windows Driver Package - zub USB COM Driver Package
   - Windows Driver Package - zub USB Driver Package

- Windows Driver Package - FTDI CDM Driver Package

2.3 Wiring the Controller

Before starting the ApossIDE, the controller should be wired according to the "Installation Manual and Hardware Reference" corresponding to the specific type of controller being used. These documents may be downloaded from the **maxon online shop**. They may also be requested free of charge by e-mail, telephone, or fax (see Contact Information).

Download them as follows:

1. Visit the **maxon online shop**, select the Controller, and then select the **Downloads** page.
2. Select the **Hardware Manuals** page.
3. Click on the link for the appropriate document to start the download.

If a controller is not available at this time or there is some reason that it cannot be wired, then it is possible to continue with the ApossIDE by using the Aposs Simulator. This is a software module included in the ApossIDE that can simulate a controller without the need of a physical controller. See Simulator Connections and Simulator for more information.

If you will be using a CAN-USB stick to connect to controllers or using a USB connection to controllers, then you must install the ApossIDE *before* plugging in either the CAN-USB stick or the controller for the first time. Otherwise, Windows will install the wrong device driver software and it will not be possible to connect to the controller.

Before turning the controller on: For safety reasons, at this point in time it should *not* be possible for the motor to accidentally drive equipment. A newly installed controller can drive a motor in unpredictable ways and damage equipment before the wiring has been tested and the internal controller parameters set for proper operation. *Make sure that the motor is not connected to equipment before turning the controller on*.

Wiring the Controller to the Computer

Controllers can be connected to the computer running the ApossIDE, using any of several different types of physical connections. The ApossIDE can connect to more than one controller at the same time if different physical connections are used for each controller. Some physical connections (e.g. CAN and USB) also support connections to multiple controllers using the same physical connection. Note that not all types of controllers support all physical connections.

The following types of physical connections are available:

V24	Serial connection using either an RS232 COM port or a USB port with a USB-to-RS232 converter.
CAN-USB	CAN network connection using a USB port with a CAN-USB stick.
USB	Direct connection using a USB port.
EtherCAT	EtherCAT network connection.
Network	Ethernet (TCP/IP) connection.

Please see the "Installation Manual and Hardware Reference" for the specific controller to determine which physical connections are supported by that controller and for instructions on how to wire the actual physical connection.

Once the physical controller connection is complete, then the controller can be powered on and the ApossIDE started.

If this is the first time that a CAN-USB stick or a USB controller is connected, then one or more messages similar to the following may be displayed at the bottom-right corner of the screen. These indicate that the computer has recognized the device and loaded the device drivers for it. Please wait for the "ready to use" message. Note that these messages may not be displayed on Windows 10 computers.

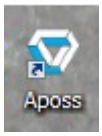
> If you see a "Found New Hardware Wizard" dialog, then the ApossIDE has not been installed. In this case, click the "Cancel" button, unplug the controller, install the ApossIDE, and finally plug the controller back in.

2.4 Starting and Exiting the ApossIDE

Starting the ApossIDE

The ApossIDE can be started in any of the following ways:

1. In the Windows task bar, click on **Start** → **All Programs** → **zub machine control AG** → **ApossIDE** to start the ApossIDE. Note that "zub machine control AG" is the default install location. The user is free to choose a different location during the ApossIDE installation.
2. If a desktop icon was created during the ApossIDE install, then double-clicking on the icon will start the ApossIDE. The ApossIDE icon is shown below.

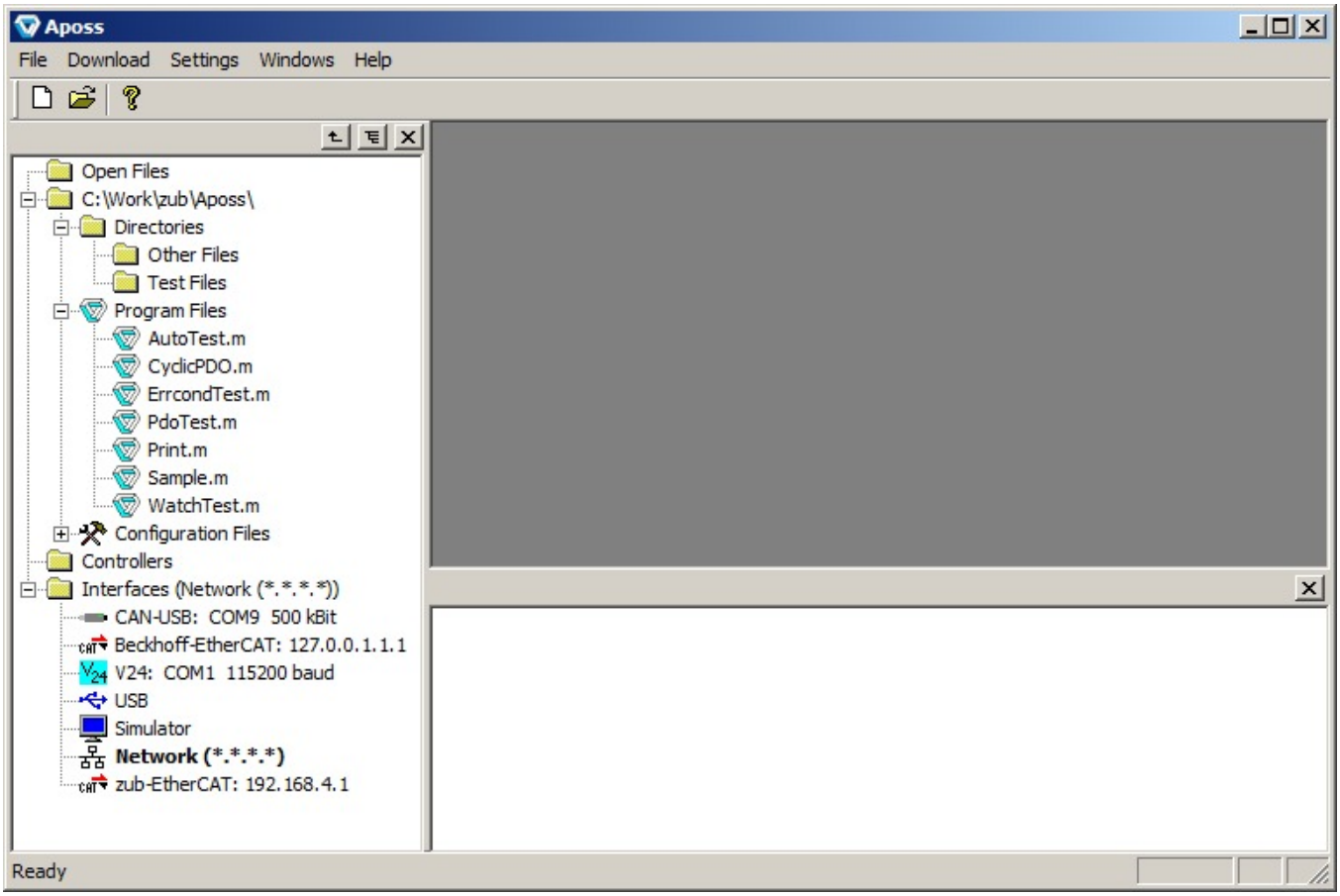


3. If a Quick Launch icon was created during the ApossIDE install, then single-clicking on the icon will start the ApossIDE. Quick Launch icons appear in the bottom-left corner of the Windows screen as is shown below.

4. If an ApossIDE file (i.e. an ".m", ".mc", ".zbo", ".zbc" file, etc.) already exists, then double-clicking on the file in the Windows folder should automatically start the ApossIDE and open that particular file.

When the ApossIDE starts, it will try to automatically open the files that were previously open when the ApossIDE was last used (unless the ApossIDE was started by double-clicking on an ApossIDE file, in which case that file is opened). This behaviour can be disabled using **Settings** → **Options**.

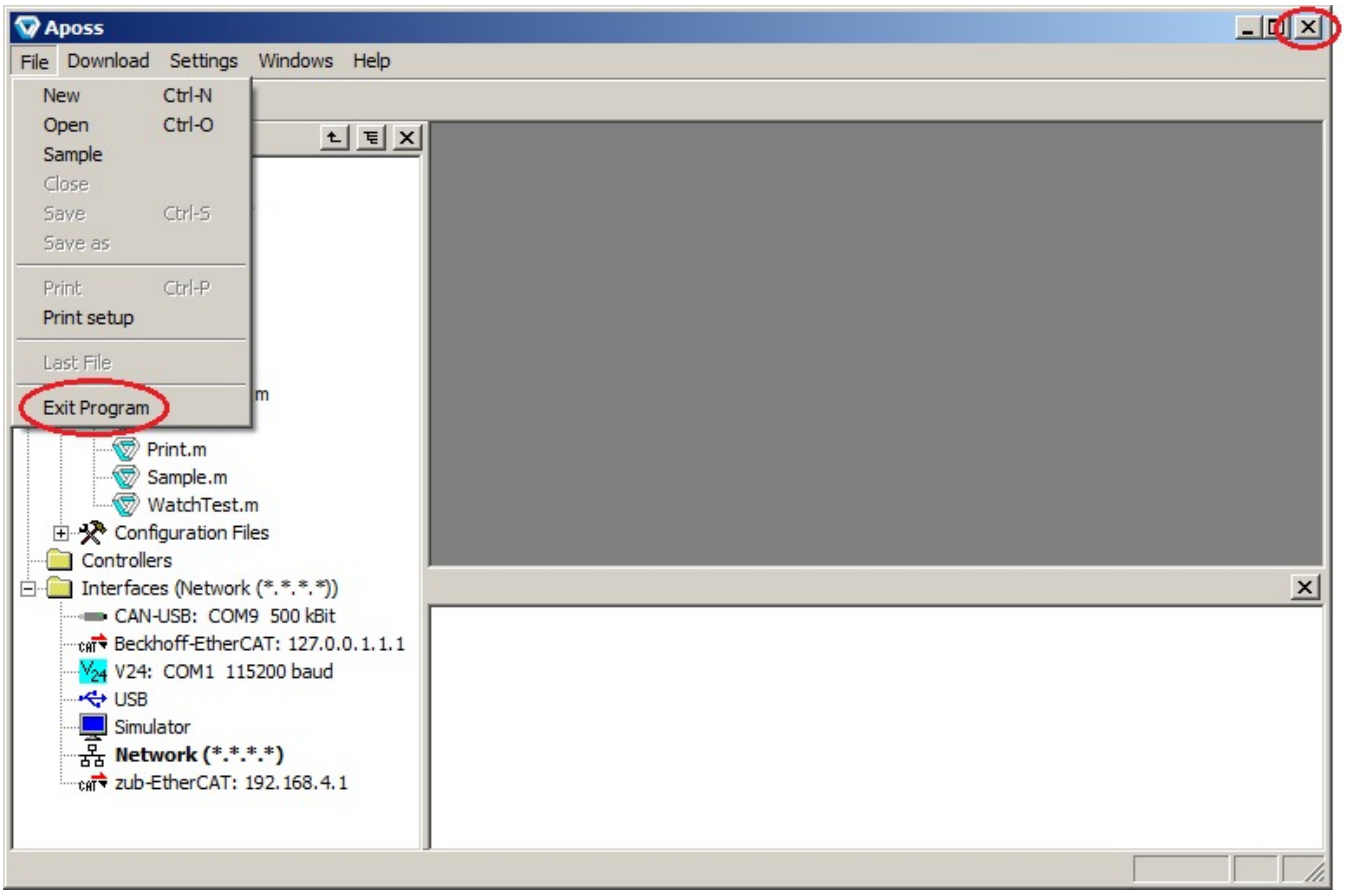
If the ApossIDE cannot find any last file to open, then the initial ApossIDE window should be similar to that shown below.



Exiting the ApossIDE

Exiting the ApossIDE can be done at any time simply by clicking on the Windows "Close" button in the top-right corner of the main ApossIDE window or by clicking on **File** → **Exit Program**.

> ⚠ Exiting the ApossIDE will *not* abort or end any program executing on the controller. If you want to abort or end the program executing on the controller, then you must do so *before* exiting the ApossIDE. To do this, you must have an ApossIDE Edit Window open and you must have opened a connection to the controller. Pressing the **Esc** key will then terminate the program executing on the controller.
>
> Note that for safety reasons, the ApossIDE allows the **Esc** key to be disabled. Please see Esc key for more information.



2.5 Connecting to the Controller

Connect the AposslDE to the controller using the following steps:

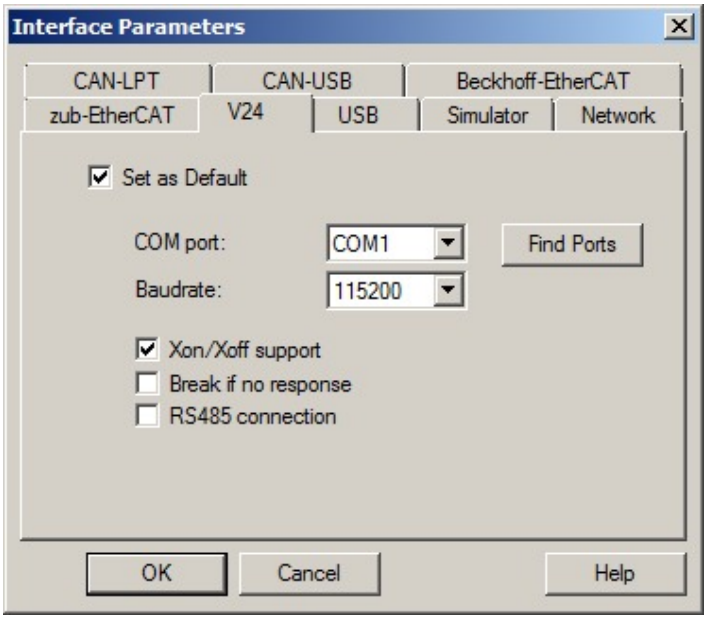
1. If an AposslDE Edit Window is not already open, then open an Edit Window by either creating a new application program (.mc or .m) file, opening an existing application program file, or opening a sample application program file. Do this using the **File** → **New**, **File** → **Open**, or **File** → **Sample** menu command.
2. Select the topic below according to the type of physical connection being used for the controller and then follow the directions.

   V24 (Serial) Connections
   CAN-LPT Connections
   CAN-USB Connections
   USB Connections
   EtherCAT Connections (Beckhoff)
   EtherCAT Connections (zub)
   Network Connections
   Simulator Connections

2.5.1 V24 (Serial) Connections

V24 connections are serial connections that use either an RS232 COM port or a USB port with a USB-to-RS232 converter.

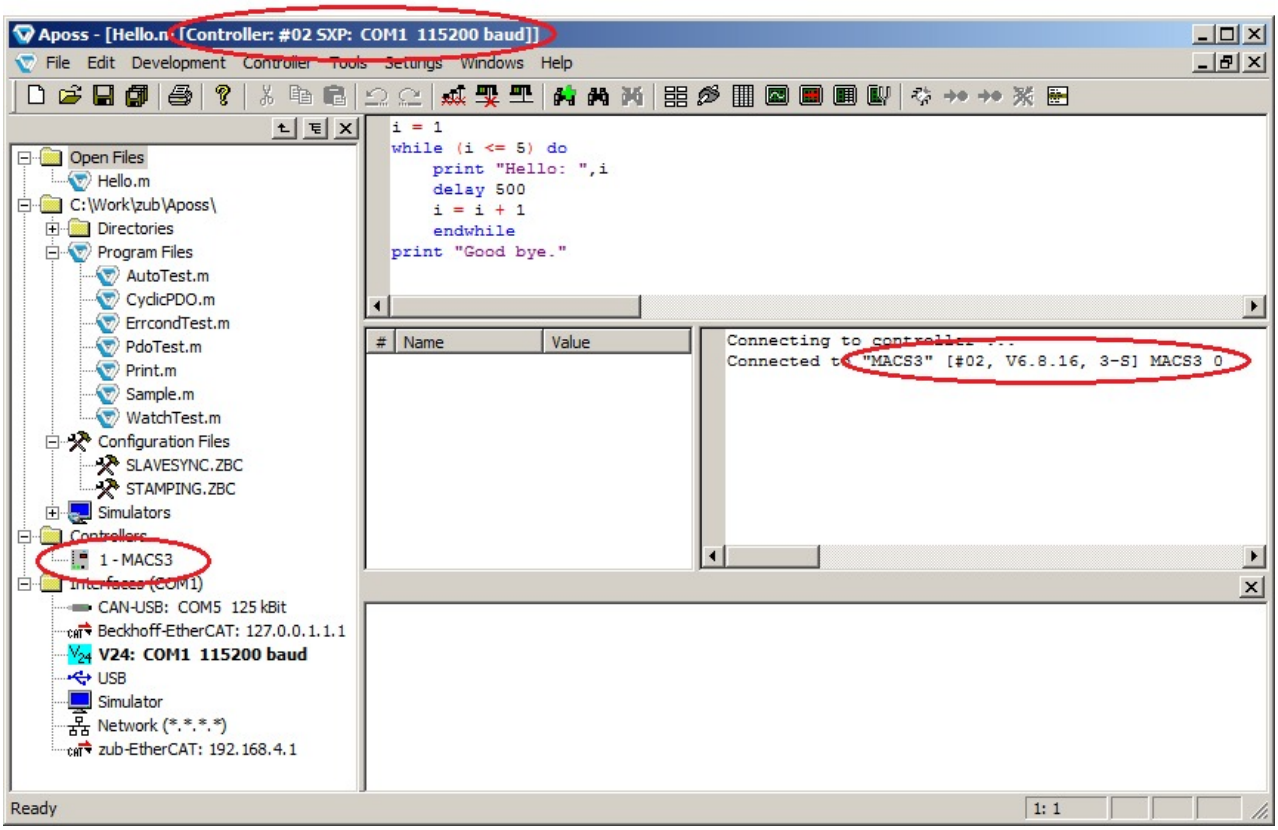
1. Click on **Settings** → **Interface** to open the Interface Parameters dialog. Select the **V24** tab.



2. Check the ☑ **Set as Default** box.
3. Select the **COM port** that you will be using. If the desired COM port is not listed, then click the **Find Ports** button and check again for the COM port. If a USB-to-RS232 converter is being used, then it will be necessary to plug the converter into the computer before Windows will be able to find the associated COM port.

   > If a USB-to-RS232 converter is being used, then the correct virtual COM port must be selected in the Interface Parameters dialog. The correct COM port can be identified by opening the Windows **Control Panel**, selecting **System**, selecting **Hardware**, clicking the **Device Manager** button, and finally opening the **Ports (COM & LPT)** item. Look for the COM port that corresponds to the USB-to-RS232 device.

4. Choose the **baudrate** that the connection will use. Note that some older controllers support only 9600 baud. For these controllers, the baud rate setting is ignored.
5. ☑ **Xon/Xoff support** should normally be enabled. It should be disabled if a USB-to-RS232 converter is being used.
6. **Break if no response** should be disabled unless it is necessary to enable it. The ApossIDE cannot successfully establish a connection to some older controllers if the controller is executing a program. If the ApossIDE returns the message "No Controller found!" when an old controller is properly connected to the correct COM port, then you should enable the ☑ **Break if no response** setting and try again. Note that this will abort the program executing on the controller!
7. ☑ **RS485 connection** should be enabled if an RS232-to-RS485 converter has been inserted into the connection. This allows the ApossIDE to adjust the communication timing to compensate for differences between RS232 and RS485.
8. Click the **OK** button. This will close the dialog and return you to the Edit Window.
9. In the Edit Window, press the **Esc** key. The ApossIDE will then try to connect to the controller. If this is successful, then the controller will be identified in the window as shown:



The above example shows a successful connection. The window title shows the controller CAN ID (#02), that a serial protocol connection (SXP) is being used on COM1, and that the connection is running at 115200 baud. The Sidebar shows a MACS3 icon, the connection ID (1), and the controller name (MACS3). The Communication Window shows the controller name (MACS3), CAN ID (#02), firmware version (6.8.16), hardware options (3-S), and the type of controller (MACS3).

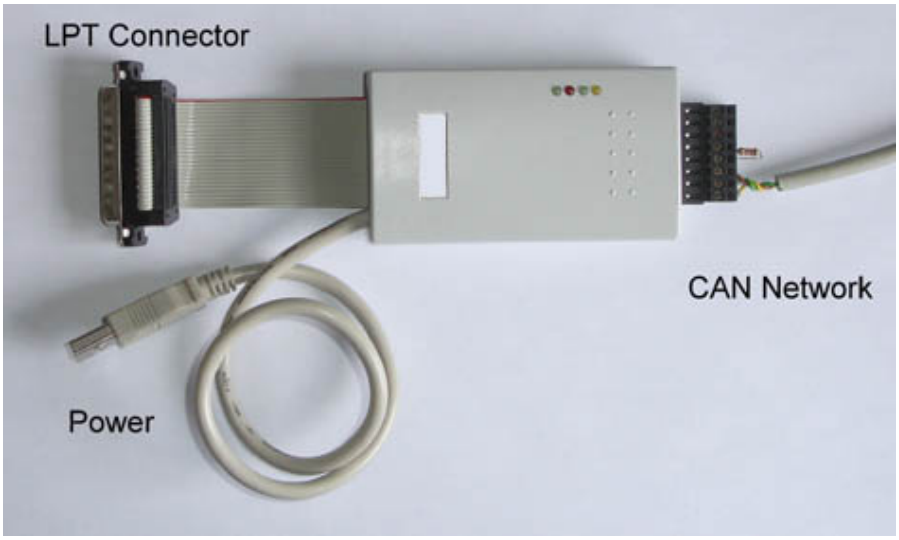
> The connection ID for V24 connections is always 1, regardless of the CAN ID. Also, the CAN ID will always be displayed with a "#" prefix in order to distinguish it from the connection ID.

If "No controller found!" is reported in the Communication Window, then the connection was unsuccessful. Please check for the following:

1. The controller is powered on.
2. The serial cable is firmly connected to the computer.
3. The serial cable is connected to the correct COM port. See below if a USB-to-RS232 converter is being used.
4. The serial cable is correctly wired to the controller serial pins (see the "Installation Manual and Hardware Reference" for the controller).
5. For some very old controllers, the ApossIDE cannot successfully establish a connection if the controller is executing a program. Enable the ☑ **Break if no response** setting and try again. Note that this will abort the program executing on the controller!

2.5.2 CAN-LPT Connections

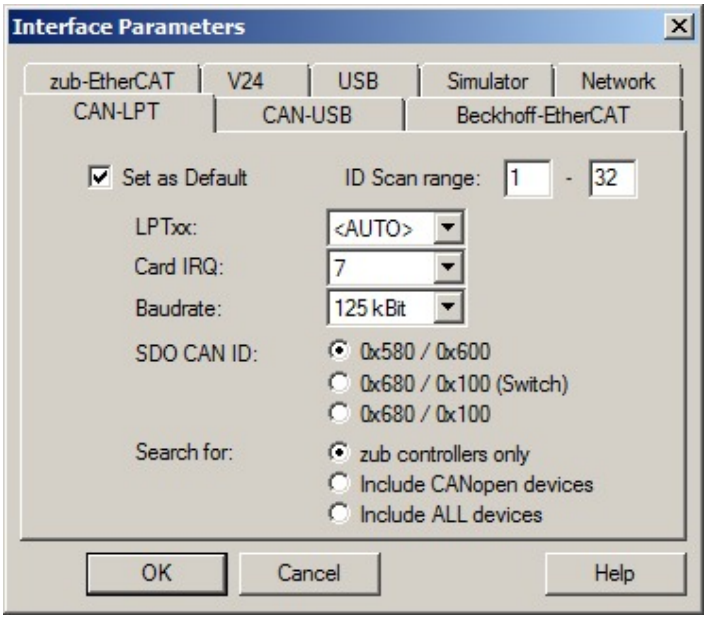
CAN-LPT Adapter with power supply via USB:



CAN-LPT connections will use a CAN-LPT adapter (similar to that shown above) to connect the computer's parallel printer port (LPT port) to a CAN network.

CAN-LPT adapters *are not supported* on Microsoft Windows 7 and later systems.

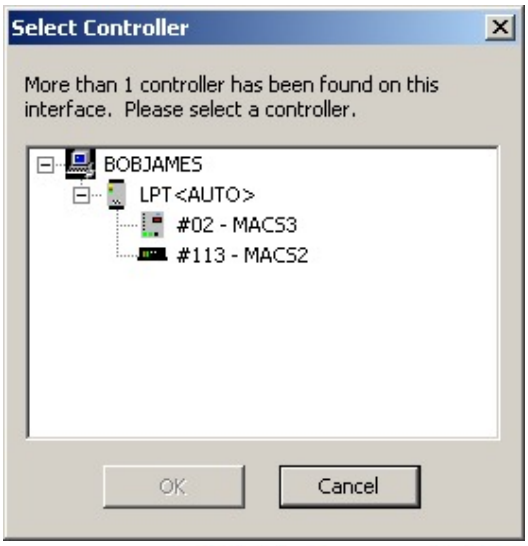
1. Click on **Settings** → **Interface** to open the Interface Parameters dialog. Select the **CAN-LPT** tab.

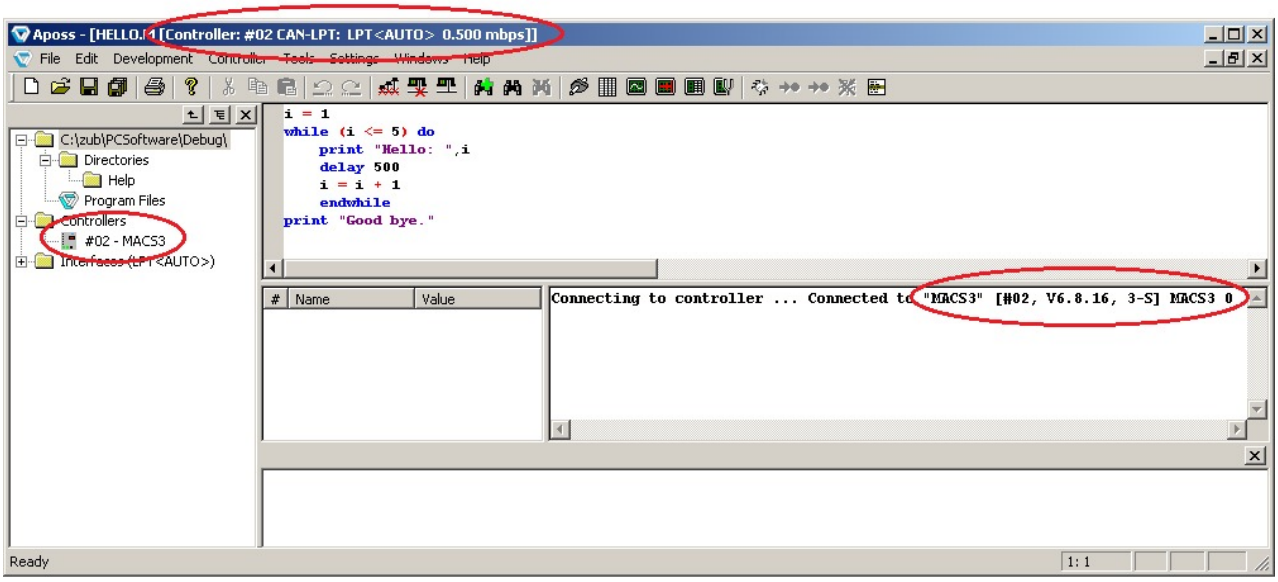


2. Check the ☑ **Set as Default** box.
3. The **LPT port** can normally be left set to <AUTO>. It will be necessary to set this to a specific LPT port if multiple CAN-LPT modules are being used.
4. Set the **Card IRQ** according to the manufacturer's installation documentation that came with the CAN-LPT module.
5. Set the **Baudrate** to the baud rate of the CAN network.
6. The **SDO CAN ID** should be left set to "0x580 / 0x600" unless there is a reason to change it. These are the CAN ID numbers that will be used to identify SDO messages sent to and received from the controller. Some controller applications may use these ID numbers for other purposes. In these cases, the alternate ID numbers may be used. Note that some older controllers may not support the alternate ID numbers. If the option labeled "0x680 / 0x100 (Switch)" is selected, then the initial connection will be made using "0x580 / 0x600". If this connection is successful,

then the ApossIDE will automatically switch to the alternate ID numbers “0x680 / 0x100”. If this switch fails, then the ApossIDE will continue to use the original ID numbers.

7. **Search for** can be left set to “zub controllers only”. The ApossIDE is capable of connecting to other types of CAN devices, not only to zub controllers. In these cases, the ApossIDE will have limited functionality.
8. The **ID Scan range** specifies the range of CAN ID numbers that the ApossIDE will use when searching the CAN network for controllers. This range must include the CAN ID of the controller to which the connection is being made. See the “Installation Manual and Hardware Reference” for the controller to see how to set the CAN ID number.
9. Click the **OK** button. This will close the dialog and return you to the Edit Window.
10. In the Edit Window, press the **Esc** key. The ApossIDE will then try to connect to the controller. If this is successful, then the controller will be identified in the window as shown. If more than one controller was found in the ID scan range, then a “Select Controller” dialog (similar to that shown below) will be displayed. Select the appropriate controller using the CAN ID and controller name shown in the dialog and click the **OK** button.





The above example shows a successful connection. The window title shows that the controller has a CAN ID of 2 (#02), that a CAN-LPT connection (CAN-LPT) is being used on the default LPT port (LPT<AUTO>), and that the CAN network is running at 0.5 mbps. The Sidebar shows the MACS3 icon, the CAN ID (#02), and the controller name (MACS3). The Communication Window shows the controller name (MACS3), CAN ID (#02), firmware version (6.8.16), hardware options (3-S), and the type of controller (MACS3).

The CAN ID will always be displayed with a "#" prefix in order to distinguish it from the connection ID used by other types of connection interfaces.

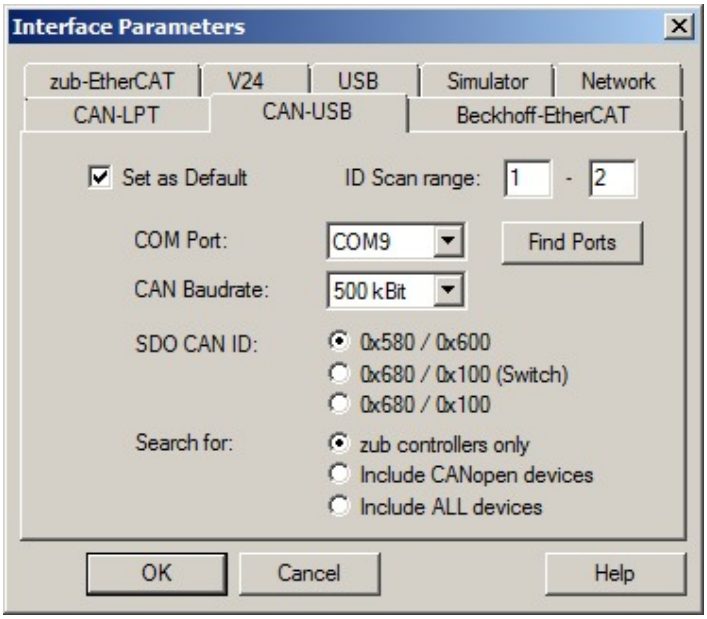
If "No controller found!" is reported in the Communication Window, then the connection was unsuccessful. Please check for the following:

1. The ID Scan range contains the CAN ID of the controller.
2. The baudrate of the CAN network has been set correctly.
3. The controller is powered on.
4. The CAN-LPT module is firmly connected to the computer and is powered on.
5. The CAN-LPT module is connected to the correct LPT port if more than one CAN-LPT module is being used.
6. The CAN network is correctly wired to both the controller CAN network pins (see the "Installation Manual and Hardware Reference" for the controller) and to the CAN-LPT module (see the manufacturer's installation documentation that came with the CAN-LPT module).

2.5.3 CAN-USB Connections

CAN-USB connections will use a CAN-USB stick (similar to that shown above) to connect a USB port to a CAN network.

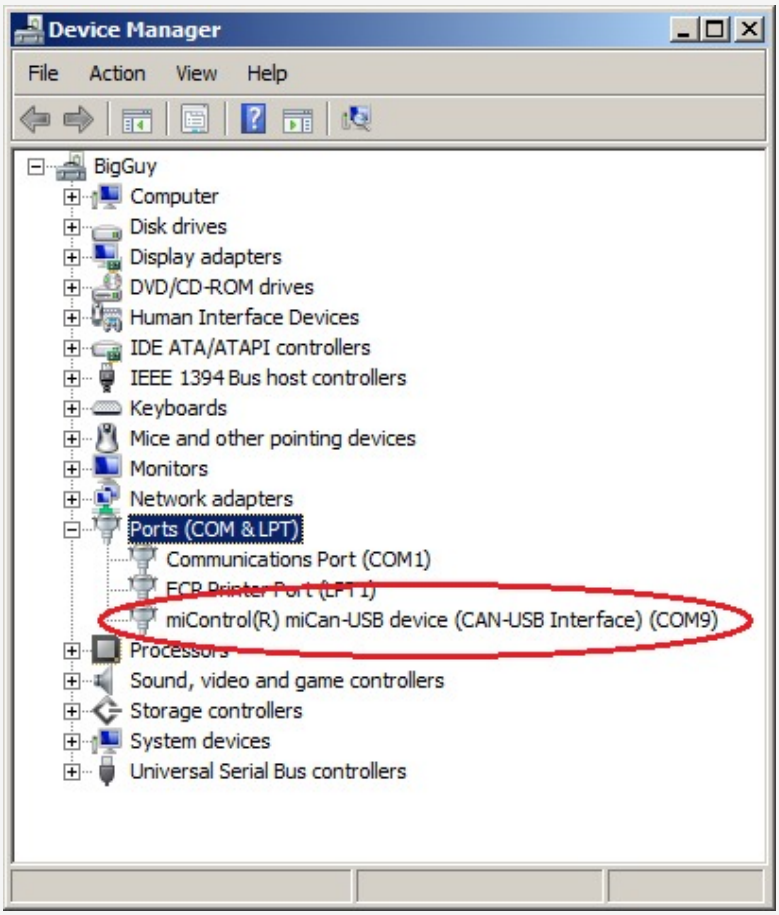
1. Click on **Settings** → **Interface** to open the Interface Parameters dialog. Select the **CAN-USB** tab.



2. Check the ☑ **Set as Default** box.

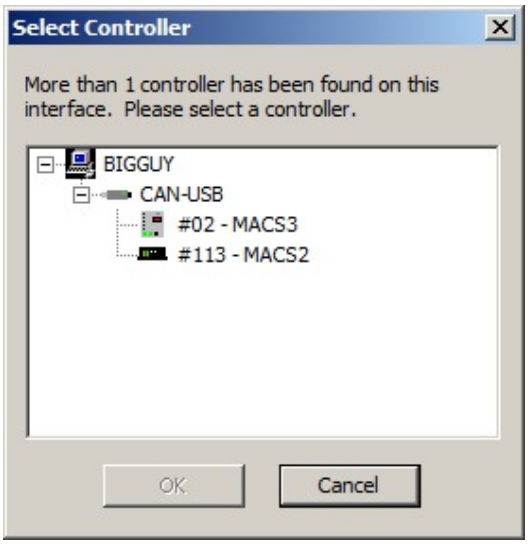
3. Select the **COM Port** that you will be using. If the desired COM port is not listed, then click the **Find Ports** button and check again for the COM port. It will be necessary to plug the CAN-USB stick into the computer before Windows will be able to find the associated COM port.

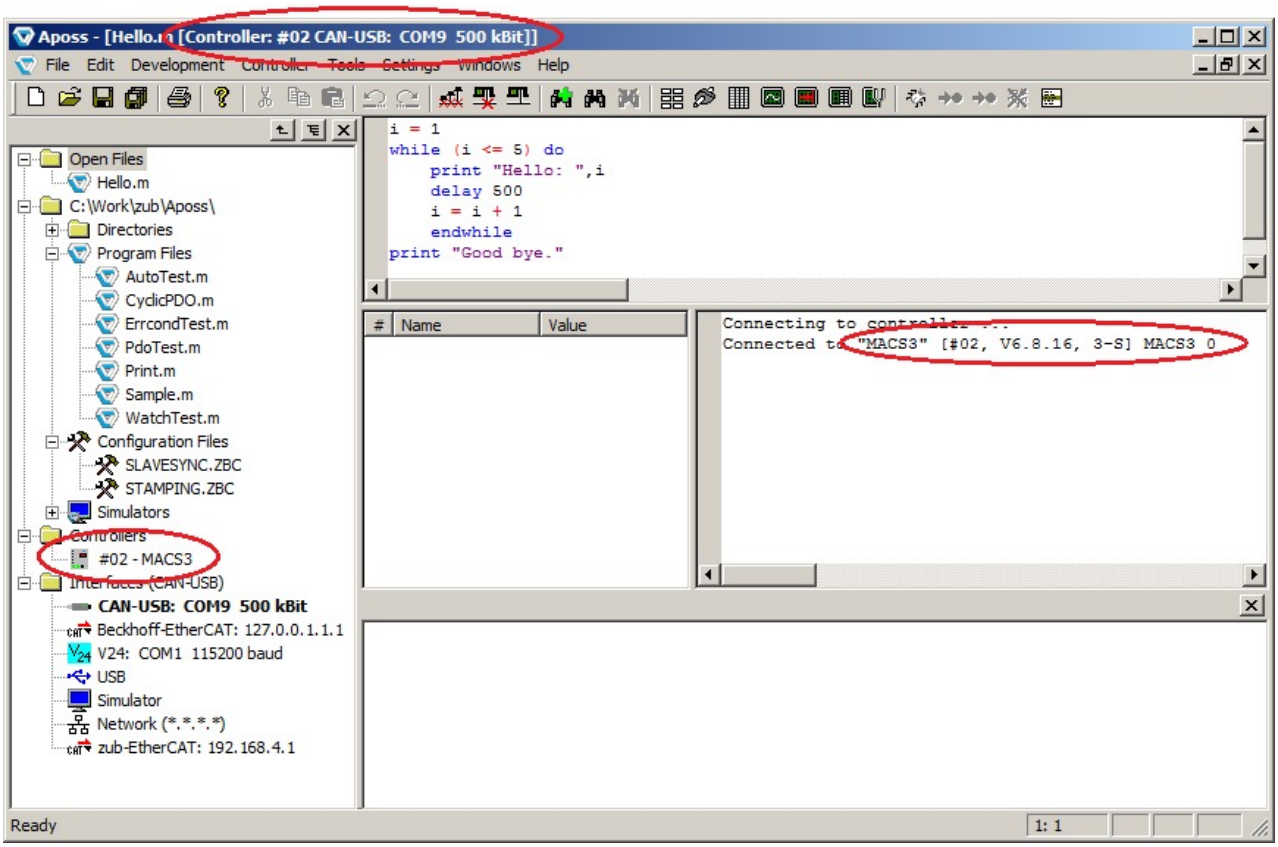
> The CAN-USB stick uses a virtual COM port. The correct COM port can be identified by opening the Windows **Control Panel**, selecting **System**, selecting **Hardware**, clicking the **Device Manager** button, and finally opening the **Ports (COM & LPT)** item. Look for the COM port that corresponds to the CAN-USB stick. It should appear similar to that shown below.



4. Set the **CAN Baudrate** to the baudrate of the CAN network.
5. The **SDO CAN ID** should be left set to “0x580 / 0x600” unless there is a reason to change it. These are the CAN ID numbers that will be used to identify SDO messages sent to and received from the controller. Some controller applications may use these ID numbers for other purposes. In these cases, the alternate ID numbers may be used. Note that some older controllers may not support the alternate ID numbers. If the option labeled “0x680 / 0x100 (Switch)” is selected, then the initial connection will be made using “0x580 / 0x600”. If this connection is successful, then the ApossIDE will automatically switch to the alternate ID numbers “0x680 / 0x100”. If this switch fails, then the ApossIDE will continue to use the original ID numbers.
6. **Search for** can be left set to “zub controllers only”. The ApossIDE is capable of connecting to other types of CAN devices, not only to zub controllers. In these cases, the ApossIDE will have limited functionality.
7. The **ID Scan range** specifies the range of CAN ID numbers that the ApossIDE will use when searching the CAN network for controllers. This range must include the CAN ID of the controller to which the connection is being made. See the “Installation Manual and Hardware Reference” for the controller to see how to set the CAN ID number.
8. Click the **OK** button. This will close the dialog and return you to the Edit Window.
9. In the Edit Window, press the **Esc** key. The ApossIDE will then try to connect to the controller. If this is successful, then the controller will be identified in the window as shown. If more than one controller was found in the ID scan range, then a “Select Controller” dialog (similar to that shown below) will be displayed. Select the appropriate

controller using the CAN ID and controller name shown in the dialog and click the **OK** button.





The above example shows a successful connection. The window title shows that the controller has a CAN ID of 2 (#02), that a CAN-USB connection (CAN-USB) is being used on COM9, and that the CAN network is running at 500 kBit. The Sidebar shows the MACS3 icon, the CAN ID (#02), and the controller name (MACS3). The Communication Window shows the controller name (MACS3), CAN ID (#02), firmware version (6.8.16), hardware options (3-S), and the type of controller (MACS3).

> The CAN ID will always be displayed with a "#" prefix in order to distinguish it from the connection ID used by other types of connection interfaces.

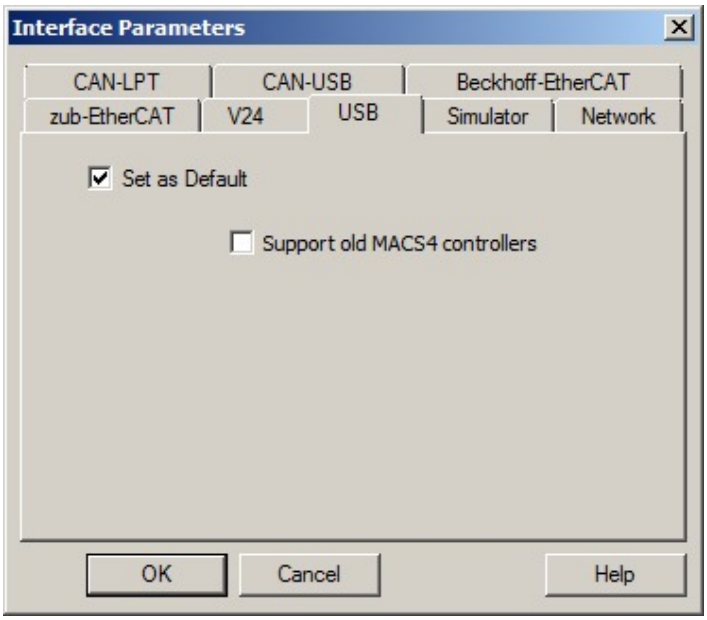
If “No controller found!” is reported in the Communication Window, then the connection was unsuccessful. Please check for the following:

1. The ID Scan range contains the CAN ID of the controller.
2. The baudrate of the CAN network has been set correctly.
3. The controller is powered on.
4. The CAN-USB stick is firmly connected to the computer.
5. The correct COM port has been selected in the Interface Parameters dialog. See below to determine the correct COM port.
6. Some USB hubs do not function properly. If the CAN-USB stick is plugged into a port on a USB hub, then try plugging the CAN-USB stick into a USB port on the actual computer.
7. The CAN network is correctly wired to both the controller CAN network pins (see the “Installation Manual and Hardware Reference” for the controller) and to the CAN-USB stick (see the manufacturer's installation documentation that came with the CAN-USB stick).

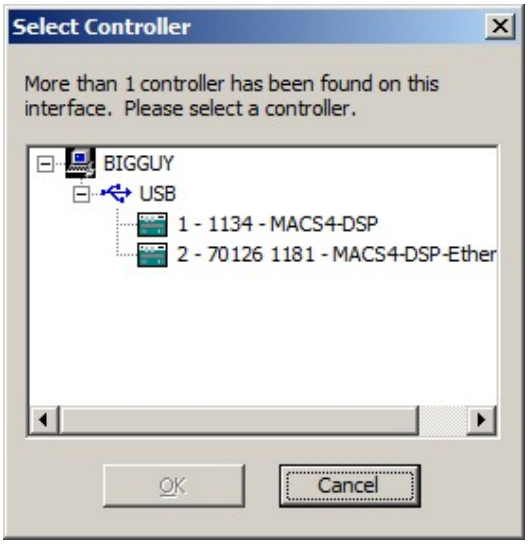
2.5.4 USB Connections

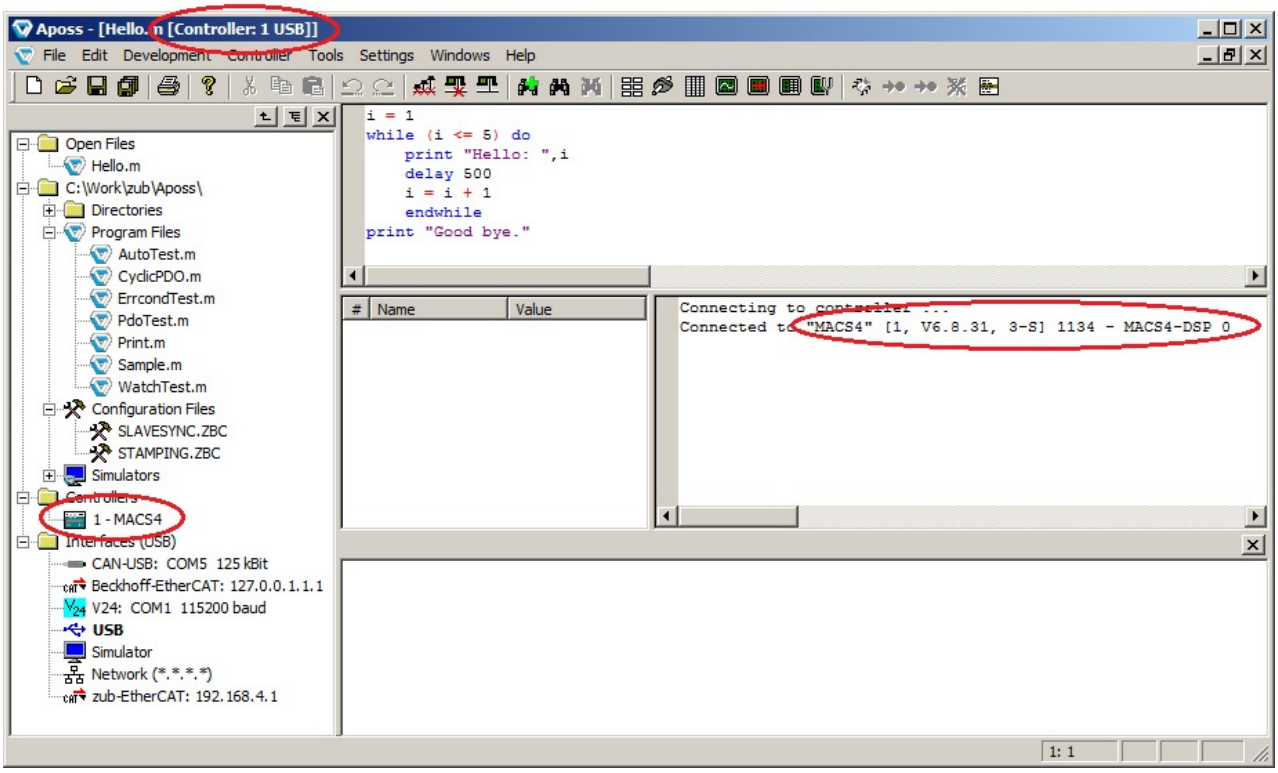
If the controller supports it, then a USB connection can be made directly from the computer to the controller.

1. Click on **Settings** → **Interface** to open the Interface Parameters dialog. Select the **USB** tab.



2. Check the ☑ **Set as Default** box. USB connections require no further setup.
3. Click the **OK** button. This will close the dialog and return you to the Edit Window.
4. In the Edit Window, press the **Esc** key. The ApossIDE will then try to connect to the controller. If this is successful, then the controller will be identified in the window as shown. If more than one controller was found, then a “Select Controller” dialog (similar to that shown below) will be displayed. Select the appropriate controller using the serial number and controller type shown in the dialog and click the **OK** button.





The above example shows a successful connection. The window title shows that the controller has a connection ID of 1 and that a USB connection (USB) is being used. The Sidebar shows the MACS4 icon, the connection ID (1), and the controller name (MACS4). The Communication Window shows the controller name (MACS4), connection ID (1), firmware version (6.8.31), hardware options (3-S), and the type of controller (1134 - MACS4-DSP).

> For USB connections, the CAN ID is not used and is not shown. Also, the connection ID is an arbitrary number assigned by the ApossIDE at the time of connection. If a controller is re-connected at a later time, then it may be assigned a different connection ID.

If "No controller found!" is reported in the Communication Window, then the connection was unsuccessful. Please check for the following:

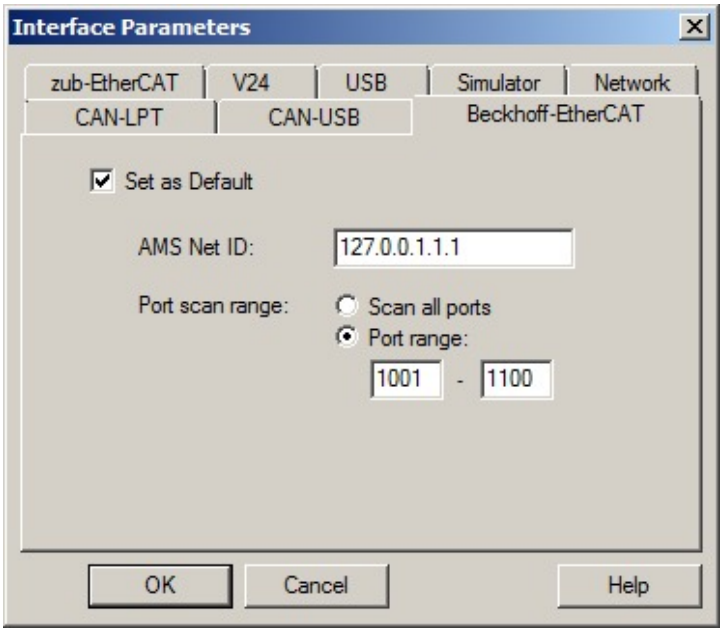
1. The controller is powered on.
2. The USB cable is firmly connected to both the computer and the controller.

3. Some USB hubs do not function properly. If the USB cable is plugged into a port on a USB hub, then try plugging the cable into a USB port on the actual computer.

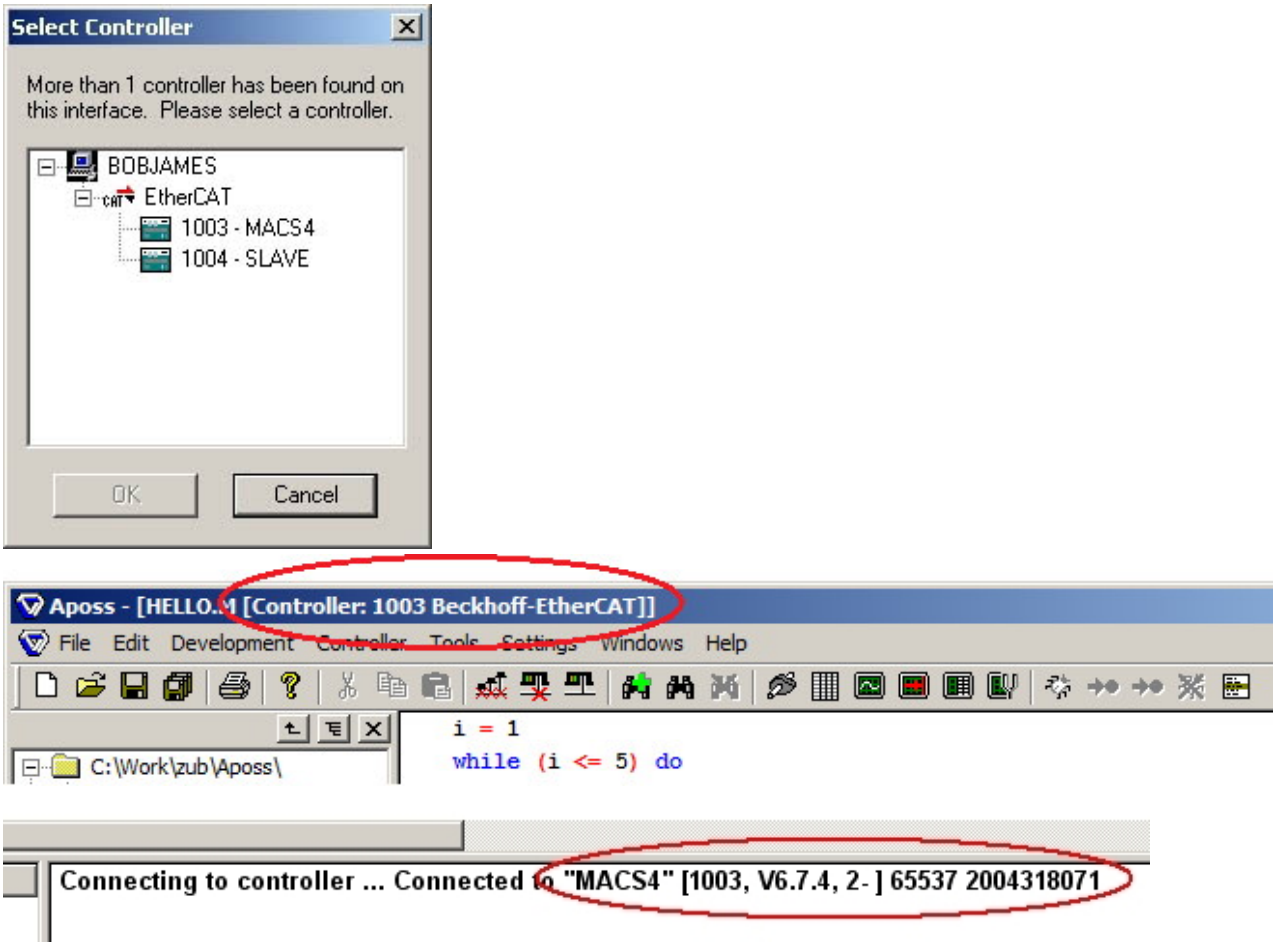
2.5.5 EtherCAT Connections (Beckhoff)

If the controller supports EtherCAT connections, then a connection can be made from the computer to the controller using either a Beckhoff EtherCAT Master or a zub EtherCAT Master. The following information applies to a Beckhoff EtherCAT Master. If you will be using a zub EtherCAT Master, then please see EtherCAT Connections (zub).

1. Click on **Settings** → **Interface** to open the Interface Parameters dialog. Select the **Beckhoff-EtherCAT** tab.



2. Check the ☑ **Set as Default** box.
3. Enter the **AMS Net ID**. This will be an address of six numbers separated by dots (e.g. 5.1.131.8.3.1).
4. The **Port scan range** can normally be left set to all ports unless there is some reason to restrict it.
5. Click the **OK** button. This will close the dialog and return you to the Edit Window.
6. In the Edit Window, press the **Esc** key. The ApossIDE will then try to connect to the controller. If this is successful, then the controller will be identified in both the Edit Window title and the Communication Window. If more than one controller was found, then a "Select Controller" dialog (similar to that shown below) will be displayed. Select the appropriate controller and click the **OK** button.



The above example shows a successful connection. The window title shows that the controller has an ID number of 1003 and that an EtherCAT connection is being used. The Communication Window shows the controller name (MACS4), ID number (1003), firmware version (6.7.4), hardware options (2-), and the type of controller (65537 2004318071).

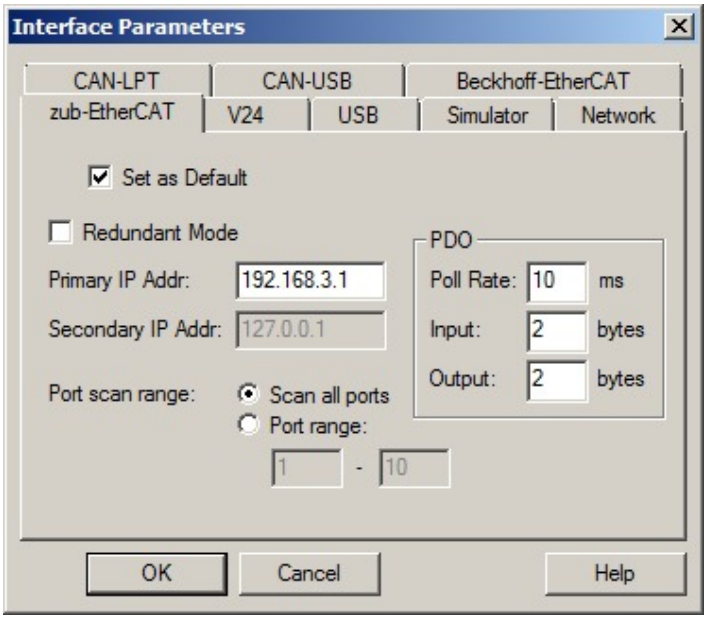
If "No controller found!" is reported in the Communication Window, then the connection was unsuccessful. Please refer to the documentation that came with the EtherCAT hardware and software. Then check for the following:

1. The controller is powered on.
2. The EtherCAT bus is up and running in either "Config" or "Realtime" mode.
3. The bus state of the controller is "OP" (viewable using the EtherCAT System Manager).
4. The AMS Address is correct.

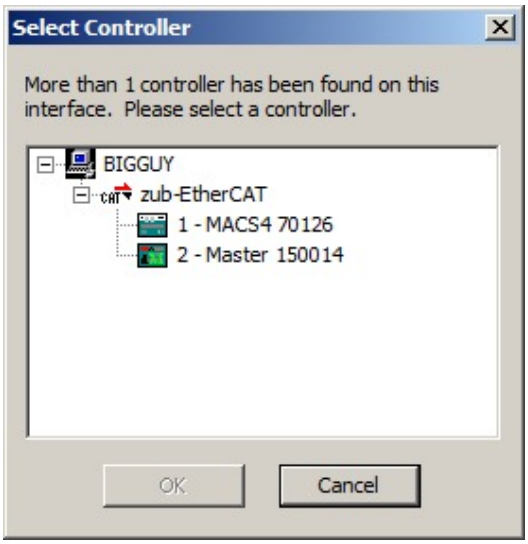
2.5.6 EtherCAT Connections (zub)

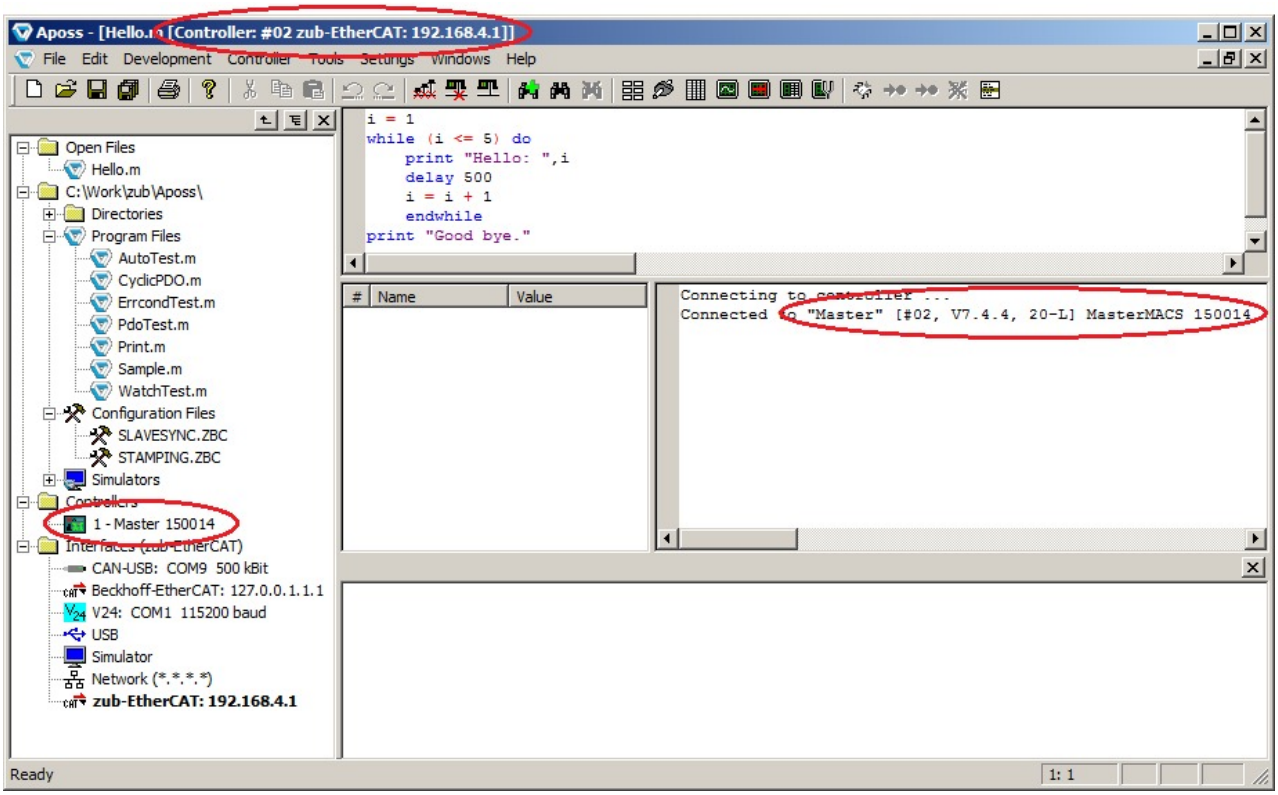
If the controller supports EtherCAT connections, then a connection can be made from the computer to the controller using either a Beckhoff EtherCAT Master or a zub EtherCAT Master. The following information applies to a zub EtherCAT Master. If you will be using a Beckhoff EtherCAT Master, then please see EtherCAT Connections (Beckhoff).

1. Click on **Settings** → **Interface** to open the Interface Parameters dialog. Select the **zub-EtherCAT** tab.



2. Check the ☑ **Set as Default** box.
3. Enter the **Primary IP Address**. This will be the IP address of the Network Interface Card (NIC) that will be used to connect to the controller. This should be four numbers separated by dots (e.g. 192.168.3.1).
4. The **Port scan range** can normally be left set to all ports unless there is some reason to restrict it.
5. Click the **OK** button. This will close the dialog and return you to the Edit Window.
6. In the Edit Window, press the **Esc** key. The ApossIDE will then try to connect to the controller. If this is successful, then the controller will be identified in both the Edit Window title and the Communication Window. If more than one controller was found, then a "Select Controller" dialog (similar to that shown below) will be displayed. Select the appropriate controller and click the **OK** button.





The above example shows a successful connection. The window title shows that the controller has an ID number of 2 and that an EtherCAT connection is being used. The Communication Window shows the controller name (Master), ID number (2), firmware version (7.4.4), hardware options (20-L), and the type of controller (MasterMACS 150014).

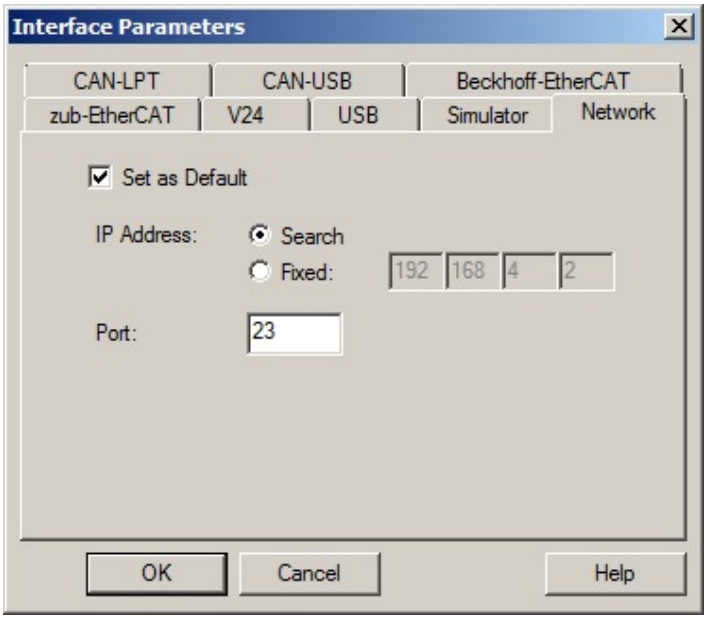
If “No controller found!” is reported in the Communication Window, then the connection was unsuccessful. Check for the following:

1. The controller is powered on.
2. The WinPcap software has been installed (and the ApossIDE re-installed).
3. The Ethernet cable has been connected to the correct port on the controller. It should be connected to the EtherCAT "IN" port and *not* to the Ethernet port.
4. The IP address of the NIC has been correctly specified.

2.5.7 Network Connections

If the controller supports it, then an Ethernet TCP/IP connection can be made directly from the computer to the controller.

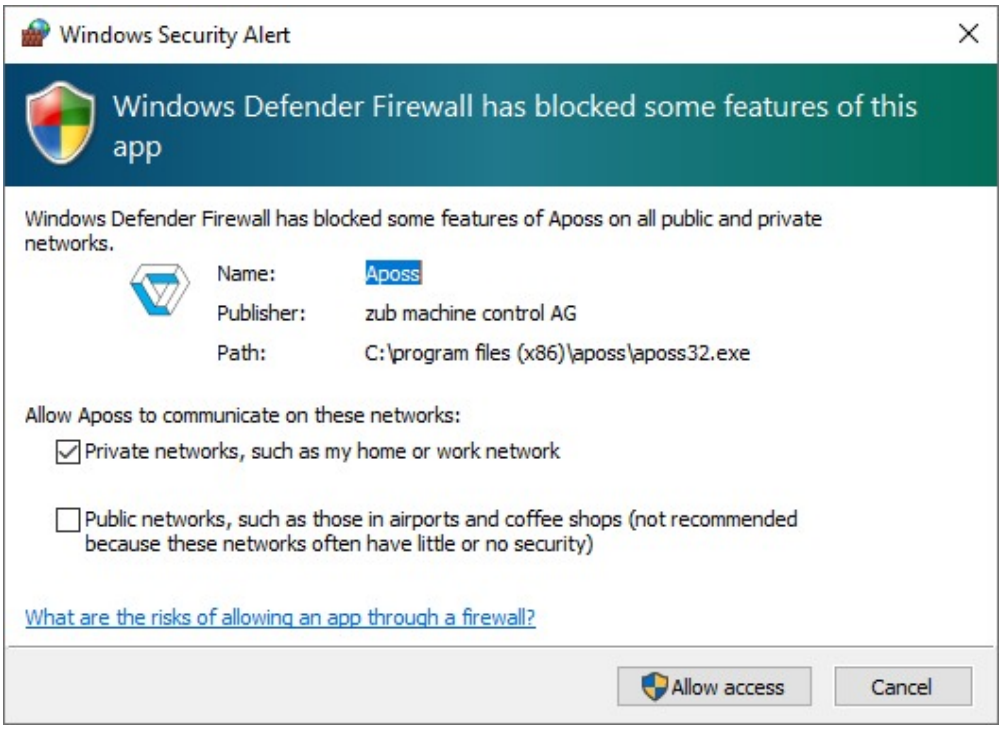
1. Click on **Settings** → **Interface** to open the Interface Parameters dialog. Select the **Network** tab.



2. Check the ☑ **Set as Default** box.
3. Select the **Search** option. **Port** must be 23.

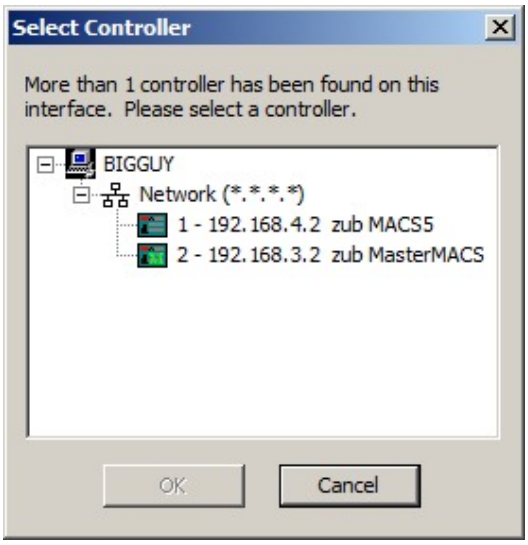
> If the controller firmware is older than 7.1.16, then the **Fixed** option must be selected and the IP address of the controller specified.

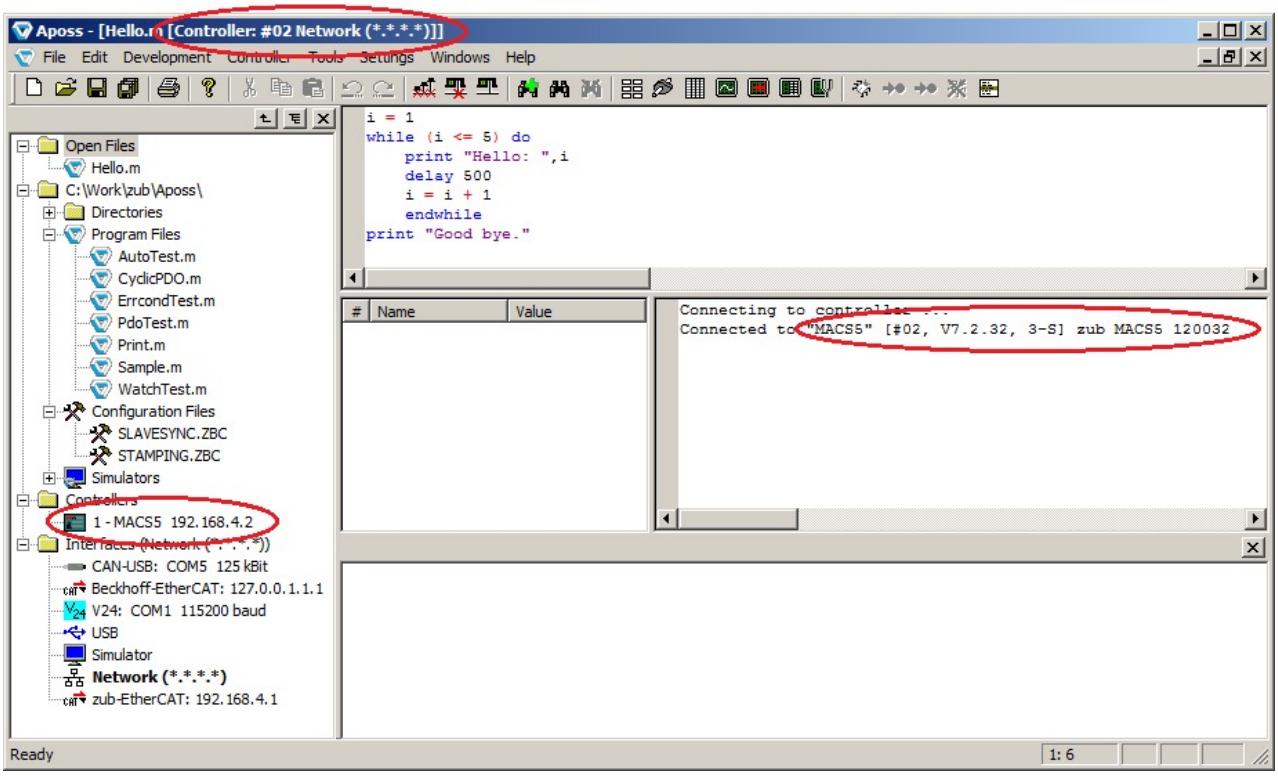
4. Click the **OK** button. This will close the dialog and return you to the Edit Window.
5. In the Edit Window, press the **Esc** key. The ApossIDE will then try to connect to the controller. This may or may not generate a security warning similar to that shown below. If it does, then allow the ApossIDE to have access to networks. Otherwise, the ApossIDE will not be able to connect to controllers over Ethernet networks.



6. If the connection is successful, then the controller will be identified in the window as shown. If more than one controller was found, then a “Select Controller” dialog (similar to that shown below) will be displayed. Select the

appropriate controller using the IP address and controller type shown in the dialog and click the **OK** button.





The above example shows a successful connection. The window title shows the controller CAN ID (#02) and that a network connection is being used. The Sidebar shows the MACS5 icon, the connection ID (1), the controller name (MACS5), and the IP address (192.168.4.2). The Communication Window shows the controller name (MACS5), CAN ID (#02), firmware version (7.2.32), hardware options (3-S), and the type of controller (zub MACS5 120032).

> The connection ID is an arbitrary number assigned by the ApossIDE at the time of connection. If a controller is re-connected at a later time, then it may be assigned a different connection ID.

If "No controller found!" is reported in the Communication Window, then the connection was unsuccessful. Please check for the following:

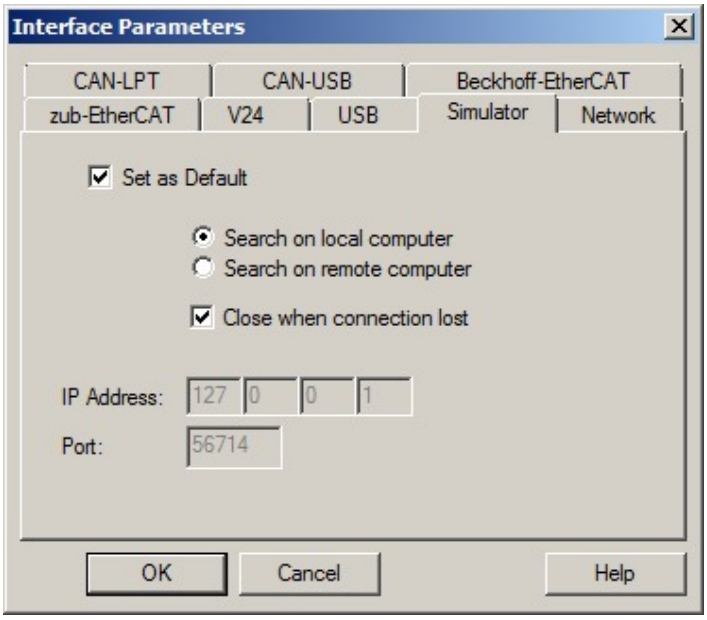
1. The controller is powered on.

2. The Ethernet cable is firmly connected to both the computer and the controller.
3. Depending on the network configuration, the Network Interface Card (NIC) settings, and the Windows firewall settings, access to the controller may be blocked. Please refer to Windows documentation for instructions on how to change the NIC and firewall settings.

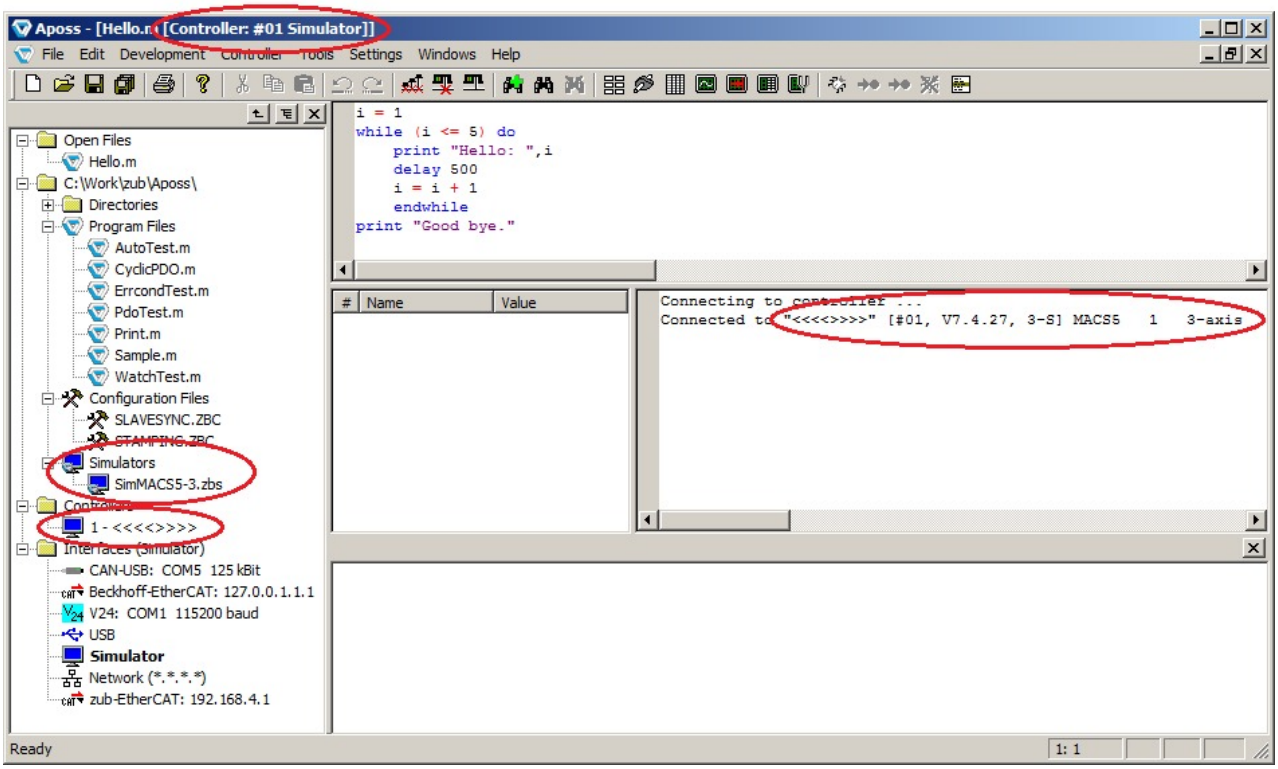
2.5.8 Simulator Connections

If a controller is not available, then the Aposs Simulator can be used to simulate a physical controller. The Simulator is able to execute most of the functions of a controller (e.g. execute application programs, react to digital inputs, etc.) but without an actual motor. See Simulator for more information.

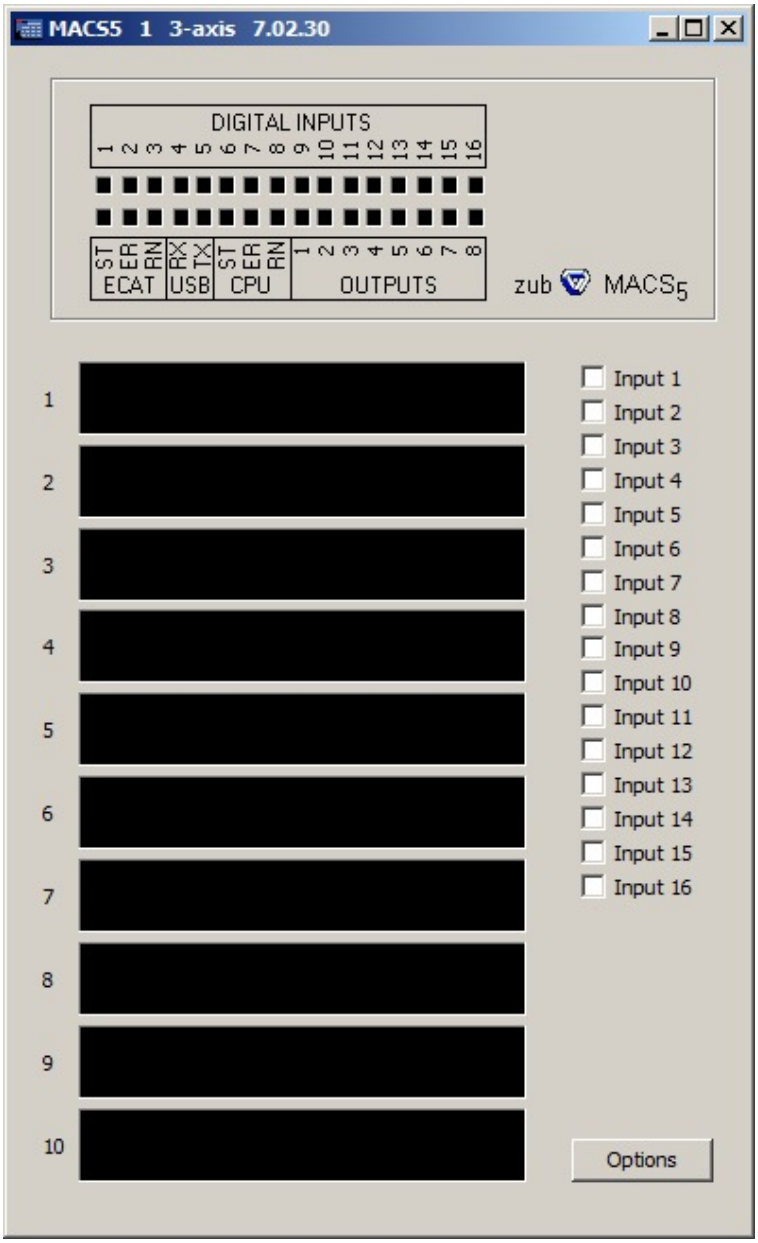
1. Click on **Settings** → **Interface** to open the Interface Parameters dialog. Select the **Simulator** tab.



2. Check the ☑ **Set as Default** box.
3. Select the **Search on local computer** option.
4. Check the ☑ **Close when connection lost** box.
5. Click the **OK** button. This will close the dialog and return you to the Edit Window.
6. In the ApossIDE Sidebar, double-click on a Simulator file. This will start the Simulator. See Simulator for details on how to create a Simulator file.
7. In the Edit Window, press the **Esc** key. The ApossIDE will then make a connection to the Simulator. If this is successful, then the controller will be identified in the window as shown:



8. A Simulator window similar to the following will also be opened. This is a stand-alone window and it can be moved to another location on the computer monitor so that it does not interfere with the ApossIDE window. See Simulator User Interface for information on how to use this window.



The above example shows a successful connection. The window title shows that the controller has a CAN ID of 1 (#01) and that a simulator is running. The Sidebar shows the Simulator icon, the Simulator ID (1), and the controller name (<<<<>>>>). The Communication Window shows the controller name (<<<<>>>>), CAN ID (#01), firmware version (7.4.27), hardware options (3-S), and the type of controller (3-axis MACS5).

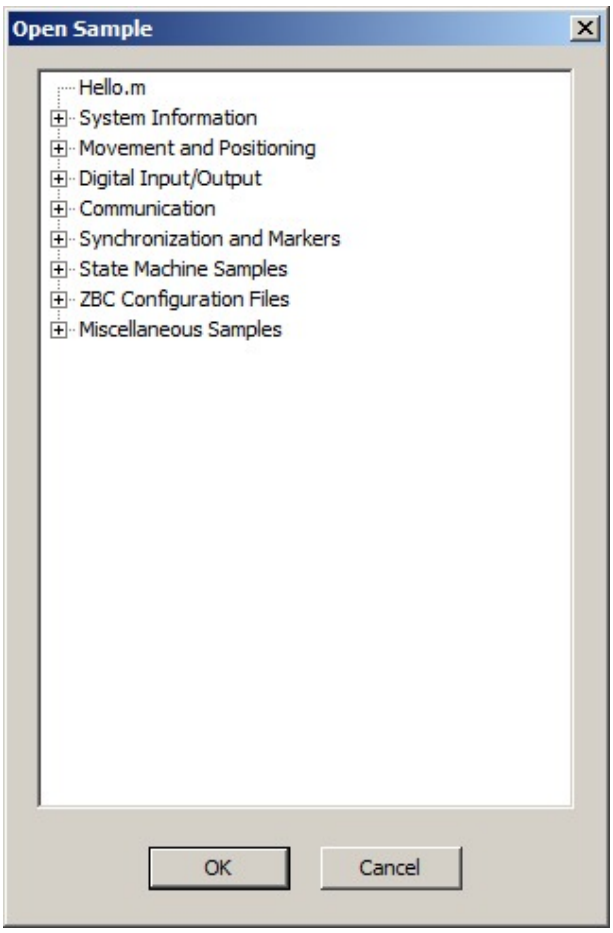
> The Simulator ID is a user-specified number assigned to the Simulator at the time the Simulator file is created. It is used to identify the Simulator and to differentiate it from other Simulators when multiple Simulators are running. The Simulator ID can be seen in the titlebar of the Simulator window. Note that the CAN ID will always be displayed with a "#" prefix in order to distinguish it from the Simulator ID.

2.6 Executing a Simple Test Program

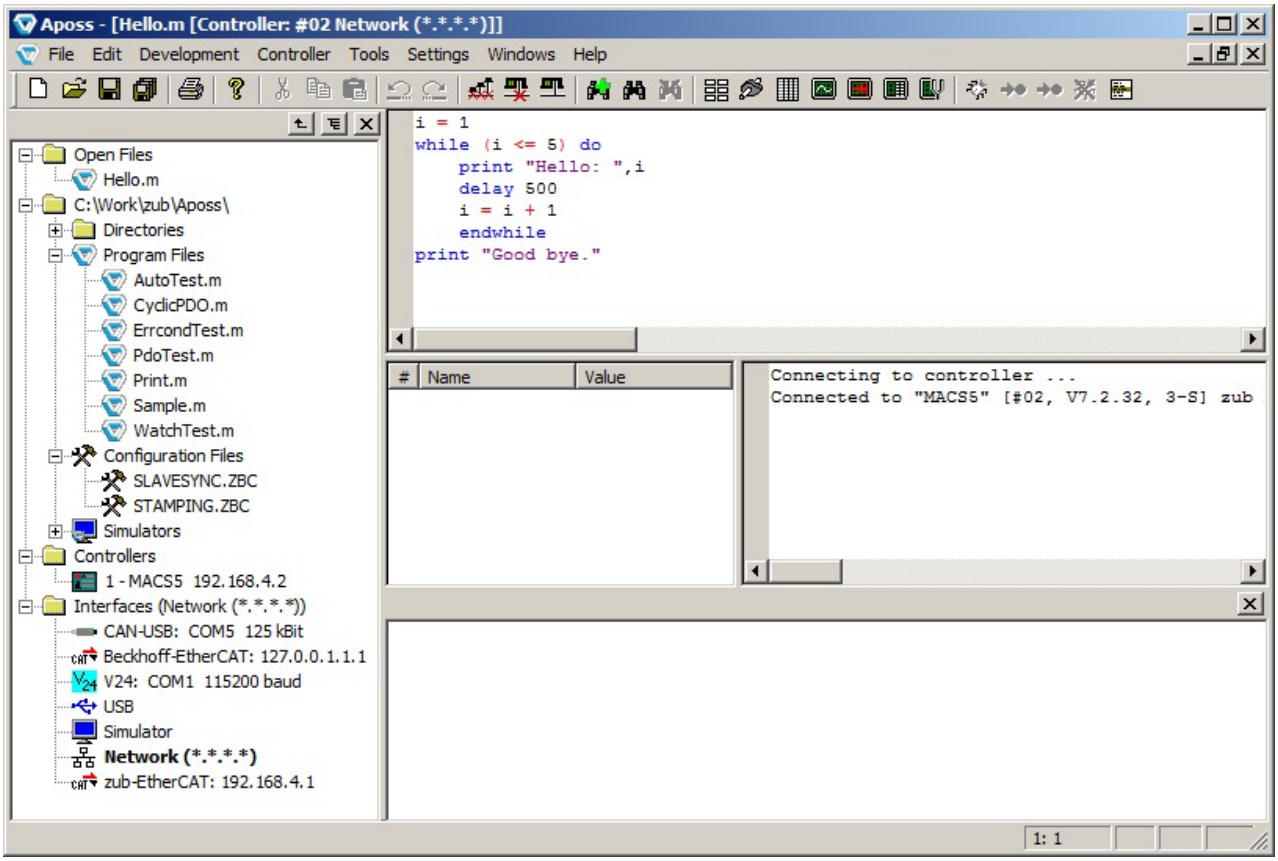
The steps in the following example, illustrate how to execute a user application program on the controller. The sample program used here is a very simple program that can be executed on the controller. It does not require any motor to be connected and any motor that is connected will be ignored. The program will simply print "Hello" five times, then print

"Good bye", and then stop.

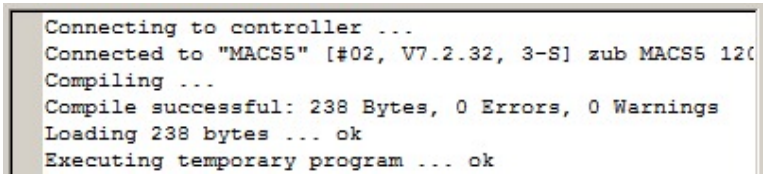
1. Start the ApossIDE.
2. Execute the **File** → **Sample** command. This will display the dialog shown below.



3. Select "Hello.mc" and press **OK**.
4. Press the **Esc** key to connect to the controller.
5. At this point, the "Hello.mc" or "Hello.m" program is ready to be executed and the ApossIDE window should appear similar to that shown below.



6. Start the program with **Development** → **Execute** or by pressing the **F5** key. This will automatically do the following: Compile the program, download the compiled program to the controller, and start the program executing on the controller. Feedback for each of these actions is displayed in the lower-right of the window as shown below.



7. The ApossIDE window title will indicate that a program is running. This is shown below. Note that the title is only updated periodically so the "running" status may not be shown if the program starts and ends quickly.

8. As the program executes on the controller, any messages printed by the program will be displayed in the lower-right of the ApossIDE window. In this example, the "[01]" at the beginning of each line is the ID number of the controller that is printing the message.

```
Connecting to controller ...
Connected to "MACS5" [#02, V7.2.32, 3-S] zub MACS5 12(
Compiling ...
Compile successful: 238 Bytes, 0 Errors, 0 Warnings
Loading 238 bytes ... ok
Executing temporary program ... ok
[01] Hello: 1
[01] Hello: 2
[01] Hello: 3
[01] Hello: 4
[01] Hello: 5
[01] Good bye.
```

9. The program can be modified by making whatever changes are needed in the upper part of the ApossIDE window. Once the changes are complete, the program can be rerun simply by using **Development** → **Execute** or by pressing the **F5** key again.

Stopping Program Execution

At any time, if it is necessary to stop a program executing on a controller, then this can be done by pressing the **Esc** key or using **Development** → **Break**.

> If the controller is operating a motor when program execution is stopped, then the controller will stop the motor using the maximum allowable deceleration.

2.7 Setting Basic Parameters

When setting up a new controller or connecting a new axis to an existing controller, the following small set of parameters must always be checked to ensure that they are set correctly for the axis and its encoder information.

These parameters are generally determined directly from the motor and encoder hardware specifications. It will normally be necessary to adjust other parameters as well, depending on the requirements of the application. However for these other parameters, the factory settings can usually be used at first and then adjusted later as the application is developed. For a detailed description of all global parameters and axis parameters, please refer to Parameter Reference.

> **These parameters are important since they are they ones that provide the controller with its basic control over the motor. The direction of the axis (internal or external amplifier) must match the direction of the ecoder feedback signal. If this is not correct, the motor might spin at high speed in a undefined direction!**
> **Incorrect settings could cause the motor to run out-of-control, resulting in a risk of injury or equipment damage.**.

Only the most important settings are described here. A list with all axis module parameter can be found here: Axis Parameters

Encoder Data

> The basic counter for the actual position (encoder value) is the virtual counter input module. By default the source of this counter is the incremental hardware counter. The source can be re-mapped to bus module (for CAN or EtherCAT encoder) or to an absolute encoder when

supported by the MACS controller.

Very helpful is to run axes in simulation mode. It allows the use of movement commands without getting a track error. Just map the command position to the virtual counter input module and set the simulation mode:

```
VIRTCOUNTIN_PARAM(axisNo, VIRTCNTIN_PISRC_COUNTER) =
AXE_PROCESS_SRCINDEX(axisNo, REG_COMPOS);
VIRTCOUNTIN_PARAM(axisNo, VIRTCNTIN_MODE) = VIRTCNTIN_MODE_SIMULATION_SLAVE;
```

To calculate the motor speed in rpm the information for impulses per revolution must be given. The following parameters must be set:

POSENCREV — Number of revolutions of the motor

AXE_PARAM(AXIS, POSENCREV) = 1;

POSENCQC — Number of quadcounts per revolutions.

AXE_PARAM(AXIS, POSENCQC) = 2000;

POSFACT_Z — Number of revolutions of the input shaft.

AXE_PARAM(AXIS, POSFACT_Z) = 1;

POSFACT_N — Number of revolutions of the output shaft.

AXE_PARAM(AXIS, POSFACT_N) = 1;

FEEDREV — Number of revolutions of the gear box output shaft.

AXE_PARAM(AXIS, FEEDREV) = 1;

FEEDDIST — Distance (in user units) in revolutions of the gear box output.

AXE_PARAM(AXIS, FEEDDIST) = 2000;

Basic Settings

The following parameters must be set to specify the motor velocity and acceleration limits:

VELMAX — Maximum velocity (in RPM) of the shaft where the encoder is mounted.

AXE_PARAM(axis, VELMAX) = 1000;

RAMPMIN — Shortest ramp (in ms). This is the minimum time required to accelerate from stopped to maximum velocity. It is used to

calculate the maximum allowable acceleration.

AXE_PARAM(axis, RAMPMIN) = 500;

Amplifier Settings

If a MACS controller with internal amplifier is used, correct settings in the amp modules must be made, as well. Listed are only the basic amplifier settings. Also settings for current and velocity controller must be made.

HWAMP_PARAM(axis, HWAMP_COMMTYPE) = 4; // type of motor: PMSM = 4
HWAMP_PARAM(axis, HWAMP_MODE) = 2; // 0 = all, 1 = P,V,PWM , 2 = P,C,PWM , 3 = P,PWM
HWAMP_PARAM(axis, HWAMP_POLES) = 4; // number of pole pairs
HWAMP_PARAM(axis, HWAMP_ENCRES) = ENC_QC; // Encoder resolution for amplifier

Please see the : SDO Dictionary > 4000 - Hardware Device Parameters > 4000 - HW Amplifiers

Example

For MACS controllers, encoder parameters must be explicitly set at the beginning of every application program. This can be done by modifying the following code and incorporating it into the application program.

```
    // AxisNo is the axis module number

        EncresQC = 1024 * 4;
    // axis parameter
    HWAMP_PARAM(AxisNo, HWAMP_COMMTYPE) = 7;         // type of motor: Brushless
motor PMSM = 7
    HWAMP_PARAM(AxisNo, HWAMP_MODE)     = 2;         // 0 = all, 1 = P,V,PWM , 2
= P,C,PWM , 3 = P,PWM
    HWAMP_PARAM(AxisNo, HWAMP_POLES)    = 4;        // number of pole pairs
    HWAMP_PARAM(AxisNo, HWAMP_ENCRES)   = EncresQC;  // Encoder resolution for
amplifier
    HWCOUNTINC_PARAM(AxisNo, HWCNTINC_RESOLUTION) = EncresQC;    // Encoder
resolution for hardware counter

    // profile generator
    AXE_PARAM(AxisNo, FEEDREV)  = 1;
    AXE_PARAM(AxisNo, FEEDDIST)  = EncresQC % 4;
    AXE_PARAM(AxisNo, POSENCQC)  = EncresQC;
    AXE_PARAM(AxisNo, POSENCREV) = 1;
    AXE_PARAM(AxisNo, VELMAX)    = 250;        // reference velocity [RPM]
    AXE_PARAM(AxisNo, VELRES)    = 1000;       // velocity reference
    AXE_PARAM(AxisNo, RAMPTYPE)  = 0;          // ramp type: trapezoidal = 0
    AXE_PARAM(AxisNo, RAMPMIN)   = 50;         // time reference [ms] to
complete acc- or deceleration
    AXE_PARAM(AxisNo, POSERR)    = 10000;      // allowed position error
```

⚠ If parameters are modified by an application program (for example, with Sysvar), then it is necessary to re-execute the Vel, Acc, and Dec commands in order to activate the new settings.

For newer MACS controllers, axis parameters must be explicitly set at the beginning of every application program. This can be done by modifying the following code and incorporating it into the application program. Each axis has its own set of axis parameters. So, the parameters must be set for *each* axis separately. See Initializing Temporary Parameters for more information on setting parameter values.

2.8 Checking the Encoder

Before trying to run the motor for the first time, verify that the encoder is connected properly and that the encoder parameters have been set properly. This is important since the controller uses the feedback from the encoder to control the motor.

> When talking about "encoder" in this chapter, the feedback signal for the drive system is meant. The feedback signal can be a encoder connected to the MACS controller directly, an encoder connected to a drive and the position information is transfered over a bus (e.g. CAN or EtherCAT) or even an analoge signal that represents a position. It is also assumed that position controller is set-up inside the MACS controller (it is possible to use the target position information only and connect a drive to the MACS controller via CAN or EtherCAT bus).

> If the encoder is not set up properly, then the controller can drive the motor in unpredictable ways causing possible damage.

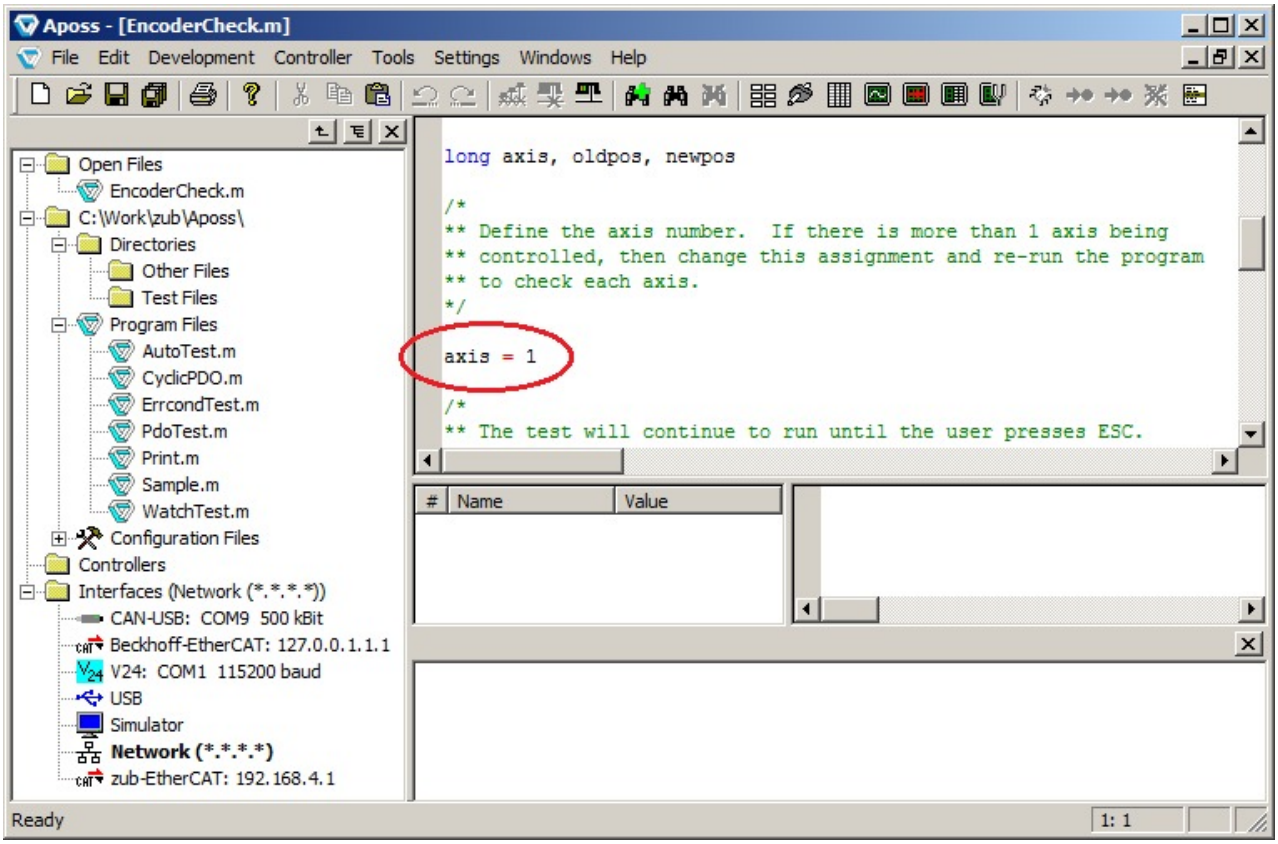
If the controller supports more than one axis, then the encoder on *each* axis must be tested.

If you have not yet done so, now is the time to connect and test the encoder with each individual axis.

> The motor power should *not* be connected.

Checking the Encoder Connections

The ApossIDE provides a sample program that can be used to simplify the process of checking the incremental encoder. Open this program by using **File** → **Sample**, selecting "Controller Verification", and then selecting the "EncoderCheck.mc" sample program.



As written, this program will check the incremental encoder for axis 1. If multiple axes are connected to the controller, then each axis should be checked individually. This can be done simply by changing the "axis" assignment (marked in the above illustration) and re-executing the program.

Execute the encoder check as follows:

1. Ensure that the motor power is *not* connected. You will be turning the motor by hand.
2. Press the **Esc** key to connect to the controller. If the controller does not connect, then see Connecting to the Controller.
3. Use **Development** → **Execute** (or press **F5**) to start the test program running.
4. An encoder position of 0 should be displayed.
5. Turn the motor by hand in the "positive" direction. "Positive" is a clockwise rotation when viewing the front of the motor (i.e. looking towards the motor along the axis). Watch the encoder position values being displayed in the Communication Window.
    1. If the values increase, then the encoder is functioning correctly.
    2. If the values decrease, then the encoder wiring is reversed. Encoder channels A and B, as well as /A and /B, must be exchanged. However, it is often easier to swap two motor-phases. See Wiring the Controller. If the wiring is correct (i.e. to specification), then this can also be corrected within the controller by changing the settings of the POSDRCT parameter. For CAM curves and kinematics applications it is recommended to change direction in parameters VIRTCOUNTIN_PARAM(axe, VIRTCNTIN_UUFACT_UNITNO) and VIRTAMP_PARAM(axe, VIRTAMP_INVERT) with value -1.
    3. If the value does not change, then this is probably the result of incorrect cable installation. Check whether the connection was made according to the hardware specifications. Measure the signals coming from the encoder and compare them to the values listed in the specifications. See Wiring the Controller.
6. If the motor is turned one full rotation, then the incremtal encoder position value should be 4 times the value of the resolution of the encoder. For example, if the encoder resolution is 500 counts per rotation, then you should see 2000 as the position value. If not, then the encoder parameters are

probably set incorrectly. See Setting Basic Parameters.
7. Turn the motor by hand in the "negative" direction. The encoder position values should decrease.
8. Press the **Esc** key to end the test and exit the program.

> ⚠ A successful test of the encoder is required before continuing with any further operations.

Checking Master Encoder for Synchronization

To check the master encoder, replace the *two* Apos commands by Mapos in the “EncoderCheck” test program and start the program again. Then run the master forward. The master position should also increase.

If the position decreases, then you must swap the master encoder's A and B channel.

2.9 Checking the Motor

Each time a new motor is connected to the controller, the controller must be "tuned" for that motor. Tuning consists of setting the various controller parameters that allow the motor to run smoothly when it is being driven by the controller. The information below is intended only to assist in doing a very basic initial tuning. When an actual application is running and the motor is under load, then it may be necessary to re-adjust the parameters. This optimization of the parameters may be done at a later. For more information on the tuning parameters, see Optimizing the PID Controller.

> When talking about encoder in this chapter, the feedback signal for the drive system is meant. The feedback signal can be a encoder connected to the MACS controller directly, an encoder connected to a drive and the position information is transfered over a bus (e.g. CAN or EtherCAT) or even an analoge signal that represents the position. It is also assumed that position controller is set-up inside the MACS controller (it is possible to use the target position information only and connect a drive to the MACS controller via CAN or EtherCAT bus).

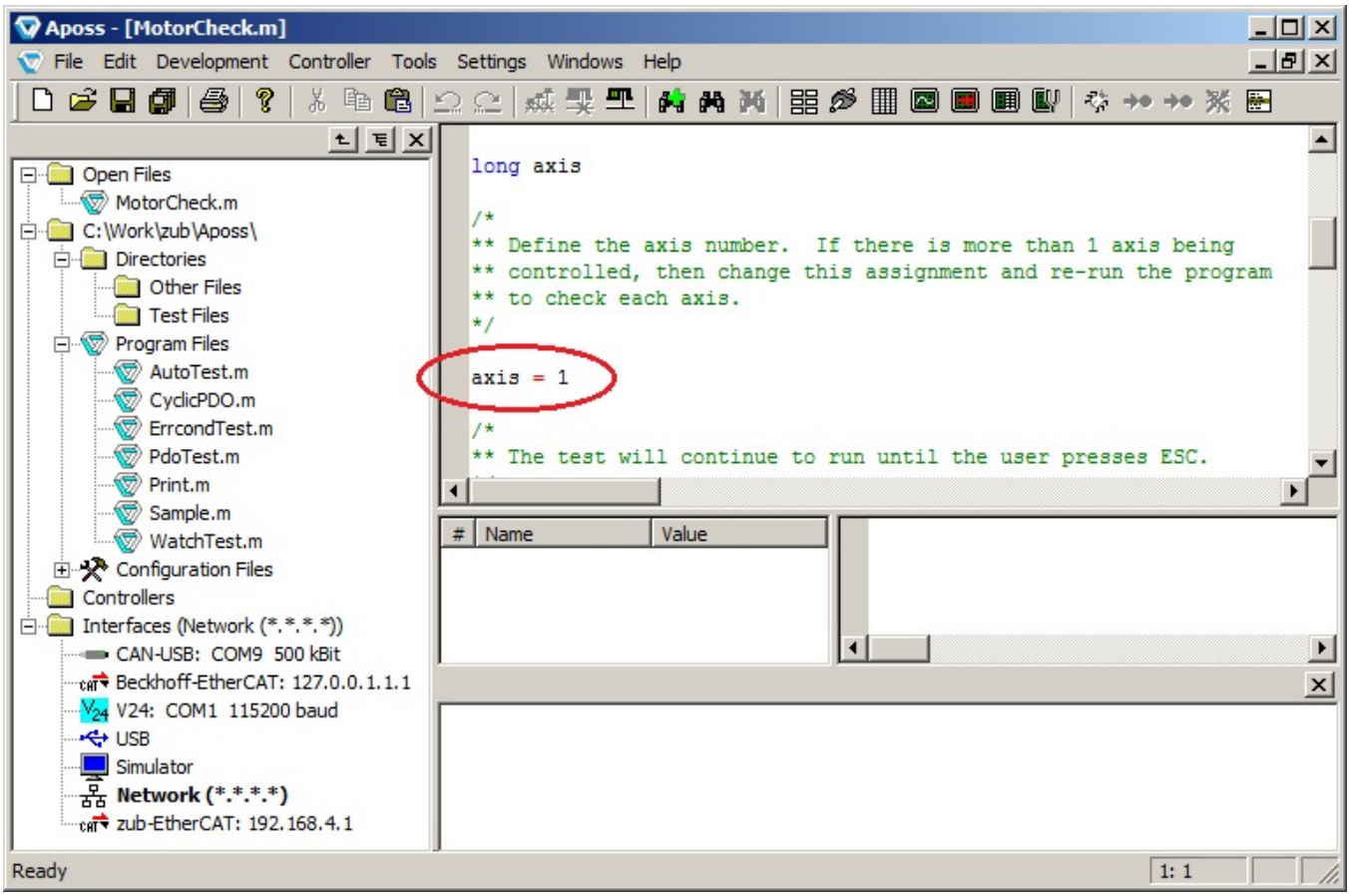
> A correctly functioning encoder is *mandatory* before a motor can be tuned. If the encoder has not yet been checked, then please see Checking the Encoder before continuing here.

If the controller supports more than one axis, then the motor on *each* axis must be tuned.

Make sure that the motor can turn completely freely. Then connect the motor to the power supply.

> The motor must be provided with an **EMERGENCY STOP** button. The motor may turn at maximum velocity in the wrong direction if the encoder has been connected incorrectly.

The ApossIDE provides a sample program that can be used to simplify the process of checking the basic tuning of the motor. Open this program by using **File** → **Sample**, selecting "Controller Verification", and then selecting the "MotorCheck.mc" sample program.



As written, this program will assist with the tuning of axis 1. If multiple axes are connected to the controller, then each axis should be tuned individually. This can be done simply by changing the "axis" assignment (marked in the above illustration) and re-executing the program.

Execute the program as follows:

1. Press the **Esc** key to connect to the controller. If the controller does not connect, then see Connecting to the Controller.
2. Use **Development** → **Execute** (or press **F5**) to start the test program running.
3. The motor should be driven smoothly in the forward direction (i.e. clockwise when looking at the face of the motor) a short distance. The motor will then stop and reverse back to its starting position. Encoder positions will be printed at regular intervals. The movement will then be repeated continuously.

   > If the motor vibrates heavily or sets off uncontrollably, then turn off the motor *immediately* with the **EMERGENCY STOP** button to prevent damage. Then press **Esc** to end the test and exit the program. Try to resolve the problem before re-executing the test.

4. If the motor does not drive smoothly or position errors are reported, then see below.
5. Press the **Esc** key to end the test and exit the program.
6. If parameters need to be modified, then do so and re-execute the program. Continue this procedure until the motor drives smoothly.

Resolving Problems

See Optimizing the PID Controller for information about the tuning parameters.

If the motor sets off uncontrollably or vibrates heavily...

If the motor sets off uncontrollably but the encoder test was previously successful, then decrease the ***Proportional Factor*** KPROP.

If the motor does not move...

If the motor do not move at all, then the proportional factor of the PID filter is probably too low or the amplifier has not been enabled. Increase the ***Proportional Factor*** KPROP for the PID control.

If the motor vibrates heavily...

If the motor vibrates heavily, then either reduce the ***Proportional Factor*** KPROP or increase the ***Derivative Factor*** KDER.

If a position error is reported...

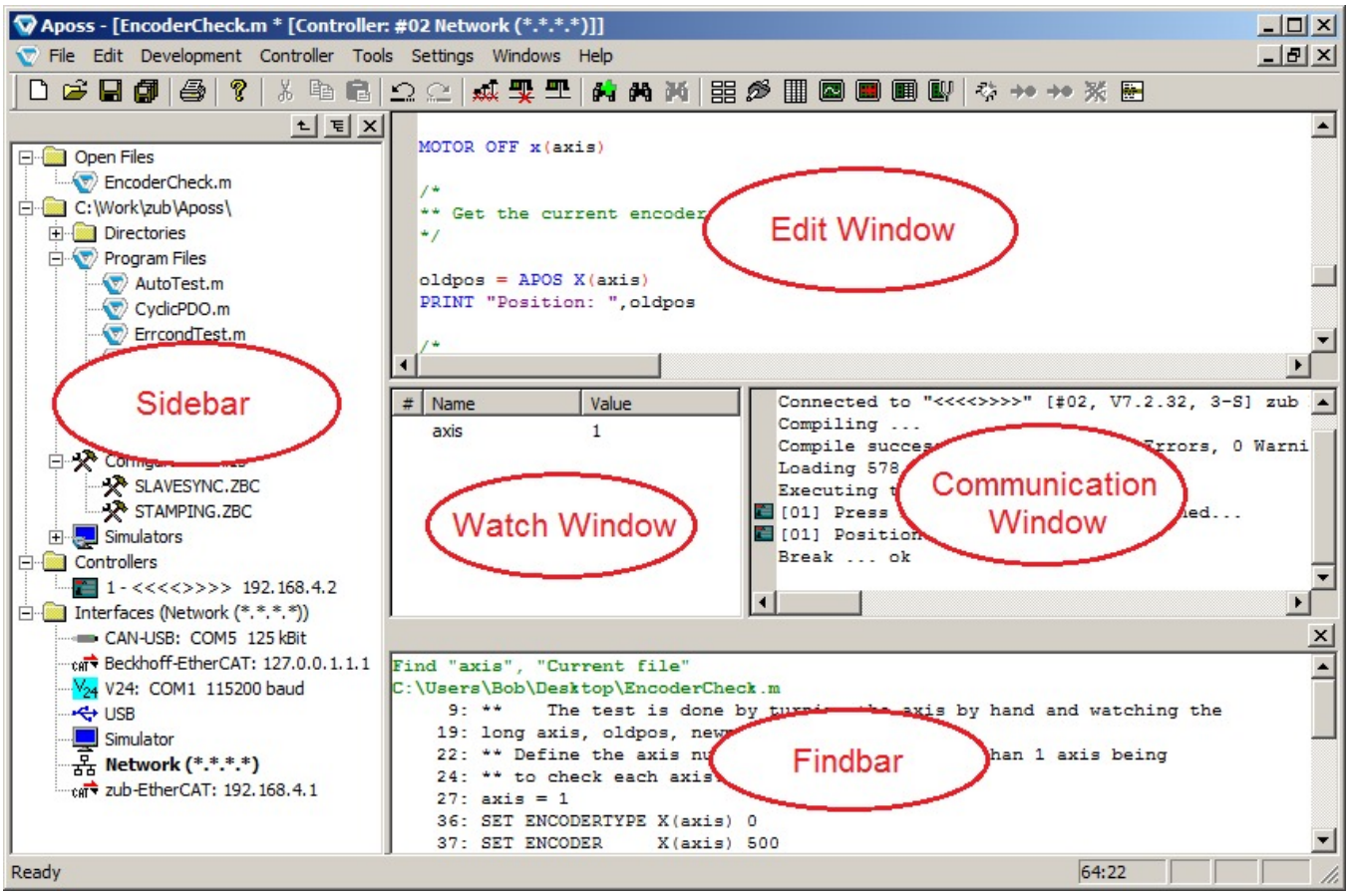
If the drive stops with a "position error" (Error 8), then repeat the Checking the Encoder step to ensure that the encoder is connected correctly and measuring in the correct direction. If the encoder is functioning properly, then increase the ***Tolerated Position Error*** POSERR.

3 ApossIDE User Interface

You should be familiar with the basic functions and terminology of the Microsoft Windows interface. This chapter only explains the specifics of the ApossIDE User Interface.

3.1 ApossIDE Window

The ApossIDE Window allows simultaneous access both to an application program and to a controller. Multiple Edit Windows can be opened, each accessing a different application program and a different controller.



From top to bottom:

Title Bar

The **Title Bar** displays the name of the application program. An "*" character after the name indicates that the progarm has been modified and has not yet been saved. If a controller is connected, then the controller ID number and the connection interface type will also be displayed. In the event of a controller error, the error is also shown in the title bar.

Menu Bar

The **Menu Bar** offers the complete selection of available functions.

Toolbar

The **Toolbar** offers "single-click" access to several commonly used functions. These functions are

described in Toolbar.

Sidebar

The **Sidebar** provides the user with quick access when working with multiple programs, multiple controllers, and multiple connection interfaces. This window is described in more detail in Sidebar.

Edit Window

The **Edit Window** displays the current application program being edited and allows it to be changed. Many functions are provided through the mouse and through keyboard shortcuts in order to assist the user in the editing process. These are described below.

Placing the mouse on a command or function keyword will cause a small popup window to be displayed showing the syntax of that command or function. An option in the **Settings** → **Editor** dialog can be used to disable these popup windows. Pressing **F1** when the mouse is on a command or function keyword will cause the online help page for that command or function to be displayed.

Different colors are used to distinguish between comments, program sections, operators, numbers, etc. You can alter the colors using **Settings** → **Editor**.

Watch Window

The **Watch Window** is useful during debugging. It allows the user to "watch" various aspects of both the controller and the executing program while the controller is operating. See **Development** → **Watch Add** for a description of how to use the Watch Window.

Communication Window

The **Communication Window** displays both messages from the ApossIDE (e.g. compiler messages) and messages from the controller (e.g. programmed print commands). This window is described in more detail in Communication Window.

Findbar

The **Findbar** displays the results of string searches when searching in multiple files. This window is described in more detail in Findbar.

Status Bar

The **Status Bar** at the very bottom of the window shows the status on the left (e.g. "Ready") and the line numbers as well as the cursor position on the right.

Within your program you can use the **line numbers** for orientation purposes. For example, the "Syntax Check" command not only places the cursor at the first line corresponding to an error, but also displays the line numbers containing errors in the Communication Window.

The current line number can be found in the status bar at the bottom right of the ApossIDE Window as is shown below. For example "13:1" means that the cursor is located on line 13 at position 1 (i.e. before the first character).

3.2 Toolbar

The ApossIDE toolbar contains the following toolbar buttons:

- **New** - Open a new window.
- **Open** - Open an existing file in a new window.
- **Save** - Save the current window in a file.
- **Save All Source** - Save all program windows in files.
- **Print** - Print the current window.
- **Help** - Display the Help window.
- **Cut** - Cut the selected text to the Windows clipboard.
- **Copy** - Copy the selected text to the Windows clipboard.
- **Paste** - Paste the content of the Windows clipboard to the current cursor position.
- **Undo** - Undo the last change.
- **Redo** - Redo the last change that was undone using **Undo**.
- **Close all interfaces** - Disconnect all controllers from all windows.
- **Close interface** - Disconnect the controller from the current window.
- **Connect** - Select a controller and connect it to the current window.
- **Watch add** - Add a new variable to the Watch Window. For more information on the Watch Window, please see **Development** → **Watch Add**.
- **Watch start** - Start watching the variables in the Watch Window.
- **Watch stop** - Stop watching the variables in the Watch Window.
- **Module Editor** - Start the Module Editor. For more information on using the Module Editor, please see Module Editor.
- **CAM Editor** - Open a new CAM Editor window. For more information on the CAM Editor, please see CAM Editor.
- **Array Editor** - Open a new Array Editor window. For more information on the Array Editor, please see Array Editor.
- **Oscilloscope (Free Run)** - Open a new Free Run Oscilloscope window. For more information on the Oscilloscope, please see Oscilloscope.
- **Oscilloscope (Single Shot)** - Open a new Single Shot Oscilloscope window.
- **Oscilloscope (Record)** - Open a new Record Oscilloscope window.
- **Debug Start** - Start the debugger. For more information on debugging application programs, please see Debugging Programs.
- **Debug Goto** - Execute the program up until the next breakpoint.

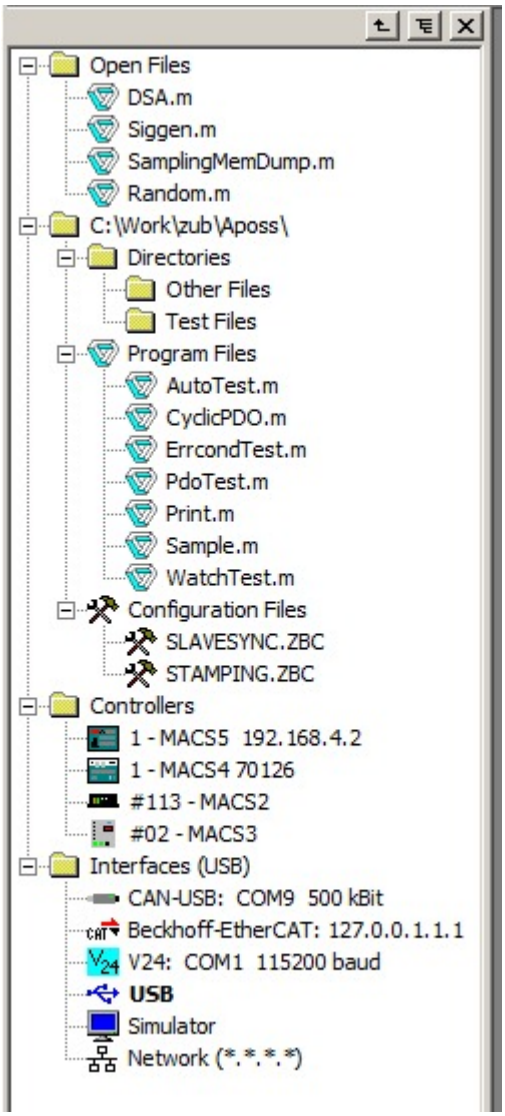
- **Debug Step** - Execute a single line of the application program.
- **Debug Stop** - Stop the debugger.
- **Debug Highlight** - Highlight the currently executing program line.

The toolbar will also contain the following "Find in Files" toolbar buttons (see Find in Files):

- **Find options** - Change the search options.
- **Find in current file** - Search in the active window only.
- **Find in current program** - Search the application program in the active window and all the include files (if any) that it uses.
- **Find in open files** - Search all open program and include file windows.
- **Find in sidebar files** - Search all program and include files listed in the Sidebar.
- **Find in folder files** - Search all files in a directory. The directory to be searched is specified using the **Find options** toolbar button.

3.3 Sidebar

The Sidebar is shown below. It is intended to be used to provide quick access when multiple programs, multiple controllers, and multiple interfaces are being used.



If the window is not wanted, then it can be closed by clicking the close button ☒ at the top right of the window. It can be re-opened at any time by using the **Windows** → **Show Sidebar** menu command. The width of the window can be adjusted by clicking and dragging the right side of the window in a manner similar to how edit window splitter bars are used.

The window will display all currently open files, all files and directories in the current directory, as well as all currently connected controllers and all available connection interfaces. The currently selected default interface will be shown in parentheses on the Interface title line. If the interface list is expanded, then the default interface will be displayed with a bold font.

Double-clicking on items in the sidebar will have the following affect:

Open Files: The Windows focus is set to this file.

Directory: The directory shown in the sidebar is changed to this directory.

File: If this file is already open, then the Windows focus is set to this file. If the file is not open, then it will be opened. Note that double-clicking on a configuration file will open the CAM Editor, double-clicking on an oscilloscope file will open the Oscilloscope, etc.

Controller: If this controller is currently associated with an Edit window, then the Windows focus will be set to that window. This allows the user to easily “find” the associated window when multiple windows are

open. If the controller is not associated with an Edit window, then the Diagnostics dialog for the controller is displayed.

Interface: If the window which currently has the focus is an Edit window, then this will automatically open a controller connection using the selected interface.

If the Edit window is already connected to a controller, then the existing controller connection will be closed and a new controller connection will be opened. Note that the selected interface does not need to be the default interface; any interface can be selected. This is a quick way to connect different controllers to different Edit windows. A warning message is displayed if the current window is not an Edit window.

Right-clicking on an interface will display a popup menu allowing that interface to be selected as the default interface or the interface parameters for that interface to be set.

The sidebar contains the following buttons in the top-right corner:

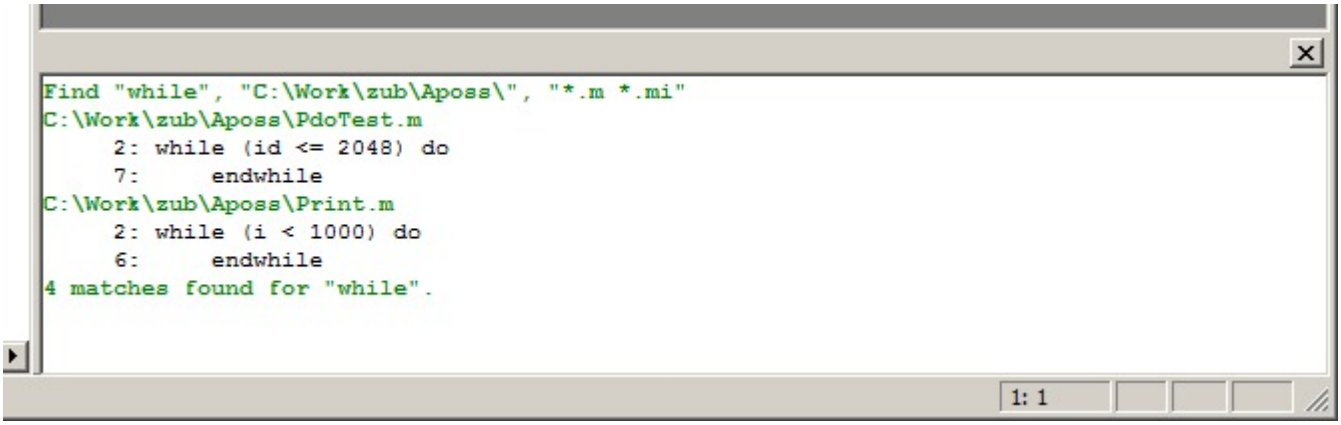
Up: The directory shown in the sidebar is changed to the parent of the current directory.

Browse: A dialog is displayed allowing the user to select the directory that will be displayed in the sidebar.

Close: The sidebar window is closed. It can be re-opened again by using the **Windows** → **Show Sidebar** menu command.

3.4 Findbar

The **Findbar** is shown below and is used to display the results of the **Edit** → **Find in Files** menu command. It is always placed at the bottom of the main ApossIDE window.



If the window is not wanted, then it can be closed by clicking the close button at the top right of the window. It will be automatically re-opened the next time that **Edit** → **Find in Files** is used. It can also be re-opened at any time by using the **Windows** → **Show Findbar** menu command. The height of the window can be adjusted by clicking and dragging the top of the window in a manner similar to how edit window splitter bars are used.

Double-clicking on any of the lines in the **Findbar** will automatically open the corresponding file (if it is not already open) and place the text cursor on that line.

3.5 Communication Window

The ApossIDE uses the **Communication Window** to display messages to the user. It is a child window of the main ApossIDE Window. Note the following points about this window:

1. New messages are always added at the bottom of the window. Messages will scroll up as new messages are

added.

2. A scrollbar will be displayed on the right side of the window if necessary. Moving this scrollbar *up* will stop the automatic scrolling as new messages are added. This allows the user to pause and view particular messages while, for example, the controller continues to print messages. Moving the scrollbar back to the *bottom* of the window will re-start automatic scrolling. Compiling a program will also re-start scrolling so that the user can see the results of the compile.
3. The window supports a maximum of 5000 messages. If more messages are displayed, then the oldest messages are deleted to make room for new messages.

> All messages displayed in the window may be logged to a file by using the **Development** → **Messages -> Log file** command. This file is not limited to 5000 messages.

> The contents of the window may be edited. This allows users to add comments if desired and to copy to/from the Windows clipboard.

The following classes of messages are displayed:

General Messages

These are messages of general interest that are generated by the ApossIDE itself. Typically, they are used to provide positive feedback to the user when interacting with the controller. These messages are displayed without any icon in the left margin of the window.

Example:

```
Connecting to controller ...
Connected to "MACS5" [#02, V7.2.32, 3-S] zub MACS5 120032
Load Program List #02 ... ok
Reading array #0 from controller #02 ... ok
Reading array #1 from controller #02 ... ok
Writing array #0 to controller #02 ... ok
Writing array #1 to controller #02 ... ok
Permanently saving arrays in controller #02 ... ok
Break ... ok
Closed connection to #02.
```

Compiler Messages

When a program is compiled (for example, by **Development** → **Execute** or **Development** → **Syntax Check**), then the results of the compile are displayed in the window. This includes a list of the #include files, if any, that are included by the program. If more than one file is included, then the list will initially be displayed in a "collapsed" format. In this format, only the first file (bordered by horizontal lines) is displayed and the + icon is displayed in the left margin of the window. Clicking on the icon will "expand" the list so that all files can be seen. Clicking on the − icon will "collapse" the list again.

> If the include file list is not wanted at all, then the **Suppress #include Messages** compiler option can be set to entirely prevent the list from being displayed. See Compiler.

Example:

```
Checking syntax ...
#include C:\Work\zub\PCSoftware\Debug\Include\sysdef.mi
Compile successful: 1128 Bytes, 0 Errors, 0 Warnings
```

```
Checking syntax ...
#include C:\Work\zub\PCSoftware\Debug\Include\sysdef.mi
#include C:\Work\zub\PCSoftware\Debug\Include\SdoDictionary.mi
#include C:\Work\zub\Test\UtilFunc.mi
#include C:\Work\zub\Test\StrFunc.mi
Compile successful: 1128 Bytes, 0 Errors, 0 Warnings
```

If there are compiler error or warning messages, then they will be highlighted with and icons in the left margin.

> Double-clicking on an error or warning message will cause the edit cursor in the Edit Window to be moved to the line containing the error. If the error is in an include file, then that file will be opened.

Example:

```
Checking syntax ...
#include C:\Work\zub\PCSoftware\Debug\Include\sysdef.mi
Errors (3):
1 Simple.m<26,4> Error #017:  Invalid command or '=' missing.  Found: yy
2 Simple.m<28,1> Error #004:  Invalid command.  Found: else
3 Simple.m<29,1> Error #004:  Invalid command.  Found: endif
Warnings (2):
4 Simple.m<31,5> Warning #158:  Parameter ENCODERTYPE is deprecated starti
5 Simple.m<27,1> Warning #126:  Variable "yy" has been assigned a value bu
Compile failed: 3 Errors, 2 Warnings
```

Controller Messages

Messages from the controller are also displayed in this window. These messages generally come from print or printf commands in the running program. They will be highlighted with the icon in the left margin. These messages will also be prefixed by the controller ID number (e.g. "[01]" in the example below). If the controller enters an error state, then an error message is displayed with the icon as in the example below.

> Double-clicking on an error message will cause the Error History window to be opened. This allows the user to see details about the error.

Example:

```
Connecting to controller ...
Connected to "MACS5" [#02, V7.2.32, 3-S] zub MACS5 120032
Compiling ...
Compile successful: 1790 Bytes, 0 Errors, 0 Warnings
Loading 1790 bytes ... ok
Executing temporary program ... ok
[01]
[01] Test 1: 80% velocity, 2% acceleration, 1% deceleration
[01] Execution time: 26736 ms
[01]
[01] Test 2: 80% velocity, 10% acceleration, 10% deceleration
[01] Execution time: 9808 ms
[01]
[01] Test 3: 80% velocity, 100% acceleration, 100% deceleration
[01] Execution time: 3800 ms
[#02] Controller Status: Error #71: Invalid array index
Break ... ok
```

3.6 Keyboard and Mouse Functions

3.6.1 Function Keys

Frequently used functions are assigned to function keys:

F1	Access on-line help. In the editor window, hover the mouse over a keyword and press F1 to obtain help for that keyword.
F2	Skip to the next bookmark.
F3	Find next (if “Find” has been used).
F4	Check the syntax of the program.
F5	Execute the program (see also **Development** → **Execute**).
F9	Single-step through the program (debug mode only).
F11	Access on-line help for system process information.

The function keys will be described later in this chapter where appropriate.

3.6.2 Esc key

In standard Windows applications, the **Esc** key normally closes the currently active window. However, in the ApossIDE Window, the **Esc** key will take the following actions:

1. If no controller is currently connected to this window, then a connection to the default controller will be opened.
2. If a controller has already been connected but no program is currently executing on the controller, then any existing controller error state will be cleared.
3. If a controller has already been connected and a program is currently executing on the controller, then the program will be aborted. The effect of aborting the program depends on the firmware version of the

controller:

Axis Control behavior:

- Action according to ESCCOND for each axis:

  0 - AxisStop()
  1 - AxisStop(), then AxisControl(..., OFF) (**default**)
  2 - Immediate AxisControl(..., OFF)

- O_BRAKE is set if configured.
- Action according to ESCCONDGLB.

Stopping an operating drive in this way may cause damage to the system to which the drive is connected.

To help avoid the possibility of this happening as a result of the user accidentally pressing the **Esc** key, an option can be set in the **Settings** → **Options** dialog to disable the abort function. If set, then the user is still able to abort an executing program by pressing **Shift-Esc** (which is less likely to be pressed by accident).

Reconnect a Lost Connection

When an active connection to a controller is lost (for example, if the controller is powered off or the communication line is disconnected), then the ApossIDE Window connected to that controller will display a "Lost connection" message in its title bar. However, the window will remain "associated" with that controller. Pressing **Esc** (or executing any command that needs to talk to the controller) will cause the ApossIDE Window to attempt to reconnect to the *same* controller. This behavior becomes relevant only when there are multiple controllers available on the communication line (e.g. CAN bus).

In certain cases, it is beneficial to have the ApossIDE attempt to automatically reconnect to a lost controller. For example, this can be useful when an unattended executing program is generating debugging information. An option can be set in the **Settings** → **Options** dialog to enable automatic reconnection.

3.6.3 Mouse functions

The Edit Window supports the following functions using the mouse buttons.

L-click	Change the cursor position and clear any current text selection.
R-click	Display the popup context menu.
L-double click	Select the word under cursor.
L-down and drag	Select text.
Alt-L-down and drag	Select a column of text.
L-down on selection and drag	Move the selected text.
Ctrl-L-down on selection and drag	Copy the selected text.
L-click in left margin	Select the entire line.
L-down in left margin and drag	Select multiple lines.

Spin wheel	Scroll window vertically.

3.6.4 Popup Menus

Popup menus are provided at certain program locations if the right mouse button is clicked. For example, a right click in the Edit Window will display the Edit popup menu. When popup menus are available, they will be described in the following sections.

The popup menus are closed automatically when the selected function is executed or if any other location on the screen is clicked with the left mouse button.

3.6.5 Keyboard functions

The Edit Window supports the following functions using keyboard shortcuts. Note that many of these functions are *only* available using keyboard shortcuts.

> Some keyboard functions are vendor dependent and may not perform the desired function. If this is the case, please ask your system administrator.

Go to

Home	Go to start of line.
End	Go to end of line.
Ctrl-Home	Go to start of program.
Ctrl-End	Go to end of program.
Ctrl-←	Go to start of word.
Ctrl-→	Go to end of word.
Ctrl-Alt-→	Go to start of next blank line.
Ctrl-Alt-←	Go to end of previous blank line.
Ctrl-G	Go to line (opens dialog).
Ctrl-B	Go to matching brace (“{([” or “})]”).
F2	Go to next bookmark.
Shift-F2	Go to previous bookmark.
Ctrl-F2	Toggle bookmark on current line.

Text Selection

Shift-←	Extend selection left.
Shift-→	Extend selection right.
Shift-↑	Extend selection up.
Shift-↓	Extend selection down.
Ctrl-Shift-←	Extend selection to start of word.

Ctrl-Shift-→	Extend selection to end of word.
Shift-Home	Extend selection to start of line.
Shift-End	Extend selection to end of line.
Shift-PageUp	Extend selection up one page.
Shift-PageDown	Extend selection down one page.
Ctrl-Shift-Home	Extend selection to start of program.
Ctrl-Shift-End	Extend selection to end of program.
Ctrl-Alt-K	Select all lines from previous blank line to next blank line.

Cut / Copy / Paste

Ctrl-C	Copy selection to clipboard.
Ctrl-Insert	Copy selection to clipboard.
Shift-Delete	Cut selection to clipboard.
Ctrl-X	Cut selection to clipboard.
Ctrl-Y	Cut line to clipboard.
Ctrl-V	Paste from clipboard.
Shift-Insert	Paste from clipboard.

Find / Replace

Alt-F3	Find.
Ctrl-F	Find.
F3	Find again.
Ctrl-R	Find and replace.

Modify

Insert	Toggle typing between “insert” and “overwrite”.
Ctrl-Z	Undo last change.
Alt-Backspace	Undo last change.
Tab (with selection)	Indent selected lines.
Shift-Tab	Un-indent selected lines.
Ctrl-Backspace	Delete to start of word.
Ctrl-Delete	Delete to end of word.
Ctrl-U	Make selection lowercase.
Ctrl-Shift-U	Make selection uppercase.

Ctrl-D	Delete entire line.

Scroll Window

Ctrl-↑	Scroll window up.
Ctrl-↓	Scroll window down.

3.6.6 Undo Function

You can use **Alt-Backspace**, **Ctrl-Z**, or **Edit** → **Undo** to undo the last action in the **Edit Window**.

> Both **File** → **Save** and **File** → **Save as** clear the Undo memory.

3.6.7 Tabs

Use tabs and indentation to visually structure your program. The tab size can be set in **Settings** → **Editor**.

3.7 File Menu

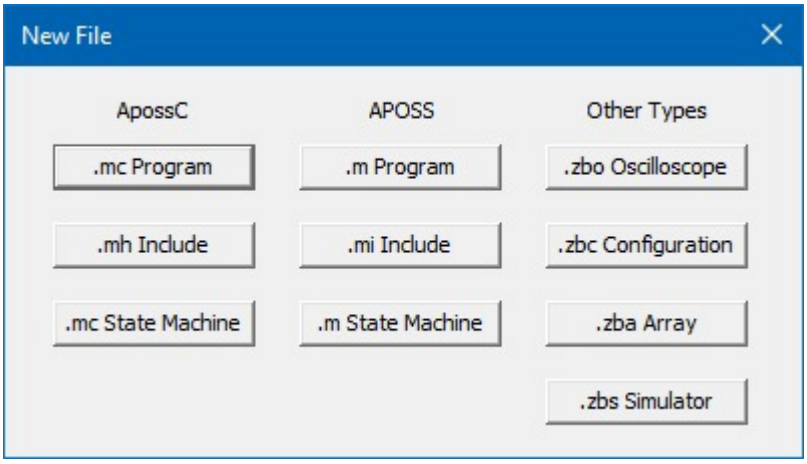
The **File** menu contains all the commands necessary to create, open, save, and print programs.

The following **File** commands are used in the usual way: **Open**, **Close**, **Save**, **Save as**, **Save all**, **Print**, **Print setup**.

> **Save all** will save only program (.m, .mc) and include (.mi, .mh) files. Other types of files (e.g. Configuration (.zbc) files, Oscilloscope (.zbo) files, etc.) must be saved individually.

3.7.1 New

To create any type of new ApossIDE file, click on **File** → **New** in the menu bar or click on the toolbar icon. This will display the following dialog:



For example, to create a new ApossC program, click **.mc Program**. An edit window will be opened with the name "Untitled" and you can begin to write your program.

3.7.2 Sample

The ApossIDE contains several sample programs. Any of these sample programs may be freely used, modified, or incorporated into other programs by the user. **File** → **Sample** can be used to edit any of these sample programs. Please see Program Samples for a list of the available sample programs.

3.7.3 Exit Program

The ApossIDE application can be ended by clicking on **File** → **Exit Program** or on the ☒ icon in the far upper right corner of the window. If you have not yet saved a new file, or saved changes to an old file, then you will have the chance to do this.

> **Exit Program** does not end a program running in the controller. You can only abort or end a program with **Esc** or with **Development** → **Break**. In order to do this, the file which is linked with the controller must be open or re-opened.

> Disconnecting from very early versions of controllers (for example, as a result of **File** → **Exit Program**), may cause the controller to stop executing if the controller is using print commands to send messages to the PC. This happens once the controller's internal print buffer becomes full. This is not a problem for all newer controllers. Newer controllers will continue to execute even after the print buffer becomes full; print messages will simple be discarded once the buffer is full.

3.8 Edit Menu

The **Edit** menu offers editing help for programming. Most of these commands can also be reached via certain keys and key combinations.

The following **Edit** commands are used in the usual way: **Undo**, **Redo**, **Cut**, **Copy**, **Paste**.

Other available functions are:

Paste Array Assignment	Paste array assignment statements into an application program.
Find and Replace	Find and replace strings within the file being edited.
Find in Files	Search for strings within files.
Go to	Scroll the edit window to a specific location in the program.
Bookmarks	Flag particular lines of interest and quickly jump between them.
Convert Tabs	Replace tab characters with space characters in the file currently being edited. The tab spacing set in the **Settings** → **Editor Settings** dialog is used.
Toggle Comments	This allows debug statements, or other optional blocks of code, to be easily compiled in or out of an application program. It is an alternative to using a statement such as #if. To use the function, select one or more lines in the application program and then execute the command. If *all*

	of the selected lines begin with "//" comment characters as the *first* two characters, then the first "//" is *removed* from all selected lines. Otherwise, "//" comment characters are *inserted* at the beginning of each selected line.
Folding	This allows switching between fully expanded functions where the whole body of the functions is seen and collapsed functions where body of the functions is hidden. This will be enabled only if Folding is enabled in **Settings** → **Editor Settings**
SDO Macro Selector	This will open dialog where all the macros from SdoDictionary.h are listed together with their parameters. The user can select one macro at the time which is inserted in the source code.
Keyboard	Display a dialog allowing characters from other languages to be inserted into the program.

3.8.1 Paste Array Assignment

This can be used to automatically paste array assignment statements into an application program based on array value tokens on the Windows clipboard. For example, if the array name is set to “list”, the start index set to “1”, and the clipboard contains the 4 tokens “1 3 5 7”, then the following will be pasted into the program:

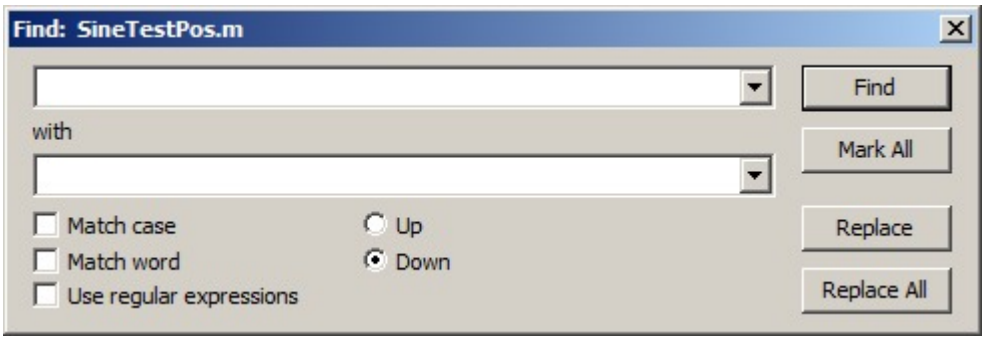
```
list[1] = 1;
list[2] = 3;
list[3] = 5;
list[4] = 7;
```

Note that the start index can be any string. For example, it could be set to “offset+1” which would produce the indices [offset+1], [offset+2], [offset+3], etc.

The value tokens on the clipboard can be copied from any other Windows application using the normal “copy” mechanism. For example, a column of values could be copied from a spreadsheet application.

3.8.2 Find and Replace

Click on **Edit** → **Find** or **Edit** → **Replace** (or press **Ctrl-F** or **Ctrl-R**) to display the Find dialog.



To find an instance of a string, enter the string in the upper box and press the **Find** button. This will jump to the next (or previous) instance of the string and highlight it. Press **Find** again to repeat the search. Clicking on **Mark All** instead of **Find** will find all instances of the string and “bookmark” them with blue triangles in the left

margin. **F2** can then be used to jump to successive bookmarks. Please note the additional capabilities of the Find dialog:

- Selecting a string in the editor (e.g. by double-clicking a word or simply selecting test with the mouse) and then clicking **Edit** → **Find**, will initialize the search string using the selected text.
- Previous search strings can be chosen using the drop-down button on the right end of the search string box.
- Selecting a new string in the editor will automatically update the search string. It is not necessary to close the dialog.
- If multiple editor windows are open, then it is possible to search in a different window simply by setting the Windows focus to the new window and pressing **Find**. It is not necessary to close and re-open the dialog. The name of the currently enabled file is displayed in the Find dialog title.

Using **Find** without the use of the dialog

If the Find dialog is closed or the Windows focus is set to an editor window rather than to the Find dialog, then it is still possible to search for strings. Simply select the search string in the editor window and press **F3** or use **Edit** → **Find Again**. This will search for the next occurance of the string using the direction orientation and other settings as they were when the Find dialog was last displayed. **Shift-F3** or **Edit** → **Find Back** will do the same search but in the opposite diection.

To replace a string, enter the string to be replaced in the upper box and the replacement string in the lower "with" box. Then press the **Find** button. This will jump to the next (or previous) instance of the string and highlight it. Then press the **Replace** button. This will replace the highlighted string with the new string and automatically search for the next instance of the search string. Pressing **Replace** again will repeat the replace and find operation. Clicking on **Replace All** instead of **Replace** will search for all instances of the search string (both before and after the current cursor position) and replace all of them with the new string.

The regular expression implementation in search and replace functionality handles the following syntax:

Wildcards	? for any single character . for any single character + for one or more of something * for zero or more of something
Sets of characters	Characters enclosed in square brackets will be treated as an option set. Character ranges may be specified with a - (e.g. [a-c]).
Logical OR	Sub expressions may be OR-ed together with the | pipe symbol.
Parenthesized sub expressions	A regular expression may be enclosed within parentheses and will be treated as a unit.
Escape characters	Sequences such as \t, etc. will be substituted for an equivalent single character. \\ represents the backslash.

Examples:

10	Search for the string “10”.
10+	Search for “1” followed by at least one “0” (e.g. 10, 100, 1000, etc.).
10*	Search for “1” followed by zero or more “0” characters (e.g. 1, 10, 100,

	1000, etc.).
vel[xyz]	Search for “velx”, “vely”, or “velz”.
vel[1-3]	Search for “vel1”, “vel2”, or “vel3”.
(vel)|(acc)	Search for “vel” or “acc”.
vel[\t]*=	Search for “vel”, followed by any number of spaces or tabs, followed by “=” (i.e. search for assignments to the variable “vel”).

3.8.3 Find in Files

The Find in Files function will display a dialog allowing the user to search for strings within files. The user can choose to search either files that are currently open in the ApossIDE or files that are on disk. All occurrences of the string which are found will be displayed in the Findbar. If the **Findbar** is not currently open, then it will be opened. Double-clicking on any occurrence in the list will open that file and position the cursor on that line.

The following options are available:

Find: Enter the string to be found. This can contain wildcard characters if “Regular expressions” are being used (see below). Use the dropdown box to select previously used search strings.

> In the edit window, preselect the string to be found (or place the cursor on the word to be found) prior to executing **Find in Files**. This string will then be used to initialize the search string.

Match case: This option specifies whether a “case-sensitive” or “case-insensitive” search is done.

Match word: When this option is enabled, the search will only match complete “words” (i.e. the character before and the character after the search string must be a “non-word” character such as punctuation or spaces). This is most useful when searching for variables; it prevents matches from being found when the string is just a substring within another longer string. This option is not available when **Use regular expressions** is selected.

Use regular expressions: Enabling this option allows wildcards to be used in the search string. The search string will be interpreted as a standard “Regular expression” (see below).

Match limit: The search will be aborted after this many string matches have been found. This prevents excessive numbers of matches from being found which can make the Findbar unusable.

Where: Select which files are to be searched:

Current file: Search only the active window. Note that the text being edited is searched so any unsaved changes *are* included in the search. Any corresponding file on disk is *not* searched.

Current program: Search the application program in the active window and all the include files (if any) that it uses. Note that the program being edited is searched so any unsaved changes *are* included in the search. Any corresponding program file on disk is *not* searched. The same is true for the include files. If an include file is being edited, then the edited version is searched along with any unsaved changes. If an include file is not being edited, then the file *on disk* is searched.

Open files: Search all open program and include file windows. Note that the text being edited is searched so any unsaved changes *are* included in the search. Any corresponding files on disk are *not* searched.

Sidebar files: Search all program and include files listed in the Sidebar. Other types of files in the Sidebar are *not* searched. Note that the files on disk are searched. If a file on disk corresponds to one of the open edit windows, then the text being edited is *not* searched; only the disk file is searched.

Directory: If "Directory" is selected, then enter (or browse to) the directory to be searched. This field also supports "drag and drop" file and directory names. If a filename is dropped, then the directory containing the file is searched. Note that if a file on disk corresponds to one of the open edit windows, then the text being edited is *not* searched; only the disk file is searched.

File filter: List the types of files to be searched. This is available only if a "Directory" search has been selected. Two wildcard characters are available:

* Match any number of characters
? Match any single character

Multiple file patterns can be separated by space characters. For example, "*.m *.mi" will search all files with file extensions of either ".m" or ".mi".

Include subdirectories: If enabled, then subdirectories are also searched. This is available only if a "Directory" search has been selected.

Toolbar Buttons

Toolbar buttons corresponding to the search methods listed above, can be added to the Edit window toolbar (see **Settings** → **Options**). This allows quick searches to be done without needing to use Find in Files dialog. The toolbar buttons are as follows:

- **Find options** - Change the search options.
- **Find in current file** - Search in the active window only.
- **Find in current program** - Search the application program in the active window and all the include files (if any) that it uses.
- **Find in open files** - Search all open program and include file windows.
- **Find in sidebar files** - Search all program and include files listed in the Sidebar.
- **Find in folder files** - Search all files in a directory. The directory to be searched is specified using the **Find options** toolbar button.

The current search string is displayed in the toolbar and is always updated with the text that the user selects in the Edit window. This string may also be edited directly in the toolbar.

Regular Expressions

The following standard "Regular expression" wildcards are supported:

.	Match any single character.
?	Match the preceding item (character or expression) zero or one time.
*	Match the preceding item (character or expression) zero or more times.
+	Match the preceding item (character or expression) one or more times.
^	Match the beginning of the line.
$	Match the end of the line.
\	The next character is a literal (i.e. not a wildcard).
[abc]	Match any character listed in the brackets. This can also be one or more character ranges such as "[a-z]" or "[A-Za-z]", etc. If the first character is ^, then

it will match any character NOT in the list. If the ^ character itself is to be matched, then it must not be placed first in the list. If the] character is to be matched, then it must be placed first in the list. If the - character is to be matched, then it must be placed last in the list.

| | Match the expression on the left or right.

( ) Used as expression delimiters if necessary.

@ (Non-standard) The following characters are case-sensitive.

~ (Non-standard) The following characters are not case-sensitive.

! (Non-standard) The exclamation character is reserved and must not be used as the first character in the regular expression. If needed as the first character, then it must be preceded by the literal character (i.e. \!).

Examples

a.b	Match any sequence of characters that contains an “a” and “b” separated by exactly one character (e.g. “axb”, “a&b”, etc.).
a\.b	Match any sequence of characters that contains the string “a.b”.
ab*c	Match any sequence that starts with “a”, contains any number of “b” characters, and ends with “c” (e.g. “ac”, “abc”, “abbc”, etc.).
^abc.*xyz$	Match all lines that begin with “abc” and end with “xyz”. Any number of characters may be between these strings.
a[0-9]+	Match any sequence that starts with “a” and is followed by one or more digits (e.g. “a1”, “a999”, etc.).
abc|xyz	Match any sequence containing “abc” or “xyz”.
(abc|xyz)[0-9]+	Match any sequence containing “abc” or “xyz” followed by one or more digits (e.g. “abc1”, “xyz99”, etc.). Note that "abc|xyz[0-9]+" would match “abc” and it would match “xyz” followed by digits, but it would not match “abc” followed by digits.

3.8.4 Go to

Go to allows the user to scroll the edit window to a specific location in the program. Either of two options may be used:

Line The edit window will be scrolled so that the specified program line number is visible. The first line in the program is line 1.

Offset The edit window will be scrolled so that the program line corresponding to the specified binary offset is visible. Binary offsets are reported in the Error History window. They can also be found in the compiler cross reference file if one is being created (see Compiler). Note that if the binary offset is actually located within an #include file rather than the main program file, then the include file will be automatically opened in a new edit window.

After the edit window has been scrolled, the line at the specified location will be highlighted with a light red color. Clicking on the line with the mouse will clear the highlighting.

3.8.5 Bookmarks

Bookmarks allow the user to flag particular lines that are of interest to him and then quickly jump between them.

If bookmarks have been used in the editor, then they are saved and restored along with the program file.

Next Bookmark

If bookmarks have been used in the editor, then this function or **F2** will scroll the program and position the edit cursor on the next line that contains a bookmark.

Previous Bookmark

If bookmarks have been used in the editor, then this function or **Shift+F2** will scroll the program and position the edit cursor on the previous line that contains a bookmark.

Toggle Bookmark

This function or **Ctrl+F2** will toggle the bookmark on the current line. If the line contains no bookmark, then a bookmark is placed on the line. If the line already contains a bookmark, then the bookmark is removed from the line.

Clear All Bookmarks

Click on **Edit** → **Clear All Bookmarks** to clear all existing bookmarks from the editor.

3.8.6 Folding

Folding allows to show and hide functions/ conditions by clicking on the displayed "+" or "-" sign. To use the feature you have to work with curly brackets {...} in the code.

Fold All

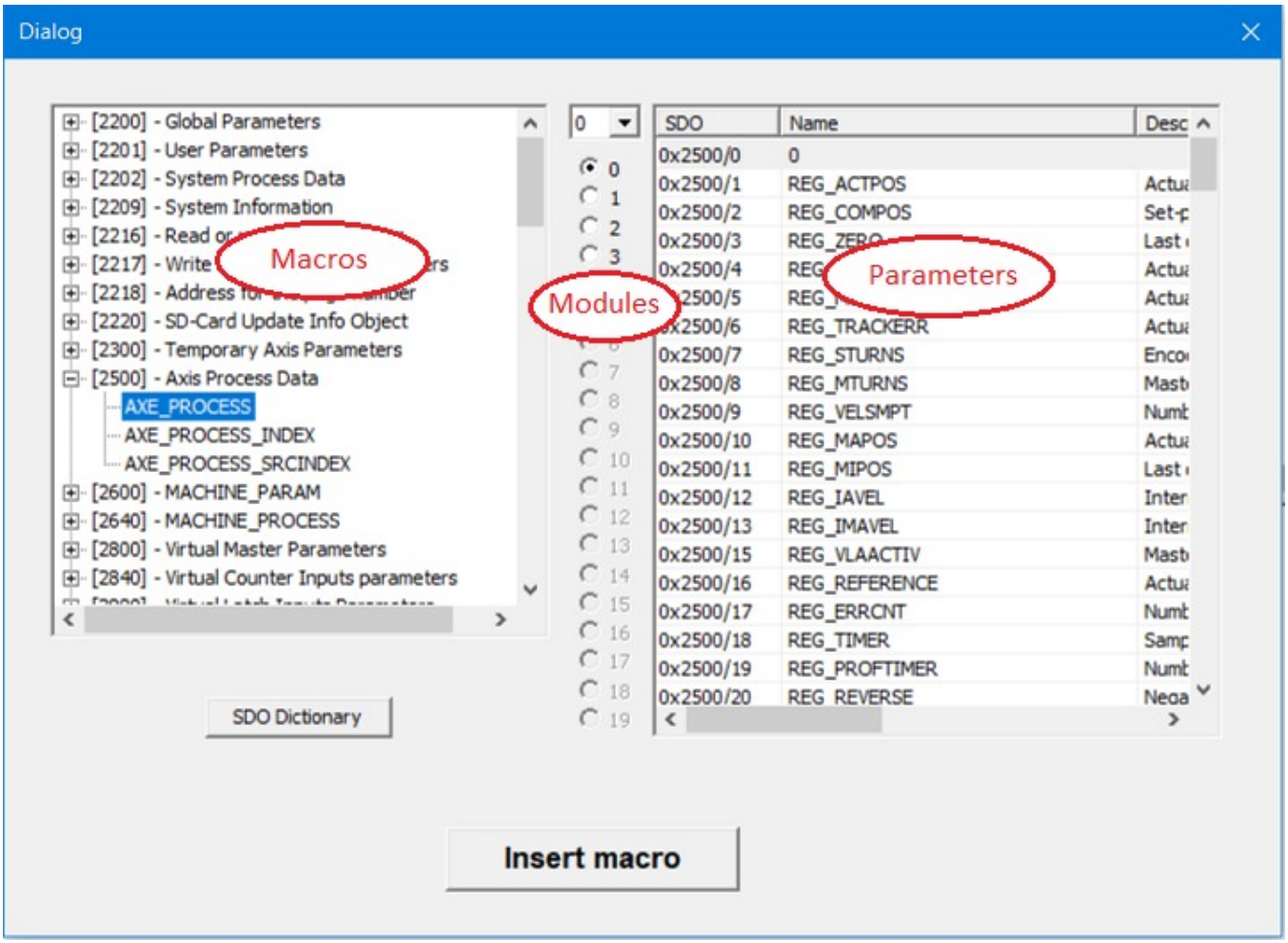
If folding has been enabled in the editor, then this function or **Alt+O** will collapse all functions in the currently opened file.

Unfold All

If folding has been enabled in the editor, then this function or **Alt+Shift+O** will expand all functions in the currently opened file.

3.8.7 SDO Macro Selector

The SDO Macro Selector dialog shows all the macros which are available in SdoDictionary.h together with their parameters. User can select a particular macro and insert it in the source code.



Macros

The **Macros** on the left side of the dialog contain macros which are grouped according to their SDO index.

Modules

The **Modules** in the middle of the dialog contain all available module numbers which are available for selected macro on the left. If this control is disabled, then macro doesn't expect a module number parameter.

Macro parameters

The **Macro parameters** on the right side of the dialog contain list of parameters that are available for selected macro on the left side.

SDO Dictionary

The **SDO Dictionary** button under the **Macros** will open new window with Help describing all SDO indexes.

Insert macro

The **Insert macro** button on the bottom of the dialog will insert the selected macro with the module number and parameter to the source code at the cursor location.

3.9 Development Menu

This menu provides various functions used during the program development phase of applications. This includes functions to compile, execute, break, and debug programs. It also includes functions that allow controllers to be connected and disconnected.

Many of these functions require a controller to be connected before the function can be executed. A controller can be connected either by pressing **Esc** to connect to the default controller or by using **Development** → **Select Controller** to connect to a specific controller if more than one controller is present. If no controller has yet been connected when a function is executed, then most of these functions will automatically connect to the default controller.

For information on debugging programs, please see Debugging Programs at the end of this chapter.

3.9.1 Execute

This will execute the program currently being edited. This involves the following steps:

1. If a controller is not currently open and connected to the Editor Window, then an attempt is made to connect to the currently defined default controller. If no controller is connected, then the **Execute** is aborted.
2. A check is made to see if the controller is already executing a program. If so, then the user is asked if the existing program can be stopped. The controller can only execute one program at a time. If the controller is not "idle", then the **Execute** is aborted.
3. The program being edited is then compiled. The compile will be done using the current version of the file being edited as well as the current versions of any include files that are opened for editing. The files do *not* need to be saved before being compiled. This produces a "machine code" version of the program that is customized for that particular controller. Note that this "machine code" is not human-readable and cannot be edited. If the compile fails for any reason, then the **Execute** is aborted.
4. If the compile is successful and the user has made changes to the program being edited, then the changes are automatically saved to the PC disk if the **Autosave** option is enabled. If this is a "new" program, then the user is requested to enter a filename for the program. Note that it is not necessary to enter a filename at this time. If no filename is given, then the main program file is not saved.
5. The "machine code" (i.e. not the original text file being edited) is then downloaded to the controller and placed in the controller's temporary memory. Note that this temporary memory will be lost if the controller is powered off. To save a program permanently in the controller, use **Controller** → **Programs**.
6. After the download completes, the controller will start to execute the program.

Each subsequent time that **Execute** is used, the previously downloaded program will be overwritten with the newly downloaded program. This allows you to quickly and easily make changes and re-test during debugging.

Run Programs in Several Controllers

You can execute different programs on different controllers all at the same time simply by opening each program in a different ApossIDE Window. Then use **Development** → **Select Controller** in each window to select the desired controller for that program. Finally, use **Development** → **Execute** in each window to download and start the programs.
Note that you must use **Select** to open the controllers before using **Execute** since **Execute** will always connect to the default controller if no controller is yet connected.

Each ApossIDE Window can be connected to only one controller at a time and each application program can be edited by only one ApossIDE Window at a time. Hence, if you want to execute the same program on several controllers, then use one of the following methods:

- Use **Development** → **Select Controller** to select and connect to the first controller. Then use **Development** → **Execute** to download and start the program executing on the controller. Once the program is executing, then use **Development** → Select Controller a second time to select and connect to the second controller. This will disconnect from the first controller but leave the program executing. Then use **Development** → **Execute** to download and start the program on the second controller. Repeat the process for as many controllers as desired.
- If the program is to be started on several controllers that all share the same network connection (for example, several controllers on a CAN network), then **Development** → **Download all** can be used. This allows the program to be compiled, downloaded, saved in permanent controller memory, and started, all in a single user operation.

3.9.2 Break [Esc] and Continue

Click on **Development** → **Break** or press **Esc** in order to stop any program currently executing on the controller.

> If an executing program is aborted while the drive is rotating, then the drive will slow down with the maximum allowed deceleration.

Development → **Break all** can be used to stop the execution of multiple controllers at the same time.

Click on **Development** → **Continue**, in order to continue a program which was just aborted. In doing so any positioning processes which were interrupted will be completed.

If a program with an error message was aborted, you can **Continue** it again with this function once you have removed the error and/or erased the error message.

3.9.3 Messages -> Log file

This function can be used to start the logging of all messages displayed in the Communication Window to a file. Similarly, **Stop logging** will stop the logging of messages.

Messages are cached and may not always be written to the file immediately. As a result, if the file is edited or copied while logging is in progress, then the most recent messages will not always be included. To avoid this, use the **Update Log file** command immediately before editing or copying the file. This will force all messages cached up until this point to be written to the file.

Note that **Stop logging** and **Update Log file** will be enabled only if logging has been started.

3.9.4 Debugging Commands

The following commands are designed to be used to assist the user in debugging newly developed programs. For detailed information about debugging and the use of these commands, please see Debugging Programs.

Start Debugger

This command will prepare both the ApossIDE and the controller for debugging. This includes compiling the program in “debug” mode, inserting breakpoints, and downloading the program.

Go to Breakpoint

This command will start execution of the program at the current program line and continue executing until the next “user breakpoint” is reached.

Singlestep

This command will execute a single program line and stop at the next line. The next line is not executed.

Stop Debugger

This command removes the ApossIDE and the controller from “debug” mode.

Remove Breakpoints

This command removes all “user breakpoints”.

3.9.5 Watch Add / Start / Stop

This function enables online monitoring of the variables, arrays, system and axis processing data and axis parameters.

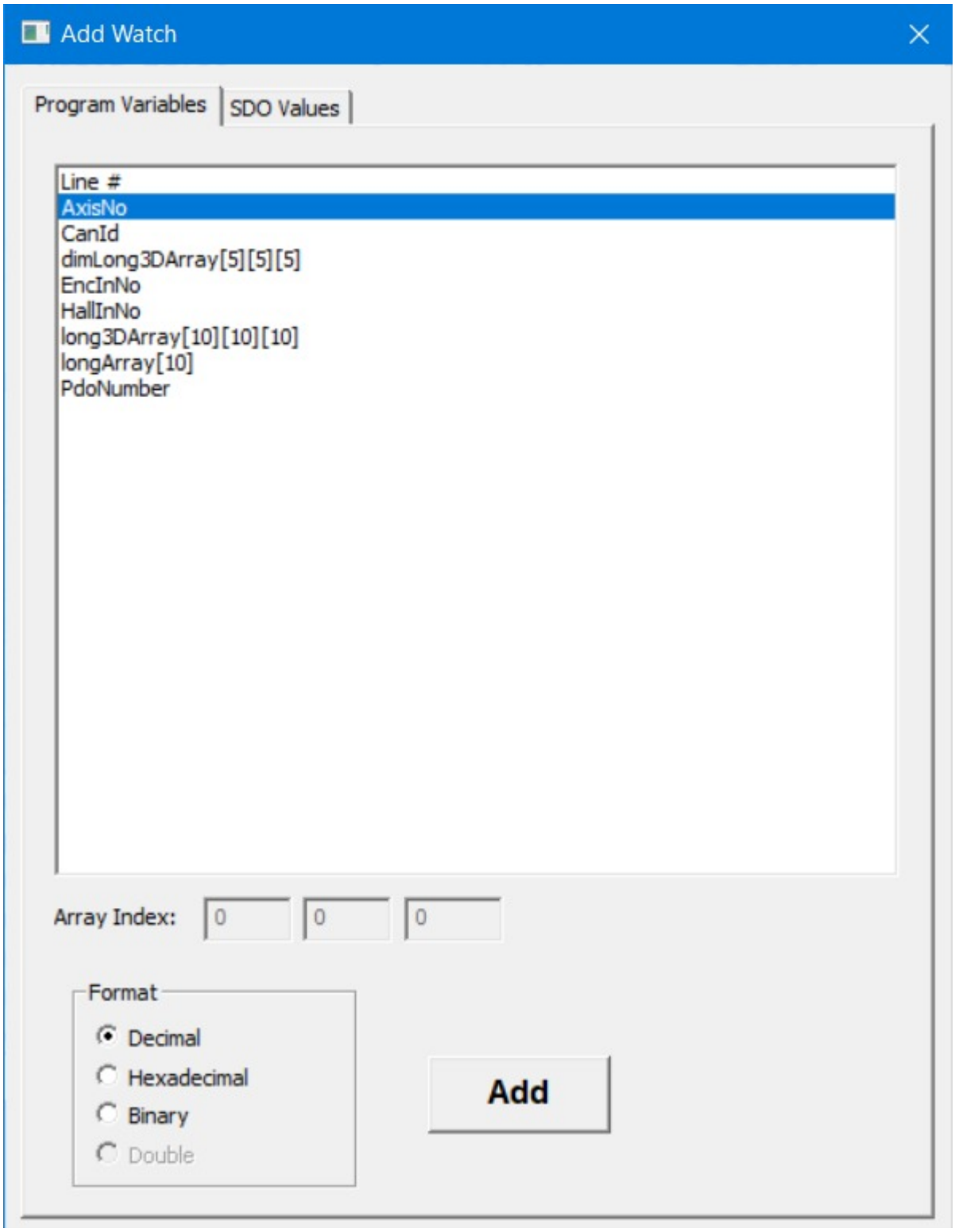
All values currently being watched are maintained in the **Watch Window**. New values can be added to this list using **Development** → **Watch Add**. Existing values can be removed from the list by clicking on the value and then pressing the **Delete** keyboard key. They can also be removed by right-clicking and using the popup menu.

Values can be written to the controller (when a controller is connected) by right-clicking on an item in the list and selecting **Set Value** from the popup menu. Monitoring does not need to be active for values to be written. **Set Value** will only be enabled if a controller is currently connected.

Activate and stop the monitoring with **Development** → **Watch Start** and **Development** → **Watch Stop**.

> When monitoring is active, the Watch Window is updated constantly. This consumes both controller resources and network connection resources. Hence, the number of values being watched should be limited to a reasonable number (usually, not more than 10-15 values, depending on network connection speed). If more values than this are required, then the **Oscilloscope** should be used.

Watch Add



The Watch Add dialog is shown above. All program variables and arrays are shown in the "Program Variables" tab and all system values are shown in the "SDO Values" tab. Add a value to the Watch Window by clicking on it, selecting the format to be used to display the value, and then pressing the **Add** button. Values can also be added simply by double-clicking on the value. Multiple values can be added before closing the dialog.

If an array value is being added to the Watch Window, then the array indices must be specified.

These fields will be disabled if the selected value is not an array. Note that only the first 250 elements of an array can be watched.

> This window can be resized allowing variables with long names to be fully displayed.

3.9.6 Syntax Check

Clicking on **Development** → **Syntax Check** will do a “test compile” of the program being edited. This can be a useful function to use while creating new programs or modifying existing programs. It is a quick way to find syntax errors in the program without having to download the program to the controller and execute it. If a syntax error is found, then the line number and an error description are displayed in the Communications Window and the cursor is automatically placed at the position of the error.

Syntax Check produces a debug file in addition to checking the syntax. This file will have that same name as the program but with the file extension “.ad$”.

> The ApossIDE will compile using the current version of the file being edited as well as the current versions of any include files that are opened for editing. The files do *not* need to be saved before being compiled.

3.9.7 Compile to file

The **Execute** command will compile a program into a “machine-readable” binary version which is then downloaded to the controller and executed. The **Compile to file** command is similar except that the compiled binary version is *not* downloaded and executed. Instead, the binary version is saved in a ".m0" or “.bin” file on the PC hard drive. Binary files can be managed using **Controller** → **Programs**.These files are intended to be useful for archiving purposes.

When **Compile to file** is selected, a dialog will be displayed allowing the user to specify the exact controller hardware for which the program is to be compiled. Note that compiled binary versions are hardware-dependent and must be compiled for the hardware on which they will be executed. After this, a **Save As** dialog will be displayed allowing the user to select the file name to be used to save the file. The file name will default to the name of the program. A file extension of either ".m0" or ".bin" may be used. Only after this is the program compiled and saved on the PC.

- **.m0** files contain the compiled binary version of the program in an ascii text file.
- **.bin** files contain the compiled binary program in a raw binary file.

> If the compile is successful and the user has made changes to the program being edited, then the changes are automatically saved to the PC disk if the **Autosave** option is enabled. If this is a “new” program, then the user is requested to enter a filename for the program. Note that it is not necessary to enter a filename at this time. If no filename is given, then the main program file is not saved.

3.9.8 Compare to Controller

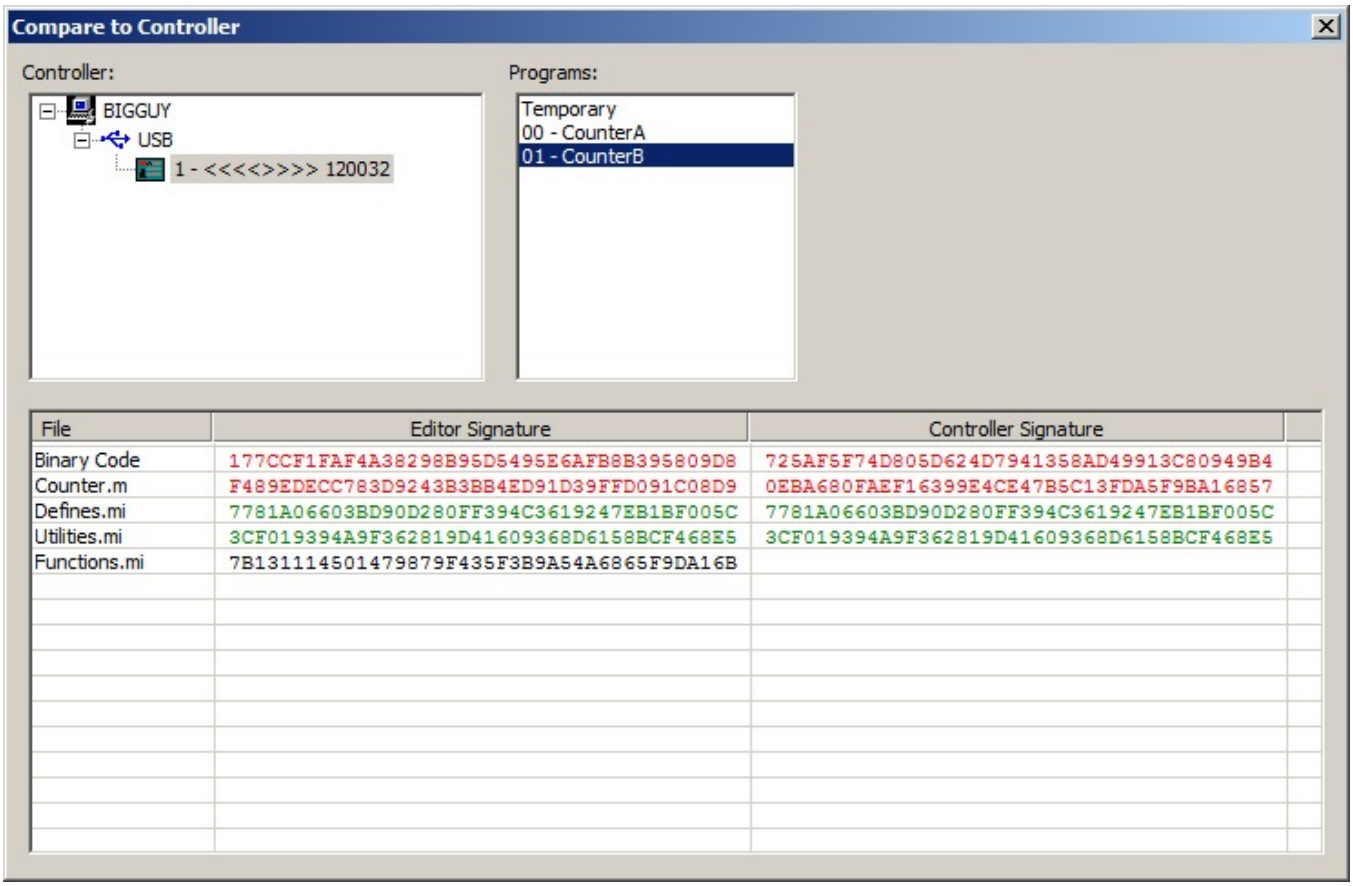
Applications sometimes require assurance that the actual application program running in the controller is identical to a "master" program. For example, a specific version of a program may have been "certified" in some way. It is then important that all controllers are running that exact version. Any modification to the program will cause it to lose its certification.

The ApossIDE supports this requirement by using cryptographic "identification signatures". These are 20-byte values that will *uniquely* identify both the application program as a whole and each of the source files used to

create the program. Any substantive modification to the application program or source files will cause the identification signatures to change. See below for a description of which modifications may and may not be made. These signatures provide maintenance personnel with the ability to verify that programs have not been modified.

If identification signatures are to be used, then "File Identification Support" must be enabled using **Settings** → **Compiler**. Note that "File Identification Support" may be enabled on a per-program basis by including #pragma FILE_IDENT in the program. When enabled, identification signatures will be included in application programs and downloaded to the controller.

The **Compare to Controller** command allows the user to view the signatures associated with both the application program currently being edited and the application programs stored on a controller. It will display a dialog similar to the following. Note that these signatures can also be viewed by using **Controller** → **Diagnostics Report** to generate a diagnostics report.



Controller

Select the controller with which the program currently being edited is to be compared.

Programs

This list shows all programs currently stored on the controller (see **Controller** → **Programs**). Select the specific program with which the program currently being edited is to be compared.

The special "Temporary" item corresponds to the program currently running on the controller. If no program is currently running, then this will correspond to the last program that was running. See System Architecture for a description of temporary programs.

File

This column lists the names of the source files referenced by either the editor program, the controller program, or both. Note that pausing the mouse in one of the "Signature" columns will display a yellow popup message containing the full pathname of the corresponding file.

The special "Binary Code" line refers to the program as a whole. The other lines refer to the actual individual source files.

Editor Signature

This column lists the signatures corresponding to the application program currently being edited. The column will be blank if controller files are found for which no corresponding editor file can be found.

Controller Signature

This column lists the signatures corresponding to the selected application program on the selected controller. The column will be blank if editor files are found for which no corresponding controller file can be found. For example, in the above illustration, the editor program references a "Functions.mi" include file that is not referenced by the controller program.

The special "Binary Code" line contains *code-based* signatures for the entire programs. If these signatures match, then they will be displayed green and the two *compiled* application programs will be exactly identical, regardless of when they were compiled, the source files from which they were compiled, the naming conventions used in the files, etc. For example, files could be moved or renamed, comments and whitespace within the files could be added, modified, or deleted, variables could be renamed, etc. If these signatures do not match, then they will be displayed red and the two programs will *not* be the same program.

All other lines contain *file-based* signatures of the individual source files. If the signatures of two files match, then they will be displayed green and the two *source* files will be exactly identical, regardless of spacing and comments within the files. For example, comments could be added, modified, or deleted, blanks lines could be added or removed, leading and trailing whitespace could be altered, etc. However, any change to the *functional* contents of the file (i.e. the actual program statements) *will* alter the signature. If these signatures do not match, then they will be displayed red and the two files will *not* contain the same program.

3.9.9 Break all

If programs are executing in several controllers, then click on **Development** → **Break all** to abort all the executing programs. Note that this will only abort the programs running on controllers that share the same connection interface as the controller connected to this ApossIDE Window (e.g. multiple controllers on a CAN bus). Controllers connected using other interfaces are not affected. For example, if this ApossIDE Window is connected using CAN-LPT, then any controller connected using USB will continue to execute any previously executing program.

⚠ Programs will be aborted even on those controllers that are not currently connected to an ApossIDE Window. As long as the controller is using the same connection interface and is within the original connection ID scan range, then execution is stopped. For example, if controllers 1 and 2 are both on a CAN network but the ApossIDE is currently only connected to controller 1, then **Break all** will still abort programs running on both controllers 1 and 2.

STOP If an executing program is aborted while the drive is rotating, then the drive will slow down with the maximum allowed deceleration.

3.9.10 Start all

The **Start all** menu will send an 'Execute' command to all connected controllers that share the same connection interface as the controller connected to this ApossIDE Window (e.g. multiple controllers on a CAN bus). If you are testing software that runs on multiple controllers in a network, then this is a quick way to start all controllers at the same time.

3.9.11 Download all

Development → **Download all** displays the **Download Mode** dialog. This allows the .m file currently being edited to be downloaded to multiple controllers.

Please read **Download** → **Programs** for more details and options.

3.9.12 Delete Programs all

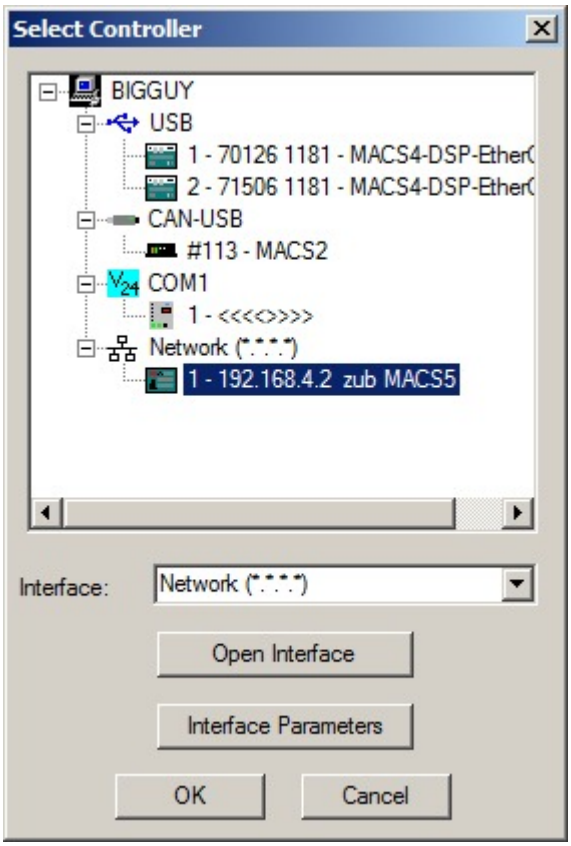
Development → **Delete Programs all** will delete all programs saved on all controllers that share the same connection interface as the controller connected to this ApossIDE Window (e.g. multiple controllers on a CAN bus). Controllers connected using other interfaces are not affected.

> Programs will be deleted even on those controllers that are not currently connected to an ApossIDE Window. As long as the controller is using the same connection interface and is within the original connection ID scan range, then programs will be deleted. For example, if controllers 1 and 2 are both on a CAN network but the ApossIDE is currently only connected to controller 1, then **Delete Programs all** will still delete programs on both controller 1 and 2.

This command is intended to be used in conjunction with **Development** → **Download all**.

3.9.13 Select Controller

If more than one controller is accessible from the PC, then **Development** → **Select controller** can be used to select the specific controller to which the ApossIDE Window is to be connected. All currently available controllers will be displayed in a tree view. Select the desired controller and click on **OK** to connect to that controller.



If no controllers are shown or the desired interface is not present, then select the desired interface from the Interface dropdown box and click on **Open Interface**. The desired interface can then be selected. Note that choosing an interface in this way does *not* change the default interface specified using **Settings** → **Interface**.

If a controller is already connected to the ApossIDE Window, then **Development** → **Select controller** can be used to switch the ApossIDE Window to another controller. Simply use the Select Controller dialog to select the new controller. The existing connection to the previous controller will then be closed and a connection to the new controller will be opened.

Since any specific controller can be connected to only one ApossIDE Window at a time, selecting a controller that is already connected to another ApossIDE Window will cause that controller to be switched from the other window to this window.

3.9.14 Close Interface / Close All Interfaces

If there is a controller currently connected in this edit window, then **Development** → **Close Interface** can be used to close the connections to all controllers that share the same connection interface as this controller (e.g. all the controllers on a CAN bus). A message will be displayed indicating which controller connections will be closed.

Close All Interfaces is similar to the **Close Interface** function except that all controllers on all connection interfaces are closed.

3.10 Controller Menu

This menu provides various functions used to view and set the controller configuration and to manage various aspects of the controller's onboard memory. The following functions are available:

Programs	Functions to manage saved programs (and mark programs for

	Autostart).
Parameters	Functions to manage configuration parameters on the controller.
Reset	Functions to reset various aspects of the controller.
Reboot	Rebooting a controller simulates turning the controller off and on again. This immediately stops any running program and then completely reinitializes the controller.
Error History	View a history of all the errors that have occurred on the controller.
Diagnostic Report	Create a text file containing the entire current state of the controller. This file is often required by zub machine control AG support personnel when assistance is required to help diagnose problems.
Diagnostics	Display a window showing diagnostic information from the connected controller.
Debug Log	This function is not a user function. It is designed to be used by zub machine control AG support personnel to help with the diagnosing of firmware bugs. Please use this function only when requested to do so by zub.
Read/Write SDO	Display a window allowing SDO values to be read from the controller and/or written to the controller.
Download Firmware	Download new firmware to the controller.
FTP Server	For controllers connected using Ethernet and that provide a microSD card slot, this function will open a normal Windows Explorer window that will allow access to the files on the microSD card. The controller must have firmware version 7.2.11 or later. Note that it may be necessary allow Windows Explorer through the Windows firewall before the window can be properly displayed. See instructions below.
State Machine	Functions related to the use of state machines.

How to Allow Windows Explorer through the Firewall

1. Allow Windows Explorer through the Windows firewall:
   a. In the Windows Control Panel, click **Windows Firewall**.
   b. On the left side of the Windows Firewall window, click **Allow a program or feature through Windows Firewall**.
   c. In the Allowed Programs window, scroll down until **Windows Explorer** is displayed. Ensure that the application itself is enabled (on the left) and that the appropriate network (private or public) is enabled (on the right). If a change is necessary, then click **Change settings**, enable the appropriate checkboxes, and then press **OK** at the bottom of the window. Do not simply close the window since this will discard your changes.
   d. Close the Windows Firewall window. It is no longer needed.
2. In some cases, you may find that the network where the controller is located, is flagged as a "public"

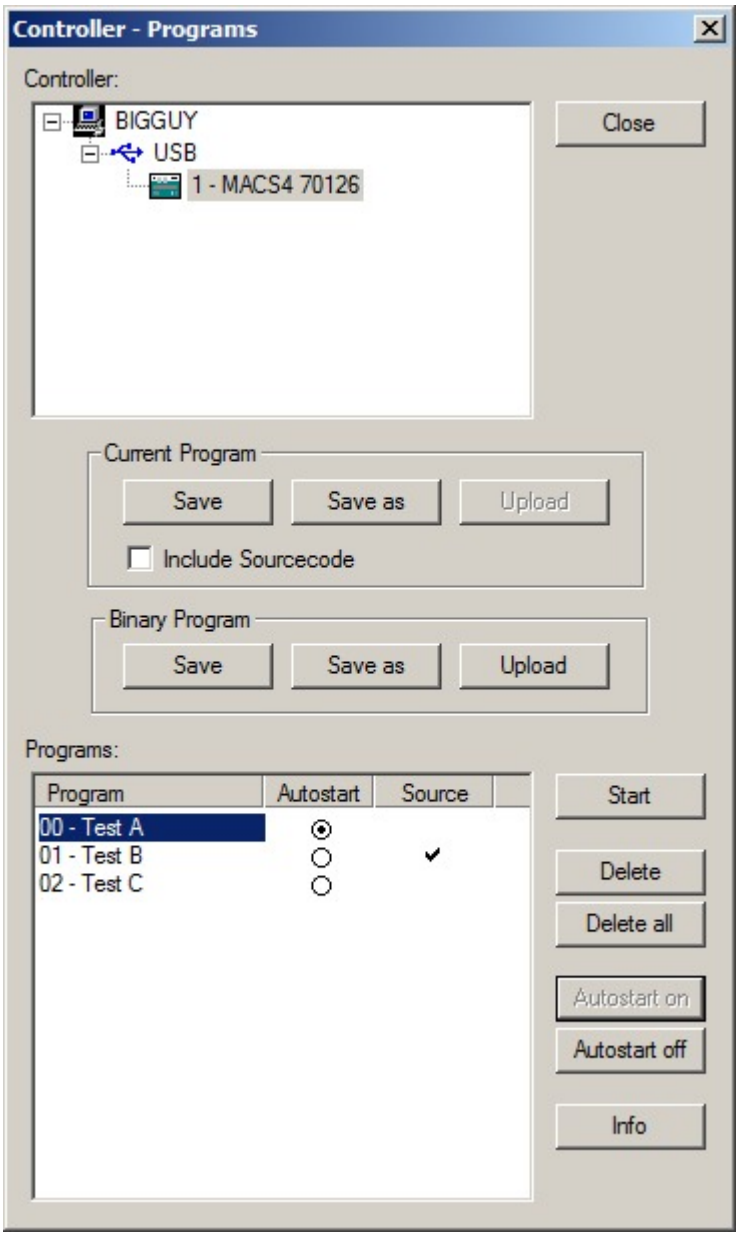
network even though the network is a completely private and trusted network. For example, the controller may be connected to a local Ethernet NIC in the computer and the network appears as an "Unidentified network". You may also not want to allow Windows Explorer full access through the firewall to public networks. If this is the case, you can change the location of the controller network from "public" to "private" as follows:

a. In the Windows Control Panel, click **Administrative Tools**.
b. In the Administrative Tools window, double-click **Local Security Policy**. It may be necessary to wait some time before the Local Security Policy window is opened.
c. In the Local Security Policy window, click **Network List Manager Policies** on the left. Then double-click **Unidentified Networks** in the right.
d. In the Unidentified Network Properties dialog, change **Location type** to "Private". Press **OK**.

3. Close the various Control Panel windows. They are no longer needed.
4. Power the controller off/on so that it can reset its network connection.

3.10.1 Programs

Click on **Controller** → **Programs** to see all the programs currently stored on the controller in permanent memory. You can also select other currently connected controllers to see (and manage) the programs on those controllers.



Under **Programs** is a list of all programs currently saved on the controller. Each program will be identified by a program number and a program name. Depending on available memory and the size of the saved programs, up to 91 programs may be saved at any one time. If a program has been designated as the **Autostart** program, then ⦿ will appear in the Autostart column. If source code for a program has been included on the controller, then a checkmark will appear in the Source column.

Current Program

Whenever you execute a program (i.e. with **Development** → **Execute**), it is compiled and downloaded into a temporary block of memory in the controller. This is overwritten with each subsequent **Execute** and it is lost if the controller is powered off. However, using **Controller** → **Programs**, you can permanently **Save** the current program in the ApossIDE Window onto the controller. The program is compiled and downloaded into a permanent block of memory in the controller. The new program will be appended to the end of the list of existing programs.

> 💡 Saving a program on the controller does *not* automatically save the original source code for the

> program. You should always use the normal **File** → **Save** menu command to save the program source to the PC hard drive.

Save and Save as

Click on **Save** and enter a name in the subsequent dialog field or confirm the file name suggested. The program number will be assigned automatically.

For controllers with firmware version 6.01.10 or later, names are saved using UTF-8 encoding. This allows any character to be used in names, even characters in foreign languages. However, names may not be longer than 8 bytes (n.b. some UTF-8 characters require more than one byte). For controllers older than 6.01.10, characters are restricted to ANSI characters in the range 0x20 to 0x5F (n.b. accented characters may not be used) and the name cannot be longer than 8 characters.

Click on **Save as** and you can also assign the program number (0 to 90) yourself, in addition to the program name.

Include Sourcecode

When this checkbox is enabled, the original source code for the program being saved will also be downloaded and saved on the controller. This source code can later be uploaded back from the controller and edited again on the PC.

When program source code is downloaded, then **Include** files used by the source code are also downloaded along with the source code. This allows a complete program, rather than only a partial program, to be stored on the controller.

All programs that have saved source code associated with them will have a checkmark in the Source column.

> If insufficient memory is available on the controller for saving the source code, then a message is displayed and other programs must be erased before saving the new program.

Create binary file

These options allow the user to create binary .m0 and/or .bin files of the program being downloaded and saved. These files are saved on the PC. These files are intended to be useful for archiving purposes.

- **.m0** files contain the compiled binary version of the program in an ascii text file.
- **.bin** files contain the compiled binary program in a raw binary file.

> Binary program files may also be created using the **Development** → **Compile to file** menu command.

Upload a Current Program

All programs with a checkmark in the Source column have the associated source code for them saved on the controller. This source code can be uploaded back from the controller to the PC for subsequent use.

Select the desired program and click on **Upload**. The default program name will contain the date and time when the original source code was saved on the controller. This is done to help avoid potential problems caused by overwriting existing files on

the PC.

Binary Program

The **Save** and **Save as** buttons will display a dialog allowing a previously compiled .m0 or .bin program file to be selected. A second dialog will then be displayed allowing the program to be saved on the controller.

The **Upload** button will read the currently selected program from the controller. The program may then be saved as either a .m0 or .bin file.

> Controllers older than version 6.1.15 do not support the uploading of binary files. In this case, the **Upload** button will be disabled.

Start

Any saved program can be started directly. Simply select the program to be started and click on **Start**.

Following actions are taken at every program start:

- All axes are set to AxisControl(..., OFF)
- All axis parameters are reset to their default values
- All global parameters are reset to their saved values in flash
- All user parameters are reset to their saved values in flash
- All digital outputs are set to 0
- Virtual digin/digout/anout/anin modules are re-initialized
- Amplifier parameters are reset to their default values
- All HW module parameters are reset to their default values
- DIM Arrays are initialized
- Internal recording is initialized
- State machine functions will be initialized

Delete

Select any program and click on **Delete** if you want to delete that individual program from the controller.

Click on **Delete all** if you want to delete all the programs in the controller.

> This *permanently* deletes the programs from the controller. Hence, you should ensure that you have previously saved the programs on the PC.

Autostart

The **Autostart** button allows any *single* program to be designated as the **Autostart** program. Please see Autostart for more information.

Info

Select any program and click on **Info** to view the compiler options used when compiling the program. Note that this function is not available on some older controllers.

3.10.1.1 Autostart

In order to support the use of the controller in “stand-alone” or “turnkey” applications (i.e. without user input), the controller can be configured so that an application program is started automatically when the controller is powered on. Typically, in these types of applications, the controller must also never be “stopped”. So the controller can also be configured to restart an application program in the event that the executing program terminates.

The first part of this mechanism is the “Autostart program”. The Autostart program (if one is defined) is normally *always* started automatically when the controller is powered on. If no Autostart program has been defined, then *no* program is started when the controller is powered on. Note that when the controller is powered on, it will run various self-test checks. If any of these fail, then the controller will not attempt to start the Autostart program.

The second part of the autostart mechanism is the “Restart program”. With some exceptions, the Restart program is started automatically when the currently executing program terminates. In particular, this includes the termination of the Autostart program.

> If the Autostart program, Restart program, or any other program *is aborted by the user*, then the autostart mechanism will be *deactivated*. The controller will not automatically start the Autostart program or any Restart program again until the controller is powered off and on.

> Note that a program designated as an Autostart program can always be *manually started* by the user. However, in this case, the program will execute as a normal program, *not* as an Autostart program, and no Restart program will be started when the program terminates.

Autostart program

Any saved program can be designated as the Autostart program by selecting it and clicking **Autostart on**. The Autostart program will be marked with ⦿ in the Autostart column. The autostart designation can be cleared again simply by selecting the program and clicking **Autostart off**. Only one program at a time can be designated as the Autostart program. So if a second program is chosen as the Autostart program, then the Autostart designation is cleared from the previous program.

One of the main purposes of an Autostart program is to select which of multiple programs is to be executed in applications that require different programs in different situations. The Autostart program will identify (usually based either on configuration parameters or on digital input settings) which program is to be executed. The Autostart program then designates that program as the Restart program and terminates. The controller will then automatically start the designated program.

Restart program

If the controller executes an Autostart program when it is powered on, then the controller’s autostart mechanism will be activated. When the autostart mechanism is active, then the controller will automatically start the designated Restart program whenever the currently executing program terminates. This will continue until the autostart mechanism is deactivated.

The autostart mechanism will be deactivated if any of the following happen:

1. The user has executed a “break” (i.e. ESC) command.
2. A program has terminated with an error (other than one of the errors listed below).

Once deactivated, the autostart mechanism cannot be reactivated again without first powering the controller off and on again.

The autostart mechanism will *not* be deactivated if any of the following errors occur:

1. Position error (Error 8)
2. Limit switch error (Error 25)

3. Software limit switch error (Error 11)

The controller determines which program is the Restart program as follows:

1. If PRGPAR is set (i.e. not -1), then this is the program number of the Restart program.
2. Otherwise, if the Autostart program is the *only* program to have been executed, then the Autostart program will be the Restart program (i.e. the Autostart program will be executed repeatedly).
3. Otherwise, there is no Restart program and the controller will not autostart any program.

The PRGPAR parameter is designed to be used for "linking" programs together. The currently executing program simply needs to set PRGPAR to specify which program is to be executed next. Note that if PRGPAR is not reset, then the current program will be executed repeatedly.

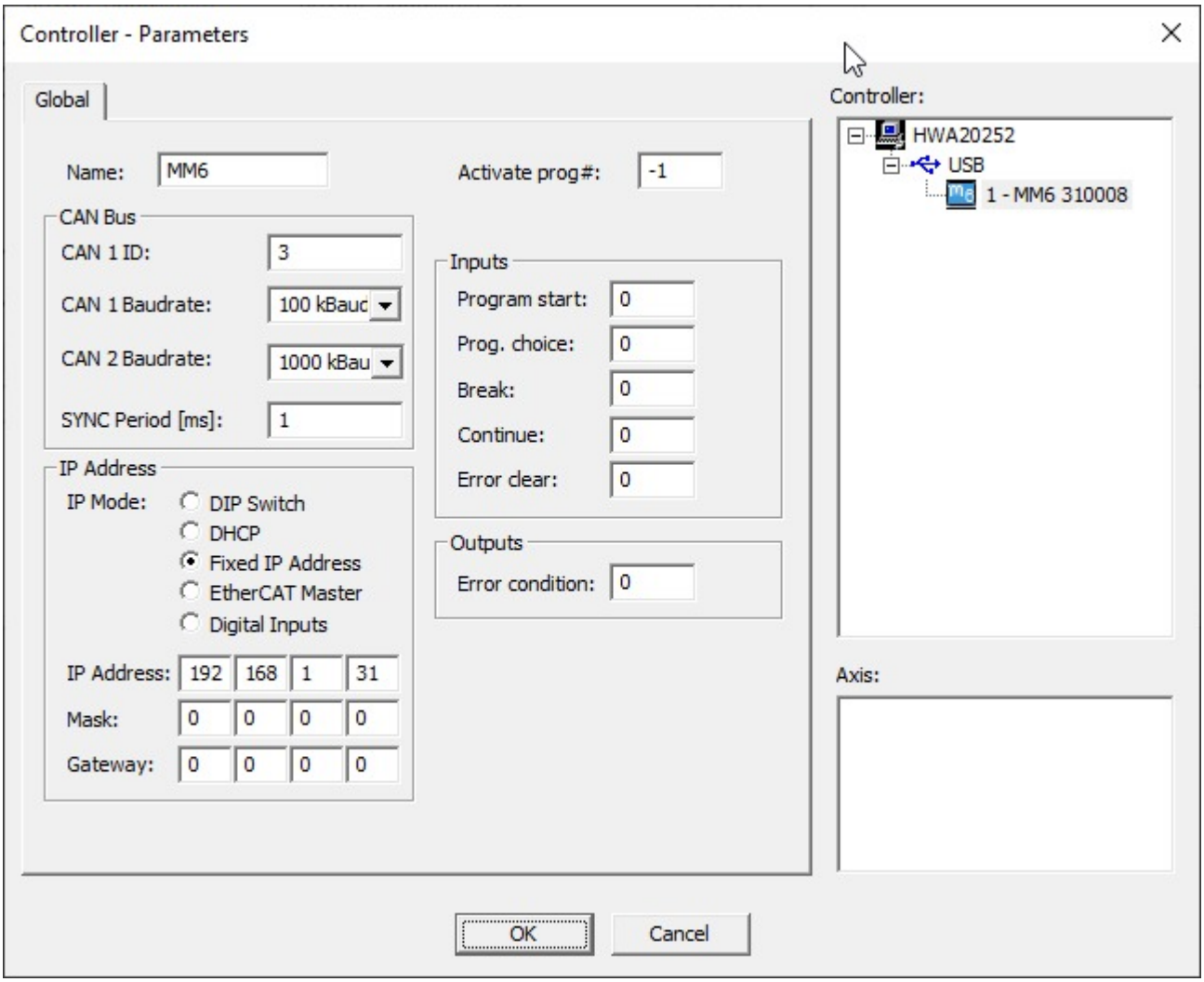
> The originally designated Autostart program is restarted *only* if PRGPAR has *not* been used. If PRGPAR has been used, then the Autostart program will only be executed once when the controller is powered on. This "one-time" execution can be useful for executing functions such as HOME. Note that the PRGPAR parameter can still be used to explicitly restart the Autostart program.

3.10.2 Parameters

The controller maintains global parameters and axis parameters. Global parameters are valid for the entire controller. Axis parameters can have different values for each axis.

3.10.2.1 Global / Axis

When parameters are being edited, then a dialog similar to the following will be displayed. A tab will be displayed for each of the various groups of parameters that are available on the selected controller. Not all tabs are available on all controllers.



Different axes and different controllers can be selected. When selected, then the parameters for that axis and controller will be displayed and can be modified. Note that *no* changes are written to *any* controller until the **OK** button is pressed. When the **OK** button is pressed, then the modified parameters will be written for all controllers.

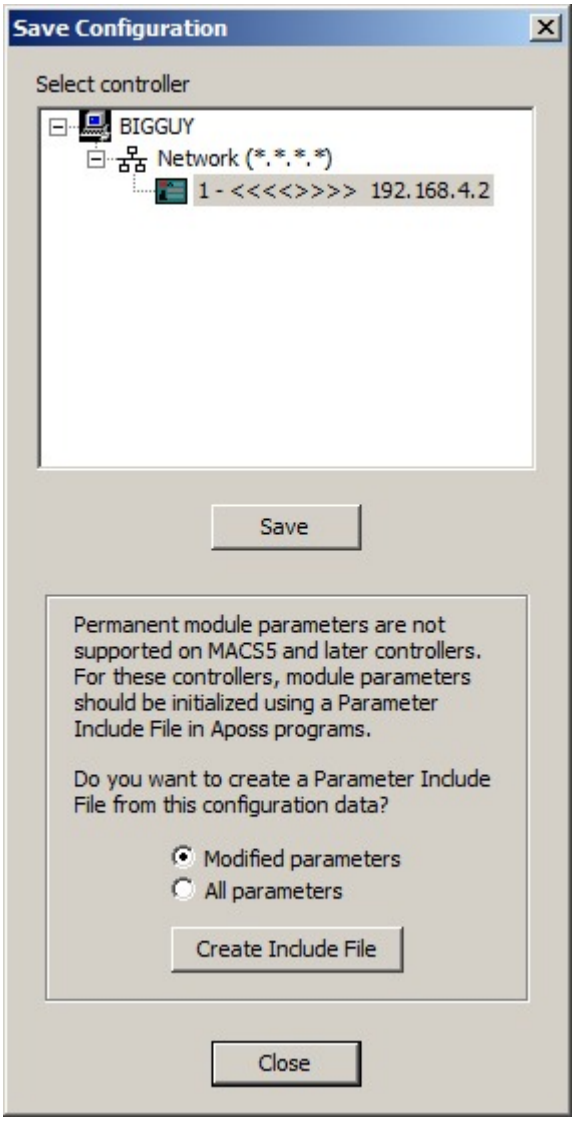
In all of the parameter tabs, the internal parameter number is listed after the name of each parameter. To find detailed information, parameter values, and initialization values for any parameter, please use this number and refer to chapter Parameter Reference. Or simply press **F1** when the text cursor is in one of the input fields and information about the corresponding parameter will be displayed.

> The global parameter "***Name***" allows you to assign a name to a controller or to change the existing name of a controller. The name is displayed in various dialogs in the ApossIDE user interface, allowing you to more easily differentiate between multiple controllers.

For controllers with firmware version 6.01.10 or later, names are saved using UTF-8 encoding. This allows any character to be used in names, even characters in foreign languages. However, names may not be longer than 8 bytes (n.b. some UTF-8 characters require more than one byte). For controllers older than 6.01.10, characters are restricted to ANSI characters in the range 0x20 to 0x5F (n.b. accented characters may not be used) and the name cannot be longer than 8 characters.

3.10.2.2 Save to file

For backup purposes, the entire controller configuration (i.e. global parameters, axis parameters, user parameters, module parameters, and arrays) can be uploaded and saved as a file on the PC. This file can then be downloaded again to the controller at a later time if it becomes necessary to restore the configuration.



Select the controller, click on **Save**, and then enter a name in the subsequent “Save As” dialog. If the parameters of multiple controllers are to be saved, then simply select another controller and **Save** again.

MACS5 and Newer Controllers

For MACS5 and newer controllers, only global parameters, user parameters, and DIM arrays are saved permanently in the controller's flash memory. Module parameters (including axis parameters) are *not* maintained permanently in flash memory. Module parameters are saved in the file but these parameters are *not* restored to the controller by **Controller** → **Parameters** → **Restore from file**. For these controllers, **Create Include File** can be used to create an include file that can be used with application programs. Please see Initializing Temporary Parameters for a more detailed description of module parameter handling in these controllers.

3.10.2.3 Restore from file

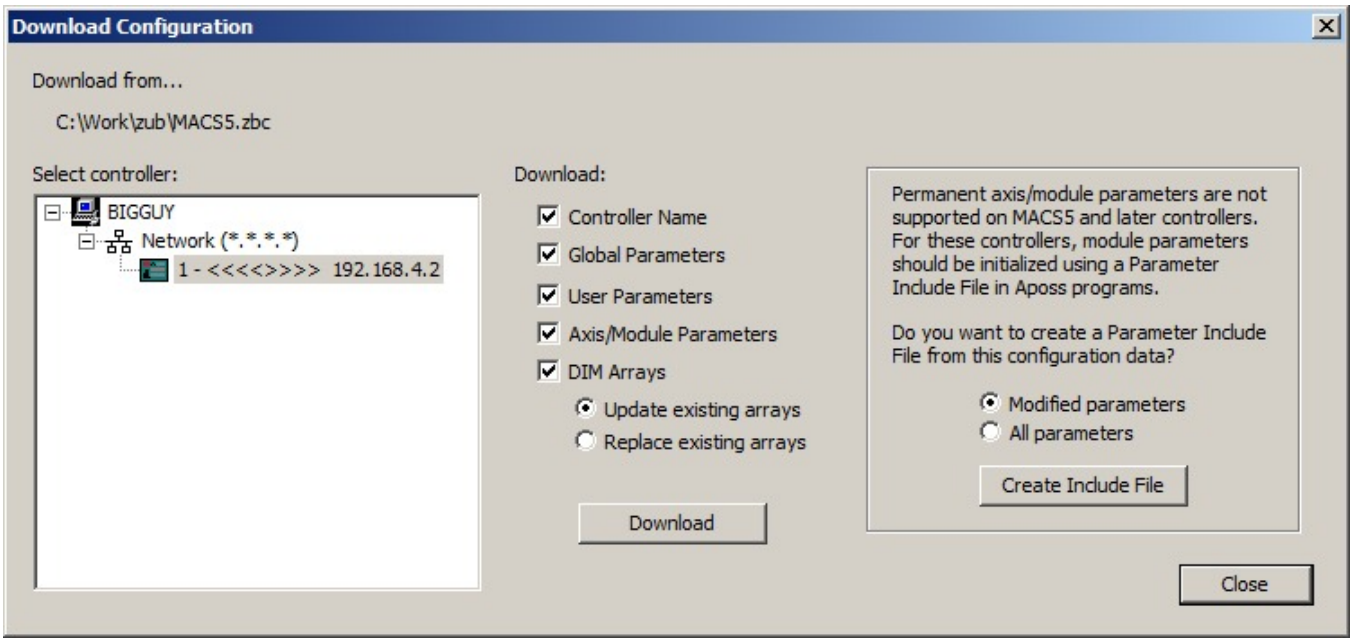
If a controller's configuration has been backed up as a file on the PC, then the configuration can be restored to the controller again by downloading the configuration file.

Click on **Parameters** → **Restore from file**. This will display an "Open" dialog. Select the file containing the data to be loaded.

In the subsequent dialog, select the controller into which the data should be loaded.

Use the checkboxes to specify which sets of parameters are to be loaded. Then click the **Download** button. The specified parameters are then loaded into the controller. Any previous parameter values are overwritten.

If the *same* data is to be loaded into more than one controller, then simply select another controller and click **Download** again.



DIM Arrays

If the configuration being downloaded contains DIM arrays, then the following options are available:

Update existing arrays Updating arrays does the following:

- If the DIM array being downloaded corresponds to a DIM array that already exists on the controller and the new array is *shorter* than the existing array, then the extra elements in the existing array will be set to 0.
- If the array being downloaded corresponds to a existing array and the new array is *longer*, then the user will be prompted whether or not to replace the existing array.
- If the array being downloaded is a new array that does not yet exist on the controller, then the new array will be created on the controller.

Replace existing arrays All previously existing DIM arrays on the controller are erased and replaced with the DIM arrays being

downloaded.

> **MACS5 and Newer Controllers**
>
> For MACS5 and newer controllers, axis parameters are *not* maintained permanently in flash memory. Any permanent or temporary axis parameters saved in the file will *not* be restored to the controller. For these controllers, **Create Include File** can be used to create an include file that can be used with application programs. The values in the include file will be obtained from the values saved in the file. Please see Initializing Temporary Parameters for a more detailed description of axis parameter handling in these controllers.

3.10.2.4 Save ALL to files

If there are multiple controllers on a single connection interface (e.g. a CAN network), then this function will allow you to simultaneously save the configuration information from all controllers on the network into files. Click on **Parameters** → **Save ALL to files**. In the subsequent "Save As" dialog, specify the "base" name of the files that you want to save the configuration into. One configuration file will be created for each controller. The files will be named "base-id.zbc" where "base" is the name specified above, "id" is the controller ID number (e.g. CAN ID), and "zbc" is the file extension. All files will have "zbc" as the file extension.

3.10.3 Reset

Arrays

The **Arrays** function will delete all arrays in permanent memory.

All Parameters

The **All Parameters** function will reset all global, axis, and user parameters in permanent memory to their initialization values.

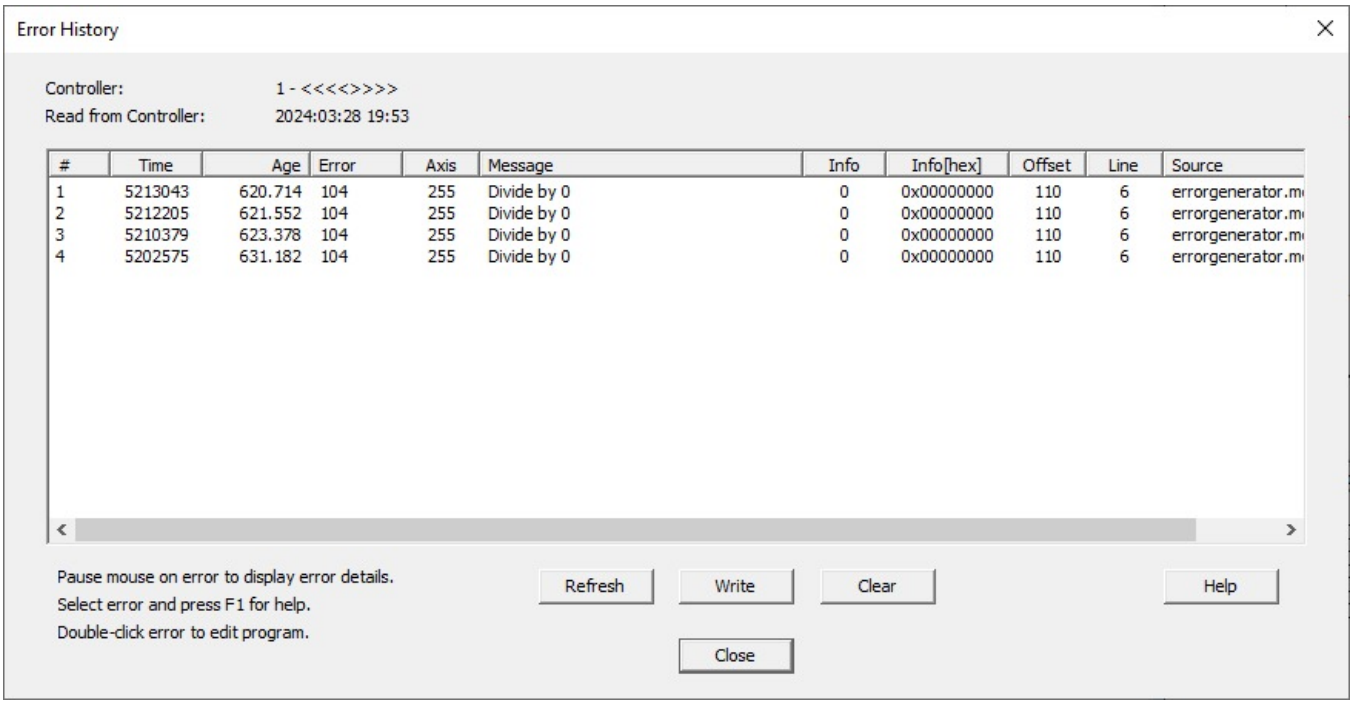
Factory Settings

The **Factory Settings** function will reset all parameters and delete all arrays and programs. The controller is reset to the original factory setting.

3.10.4 Error History

Click on **Controller** → **Error History** to see a history of all the errors that have occurred on the controller.

> Some older versions of controller do not support this function. The function will be disabled in this case.



Errors are listed from most recent at the top of the list to oldest at the bottom of the list. The list is cleared each time that the controller is powered on. The list holds a maximum of 50 errors; the oldest error will be discarded when a new error is added to the list.

Selecting an error in the list and pressing **F1** will display help for that specific error.

Double-clicking on an error in the list will close the Error History dialog and highlight this line in the Edit Window.

> ⚠ The ApossIDE cannot verify that the program currently being edited matches the program executing in the controller at the time the error occurred. This is the user's responsibility. If the programs do not match, then the wrong line will be highlighted in the Edit Window.

The following information is listed:

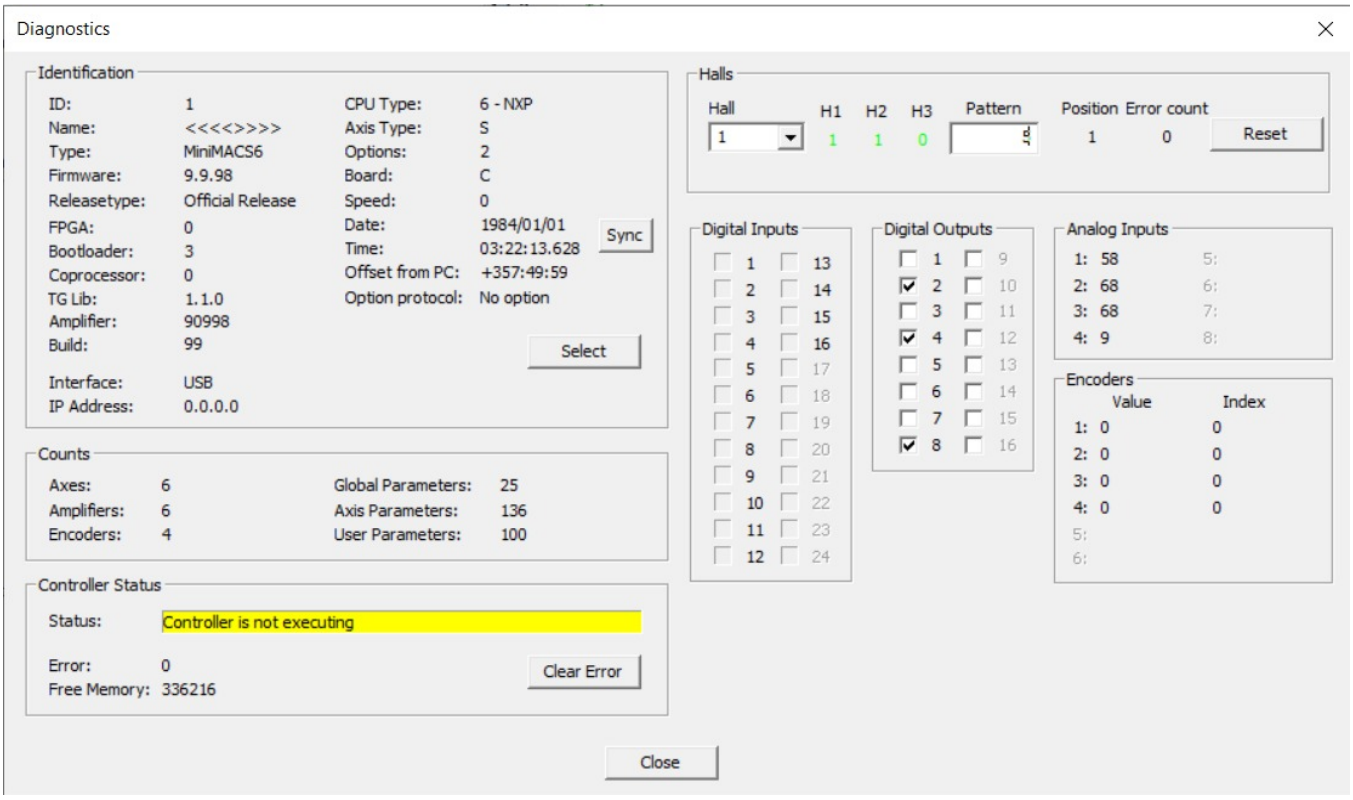
Time	Controller system time when error occurred. See "sys_clock" in System Process Data.
Age	Age (in seconds) of the error when the Error History dialog was opened (i.e. how long ago the error occurred). This will be updated each time the dialog is refreshed.
Error	Error number. See Error Reference.
Message	Error message text.
Info	Optional info value saved with error. The meaning of this value depends on the error.
Info[hex]	The optional info value displayed as a hexadecimal number.
Offset	Binary offset within the compiled program where the error occurred.
Line	Line number within the program being edited where the error occurred.
Source	Name of the program being edited where the error occurred.

The following functions are available:

Refresh	Update the list with the current error history from the controller. Note that this will update the “age” of the existing errors in the history.
Write	Write the error history into an ASCII text file so that the errors can be analyzed at a later time. This file can be edited by any text editor or imported into a spreadsheet application.
Clear	Clear the error history that is stored on the controller. This is a pemanent action and it cannot be undone.

3.10.5 Diagnostics

Click on **Controller** → **Diagnostics** to display a window showing diagnostic information from the connected controller.



The labels in the “Digital Inputs”, “Digital Outputs”, “Analog Inputs” and "Halls" boxes will be grayed for inputs and outputs that do not correspond to physical pins on the controller.

> In some cases, digital outputs that do not have a physical pin, can still be used as a “virtual” digital output.

The following functions are available:

Sync

For controllers that support a real time clock, the "Sync" button can be used to synchronize the real time clock with the time on the PC.

Select

Select a different controller. Diagnostic information from the new controller will then be displayed.

Clear Error

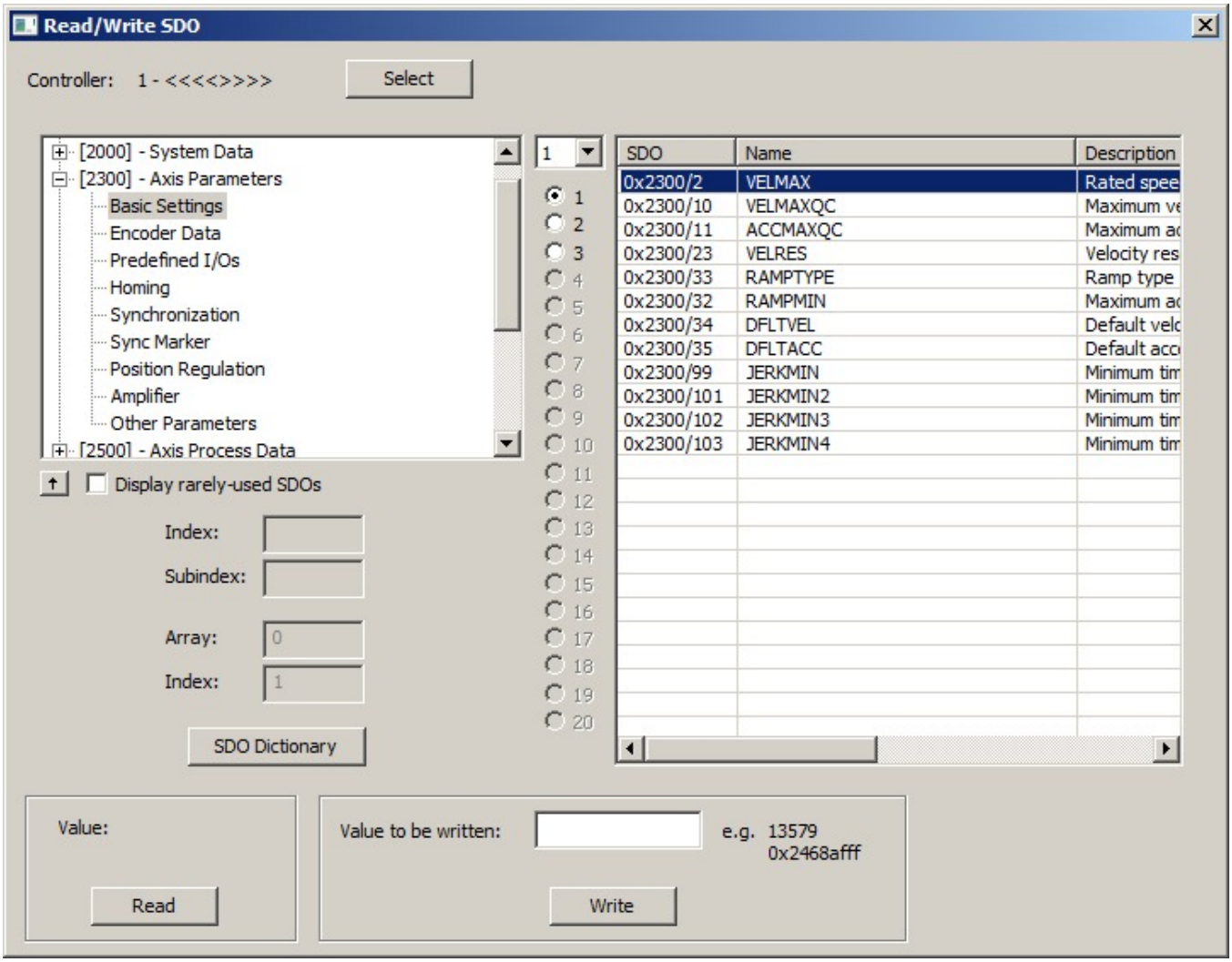
Send a "break" command (i.e. **Esc**) to the controller in order to clear any existing controller error.

> ⚠ If a program is executing, then this will stop the program.

3.10.6 Read/Write SDO

Click on **Controller** → **Read/Write SDO** to display the following window allowing SDO values to be read from the controller and/or written to the controller.

> This window can be left open while continuing to work with other windows. As well, multiple instances of this window may be opened, each connected to the same or a different controller.



Selecting an SDO

1. If the left and right SDO selection windows are not enabled (grey), then click on either of the windows to enable them.
2. Select the SDO group in the left window.
3. If the SDO corresponds to an object for which there are multiple instances (e.g. axes, encoders, etc.), then select the instance number using the radio buttons or list box in the center of the dialog.
4. Select the SDO subindex in the right window. The **Read** and/or **Write** button will be enabled if the selected SDO can be read or written.

   > The dialog can be resized to show more information in the right window.

5. The **SDO Dictionary** button will open a help window containing a description of all available SDO values.

Or if the SDO index and subindex numbers are known...

1. Click on either the **Index** or **Subindex** field to enable these fields.
2. Enter the index and subindex values. Both of these values may be entered as either integer or hexadecimal numbers.
3. Basic range validation is done of the index and subindex values. If the values appear to be valid, then the **Read** and **Write** buttons are enabled.

Or if a DIM array value is to be read or written...

1. Click on either the **Array** or **Index** field to enable these fields.
2. Enter the array number and array index.
3. Basic range validation is done of the array number and index values. If the values appear to be valid, then the **Read** and **Write** buttons are enabled. Note that only array numbers 0-254 and indices 1-250 are supported.

Reading SDO values

1. Select the SDO the be read (see above).
2. Press the **Read** button.

Writing SDO values

1. Select the SDO to be written (see above).
2. Enter the value to be written. This can be either an integer value or a hexadecimal value.
3. Press the **Write** button.

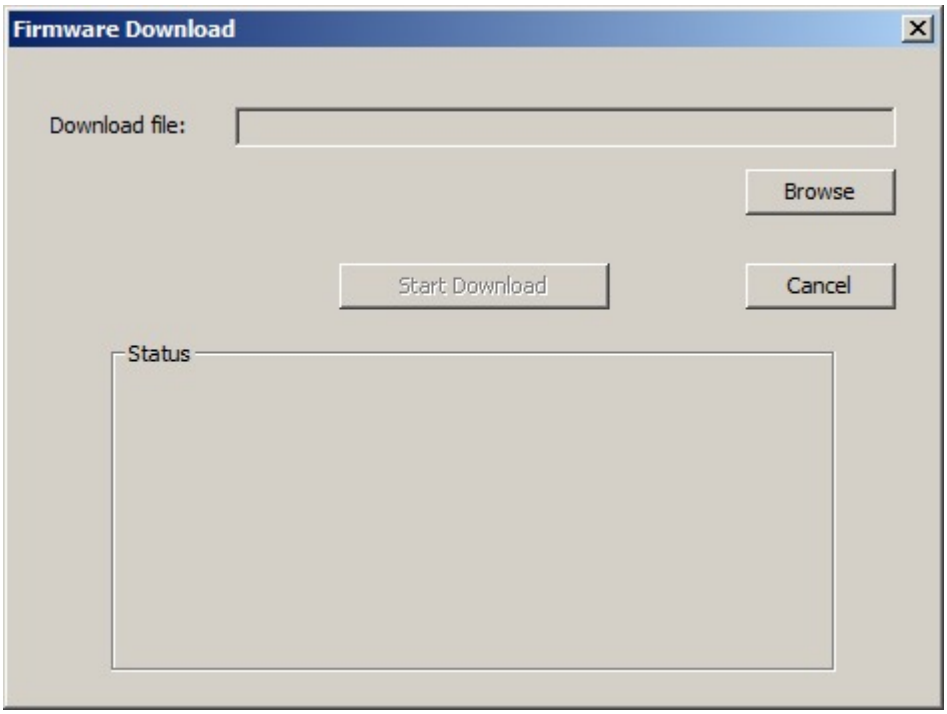
Selecting a different controller

1. Press the **Select** button.
2. Select a new controller from the controller selection dialog.

3.10.7 Download Firmware

Click on **Controller** → **Download Firmware** to download new firmware to the controller.

> This command may only be used when the ApossIDE window is the only window connected to the controller. For example, an Oscilloscope window must not be simultaneously connected to the same controller. This is necessary since the controller will be rebooted as part of the firmware download procedure. The reboot would disturb other windows that are not expecting the controller to be rebooted while they are using it.



To download firmware:

1. Press the **Browse** button and select the firmware file to be downloaded. It is also possible to use drag-and-drop to drag a firmware file to the "Download file" field.
2. Press the **Start Download** button.
3. Once the download starts, please wait for it to complete. The download should not be stopped.

3.10.8 State Machine

Configure History

The **Configure History** function allows the user to enable and disable the recording of state machine events as they are being processed by a running state machine program. This provides a valuable tool for users to use while debugging their state machine programs. See User Configuration of History for more information. Enabling and disabling can also be done within an application program. See Program Configuration of History for more information.

Upload History

The **Upload History** function allows the user to upload and view any recorded state machine history. See Viewing State Machine History for more information.

3.11 Tools Menu

The Tools menu offers the following tools:

- CAM Editor
- Array Editor
- Oscilloscope
- Log Mode
- Current Regulator Calculator

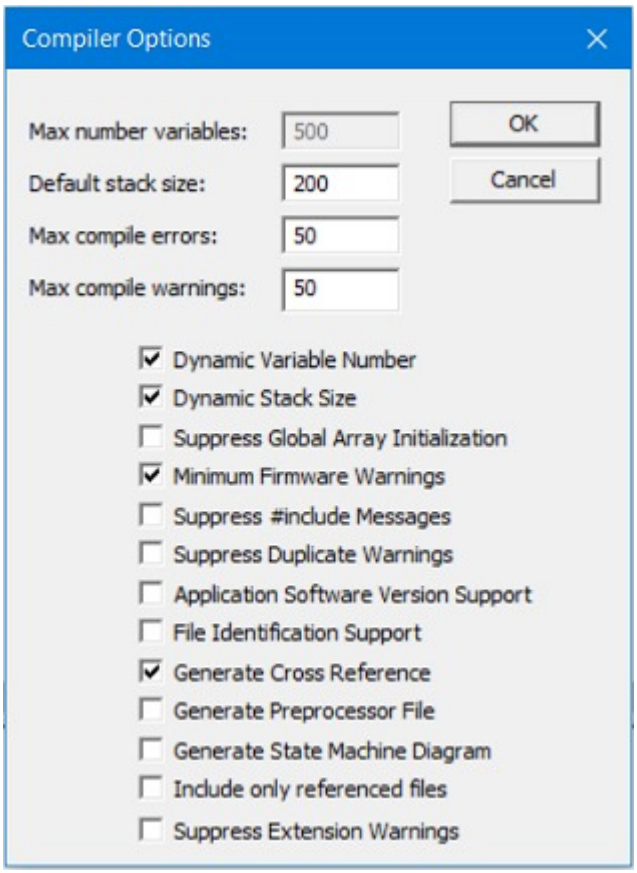
3.12 Settings Menu

This menu offers various options and settings.

Compiler	Options that control the behavior of the ApossIDE compiler.
Interface	Options that control the behavior of the various controller connection interfaces.
Editor	Options that control the behavior of the ApossIDE text editor.
Module Editor	Options that control the behavior of the Module Editor tool.
Options	Options that control the general behavior of the ApossIDE.
Oscilloscope	Options that control the behavior of the Oscilloscope tool.

3.12.1 Compiler

The default values for the compiler options are set appropriately for most applications.



Max number variables

The Maximum number of variables specifies how much memory must be reserved for global variables and arrays. This has a direct effect on the amount of free memory available to the application in the controller.

The compiler can automatically calculate the size of global memory required (see "Dynamic Variable

Number" below). If automatic calculation is disabled, then this value is used.

> The minimum required number of variables can be calculated on the basis of the application program being compiled. This will be *at least* the total of *all* of the following sizes:
>
> - 1 for each global **long** and **wchar** variable.
> - 2 for each global **double** variable.
> - 48 for each global **transform** valiable.
> - 20 for each global **dpoint** variable.
> - 1 for each array (including both DIM arrays and global arrays).
> - 1 for each element in each global **long** array (excluding DIM arrays).
> - 1/2 for each element in each global **wchar** array (excluding DIM arrays).
> - 2 for each element in each global **double** array (excluding DIM arrays).

An insufficient number of variables is detected during compilation of the program before the program is downloaded into the controller. A compiler error message is generated.

Default stack size

Stack size specifies how much memory must be reserved for the internal handling of function calls and their local data. If deeply nested function calls or a large number of local variables or local arrays are used, then a larger stack size is required.

For ApossC programs, the compiler can automatically calculate the amount of stack size required (see "Dynamic Stack Size" below). However, if the application program contains recursive function calls (or an application program is being compiled), then the compiler is unable to calculate the required stack size. In this case, the default stack size specified here is used.

> It is *highly* recommended that applications be developed *without* using recursive function calls. If at all possible, users should enable "Dynamic Stack Size" and rely on the compiler doing the calculation for the required stack size.

> The required stack size can be estimated on the basis of the program code and the logical program flow:
>
> - For each possible chain of function calls starting at "main()", calculate the following:
> - Total the memory required for all local variables and arrays in each function in the chain. This can be calculated using the description above (using local variables and arrays rather than global ones).
> - Add 3 for every function in the chain.
>
> Determine the maximum memory required by any of the chains.
> - For each possible chain of function calls starting at each non-ERROR interrupt handler routine, calculate (as above) the maximum memory required.
> - For each possible chain of function calls starting at each ERROR interrupt handler routine, calculate (as above) the maximum memory required.
> - Add the above three numbers and add a reserve/overhead (e.g. 100).
>
> This will be *at least* the total of *all* of the following sizes.

> An insufficient stack size can only be detected during program execution. If the stack space is exhausted while an application is running, then the application program will be terminated and an Error 93 is reported. ***This is a FATAL error.*** Please see Error 93 for details.

Max compile errors

The compile will often generate multiple compiler error messages. In certain cases when there is a syntax error in the application program being compiled, the compiler will be unable to reestablish a correct "starting" position following the syntax error. This can sometimes result in a large number of spurious error messages. The maximum compile errors value allows the user to limit the number of error messages displayed to some reasonable number.

Max compile warnings

The compile will often generate multiple compiler warning messages. The maximum compile warnings value allows the user to limit the number of warning messages displayed to some reasonable number.

Dynamic Variable Number

If this is enabled, then the compiler will automatically calculate the maximum number of variables required by the program and allocate memory accordingly. The program is not limited by the "Max variable number" value specified above. Note that there is an absolute limit of 65535 to the maximum number of variables.

Dynamic Stack Size

If this is enabled and an ApossC program is being compiled, then the compiler will automatically calculate the stack size required by the program and allocate memory accordingly. The program is not limited by the "Default stack size" value specified above. Note that there is an absolute limit of 65535 to the maximum stack size.

If the program being compiled contains recursive function calls, then the compiler is unable to determine the required stack size. In this case, the compiler will use the "Default stack size" value. The user will then be responsible for determining an appropriate stack size, depending on the known architecture and expected operation of the application program. See above for information about estimating stack size.

Suppress Global Array Initialization

ApossC programs will automatically initialize all global arrays to 0 during program startup. For ApossC programs that declare very large arrays, this may noticeably slow down program startup. If this is an issue, then the initialization of global arrays may be suppressed using this setting.

Minimum Firmware Warnings

The compiler will generate warning messages if an application program uses features that are not supported by the version of firmware currently installed on the connected controller. These warnnings can be enabled or disabled using this option.

Suppress #include Messages

The compiler will generate a list of each #include file encountered during a compile. This list may be useful for verifying that the correct include files have been found. If this list is not needed, then this option can be used to prevent the list from being generated.

Suppress Duplicate Warnings

In certain cases (e.g. when compiling old application programs), the compiler may generate many of the same warning messages if a particular obsolete programming construct has been used many times. Enabling this option will cause the compiler to limit the number of duplicate warning

messages to four of each message type. This will allow the user to avoid generating multiple warning messages without needing to limit all warning messages.

Application Software Version Support

"Application Software Versions" provide users with an additional mechanism to identify their application programs. They consist of 3 "long" values that can be set by the user and that are permanently compiled into the application program code that is downloaded to controllers. It is possible for external programs to use SDO 0x2209, subindices 4, 8, and 9, to extract these values from a running controller and then use them to identify loaded programs.

> Controllers must have firmware version 7.1.31 or newer to support reading these SDO values. Older firmware will return either values of 0 or an "Invalid SDO" error.

The ApossIDE itself assigns no meaning to these values; the meaning is entirely determined by the user. While they are not a secure a method of identification (since multiple programs could be assigned the same version values), they are "human-readable" and can designed to provide meaningful information to users.

The use of Application Software Versions is optional in the ApossIDE and will be disabled by default. Note that the setting will affect all application programs; the setting cannot be used on an individual per-program basis.

The versions of existing programs loaded into a controller can be seen by connecting to the controller, executing **Controller** → **Programs**, selecting one of the loaded programs, and then pressing the "Info" button.

Using "Application Software Versions" in Application Programs

Application Software Version numbers are set within the application program source (.m) file by the programmer himself (and must also be changed/managed by the programmer when necessary). They are set using the following #define statements:

```
#define APP_SW_VERSION version1
#define APP_SW_VERSION_1 version1
#define APP_SW_VERSION_2 version2
#define APP_SW_VERSION_3 version3
```

where "version1", "version2", and "version3" are the three version numbers. The following restrictions apply when using these versions:

1. The version numbers can be specified either as decimal numbers (e.g. 123) or as hex numbers (e.g. 0x0102). Only numbers are supported, so strings such as "v1.0" cannot be used.
2. If the values are defined using nested #define statements, then the final processed expansion of the #define must result in a simple number. Arithmetic expressions such as

   ```
   #define APP_SW_VERSION (MAJOR*10000 + MINOR*100 + MICRO)
   ```

   are not allowed since they are not evaluated at the time the #define is processed.
3. APP_SW_VERSION and APP_SW_VERSION_1 are synonyms for the first version number. Either one or the other should be used but not both. If both are used, then both must have the same value.
4. The first version number (i.e. APP_SW_VERSION and APP_SW_VERSION_1) is mandatory and cannot be set to 0. The other version numbers are optional and can be set to 0 if desired.
5. All versions numbers that are not defined in the application program, will be set to 0 in the compiled code.

6. These are normal #define statements. They can be placed anywhere within the program source (or include files) and the values can be used in the same ways as any other #define.
7. The version numbers that are compiled into the binary program code, are only evaluated at the end of the program source. This means that at least the APP_SW_VERSION or APP_SW_VERSION_1 definition must still exist since the first version is mandatory. The #undef statement must not be used to remove the definition before the program source ends.

> Starting with Firmware Version 7.1.31:
>
> If the compiler option "Support Application Version" is enabled and the program contains the #define statements, then the firmware now copies that information automatically into the SDO entries 0x2209/4, 0x2209/8 and 0x2209/9. This information is also used and returned by the corresponding entries of 0x1F56.

If Application Software Version Support is not enabled in the ApossIDE, then these #defines (if used) behave exactly like other #defines. There are no restrictions. All three version numbers will be set to 0 in the compiled code even if these #defines are used.

File Identification Support

Applications sometimes require assurance that the application program running in the controller is identical to a "master" program. For example, a specific version of a program may have been "certified" in some way. It is then important that all controllers are running that exact version. Any modification to the program will cause it to lose its certification.

Enabling "File Identification Support" provides maintenance personnel with the ability to verify whether or not programs running on a controller have been modified. When this option is enabled, then identification signatures are included in application programs and downloaded to the controller. These signatures can be viewed using **Development** → **Compare to Controller** or written to a diagnostics report using **Controller** → **Diagnostics Report**.

"File Identification Support" may be enabled on a per-program basis by including #pragma FILE_IDENT in the program.

For more information about identification signatures, see Compare to Controller.

Generate Cross Reference

Cross reference files are not necessary for the proper functioning of the ApossIDE and the user may safely disable this option. These files are normally of interest only to zub machine control AG personnel when debugging ApossIDE compiler issues.

Generate Preprocessor File

All application program files are preprocessed (see Preprocessor) before being compiled. This will make various string substitutions in the original program. If the user needs to see exactly what the compiler is compiling, then this option can be enabled. Enabling the option will cause the compiler to generate a ".txt" file with the same name and in the same directory as the original file. This file will contain the output from the preprocessor. The file will be created each time that the user compiles the program.

Generate State Machine Diagram

If an application program contains a state machine, then sometimes it can be useful to generate a state machine diagram corresponding to the application program. Enabling this option will cause the compiler to generate a ".gml" file that can be imported into the yEd graphical editor. For more

information about the yEd editor, see **www.yworks.com**.

Include only referenced files

If this is enabled and an AposssC program is being compiled, then the compiler will use only the include files whose content is being used in the program. That means that at least one function, one variable or one other identifier form the include file is being referenced in the program.

If certain include files will not being included that means their content will also not be present in the final binary code and that will make the binary code smaller.

Supress Extension Warnings

The compiler will generate a warning if file with extension .mh or .mi is included. Sometimes when there are a lot of source files used in the project, it may be desirable that some of the source files also include source files and not only header files and in this case this option can be used to prevent the warning from being generated.

3.12.2 Interface Parameters

Normally, connection interface parameters will have been set correctly when the ApossIDE is first installed and used (see Connecting to the controller). However, if they need to be changed for any reason (e.g. the baud rate or COM port has changed, the controller ID scan range needs to be extended, a new interface type has been added to the system, etc.), then **Settings** → **Interface** can be used to update the parameters. Simply select the appropriate interface type and change the necessary parameters.

> Only one interface type can be selected as the default interface. The default interface is used to establish connections to controllers when the **Esc** key is pressed.

> Setting the interface parameters will affect only subsequent connections to controllers. Any controller to which there is already an established connection, will remain connected and will not be changed. If an established connection to a controller needs to be changed, then it will be necessary to close the connection to that controller, change the interface settings, and then re-connect to the controller.

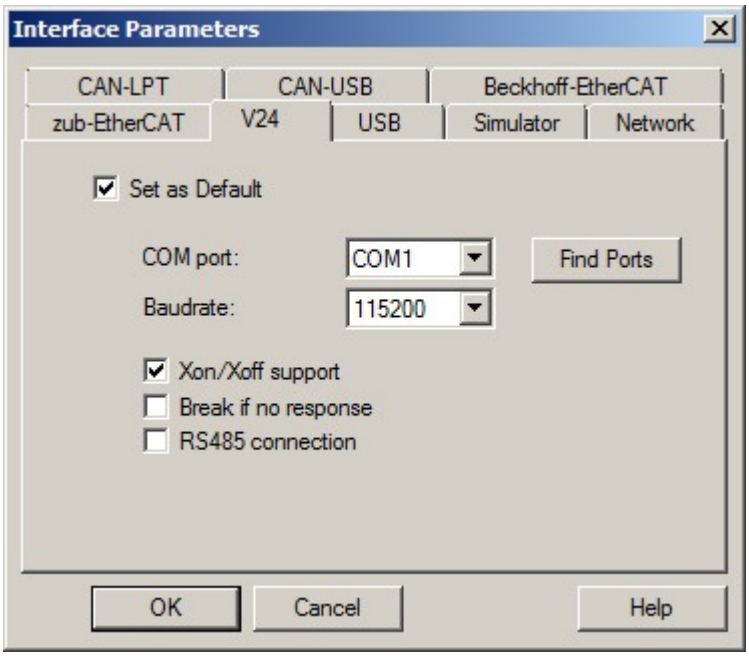
Currently Available Interfaces

V24	V24 connections are serial connections that use either an RS232 COM port or a USB port with a USB-to-RS232 converter.
CAN-LPT	CAN-LPT connections use a CAN-LPT adapter to connect the computer's parallel printer port (LPT port) to a CAN network.
CAN-USB	CAN-USB connections use a CAN-USB stick to connect a USB port to a CAN network.
USB	If the controller supports it, then a USB connection can be made directly from the computer to the controller.
EtherCAT (Beckhoff)	If the controller supports it, then an EtherCAT connection can be made from the computer to the controller using Beckhoff hardware.
EtherCAT (zub)	If the controller supports it, then an EtherCAT connection can be made from the computer to the controller using generic

	EtherCAT drivers.
Network	If the controller supports it, then an Ethernet TCP/IP connection can be made directly from the computer to the controller.
Simulator	The ApossIDE provides a Simulator that can simulate a controller in software without the need of a physical controller. In certain types of applications and at certain stages of application development, the Simulator can be a valuable debugging tool. See Simulator for more information on using a Simulator.

3.12.2.1 V24 Parameters

V24 connections are serial connections that use either an RS232 COM port or a USB port with a USB-to-RS232 converter.



Set as Default

Enabling **Set as Default** will select this interface as the default interface to be used for connections when the **Esc** key is pressed.

COM port

The COM port being using. If the desired COM port is not listed, then click the **Find Ports** button and check again for the COM port. If a USB-to-RS232 converter is being used, then it will be necessary to plug the converter into the computer before Windows will be able to find the associated COM port.

> If a USB-to-RS232 converter is being used, then the correct virtual COM port must be selected. The correct COM port can be identified by opening the Windows **Control Panel**, selecting **System**, selecting **Hardware**, clicking the **Device Manager** button, and finally opening the **Ports (COM & LPT)** item. Look for the COM port that corresponds to the USB-to-RS232 device.

Baudrate

The baudrate that the connection will use. Note that some older controllers support only 9600 baud. For these controllers, the baud rate setting is ignored.

Xon/Xoff support

Xon/Xoff support should normally be enabled. It should be disabled if a USB-to-RS232 converter is being used.

Break if no response

Break if no response should be disabled unless it is necessary to enable it. The ApossIDE cannot successfully establish a connection to some older controllers if the controller is executing a program. If the ApossIDE returns the message "No Controller found!" when an old controller is properly connected to the correct COM port, then you should enable the **Break if no response** setting and try again. Note that this will abort the program executing on the controller!

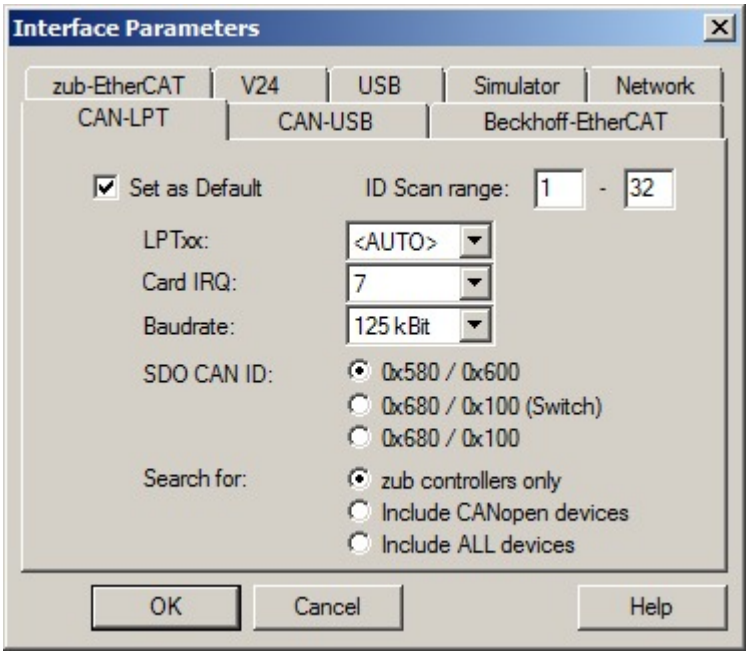
RS485 connection

RS485 connection should be enabled if an RS232-to-RS485 converter has been inserted into the connection. This allows the ApossIDE to adjust the communication timing to compensate for differences between RS232 and RS485.

3.12.2.2 CAN-LPT Parameters

CAN-LPT connections will use a CAN-LPT adapter to connect the computer's parallel printer port (LPT port) to a CAN network.

CAN-LPT adapters *are not supported* on Microsoft Windows 7 and later systems.



Set as Default

Enabling **Set as Default** will select this interface as the default interface to be used for connections when the **Esc** key is pressed.

ID Scan range

The **ID Scan range** specifies the range of CAN ID numbers that the ApossIDE will use when searching the CAN network for controllers. This range must include the CAN ID of the controller to which the connection is being made.

LPTxx

The **LPT port** being used. This can normally be left set to <AUTO>. It will be necessary to set this to a specific LPT port if multiple CAN-LPT modules are being used.

Card IRQ

Set the **Card IRQ** according to the manufacturer's installation documentation that came with the CAN-LPT module.

Baudrate

The baud rate of the CAN network.

SDO CAN ID

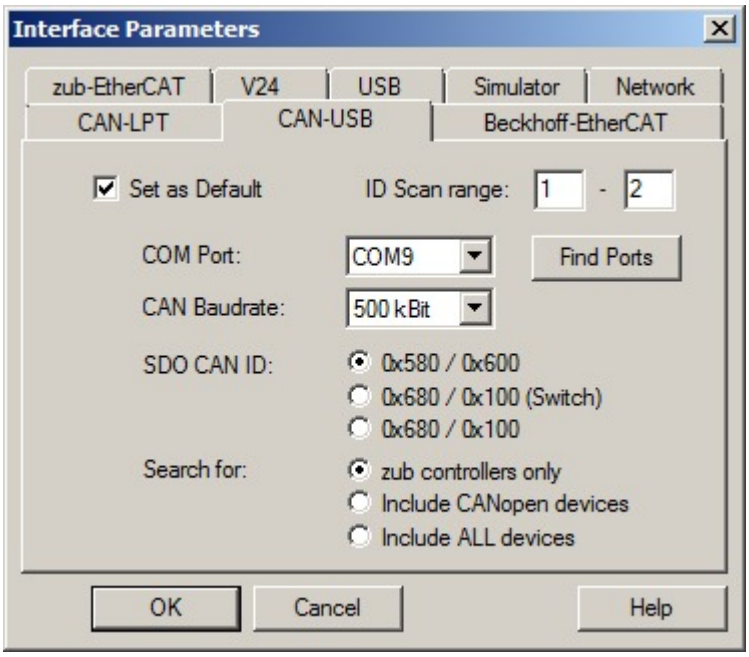
The **SDO CAN ID** should be left set to “0x580 / 0x600” unless there is a reason to change it. These are the CAN ID numbers that will be used to identify SDO messages sent to and received from the controller. Some controller applications may use these ID numbers for other purposes. In these cases, the alternate ID numbers may be used. Note that some older controllers may not support the alternate ID numbers. If the option labeled “0x680 / 0x100 (Switch)” is selected, then the initial connection will be made using “0x580 / 0x600”. If this connection is successful, then the ApossIDE will automatically switch to the alternate ID numbers “0x680 / 0x100”. If this switch fails, then the ApossIDE will continue to use the original ID numbers.

Search for

Search for can be left set to “zub controllers only”. The ApossIDE is capable of connecting to other types of CAN devices, not only to zub controllers. In these cases, the ApossIDE will have limited functionality.

3.12.2.3 CAN-USB Parameters

CAN-USB connections will use a CAN-USB stick to connect a USB port to a CAN network.



Set as Default

Enabling **Set as Default** will select this interface as the default interface to be used for connections when the **Esc** key is pressed.

ID Scan range

The **ID Scan range** specifies the range of CAN ID numbers that the ApossIDE will use when searching the CAN network for controllers. This range must include the CAN ID of the controller to which the connection is being made.

COM port

The COM port being using. If the desired COM port is not listed, then click the **Find Ports** button and check again for the COM port. It will be necessary to plug the CAN-USB stick into the computer before Windows will be able to find the associated COM port.

> The CAN-USB stick uses a virtual COM port. The correct COM port can be identified by opening the Windows **Control Panel**, selecting **System**, selecting **Hardware**, clicking the **Device Manager** button, and finally opening the **Ports (COM & LPT)** item. Look for the COM port that corresponds to the CAN-USB stick.

CAN Baudrate

The baud rate of the CAN network.

SDO CAN ID

The **SDO CAN ID** should be left set to “0x580 / 0x600” unless there is a reason to change it. These are the CAN ID numbers that will be used to identify SDO messages sent to and received from the controller. Some controller applications may use these ID numbers for other purposes. In these cases, the alternate ID numbers may be used. Note that some older controllers may not support the alternate ID numbers. If the option labeled “0x680 / 0x100 (Switch)” is selected, then the initial connection will be made using “0x580 / 0x600”. If this connection is successful, then the ApossIDE will automatically switch to the alternate

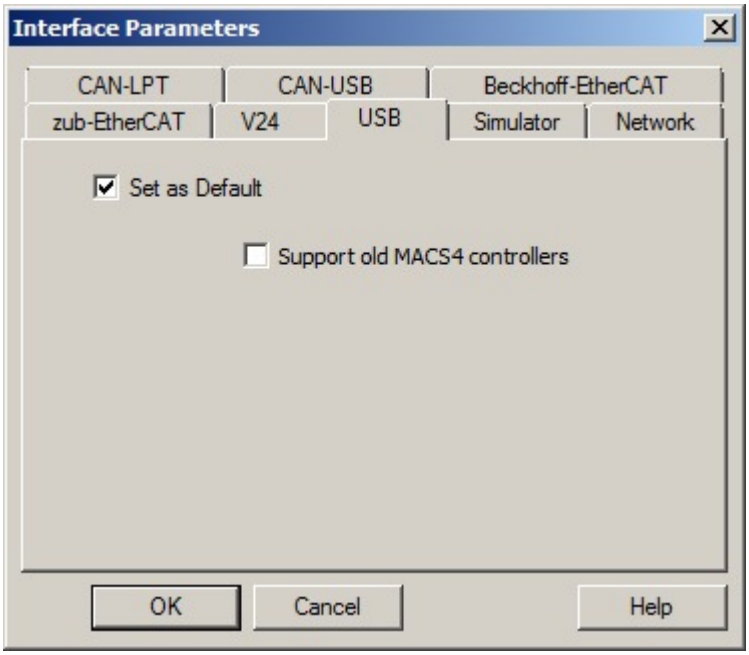
ID numbers “0x680 / 0x100”. If this switch fails, then the ApossIDE will continue to use the original ID numbers.

Search for

Search for can be left set to “zub controllers only”. The ApossIDE is capable of connecting to other types of CAN devices, not only to zub controllers. In these cases, the ApossIDE will have limited functionality.

3.12.2.4 USB Parameters

If the controller supports it, then a USB connection can be made directly from the computer to the controller.



Set as Default

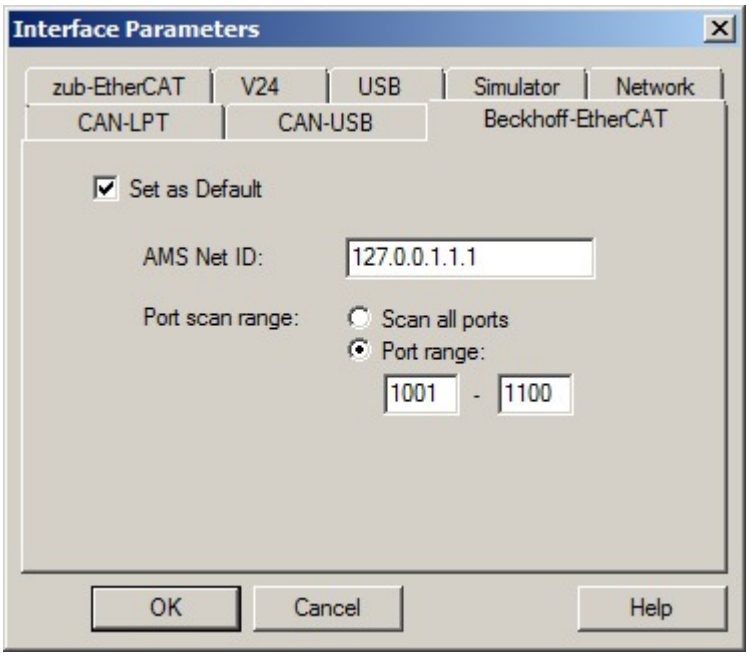
Enabling **Set as Default** will select this interface as the default interface to be used for connections when the **Esc** key is pressed.

Support old MACS4 controllers

Some very old MACS4 controllers contain old USB connection characteristics that are no longer used. Enabling this option will allow the ApossIDE to identify and connect to these old MACS4 controllers. Note that this may cause the ApossIDE to mistakenly identify some other USB devices as zub controllers. In this case, the ApossIDE will attempt to communicate with the device, possibly disrupting the device. Hence, this option should only be enabled if it is known that one of these old MACS4 controllers is being used.

3.12.2.5 EtherCAT Parameters (Beckhoff)

If the controller supports EtherCAT connections, then a connection can be made from the computer to the controller using either a Beckhoff EtherCAT Master or a zub EtherCAT Master. The following information applies to a Beckhoff EtherCAT Master. If you will be using a zub EtherCAT Master, then please see EtherCAT Parameters (zub).



Set as Default

Enabling **Set as Default** will select this interface as the default interface to be used for connections when the **Esc** key is pressed.

AMS Net ID

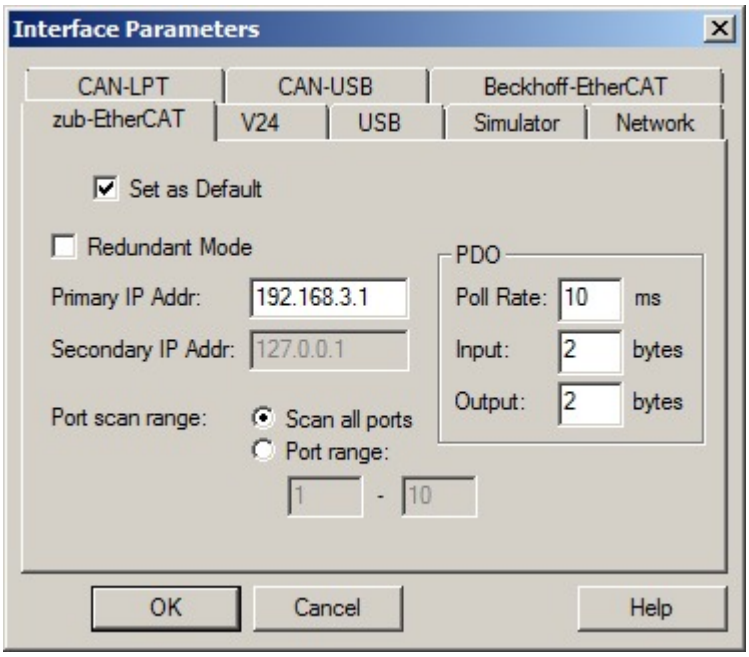
The ID of the EtherCAT network controller. This will be an address of six numbers separated by dots (e.g. 5.1.131.8.3.1).

Port scan range

The range of controller port numbers that will be scanned when searching for controllers. This can normally be left set to all ports unless there is some reason to restrict it.

3.12.2.6 EtherCAT Parameters (zub)

If the controller supports EtherCAT connections, then a connection can be made from the computer to the controller using either a Beckhoff EtherCAT Master or a zub EtherCAT Master. The following information applies to a zub EtherCAT Master. If you will be using a Beckhoff EtherCAT Master, then please see EtherCAT Parameters (Beckhoff).



Set as Default

Enabling **Set as Default** will select this interface as the default interface to be used for connections when the **Esc** key is pressed.

Redundant Mode

If EtherCAT "redundant mode" is being used, then enable this option. Redundant mode requires two Network Interface Cards (NIC) in the PC. The final EtherCAT controller in the chain will be connected back to the second NIC.

Primary AP Addr

This will be the IP address of the NIC that will be used to connect to the controller. This should be four numbers separated by dots (e.g. 192.168.3.1).

Secondary IP Addr

If EtherCAT "redundant mode" is being used, then this will be the IP address of the second NIC that terminates the controller chain. This should be four numbers separated by dots (e.g. 192.168.4.1).

Port scan range

The range of controller port numbers that will be scanned when searching for controllers. This can normally be left set to all ports unless there is some reason to restrict it.

PDO Poll Rate

The EtherCAT Master will poll all controllers for PDO messages at this rate.

PDO Input

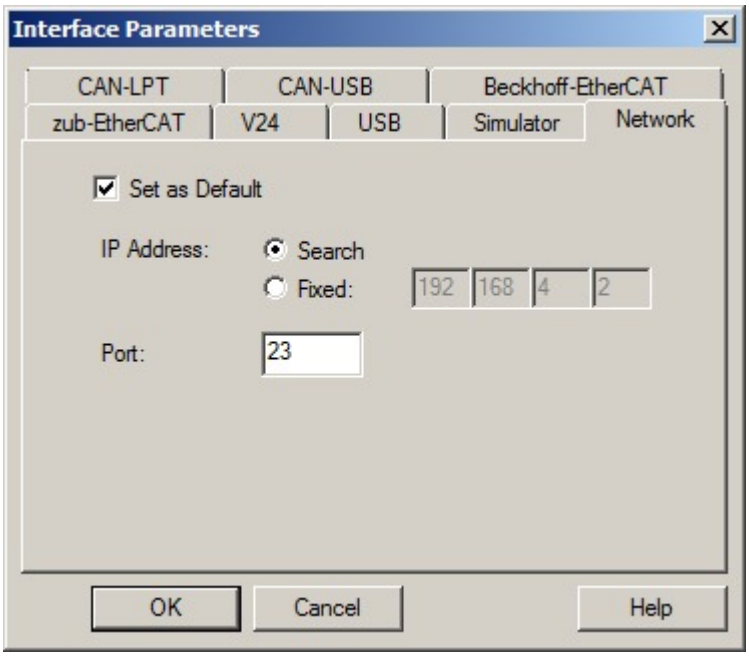
The number of bytes to be allocated for the PDO input buffer for each controller.

PDO Output

The number of bytes to be allocated for the PDO output buffer for each controller.

3.12.2.7 Network Parameters

If the controller supports it, then an Ethernet TCP/IP connection can be made directly from the computer to the controller.



Set as Default

Enabling **Set as Default** will select this interface as the default interface to be used for connections when the **Esc** key is pressed.

IP Address

If **Search** is enabled, then the ApossIDE will try to locate all Ethernet controllers that are accessible locally to the computer. The ApossIDE will not search the Internet. If **Fixed** is enabled, then the IP address of a specific controller must be given. This IP address can be a remote address on the Internet. Note that the computer firewall must be set to allow access to the controller and Internet response times must be fast enough to allow the ApossIDE to maintain a connection.

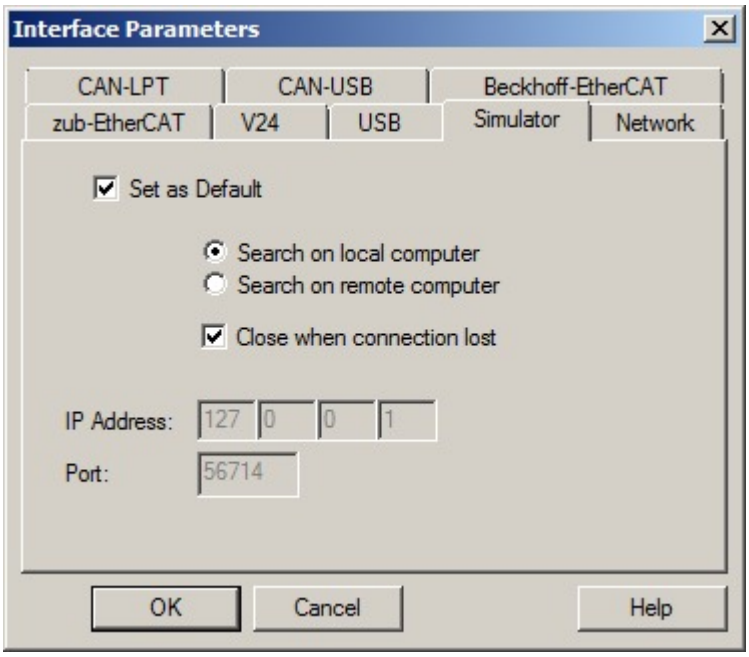
> If the controller firmware is older than 7.1.16, then the **Fixed** option must be selected and the IP address of the controller specified.

Port

The TCP/IP port to be used. This must be 23 unless there is some specific reason to change it.

3.12.2.8 Simulator Parameters

The ApossIDE provides a Simulator that can simulate a controller in software without the need of a physical controller. In certain types of applications and at certain stages of application development, the Simulator can be a valuable debugging tool. See Simulator for more information on using a Simulator.



Set as Default

Enabling **Set as Default** will select this interface as the default interface to be used for connections when the **Esc** key is pressed.

Search on local computer

If **Search on local computer** is enabled, then the ApossIDE will expect the Simulator to be already executing on the local computer (i.e. the same computer executing the ApossIDE). Otherwise, **Search on remote computer** must be enabled and the ApossIDE will expect the Simulator to be already executing on a different computer on the network. If a remote computer is being used, then an IP address and port must be specified.

Close when connection lost

If **Close when connection lost** is enabled, then the ApossIDE will automatically close its connection to a Simulator if that connection is lost for any reason. This prevents unnecessary "Connection lost" messages from being displayed. The most common case of this happening is when the user terminates the Simulator program before closing the Simulator connection in the ApossIDE.

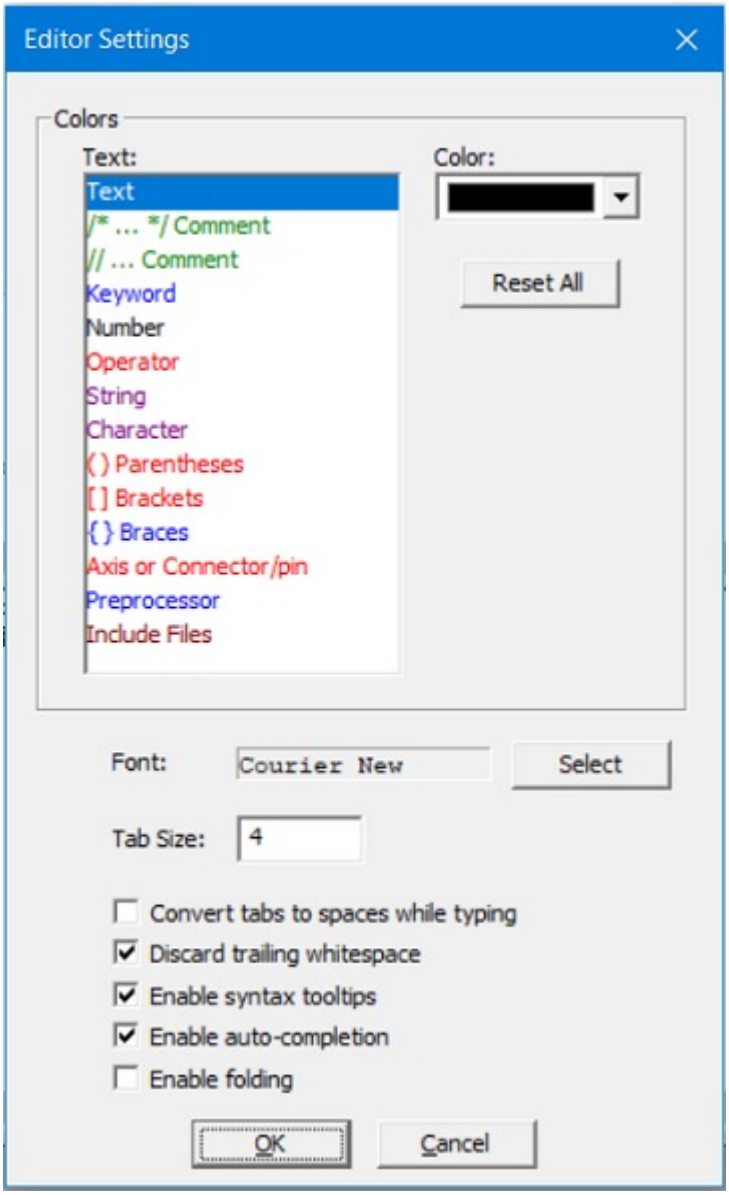
IP Address

If a Simulator is running on a remote computer, then this must specify the IP address of the remote computer.

Port

If a Simulator is running on a remote computer, then this must specify the TCP/IP port assigned to the Simulator. See Simulator for more information on specifying the IP port.

3.12.3 Editor Settings



In order to provide greater clarity, different colors can be assigned to the various program elements such as comments, key words, numbers, etc. Tab settings can also be chosen to make the program more readable.

Colors

Select the type of program element (e.g. Comment) and select the desired color. **Reset All** can be used to reset all the colors back to default colors.

Font

Press **Select** to select the character font and character height to be used in the editor.

Tab Size

Choose the preferred tab size. This can be set to 2 or more.

Convert Tabs

Click on **Convert Tabs to spaces while typing** to convert all tab characters to space characters while

typing. Note that this does not affect any tab characters that already exist in the program. Use **Edit** → **Convert Tabs** if you want to convert all existing tabs in a program to spaces.

Discard trailing whitespace

Click on **Discard trailing whitespace** to remove all trailing space and tab characters at the ends of lines before saving files.

Enable syntax tooltips

Click on **Enable syntax tooltips** to enable tooltips in the editor. These tooltips will display the syntax of the command when the mouse is paused on a line.

Enable auto-completion

Click on **Enable auto-completion** to enable the display of the popup auto-completion menu. This menu is displayed when the editor determines that some type of keyword is being entered. Pressing either the **Tab** button or double-clicking on a keyword in the menu, will automatically complete the word in the text being edited. The popup menu can be completely ignored simply by continuing to type text in the editor.

Enable folding

Click on **Enable folding** to activate the folding functions. It will be possible to show and hide functions and conditions by clicking on the displayed "+" or "-" sign. To use the feature you have to work with curly brackets {...} in the code.

3.12.4 Options

This dialog allows you to select various options that control the behavior of the ApossIDE. When settings are changed in the **Options** dialog, the settings will take effect the next time that they are used.

Enable Shift-Esc break only

Normally in the ApossIDE, the **Esc** key will send a “break” command to a connected controller to abort any running program. If the drive is rotating, then the drive will slow down with the maximum allowed deceleration. Stopping an operating drive in this way may cause damage to the system to which the drive is connected. To help avoid the possibility of this happening as a result of the user accidentally pressing the **Esc** key, this option can be set to disable the abort function. If set, then the user is still able to abort an executing program by pressing **Shift-Esc** (which is less likely to be pressed by accident).

Reopen previous files

If this option is enabled, then when the ApossIDE starts, it will automatically try to reopen all the files that were open when the ApossIDE was last used.

Reopen previous connections

If this option is enabled, then when the ApossIDE is started and previously open files are reopened, then the ApossIDE will automatically try to reconnect the files to controllers for those files that were connected when the ApossIDE was last used.

Auto-reconnect lost connections

If this option is enabled, then the ApossIDE will try to automatically reconnect to controllers when connections are lost. The ApossIDE will attempt to reconnect once every two seconds for up to one minute.

Open windows maximized

If this option is enabled, then all windows opened in the ApossIDE will be opened full-size. They can subsequently be reduced in size, if desired; they will simply be opened full-size. If this option is not enabled, then only the first window opened will be opened full-size. If a second window is opened, then all windows will be made smaller so that all windows can be seen simultaneously.

Autosave

If this option is enabled, then *before* executing the program, the ApossIDE will automatically save the current file being edited (if it has been modified) and all the include files that it references (if they have been modified). If not enabled, then the ApossIDE will compile using the current version of the file being edited as well as the current versions of any include files that are opened for editing. Note that if the compile fails, then the files are *not* saved.

Binary file support

If enabled, then the ApossIDE will allow the direct handling of binary compiled program images in the **Controller** → **Programs** window. This includes uploading compiled programs from the controller so that they can be saved on the PC and downloading previously saved binary images back to the controller. Note that these binary images cannot be edited; they can only be saved and restored. Also note that some older controllers do not support this feature.

If enabled, this option will also cause the item **Compile to file** to be added to the **Development** menu (once the ApossIDE has been re-started the next time). This item allows to user to manually compile and save the current program file to a binary file.

See also BinFileMap.

Print color

If this is enabled, then programs printed from the Edit Window will be printed in color using the same colors as the Edit Window. Otherwise, programs will be printed in black and white.

Array Editor parameter support

Normally, the Array Editor will read and display all user parameters and arrays. However, if this setting is enabled, then the Array Editor will also read and display global parameters, temporary and permanent axis parameters, axis process parameters, and system process parameters.

Create debug file

Enabling this option will cause the ApossIDE to generate debug log files. This is normally useful only to product support personnel. This option should normally be disabled.

Debug level determins how detailed data is generated in the debug file.

Level	APOS.DM	ZbMoc.dm
0	Only errors that occure are recorded.	Only errors that occure are recorded.
1	In addition to errors, basic	In addition to errors, read/write raw

Level	APOS.DM	ZbMoc.dm
	information about the operation of the application is recorded.	messsages and network statistics are recorded.
2	Extends level 1 with more detiled information about Communication Interface, Compiler output and Oscilloscope.	In addition to level 1, more information about connection and transport layer are recorded, including info about SDO index and subindex.
3	Extends level 2 with detailed data about binary program after compling, add information about Batch Mode, more connection details. Details about oscilloscope curve test run and tune.	Level 2 expanded with more detailed infromation about Virtual Link layer.
4	Info about PDO queue and polling messaging.	Level 3 expanded with more detailed infromation about Frames in Data Link layer.
5	/	All of the previous and information about frames to physical layer are recorded.

Protocol Font

This is the type font that is used to display messages in the Communications Window. Changing this value will only affect newly opened windows.

Watch Timeout

This specifies the rate at which values in the Watch Window are polled. This value will be used only the next time that Watch Window polling is started.

Customized Sections

The **SDO Lookup Tables** allow more experienced users to customize the contents of the SDO selection lists used in the ApossIDE. For example, these are used in the Watch Window and the **Oscilloscope**. This field will accept drag-and-drop filenames.

The **Debug File Directory** allows users to specify where debug log files are to be created. This field will accept drag-and-drop file or directory names. If a file is dragged to this field, then the directory will be set to the directory containing the file.

The **Authentication Key File** allows users to specify where authentication key file is located. This field will accept drag-and-drop filenames.

3.13 Windows Menu

The commands on the **Window** menu follow the Windows standards (i.e. **Cascade** for overlapping windows arranged down and across, **Tile vertically** for vertical windows arranged beside each other, or **Tile horizontally**

for horizontal windows arranged above/below each other).

The menu also includes the following commands:

Show Sidebar	Show/hide the Sidebar window.
Show Findbar	Show/hide the Findbar window.

3.14 Help Menu

ApossIDE Help

Contents will start the ApossIDE Help and display the “Contents” tab.

SDO Dictionary will start the ApossIDE Help and immediately jump to the “SDO Object Dictionary” page.

SYSDEF Defines will start the ApossIDE Help and immediately jump to the “SYSDEF Defines” page.

Axis Process Data (or **F11**) will start the ApossIDE Help and immediately jump to the “Axis Process Data” page.

> Many of the ApossIDE dialog windows and parameter fields within dialog windows, offer direct access to help. Select one of the dialog fields and press **F1**. Help will be displayed for that specific item if it is available.

> The look and feel of the Help subsystem will depend on the version of the Windows operating system being used.

APOSS Help (Legacy) will start Help for the APOSS Programming Language.

zub Website will start the default Internet browser and direct it to the zub home page.

About Program will display the version numbers of the ApossIDE, the low level interface driver, and the compiler.

3.15 Download Menu

The **Download** menu provides a simple interface to support the downloading of firmware and programs to multiple controllers. Note that all ApossIDE Windows, with the exception of the main ApossIDE application window, must be closed before this menu will be available.

3.15.1 Download Firmware

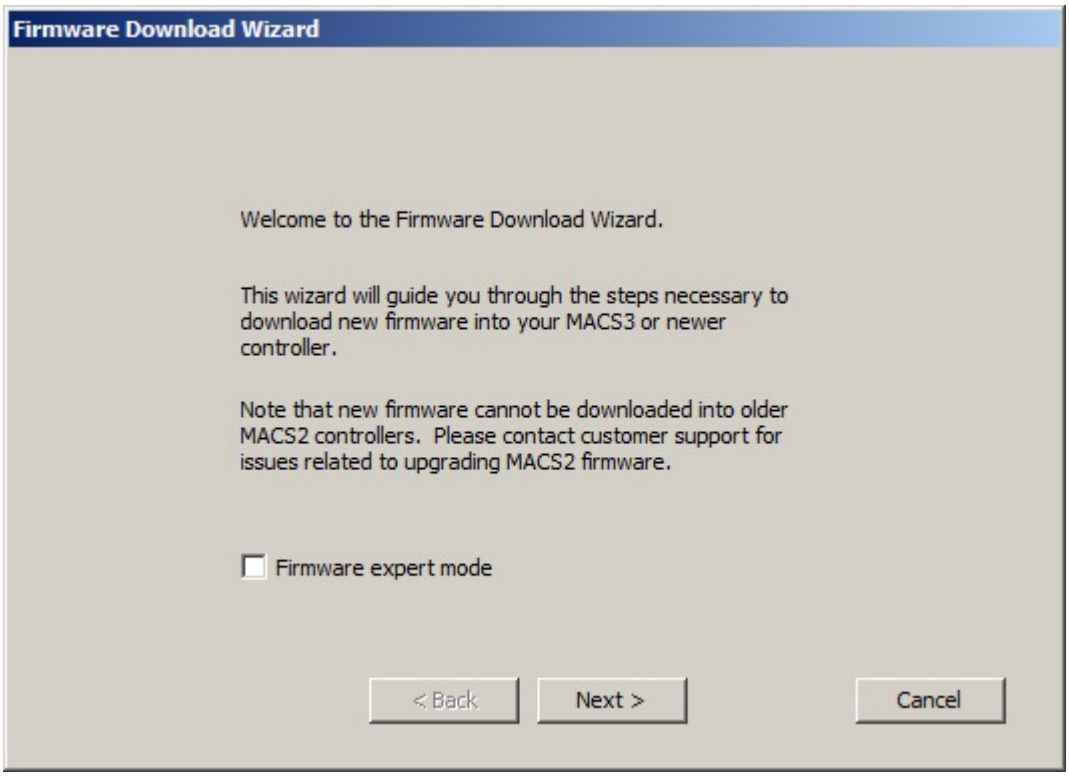
For all newer versions of controllers, the controller firmware (i.e. not compiled application programs) can be updated directly from within the ApossIDE.

Updated firmware for different types of controllers may be downloaded from the zub machine control AG website (**www.zub.ch**) if available. Otherwise, please contact zub (info@zub.ch). zub will be happy to provide you with replacement firmware or optimized firmware specifically designed for your hardware and application. On request, zub will also assist in customizing products and will adapt special functions to customize firmware. Stay in contact with zub to get the most out of our products and your machines!

> Firmware may also be updated automatically using a microSD card if the controller contains a microSD card slot. For more information, see External Storage Device (microSD Card/ USB Memory Stick).

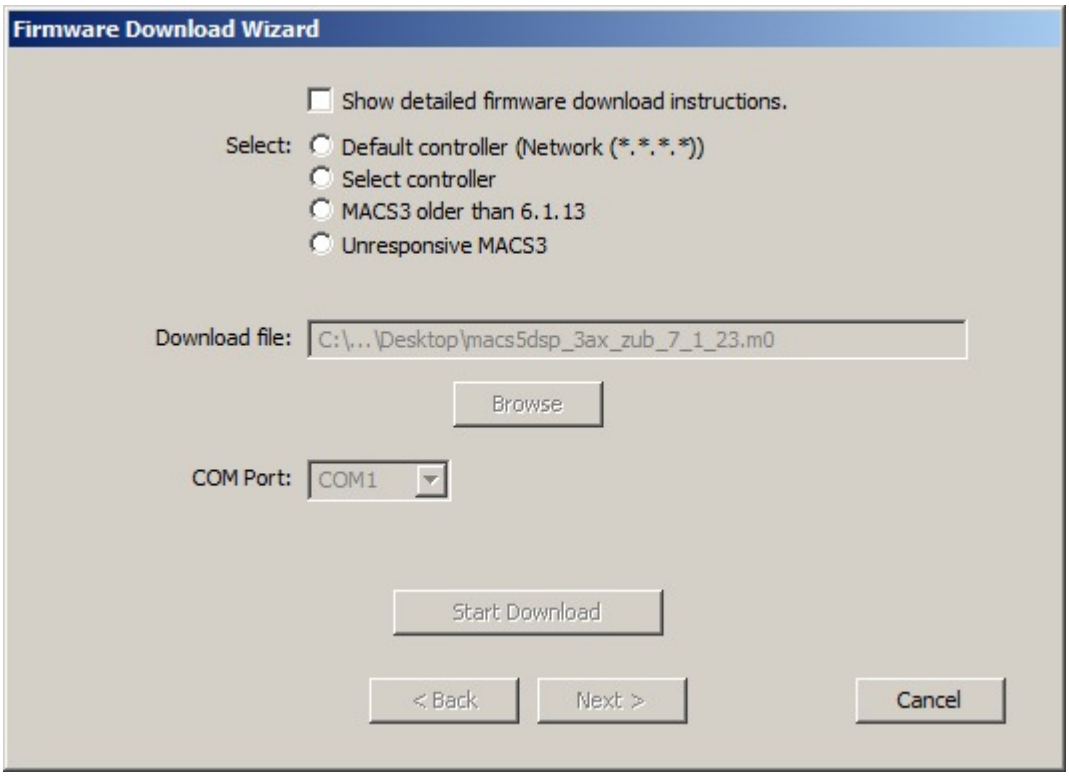
Using the Download Firmware Wizard

Make sure that all ApossIDE windows, with the exception of the main ApossIDE application window, are closed. Then click on **Download** and select **Firmware**. You will be presented with the "Welcome to the Firmware Download Wizard" dialog shown below.



If this is the first time that you have downloaded firmware, or if you are unfamiliar with the firmware download process, then press **Next** and follow the instructions. These will guide you through the download process.

If you are familiar with the download process, then select the **Firmware expert mode** checkbox and press **Next**. The entire download process can then be completed using a single dialog (shown below) and with a minimum of instructions.



If you are not an “expert” and the “expert dialog” is displayed, then you can switch back to the normal dialogs by selecting the **Show detailed firmware download instructions** checkbox and pressing **Next**.

The filename of the last file downloaded by the firmware downloader is saved and will be used as the default, the next time the firmware downloader is used.

⚠ Please read the hardware manual for more details and safety instructions.

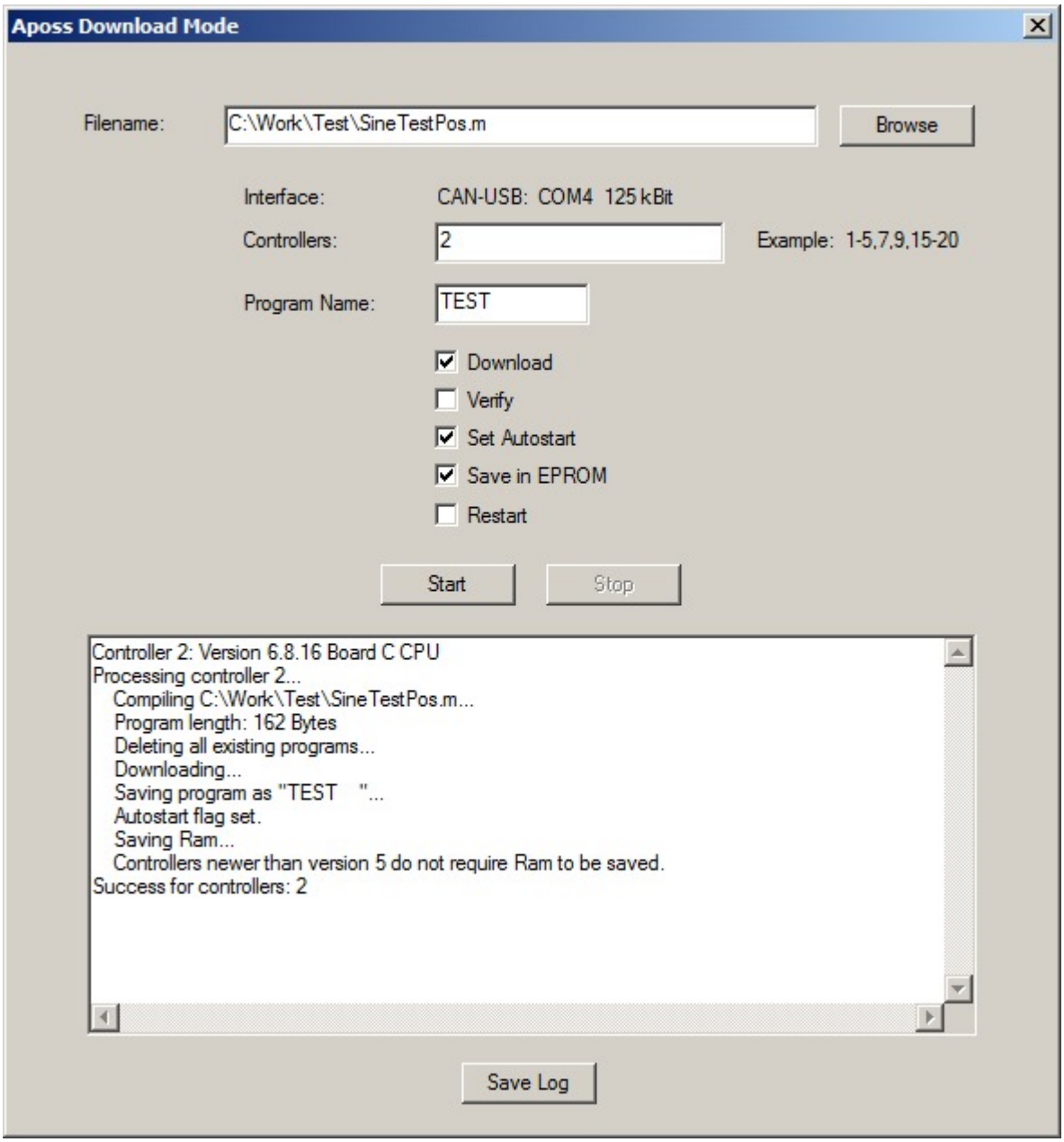
3.15.2 Download Programs

Select **Programs** and use the **Browse** button to select the application program that should be downloaded. If the controller interface supports multiple controllers, then enter the range of controller ID’s to which the program should be downloaded.

Enter a program name to be used to identify the program on the controller.

For controllers with firmware version 6.01.10 or later, names are saved using UTF-8 encoding. This allows any character to be used in names, even characters in foreign languages. However, names may not be longer than 8 bytes (n.b. some UTF-8 characters require more than one byte). For controllers older than 6.01.10, characters are restricted to ANSI characters in the range 0x20 to 0x5F (n.b. accented characters may not be used) and the name cannot be longer than 8 characters.

Select the desired operations using the check boxes **Download, Verify**, **Set Autostart**, **Save in EPROM**, and/or **Restart**. For example, you might check **Verify** but not **Download**. In that case the Downloader will only verify that the specified file is the one that is already on the controller.



Start Download Programs

Click the **Start** button. A connection is established to the controllers using the default ApossIDE interface. Any controller that is currently executing will be stopped. The following steps are performed for each accessible controller. All steps are performed for one controller before going on to the next controller.

- The program is compiled using compiler settings appropriate to the controller.
- All existing programs on the controller are deleted.
- The program is downloaded.
- The program is saved as program number 0 with the specified program name. If no program name was specified, then the first 8 characters of the filename are used.
- If the Verify flag is set, then the downloaded program is uploaded again and the downloaded and uploaded versions are compared. If they differ, then processing is halted for this controller. Note that not all older controllers contain firmware that will support uploading the

program. This flag is ignored if the controller does not support uploading.
- The Autostart flag is set for the program if requested.
- All data is saved in EPROM if requested. Note that this will be done automatically for newer controllers.
- The controller is restarted with the new program if requested.
- The connection to the controllers is closed.

If any error occurs during the processing of a controller, then processing is halted for that controller. However, processing will continue for subsequent controllers in the ID range. When the entire download operation is complete, a summary is displayed in the text box listing the ID's of those controllers that encountered no errors and those controllers that encountered errors. Use the scroll bars to scroll up in the text box in order to determine the nature of any problems encountered.

Save Log

The progress of the download is displayed in the large text box. If desired, this information can be saved to a text file by clicking the **Save Log** button.

If the download fails for one or more controllers, then you will get a "Failed for controllers" message. Saving the log will let you go back and review the failures at a later time.

4 ApossIDE Programming

The ApossIDE provides two programming languges for developers:

- The ApossC Programming Language is the *preferred* programming language. It is a subset of the popular C programming language and will be very familiar to developers that have previously used the C programming language. It is a more powerful language to use than APOSS.
- The **APOSS Programming Language** is a legacy programming language. It was originally developed to have a simplified syntax structure that is easier to learn. However, this results in some restrictions that are not present in ApossC.

The programming language to be used is selected when a new program file is opened with the **File** → **New** command.

Users who are new to the ApossIDE, should choose to use ApossC rather than APOSS.

These Help sections do *not* attempt to teach the basics of computer programming. Users are expected to be already familiar with the concepts and structures available in common programming languages.

For developers that are familiar with C and that are just starting to use the ApossC programming language, it is *highly* recommended that the ApossC Programming Language help pages be read (or at least scanned). Hopefully, this will eliminate many common mistakes made by first-time users.

For developers that are familiar with APOSS and that are just starting to use the ApossC programming language, it is *highly* recommended that the Differences Between APOSS and ApossC help page be read (or at least scanned). Hopefully, this will eliminate many common mistakes made by first-time users.

APOSS *must* be used with controllers that have firmware versions older than 9.0.00. ApossC may only be used on controllers with firmware version 9.0.00 or newer.

ApossC Programming Language	A description of the C programming language subset that is provided by the ApossIDE.
Preprocessor	The ApossIDE provides a built-in preprocessor similar to that available in many C/C++ compilers.
Program Samples	Many samples of simple application programs are provided by the ApossIDE. These are intended to show how various APOSS-specific features can be used.
Debugging Programs	Tools and tips for debugging application programs.

4.1 ApossC Programming Language

The ApossIDE provides a subset of the **C Programming Language** (called **ApossC**) that may be used as an alternative to the original **APOSS Programming Language** when developing applications for controllers. This section assumes that the reader is already competent with programming in the C language; it does *not* attempt to teach any C concepts.

ApossC requires the controller to have at least firmware version 9.0.00. For controllers with firmware older than this, the **APOSS Programming Language** is required.

Most standard C concepts are implemented in exactly the same manner in ApossC as they are in standard C. The most noteable exceptions are:

- ApossC filenames use the extensions ".mc" and ".mh" rather than the common ".c" and ".h". This is done to differentiate ApossC files from standard C files and allows a Windows "double-click" to open files using the ApossIDE rather than with some other C development system that may also be installed.
- ApossC does not support C pointers (e.g. "long *abc;"). This was done intentionally since programming errors related to pointers very often lead to fatal crashes (i.e. crashes from which the running application is unable to recover). This is very undesirable when the controller is driving real motors.
- ApossC does not currently support C structures or enumerated types.
- ApossC does not support some of the less commonly used C concepts (e.g. static, extern, union, etc.) and some of the C concepts that are of marginal use on the controller.
- ApossC currently has no linker as we know it at C. When setting up a file structure with *.mc and *.mh files, this must be taken in consideration.

ApossC also provides several extensions that are not available in standard C. In particular, ApossC provides:

- Access to all the standard motor control functions.
- Support for interrupt handling.
- Extensions related to the use of kinematics and robot control.
- Support for state machine programming.
- Support for call-by-reference function parameters.

For a complete description of the differences between standard C and ApossC, please see Differences Between Standard C and ApossC .

ApossC provides all of the functionality available in the APOSS Programming Language, but using C concepts and syntax. It also provides some enhanced functionality over APOSS, particularly with respect to array handling. For a complete description of the differences between APOSS and ApossC, please see Differences Between APOSS and ApossC .

4.1.1 Differences Between Standard C and ApossC

The following is a comprehensive list of the differences between standard C and ApossC. ApossC extensions are marked in red.

main()

Standard C normally expects the main entry point for a program to be defined as a function similar to:

```
int main(int argc, char *argv[]);
```

ApossC expects the main entry point to be defined as:

```
void main(void);
```

It is not necessary to include a function prototype for "main" when using ApossC.

Data Types and Qualifiers

ApossC supports the following data types for variables and arrays:

`long` 32-bit signed integer.

`double`	64-bit floating point value.
`wchar`	16-bit unsigned integer. This is analogous to the standard C `wchar_t` type.
`transform`	3-dimensional tranformation matrix. See Transformations. [ApossC extension]
`dpoint`	n-dimensional point. See Working with Points. [ApossC extension]

The following types and qualifiers are not supported:

`int`, `short`, `float`, `signed`, `unsigned`, `const`, `auto`, `static`, `extern`, `volatile`, `register`

`enum`, `struct`, `union`, `typedef`, * (i.e. pointers)

ApossC supports the following additional qualifiers:

`dim`	Permanent array saved in EEPROM. [ApossC extension]

> `wchar` arrays are always aligned on "long" memory boundaries. As a result, if the total number of elements in a `wchar` array is odd, then the array will be extended by 1 element to make it even. This may affect the results returned by array functions such as arraylen. For example, "`wchar arr[5]`" will actually be allocated as if "`wchar arr[6]`" had been declared. Similarly, "`wchar arr[5][5]`" will have a *single* extra element appended, but this extra element will be awkward to access.

Numeric Constants

ApossC supports the following additional constants that are not supported by standard C:

`PI`	π (i.e. 3.14159....). [ApossC extension]
`#.#.#.#`	IP address where each # is an integer constant in the range 0-255. There may be no whitespace characters adjacent to the "." characters. [ApossC extension]

ApossC does not support the u and U constant suffixes. The l and L suffixes are supported, but are unnecessary.

Character Constants

ApossC does not support the following:

- Trigraphs (e.g. '??(' used to represent '[').
- Two (or more) letter character constants are not supported in ApossC (e.g. '\x32z' for a space and 'z' to be stored in one value).
- The wide character designation (i.e. L'x'). All characters in ApossC are treated as wide characters.

String Literals

ApossC does not support the following:

- The \" escape sequence for including a double quote character in a string.
- The wide character designation (i.e. L"xxx"). All characters in ApossC are treated as wide characters.

Operators

ApossC does not support the unary & and * operators (i.e. pointer operators).

ApossC supports the "(type) expr" style of cast operator. It does not support the "type(expr)" style of cast function.

ApossC always treats the >> shift operator as a ***signed*** shift.

ApossC places some restrictions on the use of the '?' operator:

- The operands must be scalar values (i.e. simple variables, constants, and array elements). For example, the following are not supported:

```
(test) ? "abc" : "def"
((test) ? array1 : array2)[0]
```

- The expression may not be used an "lvalue" (i.e. on the left side of an assignment). For example, the following is not supported:

```
((test) var1 : var2) = 123;
```

ApossC provides a "." bit operator to access bits, bytes, and words within a **long** variable or coordinates within a **dpoint** variable. [ApossC extension]

- The "." operator behaves as if it were referencing "virtual" C structure members that have been mapped onto variables.
- The following array names are supported for **long** variables:

.i[n] Accesses individual bits with a **long** variable. "n" may be any integer constant or expression that evaluates to a value from 0 to 31. Bit 0 corresponds to the least significant bit and bit 31 corresponds to the most significant bit.

.ub[n] Accesses individual bytes with a **long** variable. The bytes are treated as *unsigned* values. "n" may be any integer constant or expression that evaluates to a value from 0 to 3. Byte 0 corresponds to the least significant byte and byte 3 corresponds to the most significant byte.

.sb[n] Accesses individual bytes with a **long** variable. The bytes are treated as *signed* values. "n" may be any integer constant or expression that evaluates to a value from 0 to 3. Byte 0 corresponds to the least significant byte and byte 3 corresponds to the most significant byte.

.uw[n] Accesses individual 16-bit words a **long** variable. The words are treated as *unsigned* values. "n" may be any integer constant or expression that evaluates to to either 0 or 3. Word 0 corresponds to the least significant word and word 1 corresponds to the most significant word.

.sw[n] Accesses individual 16-bit words a **long** variable. The words are treated as *signed* values. "n" may be any integer constant or expression that evaluates to to either 0 or 3. Word 0 corresponds to the least significant word and word 1 corresponds to the most significant word.

- When accessing **long** variables, if the "n" is a known integer constant, then the brackets shown above are not necessary. However, there may be no whitespace between the '.' operator and the constant. For example, ".i3" may be use to reference bit 3 of a variable.

  > `#define` cannot be used to represent integer constants. For example, the following is not valid:
  >
  > ```
  > #define MYBIT 3
  > var.iMYBIT
  > ```
  >
  > However, the following two alternatives may be used:
  >
  > ```
  > #define MYBIT 3
  > var.i[MYBIT]
  >
  > #define MYBIT i3
  > var.MYBIT
  > ```

- The following names are supported for **dpoint** variables (see Working with Points):

  .x Accesses the X coordinate of the **dpoint** variable.

  .y Accesses the Y coordinate of the **dpoint** variable.

  .z Accesses the Z coordinate of the **dpoint** variable.

  .a to **.g** ".a" accesses the first non-spatial coordinate of the **dpoint** variable, ".b" the second non-spatial coordinate, etc.

- As with standard C, when using the '.' operator, the complete variable/operator construction may be used as either an "rvalue" or an "lvalue" (i.e. used on either side of the '=' in an assignment expression). For example:

```
vara.i1 = varb.i0;
vara.ub[expra] += varb.ub[exprb];
arr[indx].sw0++;

pnta.x = pntb.x;
pnta.y *= 100.0;
```

Variable Initialization

- Standard C restricts the initialization of global variables to constants. ApossC allows expressions with variables to be used as long as the variables are declared before the variable being initialized. For example, the following is not allowed in standard C, but it is allowed in ApossC:

```
double two  = 2.0;
double root = sqrt(two);
```

Standard Functions

ApossC supports only the following subset of common C library functions:

`min`, `max`, `abs`, `fabs`, `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `log`, `pow`

In particular, the following standard functions are *not* supported:

`sizeof`, `offsetof`, `memcpy`, `memmove`, `strlen`, `strcpy`

The standard "exit" function is implemented as the Exit function in ApossC. With ApossC, this parameter must be specified, but its value will currently be ignored.

[ApossC extension] ApossC has support for the following additional functions:

```
rnd, rand, grad, rad, round
```

Including a C header (.h) file for these functions is not necessary with ApossC.

Unlike standard C, the print functions (`print`, `printf`, and `sprintf`) in ApossC do not return the character count. These are `void` functions in ApossC.

Functions

Functions with a variable argument list are not supported. For example:

```
long Func(long parm, ...);
```

The “old” C syntax for declaring function prototypes is not supported. For example:

```
long Func(parm1, parm2)
long parm1;
double parm2;
{
...
}
```

ApossC does not support passing functions as function arguments.

ApossC does not support declaring function prototypes inside functions (i.e. local function prototypes). All function prototypes must be global.

[ApossC extension] Standard C does not support “call-by-reference” for simple arguments in functions. ApossC supports call-by-reference using the “&” syntax borrowed from C++. Call-by-reference may be used with "long", "double", and "wchar" variables only. For example:

```
void CalcSquare(long &lptr, double &dptr)
{
    lptr = lptr * lptr;
    dptr = dptr * dptr;

    return;
}

...
    long   value = 5;
    double dbl   = 4.5;

    CalcSquare(value, dbl);
```

Interrupts

[ApossC extension] Standard C does not explicitly define support for interrupt handling. ApossC does support interrupt handling. See Interrupt Handling.

State Machines

[ApossC extension] Standard C does not support state machines. ApossC supports native state machine syntax. See State Machines.

4.1.2 Differences Between APOSS and ApossC

The following is a comprehensive list of the differences between APOSS and ApossC.

Case Sensitivity

The most immediate and important difference between APOSS and ApossC, is case sensitivity. APOSS in case-*insensitive*, while ApossC is case-*sensitive* (like standard C).

This is likely to be a VERY common cause of bugs until developers are familiar with ApossC.

Statement Termination

APOSS does not use any special character to terminate statements. It determines the ends of statements, based on context.

ApossC is like standard C and uses the ';' character to terminate most statements.

This is likely to be a VERY common cause of bugs until developers are familiar with ApossC.

main()

APOSS does not use a "main" function or subroutine. The "main" part of an APOSS program begins simply with the first executable statement.

ApossC requires a "main()" function. Like standard C, executable statements may *not* appear outside of functions. The only exception to this is executable expressions used to initialize global variables.

This is likely to be a VERY common cause of bugs until developers are familiar with ApossC.

Array Indexing

APOSS uses **1-based** indexing for arrays. For example, the first element in an APOSS array is "array[1]". ApossC uses **0-based** indexing for arrays. For example, the first element in an ApossC array is "array[0]".

This is likely to be a VERY common cause of bugs until developers are familiar with ApossC.

Axis Numbers and Axis Modules

APOSS uses **axis numbers**. Axis numbers are **1-based**, so the first axis is axis 1. ApossC uses **axis modules**; it does not use axis numbers. Axis modules are **0-based**, so the first axis is axis 0.

This is likely to be a VERY common cause of bugs until developers are familiar with ApossC.

APOSS Commands, Functions, and Keywords

All APOSS commands are implemented in ApossC as functions. If the APOSS command has no return code, then the ApossC function will be a `void` function. If the APOSS command returns a value, then the ApossC function will be a `long` function. Note that command names are always implemented in ApossC using mixed case. This will differentiate them from standard C functions.

Examples:

APOSS	ApossC
`DELAY expr`	`Delay(expr);`
`pos = APOS X(1)`	`pos = Apos(0);`

APOSS	ApossC
`POSA X(1) 1000 X(2) 2000`	`AxisPosAbsStart(0,1000,1,2000);`
`CANOPENSLAVE`	`CanOpenRestart();`

APOSS functions and keywords which are either equivalent to standard C functions or have functionality comparable to standard C, are implemented in lower case.

Examples:

```
long
double
wchar
dim
res = sin(angle)
res = min(expr1,expr2)
```

Keywords used with APOSS Commands

Many APOSS commands use keywords like ON, OFF, STOP, etc. Global and axis parameters names are also used. These are case-insentive. All of these keywords (excpet for global and axis parameter names) will be implemented in ApossC using *uppercase* names. This is similar to the common programming practice of using uppercase characters for names of constants and lowercase or mixed-case for other identifiers.

Examples:

APOSS	ApossC
`motor on X(1) X(2)`	`AxisControl(0, ON, 1, ON);`
`delete(arrays)`	`Delete(ARRAYS);`

General Syntax

APOSS and ApossC have the following operator differences. These were changed to make operators conform to standard C.

APOSS	ApossC
% [division]	/
mod [module division]	%
# [not equals]	`!=`
and	&&
or	||
not	!
exp	`pow(...)` [Implemented as a function]
ln	`log(...)` [Implemented as a function]
rnd	`rnd(...)` [Implemented as a function]

Other differences:

- APOSS supports hexadecimal numbers with embedded "_" characters (e.g. 0x12_34_56_78). ApossC does not support embedded "_" characters. Only standard C hexadecimal number are supported.
- The APOSS **string** data type has been replaced with the **wchar** type in ApossC. Note that **wchar** is a "wide char" in ApossC (i.e. a Unicode character).
- APOSS does not support "segmented" string literals (i.e. multiple quoted strings with whitespace between them). ApossC does support this. For example:

  ```
  "abcdef" "ghijkl"
  ```

  is equivalent to:

  ```
  "abcdefghijkl"
  ```

- APOSS does not support pre-increment and pre-decrement (e.g. ++abc and --abc). ApossC does support these.
- If a "default" label is used in a "switch" statement in APOSS, then it must be the last case statement defined. In ApossC (as in standard C), the "default" statement can be anywhere within the "switch" statement.
- The RETURN statement in APOSS requires parentheses if a value is to be returned (e.g. "return(1)"). ApossC does not require parentheses (although they may be used).
- APOSS does not allow **transform** variables to be assigned. ApossC allows this. For example:

  ```
  transform t1, t2;
  t2 = t1;
  ```

- APOSS allows labels to immediately precede closing block keywords (e.g. ENDFOR, etc.). ApossC does not allow labels to immediately precede "}". If this is required, then it must be specified this way using an empty statement:

  ```
  label:
  ;
  }
  ```

- The **ELSEIF** structure is not supported in ApossC. Use the standard C “if” structure instead.
- The **LOOP** statement is not supported in ApossC.
- The **REPEAT** statement has been replaced with the "do" statement in ApossC.

  > ⚠ If an existing "REPEAT...UNTIL" structure is being converted into an ApossC "do...while" structure, then it is important to realize that the sense of the stopping test is *reversed* in ApossC. APOSS repeats the loop until the test is *true*; ApossC repeats the loop until the test is *false*.

- The “shortcut” array copy commands that exist in APOSS are not supported in ApossC. Use the “arraycpy()”, etc., functions instead. For example:

  ```
  long arr1[20], arr2[20]
  arr2 = arr2
  ```

Variable and Array Declarations

- APOSS allows variables to be defined implicitly (as **long**). In ApossC, variables must be defined before they are referenced. ApossC does not support `#pragma implicit`.
- In APOSS, DIM arrays must be defined before any other code. In ApossC, DIM arrays need not be defined

first. They may be defined anywhere that a global variable may be defined.
- APOSS supports arrays of only 1 and 2 dimensions. ApossC supports arrays with an arbitrary number of dimensions.
- APOSS supports the CAMDIM declaration for declaring arrays that will be used as CAM arrays. ApossC does this using the CamArrayLen function.

Variable and Array Initialization

- ApossC allows both global and local variables to be initialized when they are declared. This is done in the same manner as it is in C. For example:

```
double two   = 2.0;
double root2 = sqrt(two);
```

- ApossC allows both global and local arrays to be initialized when they are declared. This is done in the same manner as it is in C. For example:

```
long  arr[] = { 1, 2, 3, 4 };
wchar str[] = "This is a string.";
```

  As in C, the first array dimension is taken from the initialization if it is omitted. The second form (i.e. using a quoted string) may only be used with 1-dimensional **wchar** arrays. Note that this form of array initialization may also be used with "dim" arrays.
- If multi-dimensional arrays are initialized with a list of quoted strings, then the string are always associated with the *last* array dimension. For example:

```
wchar msg[][2][50] = { "abcdef", "ghijkl", "mnopqr", "stuvwx" };
```

  This will assign the string "abcdef" to msg[0][0], "ghijkl" to msg[0][1], "mnopqr" to msg[1][0], and "stuvwx" to msg[1][1].
- With APOSS, global variables and arrays are *uninitialized* when the program starts. ApossC will *initialize* all global variables and arrays to 0 if they are not initialized by the user as above. Note that *local* variables and arrays are always *uninitialized* if the user does not explicitly initialize them. For ApossC programs that declare very large arrays, this may noticeably slow down program startup. If this is an issue, then the initialization of global arrays may be suppressed using **Settings** → **Compiler**.

Subroutines

- APOSS has the concept of subroutines (i.e. in a SUBMAINPROG section) and a GOSUB command. ApossC does not support subroutines; normal functions should be used instead.

Function Parameters

- ApossC allows local arrays and arrays that are declared as function parameters, to be passed to functions. APOSS does not allow this; local arrays cannot be passed to functions. For example:

  ApossC:

```
void Func1(long arr[])
{
    arr[0] = 123;

    return;
}

void Func2(long prmarr[])
{
```

```
    long lclarr[10];

    Func1(lclarr);   // This is allowed.
    Func1(prmarr);   // This is allowed.

    return;
}
```

APOSS:

```
long Func1(long[] arr)
{
    arr[0] = 123

    return(0)
}

long Func2(long[] prmarr)
{
    long lclarr[10]

    Func1(lclarr)    // This is NOT allowed.
    Func1(prmarr)    // This is NOT allowed.

    return(0)
}
```

- Both ApossC and APOSS allow parameters to be passed to functions using "call-by-reference". However, ApossC uses the "&" syntax borrowed from C++ and does not require any special syntax to be used within the body of the function; variables are used in the normal way. APOSS uses the "@" syntax both in the function prototype and within the function body. In addition, ApossC allows the pre/post-incrment/decrement operators to be used with these parameters (for **long** and **wchar** variables only); APOSS does not. For example:

  ApossC:

  ```
  void CalcSquare(long &lptr, double &dptr)
  {
      lptr = lptr * lptr;
      dptr = dptr * dptr;

      return;
  }
  ```

  APOSS:

  ```
  LONG CalcSquare(LONG @lptr, DOUBLE @dptr)
  {
      @lptr = @lptr * @lptr
      @dptr = @dptr * @dptr

      return(0)
  }
  ```

Behaviour of DOUBLE Values

- In APOSS, the logical operators >, <, >=, <=, AND, and OR when used with double operands, produce a double result. In ApossC, the result is a long which more closely matches standard C.

- The "!" (not) operator when used with double values behaves like this:

APOSS	ApossC	APOSS Behaviour	ApossC Behaviour
`not dbl`	`!dbl`	`((long) dbl) == 0`	`dbl == 0.0`
`not 0.5`	`!0.5`	`1`	`0`

- In APOSS, statements such as WHILE, FOR, IF, etc., that specified "double" expressions as test expressions, cast the test expression to **long** and then executed the test. In ApossC, **double** test expressions are tested as "expr != 0.0" which more closely matches standard C. For example:

```
if (0.5) then
    print "Yes"
else
    print "No"
endif
```

will print "No" in APOSS. However:

```
if (0.5)
    print("Yes");
else
    print("No");
```

will print "Yes" in ApossC.

#pragmas

In APOSS, the CAM_MAXFIXPOINT, PNT_LENGTH, and DEPRECATEDWARNINGS #pragmas accept integer constants as parameters. ApossC accepts any expression that evaluates to an integer constant. For example:

```
#define SPATIALCOUNT 3 // Number of spatial axes in kinematics.
#define EXTRACOUNT   2 // Number of extra non-spatial axes in
kinematics.
#pragma PNT_LENGTH SPATIALCOUNT + EXTRACOUNT
```

GET/SET Commands

The **GET** and **SET** commands used in APOSS to query and set global and axis parameters do not exist in ApossC. Instead, all parameters should be queried and set using Sysvar and the definitions in the "SysDef.mh" file. Also keep in mind that APOSS uses 1-based axis numbers while ApossC and Sysvar use 0-based axis *module* numbers. For example:

APOSS	ApossC
`set canbaud 8`	`GLB_PARAM(CANBAUD) = 8;`
`set posdrct x(1) -1`	`AXE_PARAM(0, POSDRCT) = -1;`

APOSS supports synonyms (shown in lower case below) for several axis parameter names (shown in upper case below):

gain (KPROP), zero (KDER), pole (KINT), orgvel (SYNCOFFTIME), orgacc (SYNCMFPAR), hometype (HOME_TYPE), ramp_min (RAMPMIN), posunits (PROFTIME), gearmotorrev (POSFACT_Z), gearshaftrev (POSFACT_N)

ApossC will not support the synonyms.

Renamed Commands

The following APOSS commands have been renamed in ApossC:

APOSS	ApossC
CANDEL	CanDelete(...);
CANINI	CanInit(...);
CANOPENSLAVE	CanOpenRestart(...);
CSTART	AxisCvelStart(...);
CSTOP	AxisCvelStop(...);
DCPINIT	SerialInit(...);
DCPREADCOMM	SerialReadComm(...);
DCPWRITECOMM	SerialWriteComm(...);
DEFCORIGIN	DefCposOrigin(...);
DEFMCPOS	DefMasterCpos(...);
DEFMORIGIN	DefMasterOrigin(...);
ERRAX	ErrorAxis(...);
ERRCLR	ErrorClear(...);
ERRINFO	ErrorInfo(...);
ERRNO	ErrorNo(...);
IN	DigInput(...);
INAD	AnalogInput(...);
INB	DigInputByte(...);
INKEY	SerialReadChar(...);
LINA	AxisLinAbsStart(...);
LINR	AxisLinRelStart(...);
MEMEORYDUMPSTATE	MemoryDumpStatus(...);
MOTOR	AxisControl(...);
MOTOR STOP	AxisStop(...);
MOVESYNCORIGIN	SyncShiftOrigin(...);
OUT	DigOutput(...);
OUTB	DigOutputByte(...);
OUTDA	AnalogOutput(...);

APOSS	ApossC
PATHJRK	PathJerk(...);
POSA	AxisPosAbsStart(...);
POSR	AxisPosRelStart(...);
RSTORIGIN	ResetOrigin(...);
SDOREAD	SdoRead(...);
SDOWRITE	SdoWrite(...);
SDOSTATE	SdoStatus(...);
SETCURVE	SyncSetCurve(...);
SETMORIGIN	SetMasterOrigin(...);
STAT	AxisStatus(...);
STATCLR	AxisStatusClear(...);
SYNCC	SyncCam(...);
SYNCCMM	SyncCamMarkerMaster(...);
SYNCCMS	SyncCamMarkerSlave(...);
SYNCCSTART	SyncCamStart(...);
SYNCCSTOP	SyncCamStop(...);
SYNCERR	SyncError(...);
SYNCM	SyncMarker(...);
SYNCMARKERSTART	SyncMarkerRestart(...);
SYNCP	SyncPos(...);
SYNCSTAT	SyncStatus(...);
SYNCSTATCLR	SyncStatusClear(...);
SYNCV	SyncVel(...);
TESTSETDEST	RecordSetDest(...);
TESTSETINDEX	RecordIndex(...);
TESTSETTIME	RecordTime(...);
TESTSETTYPE	RecordType(...);
TESTSTART	RecordStart(...);
TESTSTOP	RecordStop(...);
TPOSA	AxisTPosAbsStart(...);
TRACKERR	TrackError(...);

APOSS	ApossC
USRSTAT	UserStatus(...);
WAITAX	AxisWaitReached(...);
WAITI	WaitDigInput(...);

The following APOSS 2-word commands have been renamed to 1-word functions in ApossC:

APOSS	ApossC
lin a	AxisLinAbsStart(...);
lin r	AxisLinRelStart(...);
pos a	AxisPosAbsStart(...);
pos r	AxisPosRelStart(...);
wait ax	AxisWaitReached(...);
wait i	WaitDigInput(...);
def origin	DefOrigin(...);
set origin	SetOrigin(...);
rst origin	ResetOrigin(...);
def corigin	DefCposOrigin(...);
def canin	DefCanin(...);
def canout	DefCanout(...);

Obsolete Commands

The following APOSS commands are not supported in ApossC:

#DEBUG
AXEND [Use AxisTargetReached, etc., instead.]
CONTINUE
GET [Use Sysvar, etc., instead.]
GETSYS
_GETVEL
HOME
INDEX
INGLB
INMSG
MOVEON
MSGVAL
NOWAIT
OUTAN
OUTMSG
PID
PULSACC
PULSVEL

SAVEPROM
SET [Use Sysvar, etc., instead.]
SETSYS
SWAPMENC
TESTSETP [Use RecordIndex, etc., instead.]
TOA and TO A
TOR and TO R
VIAA and VIA A
VIAR and VIA R
VLTCOMMAND
VLTCOMSTATE
VLTREAD
VLTSTATUS
VLTWRITE
WAITNDX
WAITP
WAITT

APOSS Commands with Axes Parameters

APOSS has many commands that take axis numbers as parameters (e.g. POSA). These have the forms similar to the following:

1. `POSA expression`

   This form (where the axis number defaults to axis 1) is not supported by ApossC. ApossC requires an axis be specified.

2. `POSA x(*) expression`

   This form is implemented as: `AxisPosAbsStart(AXALL, expression);`

3. `POSA x(axis) expression x(axis) expression ...`

   This form is implemented as: `AxisPosAbsStart(axis1, expression1, axis2, expression2, ...);`

4. `POSA expression / expression / ...`

   This form (where the axis numbers default to 1, 2, 3, ... is not supported by ApossC.

Execution Order of Axis Commands

APOSS axis commands such as CVEL, executed the expression for every axis and it did this in the reverse order of the axes. For example:

```
vel = 5
CVEL x(1) vel++ x(2) vel++ x(3) vel++
```

would be equivalent to:

```
CVEL x(3) 5
CVEL x(2) 6
CVEL x(1) 7
```

Also:

```
vel = 5
CVEL x(*) vel++
```

would execute in the same order as above (for a 3-axis controller).

For ApossC, the expression is evaluated only when it is seen and this is done in the forward order of axes. For example:

```
vel = 5;
Cvel(1, vel++, 2, vel++, 3, vel++);
```

would be equivalent to:

```
Cvel(1, 5);
Cvel(2, 6);
Cvel(3, 7);
```

Also, the APOSS "all" (i.e. "*") case is executed *differently* in ApossC:

```
vel = 5;
Cvel(AXALL, vel++);
```

would be equivalent to:

```
Cvel(1, 5);
Cvel(2, 5);
Cvel(3, 5);
```

NOWAIT Command

APOSS commands that move axes are designed to wait until the axes have reached their target positions before the command completes and program execution contunues. The **NOWAIT** command may be used to disable the "wait until complete" behaviour. ApossC has no "NOWAIT" command and application programs will never wait for axes to complete movements. ApossC commands initiate movements and then return immediately. ApossC applications may wait for axis movement to complete by using any of the various "wait" methods (e.g. AxisWaitReached).

Interrupts

- In APOSS, interrupts handlers are assigned to specific interrupts using the ON command. In ApossC, this is done using InterruptSetup.
- In APOSS, interrupts handlers are removed using the ON command with the special "DELETE" keyword. In ApossC, this is done using InterruptDelete.
- In APOSS, interrupts handling is enabled using the ENABLE command. In ApossC, this is done using InterruptEnable.
- In APOSS, interrupts handling is disabled using the DISABLE command. In ApossC, this is done using InterruptDisable.

- In APOSS, interrupt handlers are implemented as **SUBPROG** subroutines. In ApossC, they are implemented as `void` functions with no parameters. See Interrupt Routines for more information.

- The APOSS **INT** interrupt has been renamed to DIGINPUT in ApossC.
- The APOSS **KEYPRESSED** interrupt has been renamed to SERIALCHAR in ApossC.
- In APOSS, the axis number is optional for the STATBIT interrupt. It will default to 1. In ApossC, the axis number is mandatory.

Labels

- ApossC allows label names to be the same as function names (as does C). The namespace for label names is kept separate from the global namespace. Names are always local to the function in which they are defined.

Bit Operators

The APOSS use of the "." bit operators to access bits within a **long** value (e.g. abc.uw1 = 1) or coordinates within a **dpoint** value, is supported slightly differently in ApossC:

- ApossC treats the "." operators as if they were referencing "virtual" C union members that have been mapped onto variables.
- APOSS uses 1-based numbering for bits, bytes, and words (16-bit integers). For example, the first byte is accessed using ".ub1". ApossC uses 0-based numbering. So for ApossC, the first byte is accessed using ".ub0".
- ApossC does not support the ".b", ".w", or ".l" bit operators.
- If an expression is used to identify the bit, byte, or word number, then ApossC requires the expression to be specified inside "[...]" characters. The only exception to this is when the number is a constant. In this case, there may be no whitespace between the prefix and the number.
- If an expression is used to identify the bit, byte, or word number, then APOSS uses 1-based numbering. ApossC uses 0-based numbers. The expression will need to take this into account.
- APOSS allows the type prefix to be omitted. In this case, the prefix defaults to ".i" (i.e. bits). In ApossC, the prefix is mandatory.

Examples:

APOSS	ApossC
`var.i1`	`var.i0`
`var.i32`	`var.i31`
`var.ub1`	`var.ub0`
`var.ub4`	`var.ub3`
`var.ub 4`	`var.ub3`
`var.ub expr`	`var.ub[expr]`
`var.expr`	`var.i[expr]`
`var.ZBCOS_STD_ENABLE`	`var.i[ZBCOS_STD_ENABLE-1]`

4.2 Special Arrays

The ApossC Programming Language support the following special types of arrays:

ANSI Arrays	Controllers have native support only for Unicode character strings. In some cases, programs may need to work with ANSI character strings rather than Unicode strings. ANSI arrays provide a certain amount of support for these character strings.
DIM Arrays	Normal global and local arrays exist only when a user program is running. DIM arrays provide a mechanism for storing arrays in

EEPROM memory. This allows arrays to persist between program executions and even through power down/up cycles.

4.2.1 ANSI Arrays

In some cases, user programs may need to work with ANSI character strings rather than Unicode strings. These strings are expected to be packed into **long** arrays, four 1-byte characters per **long**, in little-endian order with 0x00 indicating the end-of-string. For example:

```
long str[4];

str[1] = 0x44434241;   // ABCD
str[2] = 0x64636261;   // abcd
str[3] = 0x34333231;   // 1234
str[4] = 0x00868024;   // $€†
```

When working with arrays such as this, they are treated in the same way as any other **long** arrays (i.e. as an array of **long** values). The controller has no way of knowing that the array contains a character string. This can be an inconvenience, particularly with respect to printing the character strings. For example, printing the array will cause the array to be interpretted as the Unicode characters 0x4241, 0x6261, 0x3231, and 0x8024.

Two standard functions are provided to assist users when working with ANSI arrays:

ansicpy Converts an ANSI string into Unicode characters and copies them into another array (usually a **string** or **wchar** array).

ansilen Returns the number of characters in an ANSI string.

The above string can then be printed correctly:

```
long  str[4];
wchar msg[20];

str[1] = 0x44434241;   // ABCD
str[2] = 0x64636261;   // abcd
str[3] = 0x34333231;   // 1234
str[4] = 0x00868024;   // $€†

ansicpy(msg, str);

print("msg = ", msg, "  length = ", ansilen(str));
```

will print:

```
msg = ABCDabcd1234$€†  length = 15
```

4.2.2 DIM Arrays

DIM arrays are similar to normal arrays, but they have the following important differences:

- The size of each DIM array is limited only by the amount of memory available on the controller. For normal arrays, the size is limited by the amount of variable and stack memory allocated for the application program (see Compiler).
- The values are "permanent" and do not need to be initialized each time a user program starts. This includes when the controller is powered off and on again. See User Data Arrays for a description of how this is

handled.

- Since the values are "permanent", DIM arrays can be used to share data between multiple user programs. Note that it is the *order* in which the DIM arrays are defined within programs, *not* their names, that is important. Specifically, the first DIM array defined in each program always refers to the first DIM array in memory, the second DIM array to the second array in memory, and so forth, regardless of the array name used in the program.
- DIM arrays may be initialized by downloading configuration files, for example by using **Controller** → **Parameters** → **Restore from file** or from the CAM Editor. They may also be initialized using the #config statement.
- DIM array data may be accessed by external devices using SDOs (see 2100 - Arrays and 21FF - General Array Access).
- When a user program modifies DIM array values, the changes are temporary. However, the user program can make the changes permanent by executing the Save command.
- DIM arrays can be deleted with **Controller** → **Reset** → **Arrays** or by executing the Delete command.

DIM arrays are declared using the "dim" declaration specification. For example:

```
dim long   target1[20], target2[20], target3[20];
dim double offsets[50], parameter[10];
dim wchar  message[100];
dim long   matrix[3][3];
```

As with standard C, both the "dim" specification and the data type will apply to all arrays up to the terminating ';'.

DIM arrays may be specified anywhere that global variables may be specified (i.e. outside of functions). There is no restriction that they precede other "code-generating" statements in the program or other global declarations. Only the order of the arrays in important (see below).

> The number of DIM arrays that can be used is limited. The limitation is controller dependent. Please check Controller Dependent Feature List for the limits.

> ApossC supports n-dimensional arrays (e.g. "long mtx[3][3][4]").

> The ApossIDE tools (e.g. Array Editor) always interpret and display array elements as data type **long**, since these tools do not know about the data type of the array. The array data type is only determined at compile time and can vary from one application program to another. Similarly, a two-dimensional DIM arrays will appear as a one-dimensional array with a size given by the total size of the two-dimensional array.

Starting Behaviour

If a user program containing DIM statements is started and no DIM arrays already exist, then the DIM arrays will be created. If the program is started and DIM arrays *do* already exist (i.e. perhaps from a previous user program), then the sizes of the existing arrays are compared to the sizes of the corresponding arrays declared in the new user program. As long as the sizes of the arrays, *in the order declared in the user program*, matches the existing DIM arrays, then the user program is started. If the sizes differ, then the program is stopped with Error 76.

If the user program declares more DIM arrays than already exist on the controller, then the extra DIM arrays are created. If the program declares fewer DIM arrays than already exist, then the extra existing arrays are ignored; they are *not* deleted.

Note that only the position and size of the arrays (in longs) is checked; the data types of the arrays are *not* checked. Also, the controller does not save the names of DIM arrays; DIM array names are entirely local to the user program. Hence, a different user program containing this statement will also be valid:

```
dim long xpos[20], ypos[20], zpos[20], offs[100];
```

In this case, the "xpos" array is identical to the "target1" array, "ypos" to "target2", and so forth. Note that the original "offsets" array contains 50 doubles and is therefore 100 longs in length. This will correctly match the length of the "offs" array which contains 100 long values.

4.3 Interrupt Handling

The controller supports many different types of interrupts that allow programs to respond to real-time events. For example, changes to digital inputs, communication events, positioning events, and general errors can all be trapped and then handled by the program.

Controller interrupts behave as expected: They will interrupt a running user program at *any* point in the program (even mid-way through the execution of a single statement) and invoke an interrupt handling routine. Once the handler has dealt with the interrupt and returned, then execution of the user program will continue exactly where it left off.

The following programming steps are necessary to be able to use interrupts in a user program:

1. An interrupt handling routine must be implemented for each interrupt of interest. In ApossC, they are implemented as void functions with no parameters.
2. The interrupt handling routine must be assigned to the specific type of interrupt that it is intended to handle. In ApossC, this is done using the InterruptSetup function.
3. Once the interrupt is not used anymore it can be deleted with the InterruptDelete function.
4. Interrupt handling is enabled by default when a user program starts. However, interrupts may be disabled at any time using the InterruptDisable function. They may be enabled again using the InterruptEnable function.

The following types of interrupts are available:

ERROR	Interrupt when any error is detected. Note that the error interrupt handling is slightly different from all other interrupt handling. See Error Handling for more information.
TIME	Interrupt after a certain length of time.
PERIOD	Interrupt on a regular time interval.
DIGINPUT	Interrupt at the edges of a change to a digital input.
SERIALCHAR	Interrupt when a serial interface character is received.
ETHERNET	Interrupt when a ethernet telegram is received.
CANINPUT	Interrupt when a CAN telegram of a specific type is received.
PARAM	Interrupt when a parameter value changes.
STATBIT	Interrupt when an axis status bit is set.
Position	Interrupt when a specific motor position is reached.
SETOUT	Simulate a CAM Box based on motor position.

Working examples of how to use many of these interrupt types, can be found using **File** → **Sample**. In particular, look at the sample programs grouped under "State Machine Samples/Operational Events".

4.3.1 Interrupt Behaviour

Response Times

The interrupt response time is the length of time between when the actual interrupt event happened and when the interrupt handler routine is actually called to handle the event.

During the execution of a user program, the existence of an interrupt condition is normally checked every **1 ms**. If found, then the interrupt handler is called. However, the check is always delayed until the current machine code command is complete. This is done to prevent the interrupt from occurring *within* the execution of a single machine code command. This means that the effective response time may be slightly longer than 1 ms.

It is important to note that most single user program statements will consist of multiple machine code commands. This means the interrupt handler may be called even mid-way through the execution of a single user statement. See Interrupt Routines for more details.

For user commands that take some time (such as Delay or AxisWaitReached), the interrupt check continues to be done *during* the waiting period. If the interrupt is triggered, then the command is interrupted and continued again once the interrupt has been processed.

> In general, it is important to make sure that interrupt routines execute in as short a time as possible. For routines that handle the PERIOD interrupt, it is particularly important to ensure that the routine does not execute for longer than the interrupt period. Otherwise, the interrupt handler will be called continuously and other parts of the user program will never get a chance to execute.

Interrupt Priorities

The controller is designed to handle multiple simultaneous interrupt events. However, the interrupt handlers are called sequentially and it is not possible for one interrupt to be suspended by another. So while one interrupt is being processed, any additional interrupts will be processed once the current interrupt handler has completed. Interrupt events are not normally lost (see below for exceptions).

The only exception to the sequential interrupt processing, is the ERROR interrupt. These interrupts *will* interrupt the processing of other interrupts. An ERROR interrupt cannot be suspended by another interrupt and it is not possible to use InterruptDisable to disable ERROR interrupts. For more information, see Error Handling.

If two or more interrupt events are waiting to be processed, then the processing is prioritized as follows:

ERROR always first and will interrupt other interrupts
DIGINPUT comes before
Position before
ETHERNET before
STATBIT before
PARAM before
CANINPUT before
SERIALCHAR before
TIME / PERIOD

DIGINPUT, STATBIT

If two or more of these interrupts occur simultaneously (e.g. digital input 1 and 2 both go high at the same time), then the one with the lower number is processed first. Once the first processing is complete, then the handler will be called again to process the second and so forth. If the same interrupt occurs again *while* the interrupt is being processed (e.g. digital input 1 goes high and then low again), then this is recorded and the handler will be called again once the current processing is complete.

Interrupts will be lost if the same interrupt occurs more than once before the first occurance can be processed. For example, if digital input 1 goes high and then low again before the first transition is

processed, then both transitions will be lost, neither interrupt will be generated, and it will appear as if the input never changed.

PARAM

If several of these interrupts occur simultaneously, then they are processed according to the sequence of PARAM setup commands in the program.

Interrupts will be lost if the same interrupt occurs more than once before the first occurance can be processed. For example, if a parameter values changes and then changes again before the first value change is processed, then once the interrupt handler is called, it will see only the second value. It will appear as if the first value change never happened. Similarly, if the values changes and then changes back to the original value, then it will appear as if the value never changed and the interrupt handler will never be called.

CANINPUT

During the processing of a CAN interrupt, if other CAN telegrams arrive, then these are stored in a buffer. Processing of the telegrams will be done sequentially as long as the buffer contains telegrams. Up to 32 telegrams may be buffered. Telegrams will be lost if the buffer is filled faster than it can be emptied.

TIME, PERIOD

If several time-based interrupts occur at the same time, then they are processed according to the sequence of the PERIOD and TIME setup commands in the program.

4.3.2 Interrupt Routines

Interrupt handlers for ApossC are written as void functions with no parameters. These functions are then set up as interrupt handlers using the InterruptSetup function. For example, the following is a very simple ApossC program that will trigger a single timer interrupt after 3 seconds:

```
long main(void)
{
    // Assign the TimeIsUp function to a 3-second timer.
    InterruptSetup(TIME, TimeIsUp, 3000);

    // Run forever.
    while (1);

    return(0);
}

void TimeIsUp(void)
{
    print("The time is now up.");

    return;
}
```

It is *not* necessary to explicitly declare function prototypes for interrupt handler functions. An appropriate prototype is declared automatically for the function when its name is referenced in the InterruptSetup (or InterruptDelete) function.

The InterruptSetup function should be used as early in the program as possible to set up interrupt handlers so

> that they are valid for the entire program.

Minimizing Execution Time

In general, interrupt handler routines are expected to handle the interrupt as quickly as possible and then return to "normal" execution. Applications should not be designed to spend long periods of time within interrupt handler routines. Since interrupts are disabled while the interrupt handler is executing, spending a long time within the routine increases the probablility that other, perhaps more important, interrupts are lost or are handled too late. For example, if a timer interrupt is used to execute a periodic, slow process and the application relies on digital input pulses, then the application will miss any pulse that occurs while the periodic process is busy.

Enabling and Disabling Interrupts

When a routine is called to handle an interrupt, then all enabled interrupts will be automatically disabled. These interrupts will be automatically enabled again when the routine returns. This has the following implications:

1. It is not necessary for the routine to do its own interrupt disabling and enabling.
2. An interrupt handler routine will never be interrupted by another interrupt (since interrupts are disabled). It will always run to completion. An exception to this is ERROR interrupts (see Error Handling).
3. An interrupt handler routine should not enable any interrupts.

Use of Global Variables

Extreme care must be taken if interrupt handler routines assign values to global variables that are used outside of the interrupt routine. The reason for this is that interrupts can be triggered even *within* the execution of a single user program statement. As an example, consider the following simplified program. It expects to increment a "target" position during normal processing (i.e. outside of the interrupt rutine), but then reset the target back to 0 every 5 seconds.

```
long target = 0;

void ResetTarget(void)
{
    target = 0;
    return;
}

long main(void)
{
    // Assign the ResetTarget function to a 5-second periodic timer.
    InterruptSetup(PERIOD, ResetTarget, 5000);

    // Run forever.
    while (1)
        {
        ... Some processing is done here ...
        target = target + 1000;   // Increment to next target
position.
        }

    return(0);
}
```

The statement that increments the target position, is actually executed as a series of machine code commands that use temporary registers to store partial results during the expression evaluation. Only at the end of the sequence is the result stored in "target". Now, consider what happens if the timer interrupt is triggered in the middle of the expression evaluation:

1. The expression is evaluated and placed in a temporary register.
2. The interrupt is triggered, which sets "target" to 0.
3. The interrupt routine returns and execution continues where it left off with the expression evaluation.
4. The expression result is extracted from the temporary register and saved in "target". This *overwrites* the 0 value and the target reset is effectively lost!

One possible solution is to disable interrupts while the target is being incremented:

```
while (1)
    {
    ... Some processing is done here ...
    InterruptDisable(PERIOD);
    target = target + 1000;  // Increment to next target position.
    InterruptEnable(PERIOD);
    }
```

This will allow the target increment to be fully completed before the interrupt is processed. However, this is an ugly and highly bug-prone scheme, easily leading to intermittent problems that could be *very* difficult to diagnose.

> **It is highly recommended in cases such as this, that a different architectural design be created that does not suffer from the "overwrite" problem.**

4.3.3 Interrupts

4.3.3.1 CANINPUT

Summary Interrupt when a specific CAN or serial telegram is received.

Syntax
```
InterruptSetup(CANINPUT, name, id);
InterruptDelete(CANINPUT);
```

Parameter

`name`	=	Name of interrupt routine.
`id`	=	Handle of a CAN telegram; CAN Node-Id of the module sending the PDO; receive serial telegram.

Return Value -

Description The first form of the command will set up the interrupt handler function. This will then be automatically called when a specific CAN telegram or RS485/RS232 telegram arrives. See Interrupt Routines for a description of how to write interrupt handler routines.

> Only one CANINPUT interrupt is available.

The second form of the command may be used to delete a previously set up interrupt handler routine. Once deleted, interrupts will no longer trigger the interrupt handler to be called.

The source is defined by the 'id' parameter:

id = 1-99

In this case, 'id' is a handle returned by the DefCanIn command. The interrupt is called as soon as a telegram is

Summary Interrupt when a specific CAN or serial telegram is received.

received in this mailbox. The telegram can then be read in the interrupt routine with the CanIn command.

Example:

```
handle = DefCanIn(0x701, 2);
InterruptSetup(CANINPUT, CanHandler, handle);

...

void CanHandler(void)
{
    status = CanIn(handle, 0, 0, msb, lsb);
    return;
}
```

id = 100*n

This will react on reception of an Rx PDO from a device with CAN Node-id n.

id = -1

This will react to an incoming telegram on the RS485/RS232 line. The function will be called when a complete telegram (ending with EOT or after idle detection) was received by the low level RS485/RS232 functions. See SerialSetup for details.

Command Group Interrupt Functions

Syntax Example

```
InterruptSetup(CANINPUT, Receive, 0);
InterruptDelete(CANINPUT);
```

Program Sample See **File** → **Sample**: Communication → ISR_CanIn.mc

4.3.3.2 ERROR

Summary Interrupt when an error is detected.

Syntax

```
InterruptSetup(ERROR, name);
InterruptDelete(ERROR);
```

Parameter `name` = Name of interrupt routine.

Return Value -

Description The first form of the command will set up the interrupt handler function. This will then be automatically called when an error is detected. See Interrupt Routines for a description of how to write interrupt handler routines.

> If no interrupt handler is assigned to handle **ERROR** interrupts, then the current program will be automatically aborted when an error is detected.

The second form of the command may be used to delete a previously set up interrupt handler routine. Once deleted, interrupts will no longer trigger the interrupt handler to be called.

The detection of an error condition and the invoking of the error handler, requires a maximum of 2 milliseconds.

The error handler routine must take whatever actions are necessary to deal with the error. The following are possible actions:

- Use ErrorClear to clear the error.
- Execute some recovery action, based on the type of error that occurred.
- Wait for user intervention.
- Abort the program using Exit in the case of unrecoverable errors.

> If the error handler is exited with the error still active (i.e. because ErrorClear was not called) or a subsequent error has occurred, then a new interrupt call takes place as soon as the current handler returns.

Summary Interrupt when an error is detected.

> The **ERROR** interrupt does not terminate the Home and Index commands. These commands will continue after the error has been cleared. To prevent this, the error interrupt handler can start a 1 millisecond TIME interrupt. The timer interrupt can then be used to terminate the commands.

> **ERROR** interrupts cannot be interrupted by other interrupts.

Command Group Interrupt Functions

Syntax Example

```
InterruptSetup(ERROR, ErrHandle);
...
void ErrHandle(void)
{
    ErrorClear();
    return;
}
```

Program Sample See **File** → **Sample**: Miscellaneous Samples → ISR_Error.mc

4.3.3.3 ETHERNET

Summary Interrupt when a telegram is received via the ethernet interface.

Syntax

```
InterruptSetup(ETHERNET, name, handle);
InterruptDelete(ETHERNET, handle);
```

Parameter

`name`	=	Name of interrupt routine.
`handle`	=	The value returned from function call to EthernetOpenServer or EthernetOpenClient.

Return Value -

Description The first form of the command will set up the interrupt handler function. This will then be automatically called when when a telegram is received via the ethernet interface. See Interrupt Routines for a description of how to write interrupt handler routines.

See Ethernet Commands for a description of how to set up ethernet interfaces.

The second form of the command may be used to delete a previously set up interrupt handler routine. Once deleted, interrupts will no longer trigger the interrupt handler to be called.

Command Group Interrupt Functions

Syntax Example

```
handle = EthernetOpenServer(PROT_TCP, 23);
InterruptSetup(ETHERNET, EthernetHandler, handle);

...

void EthernetHandler(void)
{
    result = EthernetReceiveTelegram(handle, array);
    return;
}
```

4.3.3.4 DIGINPUT

Summary Interrupt when a digital input changes.

Syntax

```
InterruptSetup(DIGINPUT, name, n, dir);
InterruptDelete(DIGINPUT, n, dir);
```

Parameter

`name`	=	Name of interrupt routine.
`n`	=	Input number to be monitored

Summary Interrupt when a digital input changes.

`dir` = Direction of change:
RISING = Interrupt on rising edge
FALLING = Interrupt on falling edge

Return Value -

Description The first form of the command will set up the interrupt handler function. This will then be automatically called when an edge is detected at the monitored input. Note that the interrupt is edge-triggered and not level-triggered. See Interrupt Routines for a description of how to write interrupt handler routines.

> A minimum signal length of about 1 ms (hardware dependent) is necessary for the reliable identification of a level change. More information regarding the input circuit and input technical data can be found in the hardware manuals.

The second form of the command may be used to delete a previously set up interrupt handler routine. Once deleted, interrupts will no longer trigger the interrupt handler to be called.

A maximum of two interrupt handlers per input may be defined (i.e. one for the rising edge and one for the falling edge). The handlers assigned to the rising and falling interrupts may be different:

```
InterruptSetup(DIGINPUT, PosEdge, 1, RISING);
InterruptSetup(DIGINPUT, NegEdge, 1, FALLING);
```

> The interupt on a digital input works also when this inputs is linked over a bus module from a CAN or EtherCAT bus.

> The identification of an interrupt and the invoking of the handler, requires a maximum of 2 milliseconds.

Priority If a number of interrupts occur simultaneously, then the handler assigned to the lowest input number is executed first. The other interrupts will be processed afterwards. If the same interrupt occurs while the interrupt handler is executing, then it will be ignored and thus lost.

Command Group Interrupt Functions

Syntax Example

```
InterruptSetup(DIGINPUT, PosIn, 4, RISING);
InterruptSetup(DIGINPUT, NegIn, 5, FALLING);
InterruptDelete(DIGINPUT, 4, RISING);
InterruptDelete(DIGINPUT, 5, FALLING);
```

Program Sample See **File** → **Sample**: Digital Input/Output → ISR_Input.mc

4.3.3.5 PARAM

Summary Interrupt when a parameter value changes.

Syntax

```
InterruptSetup(PARAM, name, indx);
InterruptDelete(PARAM, indx);
```

Parameter

`name` = Name of interrupt routine.

`indx` = Sysvar index of parameter.

Return Value -

Description The first form of the command will set up the interrupt handler function. This will then be automatically called when the parameter value changes. See Interrupt Routines for a description of how to write interrupt handler routines.

The second form of the command may be used to delete a previously set up interrupt handler routine. Once deleted, parameter changes will no longer trigger the interrupt handler to be called.

The instruction InterruptSetup(PARAM, ...) can be used to respond when parameters are altered and to call up a function.

> The handler will only react to a write action to an SDO, e.g. when a PLC writes an SDO in the controller over CAN,

Summary Interrupt when a parameter value changes.

> or when the application program writes an SDO by the Sysvar command.
>
> The handler cannot be used to react to process data which is altered internally by the firmware, like e.g. REG_ACTPOS.

The InterruptSetup(PARAM, ...) for array elements (e.g. InterruptSetup(PARAM, ..., 0x01210005)) also works but it is limited. Only one array element can be used in an InterruptSetup(PARAM, ...) per linked array.

Example:

```
dim long MyArray[200];

////////////////////////
void ArrayChanged(void){
        print("Array has changed");
}

////////////////////////
void main(void){

        InterruptSetup(PARAM, ArrayChanged, 0x01210005);
        while (1) {

        }

        return(0);
}
```

The InterruptSetup(PARAM, ...) will become active if the array element is written by an incoming PDO or by an SDO write to the SDO 0x012100ss with the correct subindex. This works also when linked with LinkPdo.

It will not become active if the array is written in the program by test[nn] = value. And it will not become active if the whole array gets overwritten by an array write using the 0x23FF SDO.

> Interrupt handlers may be assigned to a maximum of 10 different parameters simultaneously.

Priority If the same interrupt occurs while the interrupt handler is executing, then it will be ignored and thus lost.

Command Group Interrupt Functions

Syntax Example 1 Interrupt each time user parameter 1 is modified.

```
InterruptSetup(PARAM, UserParamUpdate, USER_PARAM_INDEX(1));

void UserParamUpdate(void)
{
    print("User Parameter 1 updated: ", USER_PARAM(1));
    return;
}
```

Syntax Example 2 Interrupt each time the CANopen DS402 "Mode of operation" changes.

```
InterruptSetup(PARAM, OpModeUpdate, 0x01606000);

void OpModeUpdate(void)
{
    print("New DS402 Operation Mode: ",Sysvar[0x01606000]);
    // Execute an action depending on the new operating mode
    switch (Sysvar[0x01604000])
        {
        case 1:  print("Profile position mode");
                 // Action ....
```

Summary Interrupt when a parameter value changes.

```
                    break;
            case 2:  print("Velocity mode");
                     // Action ....
                     break;
            default: print("Operation Mode is not supported");
            }
        return;
    }
```

Syntax Example 3 Link the RxPdo with the user parameter 1 and configure an interrupt to trigger each time it changes.

```
LinkPdo(1, 32, USER_PARAM_INDEX(1), 0);
InterruptSetup(PARAM, UserParamUpdate, USER_PARAM_INDEX(1));

void UserParamUpdate(void)
{
    print("User Parameter 1 updated: ", USER_PARAM(1));
    return;
}
```

Program Sample See **File** → **Sample**: State Machine Samples → Operational Events → ISR_Param.mc

4.3.3.6 PERIOD

Summary Interrupt at regular intervals.

Syntax

```
InterruptSetup(PERIOD, name, n);
InterruptDelete(PERIOD, name);
```

Parameter

name	=	Name of interrupt routine.
n ≥ 2	=	Period in ms, after which the interrupt is repeated.
n = 0	=	Disable interrupt which is associated with 'name'.

Return Value -

Description The first form of the command will set up the interrupt handler function. An interrupt is then triggered at regular time intervals. See Interrupt Routines for a description of how to write interrupt handler routines.

The second form of the command may be used to delete a previously set up interrupt handler routine. Once deleted, interrupts will no longer trigger the interrupt handler to be called.

A previously set up interrupt may also be deleted by setting the period to 0. This method is not recommended, since it is less clear than using InterruptDelete.

The cycle time for a periodic interrupt can be redefined at any time by executing the set up again with the same routine and a different time period. Note that setting the cycle time to 0 will delete the interrupt.

The time when the first interrupt should take place is calculated at when the interrupt is set up:

START_TIME = TIME + PERIOD

As soon as this time has been reached, the function is executed and the next start time is calculated as follows:

START_TIME = START_TIME + PERIOD

This ensures that the call intervals are really the same, since the execution time does not influence the calculation. However, this means that the user must make sure that the interrupt period is longer than the execution time required by the interrupt handler. Otherwise, the interrupt handler would be executed continuously and no other part of the program would ever be executed.

> If the cycle time is too short, it can slow down the execution of the main routine of the application program significantly. In principle, the execution time of the periodic interrupt must always be shorter than the defined cycle time. Otherwise, the subroutine cannot be called often enough to maintain the requested cycle time. The

Summary Interrupt at regular intervals.

> rule of thumb is: the defined cycle time must be at least double interrupt execution time.
>
> The necessary execution time for interrupts can be measured with Time. For very quick interrupt routines (e.g. lasting just one or some milliseconds), a more accurate measurement of the execution time can be done using an Oscilloscope and setting an output at start of the interrupt and resetting the output at the end of the interrupt routine.

> The precision with which the time is kept, depends on the hardware used. Typically the precision is ±1 ms.

Command Group Interrupt Functions

Syntax Example

```
InterruptSetup(PERIOD, OneSecond, 1000);
InterruptSetup(PERIOD, OneSecond, 0);  // Delete interrupt.
InterruptDelete(PERIOD, OneSecond);    // Delete interrupt.
```

Program Sample See **File** → **Sample**: State Machine Samples → Operational Events → ISR_Period.mc

4.3.3.7 Position

Summary Interrupt when a motor reaches a specific position.

Syntax

```
InterruptSetup(sign postype, name, axis, pos);
InterruptSetup(sign postype, name, axis, pos, ONCE);
InterruptDelete(POSINT, name);
```

Parameter

sign +	=	Rising edge or when the position is passed in positive direction.
sign -	=	Falling edge or when the position is passed in negative direction. 100 xxx 0 <- - - - - - positive direction - - - - - - > negative direction
postype	=	APOS - when the slave position xxx is passed. IPOS - when the distance between the last marker position and the actual position is reached. MAPOS - when the master actual position [mu] is passed. MCPOS - when the master command position [mu] is passed. MIPOS - when the distance between two markers is reached.
name	=	Name of interrupt routine.
axis	=	Axis module number (see Command Reference).
pos	=	Position, depending on the command, in user units [UU], master user units [MU], or curve units [CU]. Note: This value is ignored if an interrupt is being deleted.
ONCE	=	The interrupt is triggered only once and then it is automatically deleted.

Return Value -

Description The first form of the command will set up the interrupt handler function. This will then be automatically called when the specified position is reached or passed. See Interrupt Routines for a description of how to write interrupt handler routines.

The second form of the command may be used to delete a previously set up interrupt handler routine. Once deleted, interrupts will no longer trigger the interrupt handler to be called.

All position interrupts support multiple axes, multiple position types, and multiple positions. If a position is already used for another interrupt, then the position automatically is increased by 1 until a free interrupt position is found. Examples:

```
InterruptSetup(+APOS, a1test,  0, 10000);
InterruptSetup(-APOS, a1mtest, 0, 10000);
InterruptSetup(+APOS, a2test,  1, 12000);
InterruptSetup(-APOS, a2mtest, 1, 12000);
```

Summary Interrupt when a motor reaches a specific position.

Up to 30 position interrupts may be specified simultaneously.

> If a position interrupt is defined using a position which lies *behind an overflow* of the encoder, then this is handled correctly. The interrupt mechansim automatically adjusts the position internally. For example, if no POSFACTs are set, then the following is handled correctly:
>
> ```
> testpos = 0x3FFFFFF0;    // Shortly before overflow
> newpos  = testpos + 200; // After overflow
> InterruptSetup(+APOS, myprog, 0, newpos);
> ```
>
> The same is true if POSFACT_Z and POSFACT_N are set and the user value will cause an overflow of the internal qc positions.
>
> See Encoder Position Values for a description of encoder overflows.

When deleting a position interrupt, *all* position interrupts associated with the specified interrupt handler name will be deleted. It is not possible to delete an individual interrupt if multiple interrupts are using the same interrupt handler. Please see example 6.

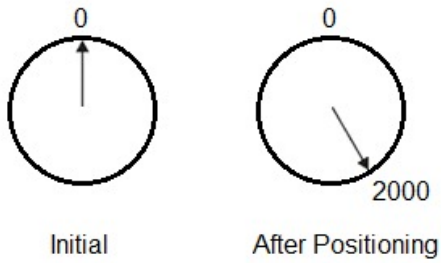
It is possible to "re-route" an existing position interrupt to another interrupt handler routine. This does not define a new interrupt; it merely modifies the routine that will be executed when an interrupt is detected. To do this, simple re-execute the original setup command with the new handler name. Please see example 7.

APOS Interrupt

Interrupt when the slave position xxx (UU) has been passed in positive or negative direction. The instruction can be useful for positioning and synchronization controls, as well as for CAM controls and CAM boxes. For example, in order to replace the increasing slave position in the case of open curves after each cycle by a recurring reference point.

Example:

```
Cstart(0);
InterruptSetup(+APOS, Stop, 0, 2000);

void Stop(void)
{
    Cstop(0);
    return;
}
```



The drive stops once it reaches the position 2000.

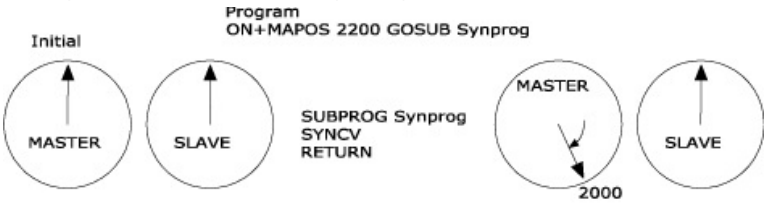
IPOS Interrupt

This position interrupt looks at the distance between the last marker position and the actual position. It is important that SYNCMPULSS is set correctly. This information is used for detection of overflow as well as for backward driving. In the case of backward driving, the information used is (SYNCMPULSS + (Apos - Ipos)) instead of (Apos - Ipos).

MAPOS Interrupt

Interrupt when the master actual position xxx (MU) has been passed in positive or negative direction. For example, in order to set an output at any point in the case of a linear drive (slave) with a traversing range from 0 to 10000 UU.

Summary Interrupt when a motor reaches a specific position.



Here according to the program Velocity Synchronization starts after Master has reached 2200 qc in positive direction. Then the slave and master move in SyncVel.

MCPOS Interrupt

Interrupt when the master command position xxx (MU) is passed.

Typical for CAM controls, is to use MCPOS to call a routine when a specific master position (MU) has been passed in positive or negative direction. This allows not only the realization of CAM boxes, but also the execution of tasks that are much more complex. For example, one could change parameters online depending on the position.

> A DefMasterCpos or a SyncSetCurve must always be placed before setting up the MCPOS interrupt, since otherwise the curve position is not known.

MIPOS Interrupt

Interrupt when the distance between two markers is reached.

This position interrupt looks at the distance between the last marker position and the actual position. It is important that SYNCMPULSM is set correctly. This information is used for detection of overflow as well as for backward driving. In the case of backward driving, the information used is (SYNCMPULSM + (Mapos - Mipos)) instead of (Mapos - Mipos).

Command Group Interrupt Functions

Syntax Example 1 Call the interrupt routine when slave position 800 is passed in the negative direction.

```
InterruptSetup(-APOS, Name, 0, 800);
```

Syntax Example 2 In this example, two markers are separated by a distance of 20000 qc. Assume that the first marker is at position 0. Then "prog1" will be called at 5000, 25000, 45000,... and "prog2" will be called at 15000, 35000, 55000,...

```
AXE_PARAM(0, SYNCMPULSS) = 20000;    // Marker separation
InterruptSetup(+IPOS, prog1, 0, 5000);
InterruptSetup(+IPOS, prog2, 0, 15000);
```

Syntax Example 3 Always call the interrupt routine at position 1200.

```
InterruptSetup(+MAPOS, name, 0, 1200);
```

Syntax Example 4 In this example, two markers are separated by a distance of 20000 qc. Assume that the first marker is at position 0. Then "prog1" will be called at 5000, 25000, 45000,... and "prog2" will be called at 15000, 35000, 55000,...

```
AXE_PARAM(0, SYNCMPULSM) = 20000;    // Marker separation
InterruptSetup(+MIPOS, prog1, 0, 5000);
InterruptSetup(+MIPOS, prog2, 0, 15000);
```

Syntax Example 5 Cardboard boxes are transported irregularly on a conveyor belt. By setting an output, the slave will always be started when the position 4500 is reached.

```
InterruptSetup(+MCPOS, output, 0, 4500);

void output(void)
{
    Out(3, 1);    // Output 3 on
    return;
}
```

Syntax Example 6 Deleting an interrupt:

Summary Interrupt when a motor reaches a specific position.

```
InterruptSetup(-APOS, hitinfo, 0, 20000);  // Interrupt #1
InterruptSetup(-APOS, hitinfo, 0, 10000);  // Interrupt #2
InterruptSetup(+APOS, hitinfo, 0, 10000);  // Interrupt #3
InterruptSetup(+APOS, hitzero, 0, 0);      // Interrupt #4
InterruptSetup(-APOS, hitzero, 0, 0);      // Interrupt #5
InterruptSetup(DIGINPUT, hitinfo, 3);      // Interrupt #6
...
InterruptDelete(POSINT, hitinfo);
...
InterruptSetup(+APOS, hitinfo, 0, 99999);  // New +APOS
```

When the "hitinfo" interrupt is deleted, then all position interrupts referencing "hitinfo" are deleted (i.e. #1, #2, #3). All position interrupts referencing *other* handlers remain unchanged (i.e. "hitzero" used by #4 and #5). As well, all non-position interrupts (i.e. #6) remain unchanged *even* if they use the same handler as the position interrupt being deleted.

A "deleted" interrupt handler (i.e. "hitinfo") can be reused again, simply by setting up a new interrupt to use it.

Syntax Example 7 Re-routing an interrupt:

```
InterruptSetup(-APOS, hitinfo, 0, 10000); // Interrupt #1
InterruptSetup(+APOS, hitinfo, 0, 10000); // Interrupt #2
...
InterruptSetup(+APOS, hitposdir, 0, 10000); // Re-routed #2
```

As soon as the second +APOS is executed, interrupt #2 is “re-routed” from "hitinfo" to "hitposdir". Note that the original definition of interrupt #1 is still valid and has not been changed since it uses a different position type.

Program Sample See **File** → **Sample**: Movement and Positioning → ISR_Position.mc

4.3.3.8 SERIALCHAR

Summary Interrupt when a character appears via the serial interface.

Syntax

```
InterruptSetup(SERIALCHAR, name);
InterruptDelete(SERIALCHAR);
```

Parameter `name` = Name of interrupt routine.

Return Value -

Description The first form of the command will set up the interrupt handler function. This will then be automatically called when when a key is pressed and a character appears via the serial interface. See Interrupt Routines for a description of how to write interrupt handler routines.

See Parity baudrate and interface parameters for a description of how to set up serial interfaces.

The second form of the command may be used to delete a previously set up interrupt handler routine. Once deleted, interrupts will no longer trigger the interrupt handler to be called.

Command Group Interrupt Functions

Syntax Example

```
InterruptSetup(SERIALCHAR, KeyHandler);
while (1);    // Endless loop

void KeyHandler(void)
{
    long key = SerialReadChar(-1);  // Don't wait for key
    print(key);
    return;
}
```

Program Sample See **File** → **Sample**: Communication → ISR_Keypressed.mc

4.3.3.9 SETOUT

Summary Simulate a CAM Box based on motor position.

Syntax

```
InterruptSetup(sign postype, SETOUT, axis, pos, outno);
InterruptSetup(sign postype, SETOUT, axis, pos, outno, ONCE);

InterruptSetup(sign postype, SETOUT, axis, pos, outno, inno);
InterruptSetup(sign postype, SETOUT, axis, pos, outno, inno, ONCE);

InterruptDelete(SETOUT, axis, outno);
```

Parameter

sign +	=	Rising edge or when the position is passed in positive direction.
sign -	=	Falling edge or when the position is passed in negative direction. 100 xxx 0 <- - - - - - positive direction - - - - - - > negative direction
postype	=	APOS - when the slave position xxx is passed. IPOS - when the distance between the last marker position and the actual position is reached. MAPOS - when the master actual position [mu] is passed. MCPOS - when the master command position [mu] is passed. MIPOS - when the distance between two markers is reached.
axis	=	Axis module number (see Command Reference).
pos	=	Position, depending on the command, in user units [UU], master user units [MU], or curve units [CU]. This is ignored by the DELETE command.
outno	=	Any valid output number (or the negative output number).
inno	=	Any valid input number (or the negative input number).
ONCE	=	The interrupt is triggered only once and then it is automatically deleted.

Return Value -

Description All position interrupt functions can use this feature which simulates a cam box. This is possible with all types of POSINTs.

In the first case, the output *outno* is either set or reset, depending on whether the *outno* is positive or negative.

In the second form, the output is set to the value of the input *inno*. (Or to the opposite of the input value, if either the *inno* or the *outno* is negative.). If both *outno* and *inno* are negative, that is the same as if they both were positive.

The advantage of those commands is that they are handled in the background and do not interrupt the application program. They are also faster than calling a subroutine which in turn then sets an output. Typical reaction time is below 1 ms.

The `InterruptDelete` command deletes all interrupts which set or reset the output *outno*.

> If the *outno* is positive in the above command, then only interrupts are deleted where the outno is set. If *outno* is negative, then only interrupts are deleted where the output is reset. So if you use both types of interrupt definitions, you must also have two delete commands for that.

Command Group Interrupt Functions

Syntax Example 1 In this example, output 1 is set to the value of input 2 at a position of 500 user units after the marker.

Then output 1 is set to 0 again at a position of 1000 qc after the marker.

```
AXE_PARAM(0, SYNCMPULSS) = 20000;    // Marker separation.
InterruptSetup(+IPOS, SETOUT, 0, 500, 1, 2);
InterruptSetup(+IPOS, SETOUT, 0, 1000, -1);
```

Syntax Example 2 Deleting a SETOUT interrupt:

Summary Simulate a CAM Box based on motor position.

```
AXE_PARAM(0, SYNCMPULSS) = 20000;    // Marker separation.
InterruptSetup(+IPOS, SETOUT, 0, 500, 1);
InterruptSetup(+IPOS, SETOUT, 0, 1000, -1);
...
(program)
...
InterruptDelete(SETOUT, 0, 1);
InterruptDelete(SETOUT, 0, -1);
```

4.3.3.10 STATBIT

Summary Interrupt when bit 'b' of the axis status word is set.

Syntax

```
InterruptSetup(STATBIT, name, axis, bit, dir);
InterruptDelete(STATBIT, axis, bit, dir);
```

Parameter

`name`	=	Name of interrupt routine.
`axis`	=	Axis module number (see Command Reference).
`bit`	=	Bit number of status word: 0 ≤ b ≤ 31 (see below).
`dir`	=	Direction of change: RISING = Interrupt on rising edge FALLING = Interrupt on falling edge

Return Value -

Description The first form of the command will set up the interrupt handler function. This will then be automatically called when an edge is detected at the monitored bit of the axis status word. Note that the interrupt is edge-triggered and not level-triggered. See Interrupt Routines for a description of how to write interrupt handler routines.

The second form of the command may be used to delete a previously set up interrupt handler routine. Once deleted, interrupts will no longer trigger the interrupt handler to be called.

It is possible to assign interrupt handlers to more than one status bit at the same time and to more than one axis at the same time. For example, an interrupt handler could be assigned to bit 2 and a different handler assigned to bit 3. However, it is not possible to assign both rising and falling handlers at the same time. For example, interrupt handlers cannot be assigned for both 2 RISING and 2 FALLING at the same time.

The meanings of the 32 bits of the status word are listed below. It is recommended that "SysDef.mh" be included and the standard #define names used, rather than using explicit numeric bit numbers in the application program.

Byte 3 (MSB): SyncStatus status byte.

Bit #	#define	Meaning
31		Reserved
30	STATBIT_SYNCSMERR	Slave marker error
29	STATBIT_SYNCMMERR	Master marker error
28	STATBIT_SYNCSMHIT	Slave marker hit
27	STATBIT_SYNCMMHIT	Master marker hit
26	STATBIT_SYNCACCUR	Sync. accuracy reached
25	STATBIT_SYNCFAULT	Sync. failed
24	STATBIT_SYNCREADY	Sync. ready

Byte 2: Reserved.

Byte 1: Byte 2 of the internal status (e.g. moving).

Bit #	#define	Meaning

Summary Interrupt when bit 'b' of the axis status word is set.

15	STATBIT_OVERSHOOT	AxisPosAbsStartTvel will overshoot or not reach target
14	STATBIT_CURVE_ERR	CAM sync. error
13	STATBIT_NEGLIMERR	Negative pos. sw limit (-> NEGLIMIT)
12	STATBIT_POSLIMERR	Positive pos. sw limit (-> POSLIMIT)
11	STATBIT_POSFLOAT	Pos. within REGWMAX / REGWMIN
10	STATBIT_OVERFLOWM	Overflow of master position
9	STATBIT_OVERFLOWS	Overflow of slave position
8	STATBIT_MOVING	Axis is moving

Byte 0 (LSB): Status byte of the axis controller.

Bit #	#define	Meaning
7	STATBIT_FLOATING	Pos. control switched off (= AxisControl(..,OFF) state)
6		Reserved
5		Reserved
4	STATBIT_MINDEX_HIT	Master Index observed
3	STATBIT_INDEX_HIT	Slave Index observed
2	STATBIT_POSREACHED	Position reached
1		Reserved
0		Reserved

Priority If a number of interrupts occur simultaneously, then the handler assigned to the lowest bit is executed first. The other interrupts will be processed afterwards. If the same interrupt occurs while the interrupt handler is executing, then it will be ignored and thus lost.

Command Group Interrupt Functions

Syntax Example

```
// Interrupt if error flag Master (axis module 0)
InterruptSetup(STATBIT, MarkerError, 0, STATBIT_SYNCMMERR, RISING);

void MarkerError(void)
{
    SyncStatusClear(32);
    // clear error flag SYNCMMERR
    // value 32 of command SyncStatusClear
    return;
}
```

Program Sample See **File** → **Sample**: Movement and Positioning → ISR_Statbit.mc

4.3.3.11 TIME

Summary Interrupt once after a specified period of time.

Syntax

```
InterruptSetup(TIME, name, n);
InterruptDelete(TIME, name);
```

Parameter

`name` = Name of interrupt routine.

`n` = Time in ms, after which the interrupt handler is called.

Return Value -

Summary Interrupt once after a specified period of time.

Description The first form of the command will set up the interrupt handler function. This will then be automatically called once after the specified time period expires. See Interrupt Routines for a description of how to write interrupt handler routines.

The second form of the command may be used to delete a previously set up interrupt handler routine. If the interrupt is deleted before the time period expires, then the interrupt handler will not be called.

A previously set up interrupt may also be deleted by setting the time to 0. This method is not recommended, since it is less clear than using `InterruptDelete`.

> The precision with which the time is kept, depends on the hardware used. Typically the precision is ±1 ms.

Command Group Interrupt Functions

Syntax Sample

```
DigOutput(1, ON);                // Light on
InterruptSetup(TIME, Off, 200);  // Light off after 200 ms

void Off(void)
{
    DigOutput(1, OFF);           // Light off
    return;
}
```

Program Sample See **File** → **Sample**: State Machine Samples → Operational Events → ISR_Period.mc

4.4 Error Handling

All real-world user programs should expect errors to occur and be prepared to handle them. Errors can happen for a wide range of causes, from programming bugs to operational errors such as running motors into mechanical limit switches. See Error Messages.

Errors are expected to be handled using the interrupt mechanism (see Interrupt Handling). The user should write an interrupt handler routine to handle errors and then assign that routine to handle error interrupts (see ERROR). Whenever an error is detected, the controller will generate an ERROR interrupt and call the error handler routine. The routine must do whatever is necessary to resolve the error and then return to normal program operation.

> If an ERROR interrupt is triggered *and* no handler routine has been assigned to ERROR interrupts, then the controller will go into an error state and program execution will be terminated. Any motor that is *not* stationary, will be stopped immediately *using the maximum possible torque*.

ERROR interrupts are treated slightly differently than all other interrupts:

1. ERROR interrupts have the highest priority.
2. They *cannot* be disabled with the InterruptDisable command.
3. They *will* interrupt other interrupt handlers.
4. All errors that occur *while* the error handler routine is executing, are ignored (i.e. the error handler itself cannot be interrupted).

The following special variables/functions are particularly valuable for error interrupt routines:

ErrorNo System function returning the current error number (see Error Messages).

> If SysDef.mh is included, then predefined keywords may be used, rather than error numbers, to identify specific errors. This will make the program much more readable.

ErrorAxis System function returning the axis number which triggered the current error (for axis-related errors).

ErrorInfo System function returning some additional information to the current error. The return value depends on the error number.

At some point before returning, error handler routines must execute the ErrorClear command to clear the error. If the error is not cleared when the routine returns, then the residual error will simply cause the error handler routine to be called again *immediately*.

When an error handler routine returns, execution of the user program will continue where it was interrupted *as if no error ever happened*. This is an extremely important consideration and will have a significant impact on how the application is designed and implemented. For example, if the application has only partially completed some process, then thought must be given to whether or not the process may be continued and to the mechanism by which it must be aborted if it cannot be continued. This is a *non-trivial* task and it will vary greatly depending on the type of application, the state of the application when the error happens, and the specific error that was triggered.

In worst-case scenarios, error handler routines can be designed to take advantage of the controller's Autostart mechanism. This will allow the program to set up some type of "repair or recovery" program and then terminate itself. The controller will then autostart the designated program.

Example

The following example program illustrates the basic mechanism for handling errors. It will move a motor forward in steps of 10000, looking for a limit switch. When a limit switch is hit, then the motor will be moved back to the start position and the process will be repeated.

Note the following points about this example:

1. The global "restart" variable is used to communicate between the error handler routine and the main program.
2. A minimum amount of time is spent inside the error handler routine. For example, the return to the start position is done in the main portion of the program and *not* within the error routine.
3. The program will abort if an unrecognized error occurs. This allows an autostart program to be started if one is configured.
4. User Parameters are used to communicate with any autostart program. Other methods are also possible. This is necessary since the autostart program will have no access to the internal variables of this program.

This program is designed to execute correctly using the Simulator. Start the program, wait a short while, and then click in the checkbox for digital input 1 to simulate the limit switch being triggered. Click again to clear the limit switch. The program will abort if the limit switch is not cleared quickly enough. If this happens, then simply click the digital input checkbox to clear it and restart the program.

```
#include <SysDef.mh>

void ErrorHandler(void);  // "ERROR" interrupt handler.

#define AXIS 0    // We will be using axis module #0.

// "restart" will be set to 1 if a limit switch is hit.
// This will cause us to move back to the start position.
```

```
long restart = 0;

long main(void)
{
    // Assign "ErrorHandler" as the subroutine for ERROR
    // interrupts.

    InterruptSetup(ERROR, ErrorHandler);

    // The limit switch will be connected to digital input 1.

    AXE_PARAM(AXIS, I_POSLIMITSW) = 1;

    // This statement is here so that this sample program
    // will execute on the Simulator with no motor connected.

    VIRTCOUNTIN_PARAM(AXIS, VIRTCNTIN_PISRC_COUNTER) =
AXE_PROCESS_SRCINDEX(AXIS1, REG_COMPOS);
    VIRTCOUNTIN_PARAM(AXIS, VIRTCNTIN_MODE) = VIRTCNTIN_MODE_SIMULATION_SLAVE;
    VIRTLATCH_PARAM(AXIS, VIRTLATCH_PISRC_COUNTER) = AXE_PROCESS_SRCINDEX(AXIS,
REG_ACTPOS);
    VIRTLATCH_PARAM(AXIS, VIRTLATCH_MODE) = VIRTLATCH_MODE_SIMULATOR;

    // Turn the motor on so that we can move.

    AxisControl(ON, AXIS);

    // Set the current position as the start position.

    DefOrigin(AXIS);

    // Run forever.

    while (1)
        {

        // Take a step forward. If we hit a limit switch,
        // then the AxisPosRelStart() will be terminated, the motor
        // stopped, and an error interrupt triggered. Once
        // the error handler completes, then we will return
        // here and execution will continue with the next
        // statement.

        AxisPosRelStart(AXIS, 10000);

        // Wait for the movement to complete.

        AxisWaitReached(AXIS);

        // If "restart" is not 0, then we have hit a limit
        // switch. Go back to the start position.

        if (restart != 0)
            {
            print("Returning to start...");
            AxisPosAbsStart(AXIS, 0);
            AxisWaitReached(AXIS);
            restart = 0;
```

```
            }

        // Wait a short while for the motor to settle.

        Delay(100);

        print("Position = ", Apos(AXIS));
        }

    return(0);
}

void ErrorHandler(void)
{
    // Save the error number and then clear the error.

    long err = ErrorNo();

    ErrorClear();

    // If this is NOT a limit switch error, then some
    // disaster happened. Save the error number in
    // User Parameter 1 and then abort. If Autostart
    // is set up, then the autostart program will be
    // started and it can determine what went wrong by
    // checking the value saved in the User Parameter.

    if (err != F_ENDSCHALT)
        {
        print("Error ", err, " detected!");
        print("Saving error and aborting program.");
        USER_PARAM(1) = err;
        Save(USRPARS);
        Exit(0);
        }

    // Set "restart" to 1 to indicate that we need to
    // restart. Then we need to turn the motor back
    // on again, since the error interrupt will have
    // automatically stopped the motor and turned it
    // off. After that, we can return to the main
    // program.

    print("Found limit switch at ", Apos(AXIS), ".");

    restart = 1;

    AxisControl(ON, AXIS);

    return;
}
```

4.5 Preprocessor

The Preprocessor is designed to make various modifications to the program before it is actually compiled. Its most important functions are:

1. Handling continuation lines.

2. Including "header files" or other common blocks of code that have been saved in other files.
3. Text substitution (macro expansion).
4. Conditional compilation.
5. Handling configuration files.

Preprocessor command keywords *must be lower case only*.

All Preprocessor commands must be specified on a *single* source line. If multiple lines are necessary, then continuation lines must be used.

4.5.1 Continuation lines

Any line that ends with a "\" will be continued on the next line. Note that "\" must be very *last* character on the line; it may not be followed even by whitespace characters or comments.

Example:

```
print(" Apos = ",Apos(0),\
      " Avel = ",Avel(0));
```

is the same as:

```
print(" Apos = ",Apos(0)," Avel = ",Avel(0));
```

Note that this is normally of limited use since both the ApossC language is designed to allow statements to be continued on following lines without the use of the "\" character. For example,

```
print(" Apos = ",Apos(0),
      " Avel = ",Avel(0));
```

are also equivalent to the above examples. However, see the exception described below for Preprocessor commands.

Quoted strings may not be continued using the "\" character. For example,

```
print("This is a long \
message string on two lines.");
```

is invalid. Quoted strings must be completely specified on a single line. However, ApossC allows an exception to this. If there are only whitespace characters (and comments) between the trailing quote and the next quote, then this will be treated as a single quoted string. For example,

```
print("This is a long "    "message string"    // Comment here.
      " in three parts and on two lines.");
```

is the same as

```
print("This is a long message string in three parts and on two lines.");
```

Similarly, single-line comments (i.e. using "//") may not be continued. For example,

```
value = 0;  // This is \
a comment on two lines.
```

is invalid.

If more than one line is necessary to completely specify any Preprocessor command (e.g. #define, #if, etc.), then the "\" continuation character *must* be used. For example:

```
#define TESTVALUE if (value > VALUEMAX)    \
                      value = VALUEMAX;    \
                  else if (value < 0)      \
                      value = 0;
```

4.5.2 #include

Files can be embedded within a file by using the following statements:

```
#include "file"
#include <file>
```

where "*file*" is the name of the file to be embedded and may include directories. Either "\" or "/" may be used for directories.

If the quote syntax is used, then the Preprocessor will search for the include file in the same directory in which the file that included it was found. This is normally the same directory in which the original ".mc" file was found. However, it could be another directory if nested include files are used that come from different directories. If the include file is not found in this directory, then the ApossIDE "system include" directory is searched. The ApossIDE system include directory is a subdirectory under the ApossIDE install directory.

If the angle bracket syntax is used, then the Preprocessor will search for the include file only in the ApossIDE "system include" directory.

Examples:

```
#include "defines.mh"
#include "Library/Structures.mh"
#include "Library\Structures.mh"
#include <SysDef.mh>
```

#include statements can be used in combination with #define statements to provide more programming flexibility. For example:

```
#define INCFILE "defines.mh"
#include INCFILE
```

Often it is undesirable to include the same include file multiple times in a single application program, however the use of nested include files can make this difficult to prevent. The use of "#include <SysDef.mh>" (see SysDef Defines) is a good example of this because this file is quite large and is often included by multiple include files. In these cases, it is common practice to use #define and #ifdef statements to exclude the contents of an include file if it has already been included once.

A simpler and more efficient method is to use:

```
#pragma once
```

See #pragma for more information.

4.5.3 #define, #undef

Simple text replacement can be done using the following statement:

```
#define name string
```

All instances of "*name*" that follow this statement will be replaced with "*string*". Note that "*string*" will contain all characters up to the end of the line. The line may be continued using the "\" line continuation character if necessary (see Continuation lines).

Definitions can be cleared again using the following statement:

```
#undef name
```

> Names should not be redefined to different strings. Doing so will be successful but will generate a warning message. Names should always be undefined before being redefined.

The following names are predefined:

__DATE__	The date when the program was compiled (as a character string).
__TIME__	The time when the program was compiled (as a character string).
__FILE__	The filename of the program (as a character string).
__LINE__	The current line number in the program (as a number).
__FIRMWARE__	The firmware version of the controller for which the compile is being done (as a single integer). This is normally the connected controller. For example, a firmware version of 9.1.23 will result in the name being defined as 90123. If the firmware version is unknown (e.g. there is no connected conntroller), then the name will be defined as 999999.
__APOSSC__	This is defined (but empty) if the ApossC compiler is being used. It is undefined if the APOSS compiler is being used.

> Please note that each "__" in the above names, consists of *two* underscore characters.

Example:

```
#define ARRAY_LEN 100
#define PRINT_MSG print(" Apos = ",Apos(0)," Avel = ",Avel(0))
dim long prm[ARRAY_LEN];
PRINT_MSG;
print("This is line ",__LINE__);
```

is the same as:

```
dim long prm[100];
print(" Apos = ",Apos(0)," Avel = ",Avel(0));
print("This is line ",5);
```

Text replacement is done recursively. Once the original string has been replaced, then the Preprocessor will look for other replacements within the new string. Note that text replacement is *not* done within the **#define** statement itself and is *not* done within strings enclosed in quotes.

Example:

```
#define VALUE 10
```

```
#define PRINT_MSG print("VALUE = ",VALUE);
PRINT_MSG;
#define VALUE 20
PRINT_MSG;
```

is the same as:

```
print("VALUE = ",10);
print("VALUE = ",20);
```

Defines may also include one or more arguments separated by commas as follows:

```
#define name(arg1,arg2) string
```

In this case, instances of "*arg*" will be replaced in "*string*".

Example:

```
#define SQUARE(x) ((x)*(x))
#define SUM_SQUARE(x,y) (SQUARE(x) + SQUARE(y))
i = SUM_SQUARE(3,4);
i = SUM_SQUARE(a,b);
```

is the same as:

```
i = (((3)*(3)) + ((4)*(4)));
i = (((a)*(a)) + ((b)*(b)));
```

4.5.4 #ifdef, #ifndef, #if

Program code can be conditionally compiled using any of the following statements:

`#ifdef name`	If "*name*" has been defined using **#define**.
`#ifndef name`	If "*name*" has *not* been defined using **#define**.
`#if integer-expression`	If "*integer-expression*" evaluates to non-0.

These can be followed by any number of statements and then **#endif**.

When **#if** is used, then the expression may be any expression consisting of integer constants, character constants (e.g. 'a'), macro definitions, parentheses, and any of the ApossC operators (i.e. +, -, *, /, %, >>, <<, >=, <=, >, <, ==, !=, !, ~, &, ^, |, &&, and ||). The normal operator precedence applies. The expression may also include the special **defined(...)** function. This will return 1 if the argument name has been previously defined by a **#define** statement and 0 otherwise. This behaves like **#ifdef**, but it may be used in an **#if** statement.

#else and **#elif** (for "else if") can also be used as is shown in the following examples.

Example:

```
#ifdef INCLUDE_DEBUG
print("Position = ",Apos(0));
print("Velocity = ",Avel(0));
#endif

#ifdef START_AT_0
start = 0;
#else
start = 1;
```

```
#endif

#if 0
This will never be compiled.
#endif

#if 1
This will always be compiled.
#endif

#define OPTION 1
#if OPTION > 2
This will be compiled only if OPTION is greater than 2.
#endif

#if defined(DEBUG) && !defined(EXCLUDE)
This will be compiled if DEBUG has been defined and EXCLUDE has not been
defined.
#endif
```

Example:

```
#define CUSTOMER_A 1
#define CUSTOMER_B 2

#define CUSTOMER CUSTOMER_B

#if CUSTOMER == CUSTOMER_A
Compile this for customer A.
#elif CUSTOMER == CUSTOMER_B
Compile this for customer B.
#else
Otherwise compile this.
#endif
```

4.5.5 #config

Configuration (.zbc) files contain parameter settings and DIM array data for controllers. These files are created by **Controller** → **Parameters** → **Save to file** and by the CAM Editor.

Configuration data from these files can be embedded within an application program and used to initialize the controller, by using any of the following statements:

```
#config "file"
#config "file",flags

#config <file>
#config <file>,flags
```

where "*file*" is the name of the configuration file to be embedded and may include directories. Either "\" or "/" may be used for directories. "*flags*" is an optional ApossC expression that will be evaluated to produce set of one or more flags that can be used to select which portions of the configuration are to be embedded. Note that it must be possible to evaluate the expression *at compile time*, hence ApossC variables cannot be used. Possible flags are:

CONFIGFLG_DIM	**(0x01)**	DIM Arrays
CONFIGFLG_USER	**(0x02)**	User Parameters

CONFIGFLG_AXIS	**(0x04)**	Axis Parameters
CONFIGFLG_MODULE	**(0x08)**	Module Parameters (excluding Global, User, and Axis Parameters)

If the quote syntax is used, then the Preprocessor will search for the configuration file in the same directory in which the file that referenced it was found. This is normally the same directory in which the original ".mc" file was found. However, it could be another directory if referenced by an include file that comes from a different directory. If the configuration file is not found in this directory, then the “system include” directory is searched. The system include directory is a subdirectory under the ApossIDE install directory.

If the angle bracket syntax is used, then the Preprocessor will search for the file only in the “system include” directory.

Examples:

```
#config "AppConfig.zbc"
#config "Library/AppConfig.zbc"
#config "Library\AppConfig.zbc"
#config <AppConfig.zbc>

#config "AppConfig.zbc", CONFIGFLG_DIM | CONFIGFLG_USER
```

The expression "CONFIGFLG_DIM | CONFIGFLG_USER" in the above example will cause both DIM arrays *and* User Parameters to be initialized since "|" is an ApossC bit operator. Counterintuitively, an expression such as "CONFIGFLG_DIM & CONFIGFLG_USER" will cause nothing to be initialized.

#config statements can be used in combination with #define statements to provide more programming flexibility. For example:

```
#define ZBCFILE "AppConfig.zbc"
#config ZBCFILE
```

#config statements may be placed in an application program anywhere that global variables may be declared or they may be placed within functions. If placed with global variables, then the initialization is done once when the application first starts executing. If placed within function, then the initialization is done in the normal way whenever execution passes the statement. Note that the same file may be used in multiple #config statements. For example:

```
// Initialize all configuration parameters.
#config "AppConfig.zbc"

long main(void)
{
    // Do some processing here.
    ...

    // Reinitialize the USER parameters.
    #config "AppConfig.zbc", CONFIGFLG_USER

    // Do more processing here.
    ...

    // Reinitialize the USER parameters again.
    #config "AppConfig.zbc", CONFIGFLG_USER
```

```
        // Do more processing here.
        ...

        return(0);
    }
```

If DIM Arrays are being initialized, then the following controller errors may be generated:

1. Error 73 if the DIM array in the configuration does not exist on the controller.
2. Error 107 if the DIM array in the configuration is larger than the corresponding DIM array on the controller.

4.6 Program Samples

The ApossIDE provides many sample programs designed to illustrate various aspects of the ApossC programming language and how it may be used to implement the functionality required by actual motor controller applications. Any of these sample programs may be freely used, modified, or incorporated into other programs by the user. **File** → **Sample** can be used to edit any of these sample programs.

Please note the following requirements for using the sample programs:

1. Unless specified otherwise, all sample programs may be executed using the Simulator; an actual controller is not required. Exceptions to this are usually related to samples that require the use of physical on-board hardware (e.g. samples illustrating the use of the microSD card, CAN communication between multiple controllers, etc.).
2. All ApossC sample programs may be executed on actual controllers. However, ApossC require at least firmware version 9.0.02.
3. No sample program will move an actual drive, even if the sample is executed on an actual controller that is connected to a drive. All drive movement is simulated. However, the samples may be easily modified so that a drive *will* be moved (see below).
4. The percentage velocity and acceleration values indicated in the programs are only valid if the parameter *Velocity Resolution* VELRES = 100.

Modifying sample programs to move the drive

All sample programs that are designed to move a drive, contain the following statement:

```
VIRTCOUNTIN_PARAM(AxisNo, VIRTCNTIN_PISRC_COUNTER) =
AXE_PROCESS_SRCINDEX(AxisNo, REG_COMPOS);
VIRTCOUNTIN_PARAM(AxisNo, VIRTCNTIN_MODE) = 2;
VIRTLATCH_PARAM(AxisNo, VIRTLATCH_PISRC_COUNTER) =
AXE_PROCESS_SRCINDEX(AxisNo, REG_ACTPOS);
```

This will cause actual position to be simulated, even when no drive with motor is present. Simply locate this command in the sample program and comment it out. Executing the sample program will then cause the drive to be moved.

Most programs that move the drive do so in a positive direction, starting from the current drive position. After processing various commands for movement and positioning, the drive will be returned to the starting position.

For movement commands to be executed, the following conditions must be satisfied:

- The axis controller must be active: AxisControl(AxisNo, ON);
- Any existing error state in the controller must be cleared.

Warning

When modifying a sample program to move a drive:

- The entire unit must be installed according to the installation instructions and the functionality of the individual components and connections must be tested.
- An EMERGENCY STOP must be installed according to the specific guidelines of the respective country.
- The program samples may only be used with a drive which can turn freely without a path limitation.
- The drive should be fixed in a stable holding device.
- When using a linear unit, limit switches for path limitation must be mounted and connected.
- When using a linear unit, the area of movement of the sample programs must be calculated roughly and compared with the paths of movement which are mechanically permissible.

The sample programs are divided into the following catagories:

`Hello`	Introduction to ApossC.	
System Information	Programs illustrating various ways of accessing system information from the controller.	
	`Information`	Displays some system information.
	`AxisStatus`	Displays the status of an axis during various conditions of movement.
Movement and Positioning	`BasicPositionControl`	Illustrates some very basic position-controlled moves.
	`BasicVelocityControl`	Illustrates some very basic velocity-controlled moves.
	`AbsoluteRelative`	Illustrates the difference between absolute and relative movements and how this is affected by setting the position origin.
	`ChangingVelocity`	Varies the velocity during a position-controlled motion.
	`MotorControl`	Illustrates basic motor control.
	`AccelerationRamps`	Illustrates different acceleration profiles (ramp types).
	`ISR_Position`	Illustrates how to recognize when encoder positions are passed.
	`ISR_Statbit`	Illistrates how to trap axis statis bit changes.
Digital Input/Output	`ISR_Input`	Illustrates how to handle digital inputs.
	`DigitalOutputs`	Illustrates how to set digital outputs using a variety of programming constructs.
Communication		
	`ISR_CanIn`	Illustrates the handling of CAN messages.
	`ISR_CanPdo`	Illustrates the handling of PDO messages.
	`ISR_SerialTelegrams`	Illustrates the handling og messages received on RS485/RS232 interfaces.

	`ISR_SerialCharacters`	Receives characters from the serial interface.
Synchronization and Markers	`SimpleSynchronization`	Synchronized positioning of two axes so that both axes reach their target position simultaneously.
	`PositionSynchronization`	Position synchronization of a slave axis to a master axis.

	Stamping Sample
	`Stamping`
	`Stamping.zbc`
	`Stamping.zbo`
		CAM Control: Stamping of Boxes with Use-by Date.

	Marker Sample
	`Marker`
	`Marker.zbc`
	`Marker.zbo`
		CAM Control: Printing of Cardboard Boxes with Marker Correction.

	`slavesync`	CAM Control: Slave Synchronization with Marker.
	`slavesync.zbc`	CAM Control: Slave Synchronization with Marker (configuration file).
	`Syncc_Msim`	Simulation of a master via software command.
State Machine Samples	Operational Events `ISR_xxx.mc`	div application for ISR events

Path Curve Sample	`PathCurves`
	`Pen.mi/.mh`
	`Square.zbc`	

	Illustrates all of the various events that are related to path curve handling (see Profile Generator Events). For details on how to use this sample program, please see Path Curve Sample.	
Controller Verification	`EncoderCheck.m`	Test program for testing encoder connection.
	`MotorCheck`	Test program for basic motor tuning.
ZBC Configuration Files	`Roller.zbc`	Curve configuration file for Roller application
	`PathSquare.zbc`	Configuration file containing a path.
Miscellaneous Samples	`Printing`	Illustrates the printing functionality supported by the ApossIDE.
	`ISR_Error`	Illustrates how to handle errors
	`Recording`	Illustrates how to record debug data.
	`DigitalOutputs`	Illustrates how to set digital outputs using a variety of programming constructs.

`OtherWaitTypes`	Illustrates various types of programmed waiting.
`MarkerCount`	Measurement of marker distance in connection with SyncMarker.

4.7 Debugging Programs

The ApossIDE provides several different methods for debugging application programs. Generally, each phase of development, from the initial coding of the program to the monitoring of a production system, will have a debugging method that is appropriate to it.

Coding and early testing:	ApossIDE Communication Window ApossIDE Debugger ApossIDE Simulator
Early live and production testing:	ApossIDE Communication Window ApossIDE Oscilloscope ApossIDE Recorder
Production system monitoring:	ApossIDE Oscilloscope ApossIDE Log Mode ApossIDE Recorder microSD Card

Communication Window

The Communication Window provides the quickest and simplest method of debugging. It is useful during both initial coding and early production testing. However, since it requires a dedicated connected computer, it is less useful as a method of monitoring production systems.

print and printf statements can be included anywhere in a program and used to display any type of information in the Communication Window. Typically, they are used to print critical variable values at critical times during program execution. They are also valuable as a way of tracking the flow of execution through a program.

> If a computer is not connected to a controller during program execution, then all messages generated by print statements are simply discarded. Application programs do *not* stop simply because a print statement has nowhere to display the message.

> The Communication Window can only save a limited number of messages. If large numbers of messages are to be generated for future analysis, then **Development** → **Messages -> Log file** can be used to start the logging of Communication Window messages to a file. Note that **Development** → **Stop logging** will be enabled if logging has been started.

Debugger

The ApossIDE contains a powerful built-in debugger. This provides common debug features such as single-stepping, breakpoints, and the ability to read and set program variable values. The Debugger is most useful during the coding and initial testing of programs.

> The debugger *cannot* be used in all cases. It may not be possible to use the debugger with programs that...
>
> - **...are actively controlling a motor.**

> Stopping program execution at a breakpoint is equivalent to pressing the **Esc** key to break program execution. This will cause the motor to slow down and stop with the maximum allowed deceleration. In many cases, stopping the motor like this will invalidate the test procedure and make the debug result meaningless. As well, if program execution is continued after a breakpoint, then it is unlikely that the motor can be restarted correctly to put the system into the state it was in prior to the breakpoint.
>
> - **...rely on PERIOD interrupts.**
>
> The internal timer that triggers calls to PERIOD interrupts functions does *not* stop when program execution stops at a breakpoint. This may leave a pending interrupt which will then trigger a PERIOD interrupt call as soon as program execution is continued.
>
> - **...are implemented as state machines.**
>
> In general, state machine programs require timely and rapid handling of events in order to function properly. Disturbing the processing of the event queues can have an adverse affect on the behaviour of the state machines. Please see Debugging State Machines for more information on the tools available for debugging state machine programs.

The debugger may be used when the controller is not currently executing a program or when the controller is already executing a program. If the controller is not currently executing, then starting the debugger will debug the source that the user is currently editing. If the controller is already executing, then the user must ensure that the source being edited matches the program that is currently executing.

Click on **Development** → **Start Debugger** and the program is prepared for debug mode: It is compiled and downloaded to the controller but execution of the program is not started. Breakpoints are added to every executable line and are displayed as blue dots at the beginning of each line in the program. The "next" line to be executed is highlighted in yellow. The user can then "single step" through the program.

Please see the links below for more details.

Oscilloscope

For situations such as those above where the debugger cannot be used, the Oscilloscope provides excellent debugging capabilities. It is most useful during live testing when the application program is actively controlling motors. It can also be used in a production system if problems need to be diagnosed, although this requires a computer to be connected which is not always possible.

The Oscilloscope can watch and record program variables and system states without having to break program execution. These can then be reviewed afterwards to identify problems. For more information on the Oscilloscope, please see Oscilloscope.

Simulator

The ApossIDE provides several Simulators that can simulate a controller in software without the need of a physical controller. In certain types of applications and at certain stages of application development, a Simulator can be a valuable debugging tool. For example, partially developed application programs can be run on a Simulator without the risk of damaging equipment when something goes wrong with the application program. A Simulator can also be used when a physical controller is unavailable. For more information, see Simulator.

Log Mode

Log Mode provides a simple client/server mechanism that allows active controllers to be monitored. This is mostly intended to be used in production scenarios where unattended monitoring of controllers is needed and controllers are accessible over some type of network (e.g. Ethernet or EtherCAT). For more information, see Log Mode.

Recorder

The Recorder is a demonstration application that provides a function similar to black-box recorders. This is intended to be used in production scenarios where unattended monitoring of controllers is needed and controllers may or may not be accessible over some type of network. As this is a demonstration application, source code is available upon request. For more information, see Recorder.

microSD Card

Some types of zub controllers (including the MACS5 and the MasterMACS) provide a microSD card slot. This can be used to record diagnostic information (e.g. print messages and Oscilloscope data) for later review. It can be used in cases where production controllers are isolated and not accessible over any network. For more information, see External Storage Device (microSD Card/ USB Memory Stick).

4.7.1 Starting the Debugger

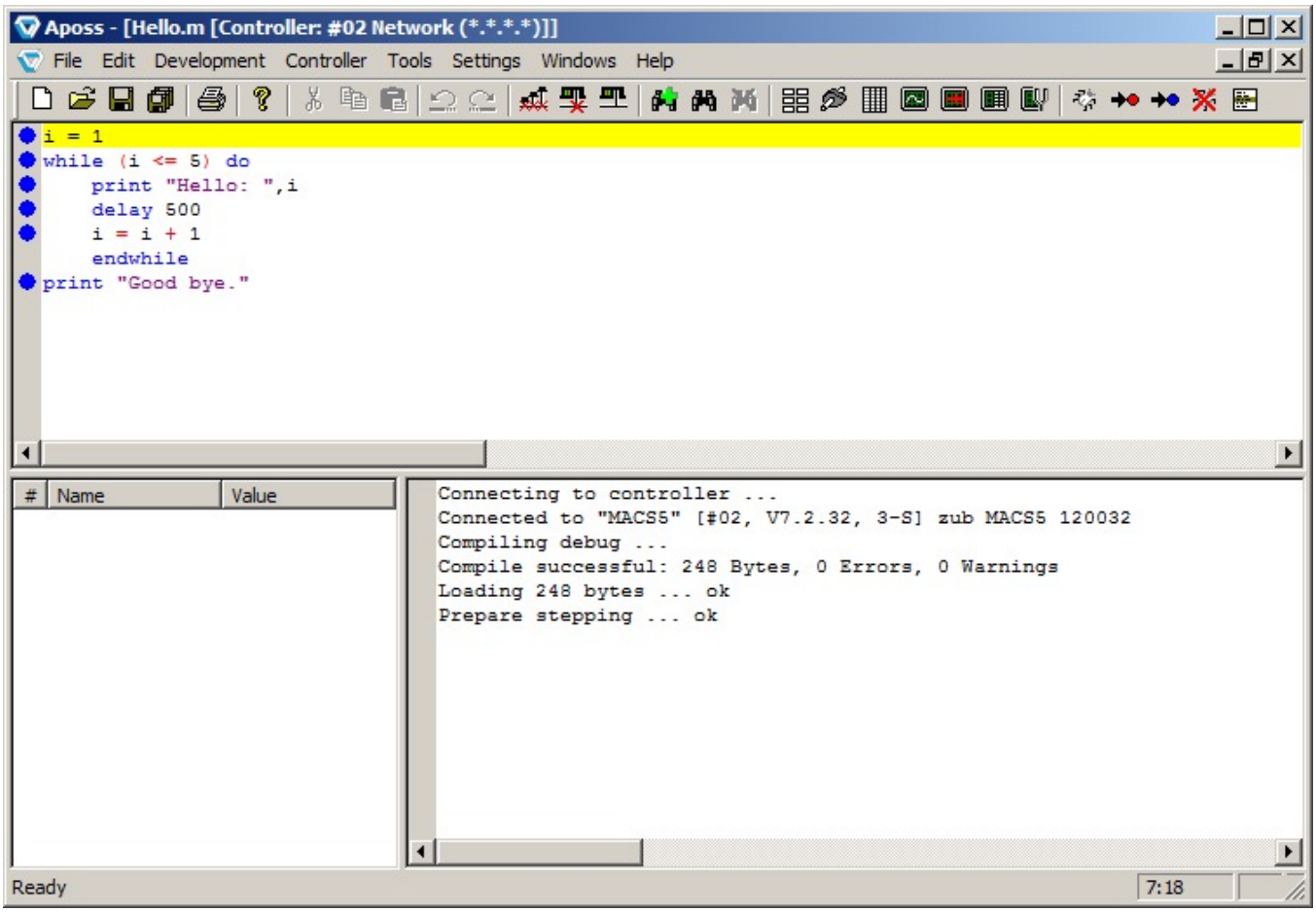
To start the debugger, edit the program to be debugged in the normal way so that it is displayed in the Edit Window. Then click on **Development** → **Start Debugger** or click in the toolbar. If the controller is not currently executing a program, then this will take the following actions:

1. The program is compiled in debug mode and downloaded to the controller. However, program execution is *not* started at this time.
2. A blue dot is placed before each executable statement in the program. These are the positions where the user may insert breakpoints.
3. The *next* statement to be executed (i.e. when execution is started or continued) will be highlighted in yellow.

If the controller is already executing a program, then starting the debugger will take the following actions:

1. A blue dot is placed before each executable statement in the program. These are the positions where the user may insert breakpoints.
2. The executing program continues to execute.
3. The user may pause the program at any time by pressing the **Esc** key. The *next* statement to be executed (i.e. when execution is continued) will be highlighted in yellow.

The following diagram shows what the Edit Window might look like.



While the debugger is active, the program should not be modified. Doing so will cause the ApossIDE to become out-of-sync with the version of the program executing on the controller and the debugger may no longer be able to properly follow the program execution. If the program must be changed, then the debugger should be stopped and the test restarted from the beginning.

4.7.2 Stopping the Debugger

To stop the debugger and end the debugging session, click on **Development** → **Stop Debugger** or click in the toolbar. This will do the following:

1. Remove the blue dots marking executable statements.
2. Remove any breakpoints that the user has set.

If the controller is executing a program when the debugger is stopped, then the controller will continue to execute the program. If the program has been stopped, then the user may restart the program at the point where it stopped, by clicking on **Development** → **Continue**.

4.7.3 Single-Stepping

To single-step through the program, use **Development** → **Singlestep**, press **F9**, or click in the toolbar. This will execute the next statement (i.e. the statement currently highlighted in yellow) and automatically stop before the next executable statement is executed (i.e. at the next statement with a blue dot). This next statement will then be highlighted in yellow.

While execution is stopped, the user is free to examine and modify the value of any program variable, set and clear breakpoints, etc.

At any time, program execution can be continued without single stepping, by using **Development** → **Go to Breakpoint** or by clicking in the toolbar.

4.7.4 Using Breakpoints

Breakpoints are set by double-clicking anywhere on the statement in the program (including on the blue dot) where the breakpoint is to be set. The dot will change from blue to red to indicate that the breakpoint has been set.

Double-clicking on a statement that already contains a breakpoint will clear the breakpoint and the dot will change back from red to blue.

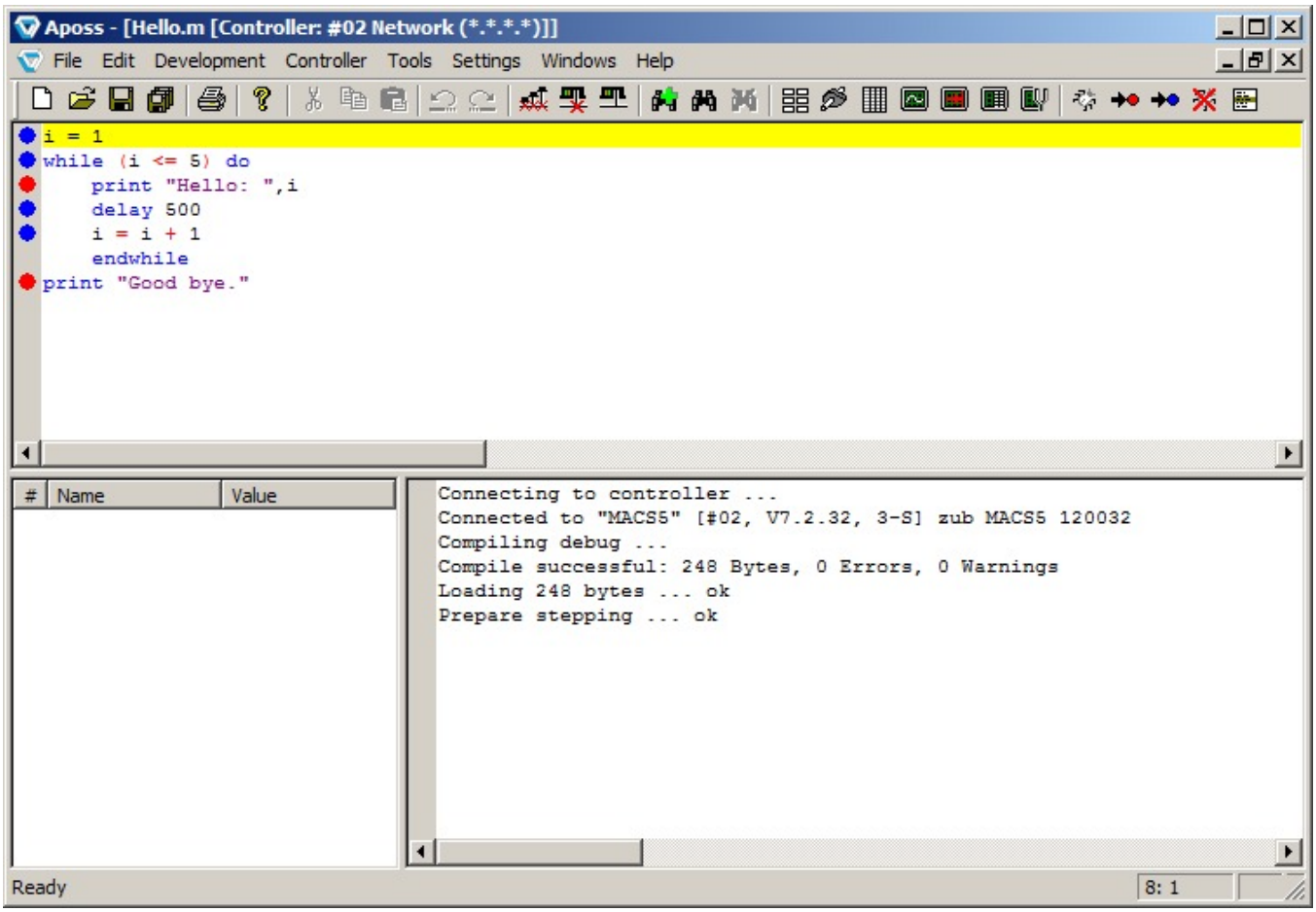
The user can set a "breakpoint" at any executable statement in the program. These are the statements with blue dots. When program execution encounters a breakpoint, execution is immediately stopped *prior* to executing the statement with the breakpoint. The statement will then be highlighted in yellow since this is the next statement to be executed.

While execution is stopped, the user is free to examine and modify the value of any program variable, set and clear breakpoints, etc.

When ready, the user can click on **Development** → **Go to Breakpoint** or on to continue program execution. The program will then execute up until it encounters another breakpoint (i.e. up until the next statement with a red dot). Note that it is also possible to single-step after stopping at a breakpoint.

At any time during program execution, the user may also press the Esc key to stop execution. Execution will then stop immediately rather than continuing to the next breakpoint. The next statement to be executed will be highlighted in yellow. Again, the user can continue with either **Development** → **Go to Breakpoint** or **Development** → **Singlestep** **[F9]**.

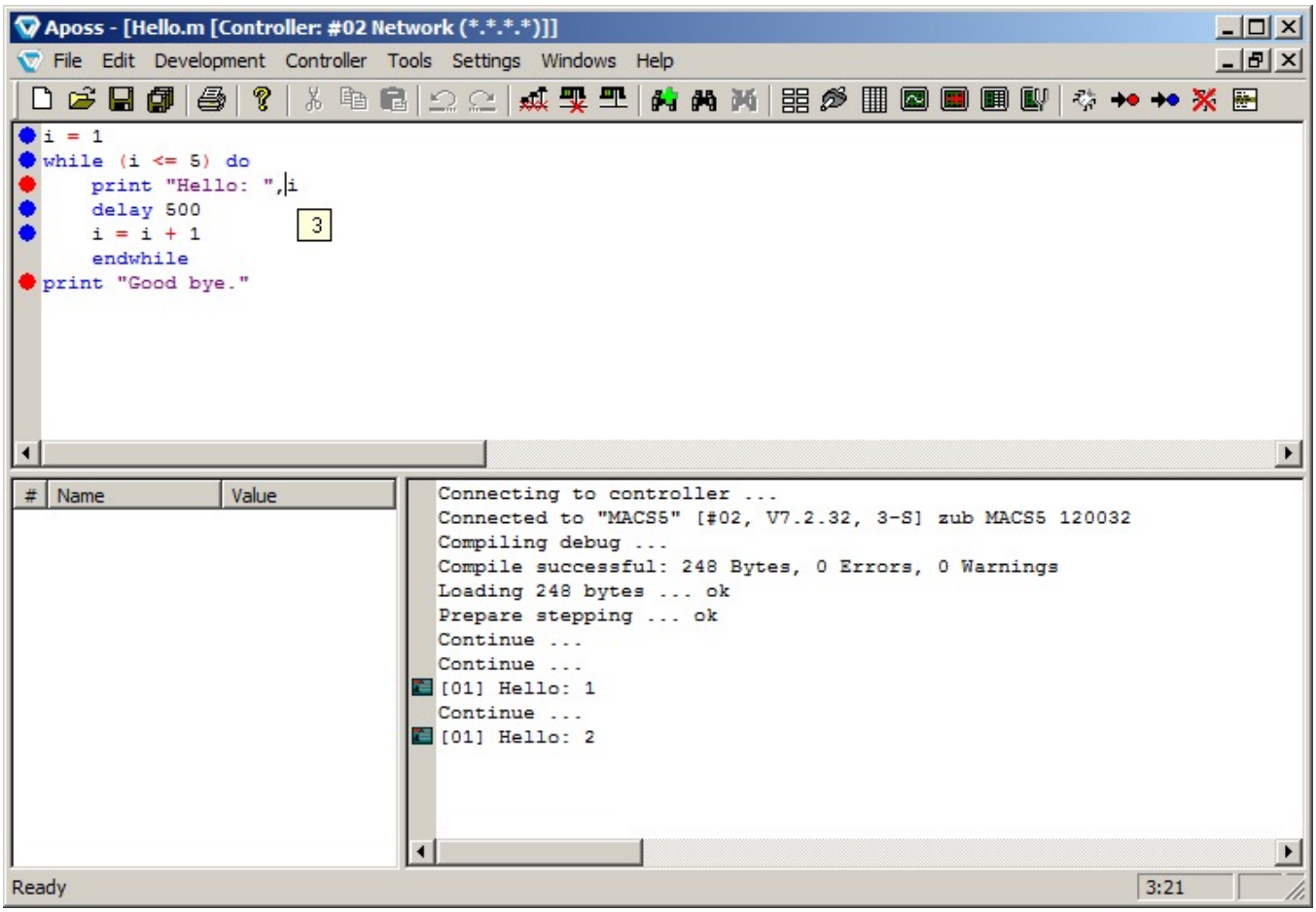
The following diagram shows what the Edit Window might look like when breakpoints have been set before each "print" statement. Note the red dot on these statements.



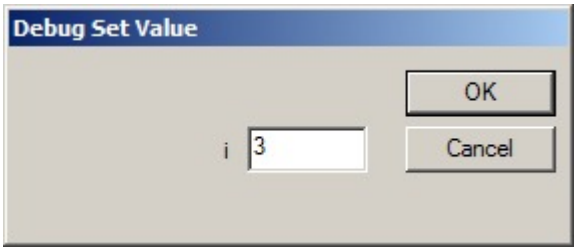
A maximum of 10 breakpoints are allowed.

4.7.5 Displaying and Modifying Variables

At any time while the debugger is active (i.e. whether the program is executing or stopped) the current value of any program variable can be displayed. This is done by clicking on any instance of the variable with the *left* mouse button. The current value will be displayed in a yellow popup box. The value will be displayed until the mouse is moved away from the variable. An example is shown below where the mouse button was clicked near the variable “i” in the “print” statement.



At any time while the debugger is active, the current value of any program variable can also be modified. This is done by clicking on (or immediately after) any instance of the variable with the *right* mouse button. The dialog shown below will be displayed allowing the user to modify the variable value.



4.7.6 Displaying the Executing Line

At any time, whether the debugger has been started or not, the current statement being executed (or about to be executed) by the controller can be displayed by pressing the toolbar button. The statement will then be highlighted in yellow. This highlight can be cleared simply by clicking on the yellow highlight. This function is useful for determining which part of a program is executing at any given time.

5 ApossIDE Tools

The following tools are available from within the ApossIDE, usually from either the Tools Menu or as buttons on the Toolbar.

CAM Editor	The CAM Editor is a tool that provides users with the ability to graphically design the CAM and Path curves that will be used in synchronization applications.
Array Editor	The Array Editor provides a list-oriented interface that allows all controller arrays and parameters to be viewed and edited. This includes user parameters, global parameters, axis parameters, axis state variables, and system state variables.
Oscilloscope	The Oscilloscope is a tool for recording and graphically displaying values from one or more controllers while those controllers are operational and executing programs.
Module Editor	The Module Editor is a tool allowing users to configure and debug Function Modules. Function Modules provide application programs with access to hardware and/or software functions such as encoders, counters, latches, pulse generators, etc. Function Modules are particularly important in MACS5 and later controllers.
State Machines	The ApossIDE and zub motor controllers provide built-in support for hierarchical state machines. The ApossIDE programming languages contain extensions that allow programmers to develop programs directly using hierarchical state machine structures.
Simulator	Simulators can simulate a controller in software without the need of a physical controller. In certain types of applications and at certain stages of application development, a Simulator can be a valuable debugging tool.
Current Regulator Calculator	With the help of the current regulator calculator the parameters needed to set the PI controller can be determined from the motor data.

The following tools are designed to run as stand-alone tools rather than being run from within the ApossIDE. However, some of these tools may also be run from within the ApossIDE.

Monitor	The Monitor feature allows the user to create customized Windows dialog interfaces for operating controllers. These could be designed for such purposes as normal system operation, system configuration, maintenance, diagnostics, etc.
Batch Mode	Batch Mode provides a simple client/server mechanism that allows

communication with controllers without requiring user input. When the ApossIDE is started in Batch Mode, it will wait for a "command file" to be created (by some external means such as another program). When a file appears, then the file is opened and the commands within it are executed. Once execution is complete, an "answer file" is created and the original command file is deleted. Batch Mode will then wait for a new command file to be created.

Log Mode — Log Mode provides a simple client/server mechanism that allows active controllers to be monitored. The monitored values for each controller will be written to a separate log file. This can include both print messages and data values. The log files can then be subsequently retrieved and analyzed. For example, the data values can be imported into the Oscilloscope. When the ApossIDE is started in Log Mode, it will wait for a "command file" to be created (by some external means such as another program). When a file appears, then the file is opened, the commands within it are executed, and logging begins. Once execution is complete and logging has begun, an "answer file" is created and the original command file is deleted. Log Mode will then wait for a new command file to be created. Note that a new command file will not necessarily stop the logging begun by a previous command file. For example, new command files can add additional controllers to the list of controllers being monitored.

Download Mode — Download Mode provides a simple interface that allows compiling and downloading of programs to multiple controllers. It will also allow downloading of configuration files to multiple controllers.

Recorder — The Recorder is a demonstration program that provides a function similar to a black-box recorder. It is designed to record information from a controller while it is running, providing the opportunity to analyze the data sometime in the future if necessary.

5.1 CAM Editor

The ApossIDE and zub controllers support the following types of curves:

CAM curves — These curves are used by applications that simulate CAM controls. See CAM Curves and Example 5: CAM Box for examples.

Path curves — Many motor controller applications require the synchronization of multiple axes to move a tool along a specific curve through space. For example, the 2-dimensional space of a pen in an XY plotter or the 3-dimensional space of a tool in a numerical milling machine or a coordinate measuring machine. The ApossIDE supports this using multi-dimensional “path curves”. See Working with Kinematics and Path Curves for a description of support for path curves.

Data arrays — In some cases, it is useful to display and edit a simple array of data

values as if it were graphical data. The CAM Editor will treat any array that is not recognized as a CAM or Path curve as a Data array.

Curves for these applications are created using the CAM Editor. Curves are defined by fixed points as well as parameters for starting and stopping velocities, engage and disengage motion, synchronization with markers, etc. The curves and parameters are displayed graphically and can be edited either graphically or textually. Each curve is stored in a single global DIM array on the controller. Multiple curves can be created simply by using multiple DIM arrays.

The CAM Editor will save the curve definitions and all parameter values in a "configuration file" with the extension ".zbc". They will be identified in Windows folders with the icon shown below. As with other types of files, configuration files may be opened, modified, and saved by the CAM Editor at any later time.

> Previous versions of the ApossIDE used the file extension ".cnf" for configuration files. This was changed from ".cnf" to ".zbc" to avoid conflicts with the usage of the ".cnf" file extension by Microsoft Windows. However, existing ".cnf" files are still supported and can be read by the CAM Editor simply by opening them in the normal way. If a ".cnf" file is opened by the CAM Editor, then the CAM Editor will save it as a ".zbc" file.

The CAM Editor is designed to use exactly the same ".zbc" configuration file format as is used by the **Controller** → **Parameters** → **Save to file** and **Controller** → **Parameters** → **Restore from file** commands to save and restore complete controller configurations. Hence, the configuration files created by the CAM Editor will store all Global, User, and Axis parameters *in addition* to the curve profiles. As a result, you should always start the CAM Editor with parameters *from the controller to which the curve profiles are intended*. Otherwise, the *wrong* Global, User, and Axis parameters may be downloaded to the controller when the curve profiles are downloaded.

Once curves have been created and saved in a ".zbc" file, they can be downloaded to the controller using **Controller** → **Parameters** → **Restore from file**. Curves can also be downloaded directly to the controller using the **Write to controller** toolbar button in the CAM Editor.

5.1.1 Starting the CAM Editor

To start the CAM Editor, click on **Tools** → **CAM Editor**, click the toolbar icon, or click on **File** → **New** and select "Create Configuration .zbc file". All of these will open a new configuration file and set all parameters to default values.

The CAM Editor can also be started by using **File** → **Open** and selecting an existing configuration file to be opened.

Simply double-clicking on a configuration file in Windows will also start the CAM Editor. Note that double-clicking a ".cnf" file will *not* start the CAM Editor; the file must be a ".zbc" file.

If you are starting with new curve profiles and a configuration file does not yet exist, then the recommended procedure is always to start with parameters *from the controller to which the curve profiles are intended*. This can be done in either of the following ways:

1. First connect to the controller and then create a configuration file using **Controller** → **Parameters** → **Save to file**. The newly created configuration file can then be opened with the CAM Editor by using **File** → **Open**.
2. Open a "new" configuration file as described above. Then use the **Read from Controller** toolbar button to read the current parameters from the controller.

> If parameters are not read from the controller, then default factory settings will be used. In this case, the existing controller parameters will be *overwritten* with the factory defaults when this configuration file is downloaded to the controller.

New configuration files will not contain any curve profiles. Use the **Add Array** toolbar button to insert a new curve profile. If the configuration file already contains arrays, then they may be selected using the **Array** field in the toolbar (see CAM Editor Toolbar).

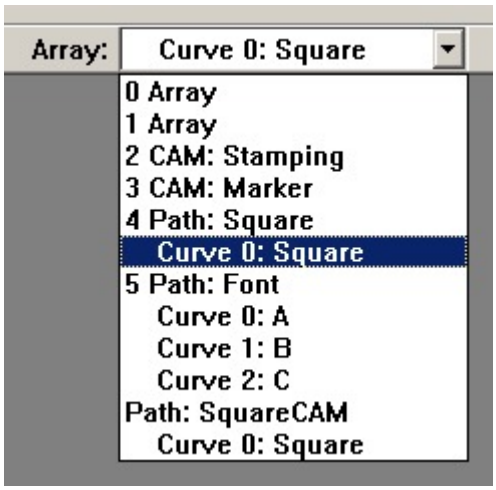
Several sample configuration files containing curve profiles can be found using **File** → **Sample**.

5.1.2 CAM Editor Toolbar

The CAM Editor toolbar contains the following special fields:

Array This field lists all the arrays currently defined and allows the user to select the array to be edited. The currently selected array is highlighted.

- Entries that correspond to actual DIM arrays, will start with the DIM array number. In the example below, there are 7 DIM arrays.
- Data arrys (i.e. non-curve arrays) will simply be labeled "Array". DIM arrays 0 and 1 below, are data arrays.
- CAM arrays will be identified with the "CAM:" prefix, followed by the curve name. DIM arrays 2 and 3 below, are CAM arrays.
- Path arrays will be identified with the "Path:" prefix, followed by the path name. Path arrays may contain multiple sub-paths, identified with the "Curve" prefix, followed by the sub-path index (see Multiple Paths). DIM array 5 below, contains a Path array with 2 sub-paths. Each sub-path is then split into its component axes (see Viewing Individual Axes).



Length This displays the minimum length required for the array. This is the minimum value needed for the DIM statement in the application program.

The non-standard toolbar buttons that are specific to the CAM Editor are as follows:

Zoom Reset - Redisplay the diagram so that the entire curve is visible.

Zoom In - Zoom in by a factor of 2 and display only the central portion of the diagram.

Zoom Out - Zoom out by a factor of 2.

Reset splitters - Restore the splitters in the window to their default positions.

Undo - Undo the last change to either a fixed point or a Start Stop point. If multiple changes were made at a single time (for example adding multiple points, or by clicking the **Snap on Grid** button), then these changes will be undone as a set. If no changes are available to be undone, then this button will be disabled and grayed.

Note that if multiple curve arrays exist, then a separate undo list is maintained for each curve.

> **Undo** is not currently available when editing path curves.

Redo - Redo a change that was undone using the Undo button.

Snap on Grid - Snap all fixed points and Start Stop points to the interpolation grid.

Read from controller - Read the existing configuration parameters and all arrays and curves from a controller. This will *replace* all existing parameters and arrays currently in the CAM Editor.

Write to controller - Write the configuration parameters and arrays (i.e. including curve arrays) to a controller. A dialog will be displayed allowing the user to select which groups of parameters to write and whether or not to write arrays.

Import - Import an ACSII text file that has been exported with the **Export** toolbar button. The imported curve will replace the currently selected curve. The format of the file must exactly match an export file or the import will not succeed.

Export - Export the curve data for the currently selected curve to an ASCII text file with the extension ".dat". This is useful if you want to process the curve outside of the ApossIDE (for example, using a spreadsheet). Please see CAM Array Structure in the chapter "Technical Reference" for a description of the exact format of the array.

Create on clipboard - Create an application function on the Windows clipboard that can be used to explicitly initialize an array with the appropriate curve profile. This can then be pasted into an application program edit window using any of the normal paste functions.

Graphics mode - The currently selected array will be displayed as a graphical chart rather than in text mode.

Text mode - The currently selected array will be displayed in text mode rather than as a graphical chart.

Add array - Add a new array. The array may be either a simple array or a CAM or

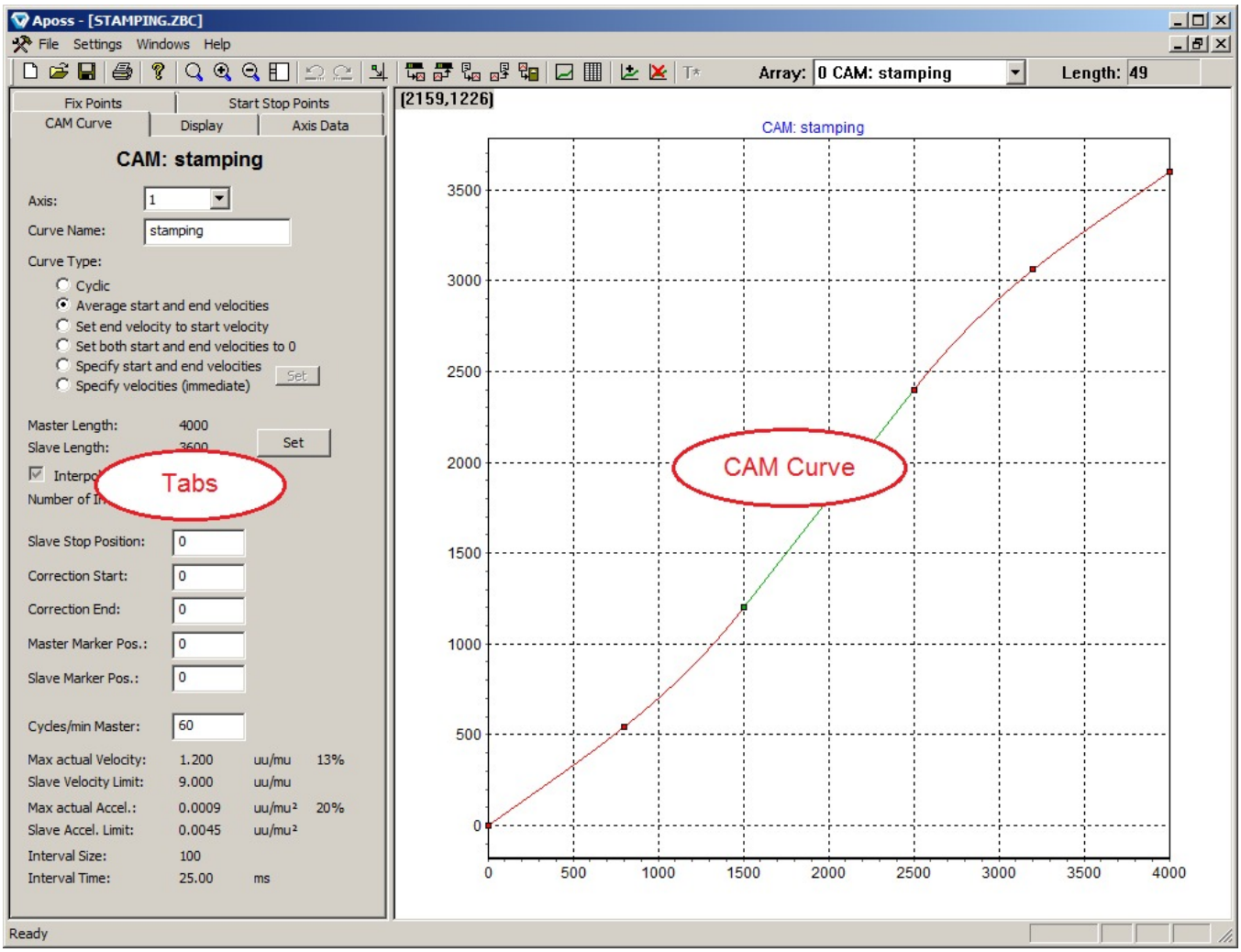
Path curve. A dialog is displayed allowing the user to set the initial attributes of the array or curve. The new array will then be automatically selected and displayed. If a curve is added, then the initial curve will contain only two fixed points and will be a straight line. Additional fixed points should then be added to define the actual shape of the curve.

Delete array - The currently selected array will be deleted. Note that any array may be deleted; the array need not be a curve profile array.

T* **Transform** - Transform the currently selected array. Note that not all types of arrays may be transformed.

5.1.3 Editing CAM Curves

When a CAM curve array is selected in the **Array** box in the toolbar, the CAM Editor window will appear similar to that shown below:



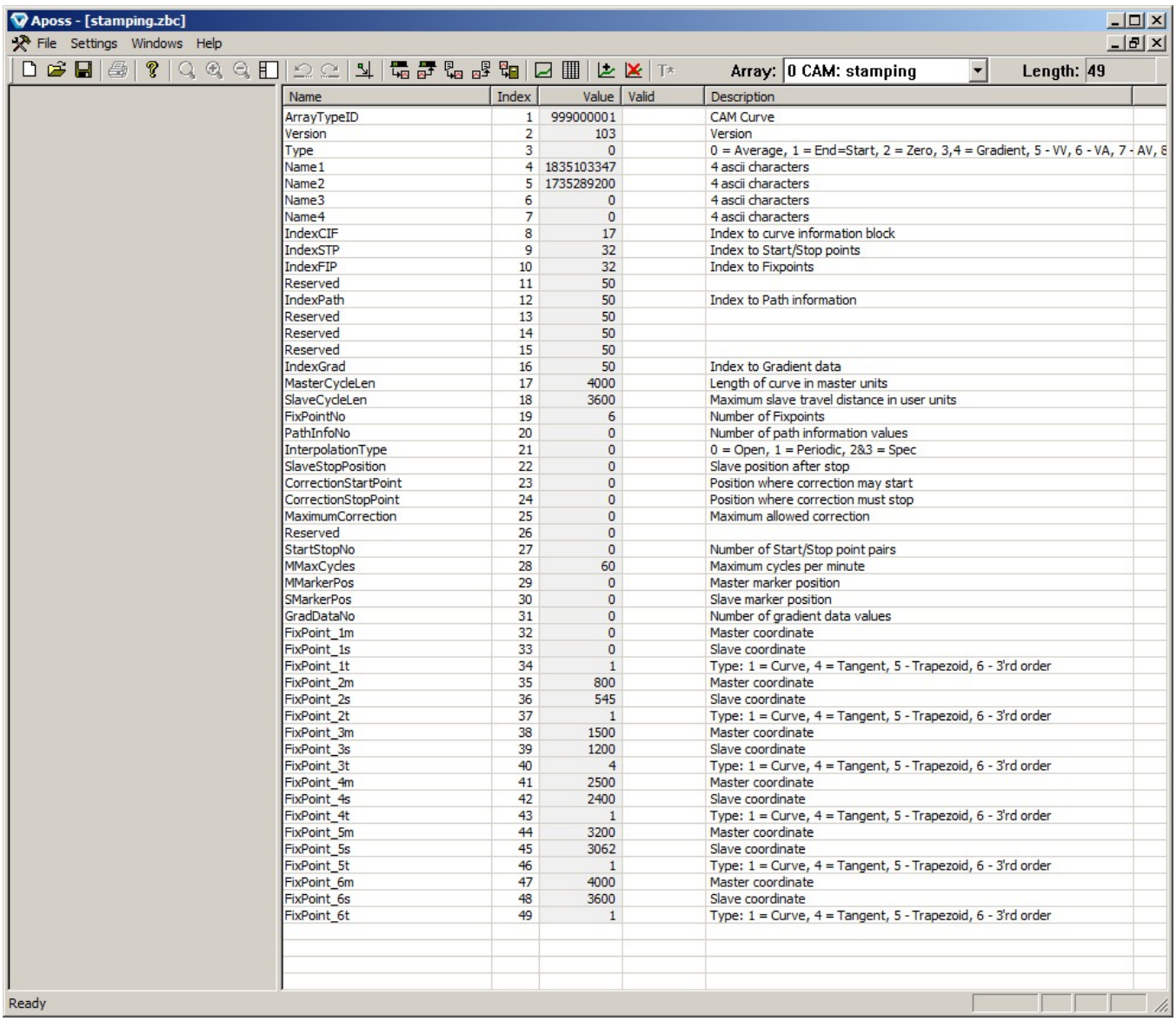
The window is divided into two sections:

- CAM Curve Diagram
- Tabs:

- CAM Curve Tab for settings various curve-related parameter values.
- Display Tab for displaying and hiding various attributes of the curve profile diagram.
- Axis Data Tab for setting various axis-related parameter values required by the curve.
- Fix Points Tab for editing the fixed points belonging to the curve.
- Start Stop Points Tab for editing the start/stop points belonging to the curve.

The window can be resized in the standard way. As well, the splitter bar between the two window sections can be adjusted using the mouse in the standard way. The splitter bar can be restored to its default positions by using the **Reset splitters** toolbar button.

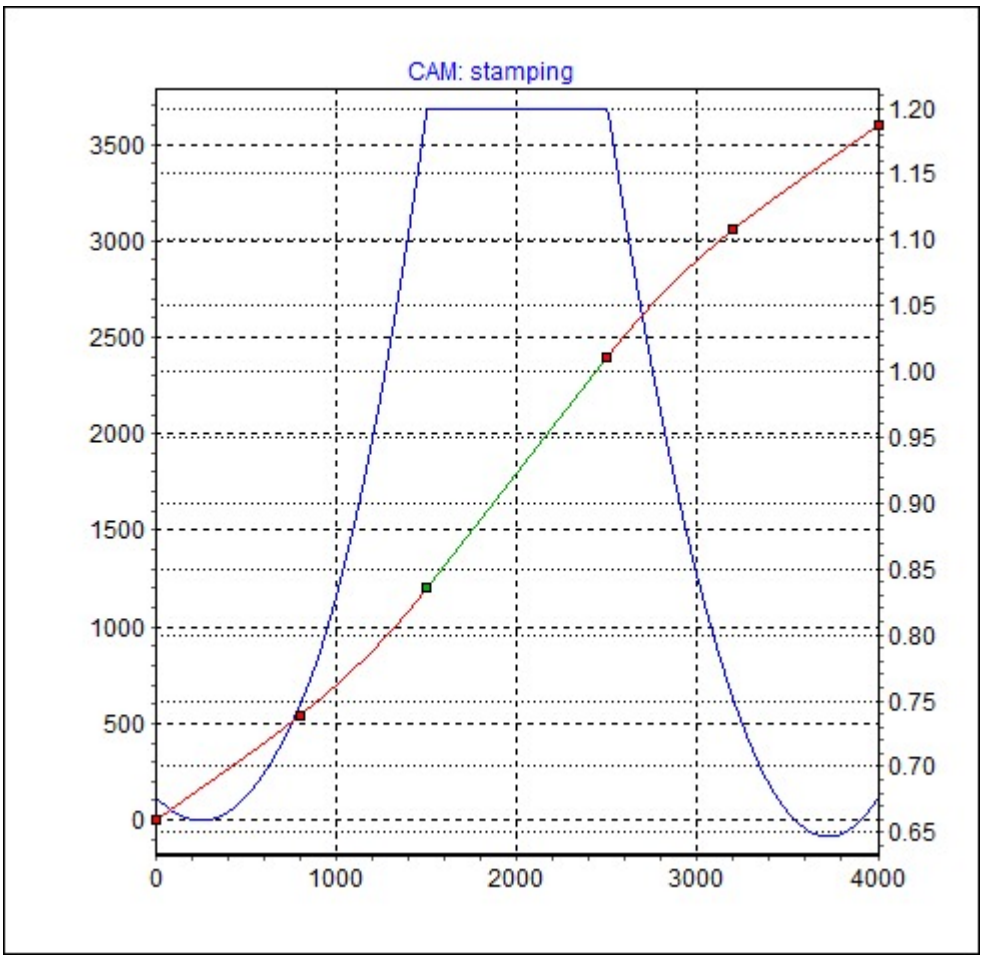
The actual CAM array that will be downloaded to the controller can be seen if the display is switched to text mode using the **Text mode** toolbar button. In this case, the CAM Editor window will appear similar to that shown below. Note that the array cannot be modified when text mode is being used. See CAM Array Structure for more information.



Name	Index	Value	Valid	Description
ArrayTypeID	1	999000001		CAM Curve
Version	2	103		Version
Type	3	0		0 = Average, 1 = End=Start, 2 = Zero, 3,4 = Gradient, 5 - VV, 6 - VA, 7 - AV, 8
Name1	4	1835103347		4 ascii characters
Name2	5	1735289200		4 ascii characters
Name3	6	0		4 ascii characters
Name4	7	0		4 ascii characters
IndexCIF	8	17		Index to curve information block
IndexSTP	9	32		Index to Start/Stop points
IndexFIP	10	32		Index to Fixpoints
Reserved	11	50		
IndexPath	12	50		Index to Path information
Reserved	13	50		
Reserved	14	50		
Reserved	15	50		
IndexGrad	16	50		Index to Gradient data
MasterCycleLen	17	4000		Length of curve in master units
SlaveCycleLen	18	3600		Maximum slave travel distance in user units
FixPointNo	19	6		Number of Fixpoints
PathInfoNo	20	0		Number of path information values
InterpolationType	21	0		0 = Open, 1 = Periodic, 2&3 = Spec
SlaveStopPosition	22	0		Slave position after stop
CorrectionStartPoint	23	0		Position where correction may start
CorrectionStopPoint	24	0		Position where correction must stop
MaximumCorrection	25	0		Maximum allowed correction
Reserved	26	0		
StartStopNo	27	0		Number of Start/Stop point pairs
MMaxCycles	28	60		Maximum cycles per minute
MMarkerPos	29	0		Master marker position
SMarkerPos	30	0		Slave marker position
GradDataNo	31	0		Number of gradient data values
FixPoint_1m	32	0		Master coordinate
FixPoint_1s	33	0		Slave coordinate
FixPoint_1t	34	1		Type: 1 = Curve, 4 = Tangent, 5 - Trapezoid, 6 - 3'rd order
FixPoint_2m	35	800		Master coordinate
FixPoint_2s	36	545		Slave coordinate
FixPoint_2t	37	1		Type: 1 = Curve, 4 = Tangent, 5 - Trapezoid, 6 - 3'rd order
FixPoint_3m	38	1500		Master coordinate
FixPoint_3s	39	1200		Slave coordinate
FixPoint_3t	40	4		Type: 1 = Curve, 4 = Tangent, 5 - Trapezoid, 6 - 3'rd order
FixPoint_4m	41	2500		Master coordinate
FixPoint_4s	42	2400		Slave coordinate
FixPoint_4t	43	1		Type: 1 = Curve, 4 = Tangent, 5 - Trapezoid, 6 - 3'rd order
FixPoint_5m	44	3200		Master coordinate
FixPoint_5s	45	3062		Slave coordinate
FixPoint_5t	46	1		Type: 1 = Curve, 4 = Tangent, 5 - Trapezoid, 6 - 3'rd order
FixPoint_6m	47	4000		Master coordinate
FixPoint_6s	48	3600		Slave coordinate
FixPoint_6t	49	1		Type: 1 = Curve, 4 = Tangent, 5 - Trapezoid, 6 - 3'rd order

5.1.3.1 CAM Curve Diagram

The curve profile diagram graphically displays the curve along with other information about curve synchronization. It also allows fixed points to be added, deleted, and moved using the mouse.



Please note the following about the curve profile diagram:

- The name of the curve, if any, is shown at the top of the chart.
- The master distance is shown horizontally across the *bottom* of the diagram. The slave position is shown vertically up the *left* side of the diagram. Various other scales (e.g. velocity) will be shown vertically up the *right* side of the diagram when necessary.
- Fixed points are displayed as small red or green squares. “Curve” points (i.e. fixed points that begin a curved segment of the profile) will be red and “tangent” points (i.e. fixed points that begin a tangent segment of the profile) will be green.
- Curved segments of the profile will be drawn red and tangent segments will be drawn green. Note that segments that are straight will be drawn green even though they may be “curve segments” (i.e. segments that begin with a “curve” point).

Zoom and Pan

The mouse buttons can be used in the curve profile diagram for the following functions:

Zoom

Zoom in to an area of interest in the diagram by pressing the left mouse button, moving the mouse across the area of interest, and then releasing the mouse button.

> The selected rectangle will be stretched so that it is exactly displayed in the window. If stretching is not

> desired, then press and hold the **Ctrl** key before pressing the left mouse button.

Use the **Zoom Out** or **Zoom Reset** toolbar buttons to zoom back out again.

Zooming the diagram will automatically clear the ☑ **Auto Scale** setting.

Pan

Pan across the diagram (i.e. move the diagram in the window) by pressing the right mouse button, moving the mouse, and then releasing the mouse button.

Panning the diagram will automatically clear the ☑ **Auto Scale** setting.

Use **Zoom Reset** to reset both the pan and zoom so that the entire diagram is displayed again.

Using the Mouse to Edit Fixed Points

Fixed points can be edited interactively using the mouse within the diagram. They can also be input directly in the Fix Points Tab .

Identify Fixed Point

Fixed points can be identified simply by moving the mouse and pausing over the fixed point in the diagram. The mouse cursor icon will change from the normal arrow icon to a hand icon and the corresponding fixed point will automatically be highlighted in the Fix Points Tab . The required "nearness" of the mouse to the fixed point can be adjusted using **Mouse accuracy** in the Display Tab.

Move Fixed Point

A fixed point can be moved by dragging it with the left mouse button. Make sure that the hand icon is displayed before trying to drag the fixed point. The point's position will be automatically updated in the Fix Points Tab as the point is being dragged.

If ☑ **Continuous Recalc** is enabled, then the curve profile will also be automatically recalculated and the CAM Curve Tab will be updated as the point is dragged.

Note that if ☑ **Snap on Grid** is enabled, then the point will always snap onto the nearest interpolation grid line and will not follow the mouse exactly.

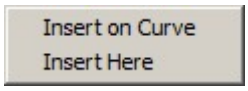
If necessary, use the **Undo** toolbar button to restore the point to its previous position.

Add Fixed Point

A new fixed point can be added by moving the mouse to the point where the fixed point is to be placed and then double-clicking the left mouse button. If necessary, use the **Undo** toolbar button to delete the point. Please see Fix Points Tab for more information on adding fixed points.

Using CAM Popup Menus

If the right mouse button is clicked in an empty part of the curve profile diagram, then the following popup menu will be displayed:



Insert on Curve

A new fixed point is inserted at the specified master position but on the curve rather than where the mouse button was clicked.

Insert Here

A new fixed point is inserted at the position where the mouse was clicked. This is the same as double-clicking with the left mouse button. The curve profile will be recalculated.

Change Fixed Point Type

If the right mouse button is clicked on a fixed point, then the following popup menu will be displayed to change this fixed point type:

Select the type to which the identified point should be changed: **Curve**, **Tangent**, **Trapezoid**, or **3'rd order**. See Fix Points Tab .

Corner

Toggle the identified fixed point between a "corner" point and a "non-corner" point. See Fix Points Tab .

Delete

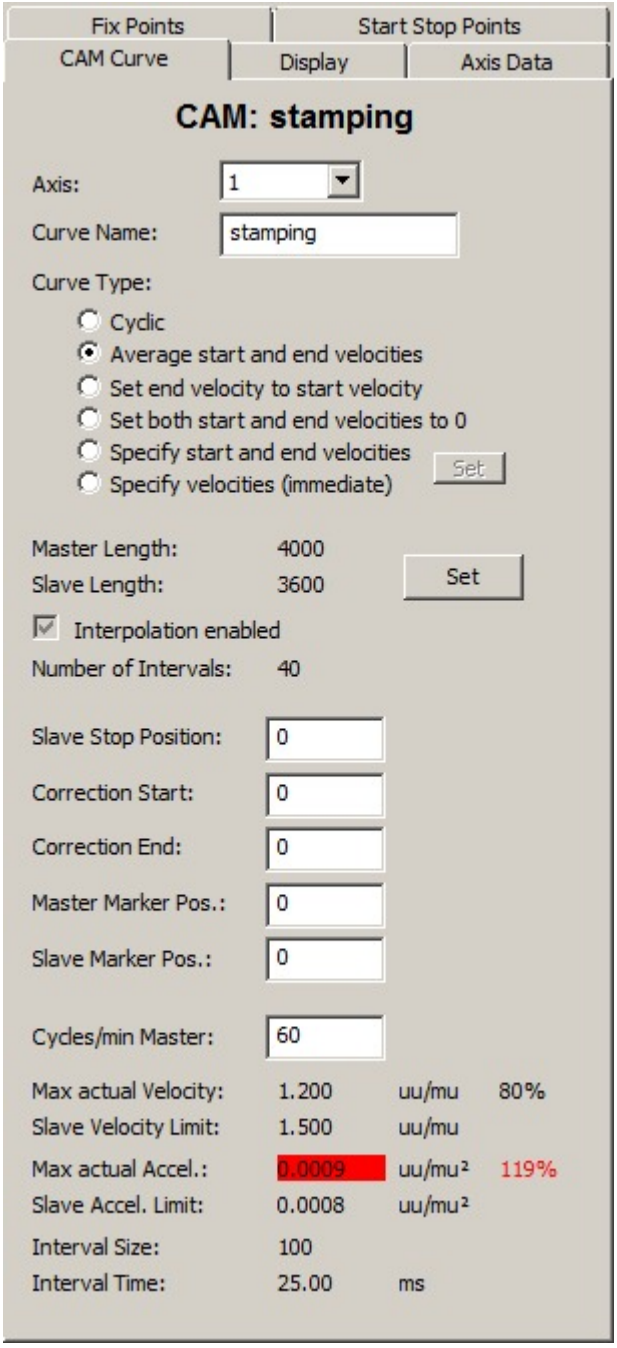
Delete the identified fixed point.

Snap on Grid

Snap this fixed point to the nearest interpolation grid line.

5.1.3.2 CAM Curve Tab

This tab is used for settings various curve-related parameter values and attributes.



Axis

This allows the user to specify which axis is intended to be used with the curve profile. This is important since the synchronization parameters are specific to the selected axis and will affect the shape of the curve profile.

Name of Curve

If you edit several curves, you can give meaningful names to the curves. Names may be up to 16 characters long and will be displayed in the “Array” field in the CAM Editor toolbar. These names are for your own information; the controller does not use them when working with curve profiles.

Type of Curve

Typically, curve profiles are cyclic; when the controller reaches the end of a curve profile, it will restart the curve profile from the beginning again. As a result, it is extremely important that the gradients (i.e. velocity) of the curve profile matches at the start and end of the profile. This allows the slave motor to run smoothly from the end of the profile back onto the start of the profile as the master cycles around the profile.

> A discontinuity in the gradient would cause a hard velocity leap which could potentially damage the motor or connected machinery.

The **Type of curve** allows the user to select how the CAM Editor will match the start and end gradients. The following curve types are available:

Cyclic	The CAM Editor will calculate start and end velocities so that the curve defined by the fixed points is a true cyclic cubic spline (i.e. third degree curve).
Average start and end gradients	The actual slave velocity is not important at either the start or end of the profile. The CAM Editor will choose an average gradient simply to ensure smoothness.
Set end gradient to start gradient	The actual slave velocity at the start of the profile is important and should not be changed. The CAM Editor will adjust only the end gradient so that it matches the start gradient. The start gradient is not changed.
Set both start and end gradient to 0	The slave motor must be stopped when the master cycles. The CAM Editor will set both the start and end gradients to 0.
Specify start and end gradients	The actual slave velocity is important at *both* the start and end of the profile. The user must explicitly set the start and end velocities using the **Set** button. The CAM Editor will warn the user if the start and end gradients do not match but it is the user's responsibility to ensure than any discontinuity does not cause problems.
Specify gradients (immediate)	The user must explicitly set the start and end velocities using the **Set** button. However, the start gradient will be replaced with the *actual* current velocity when the curve is activated in the controller. This type of curve is intended to be used when an existing curve must be replaced by a different curve before the first curve has completed its cycle. For more information on this curve type, see Dynamic CAM Curves and Immediate Curves.

> Not all versions of controllers can support all types of curve profiles.

Master Length and Slave Length

The master length is the distance that the master encoder must travel in order to complete one cycle; it is the horizontal length of the curve profile. As the master completes one cycle, the slave motor will also complete one cycle by following the curve profile from the start point to the end point. The slave length will be the difference in the slave position from the beginning to end of the

curve. Note that this is *not* the same as the vertical height of the curve. If the slave has returned to its starting position when the master completes one cycle, then the slave length will be 0 even though the slave has moved during the cycle.

The master length and slave length can be set in three ways:

1. Changing the last fixed point by double clicking on it in the Fix Points Tab.
2. Moving the last fixed point using the mouse in the curve profile diagram.
3. Pressing the **Set** button.

Both of the first two methods change the master and slave cycle lengths directly. If the **Set** button is pressed, then a dialog is displayed that allows both the master cycle length and the number of interpolation intervals (described below) to be changed simultaneously. The slave length cannot be changed using this dialog.

Note that the user is prevented from shortening the master cycle length beyond certain limits. The last fixed point cannot be moved to a position preceding

- the previous fixed point,
- any Start Stop point,
- either of the correction start or end points,
- or the master marker position.

If the last fixed point must be moved beyond these limits, then it will be necessary to first change the limiting points or positions before moving the last fixed point. For example, if the last fixed point is being blocked by the master marker position, then move the master marker position first and then move the last fixed point.

Slave Stop Position

This specifies the position to which the slave should run and stop if no SyncCamStop command has been executed to set the slave stop position.

This position will also be used if SyncCam starts with a specific number of cycles and does not use a SyncCamStop command.

If ☑ **Slave Stop** is enabled, then a horizontal grey bar will indicate this position in the curve profile diagram.

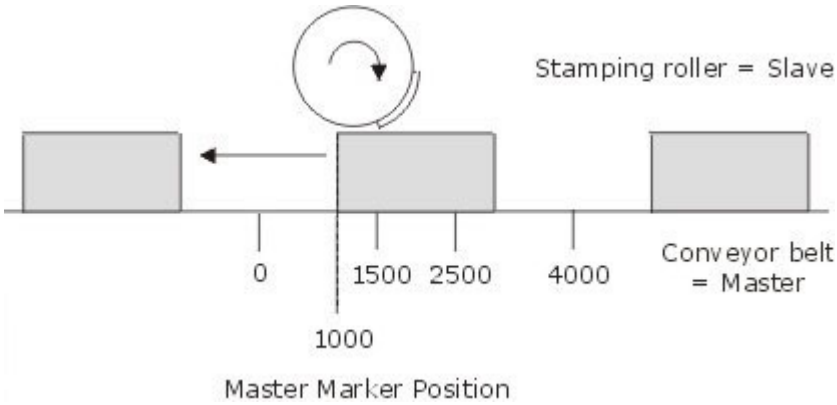
Correction Start and End

This specifies the master positions where the master correction should begin and end. Be careful to leave enough time to correct the synchronization before the processing point is reached.

If **Correction** is enabled, then the correction area is shown as a blue-grey bar along the X-axis in the curve profile diagram.

Master and Slave Marker Positions

Enter the master position (or the slave position in the case of slave synchronization with marker) for which the marker has been set. For example, the beginning of the box in the illustration below is the master marker position.



The position of the curve where the marker is detected is calculated from the master marker position and the marker distance. This allows you to fix the correction area.

If ☑ **Master Marker** is enabled, then a vertical green line will indicate the master marker position in the curve profile diagram. If ☑ **Slave Marker** is enabled, then a horizontal green line will indicate the slave marker position.

Interpolation and Number of Intervals

Controllers containing firmware older than version 6.4.44, require “interpolation points” when processing curve profiles. For newer controllers, these are no longer necessary and this topic can be skipped entirely. The firmware version of the controller is displayed in the ApossIDE Communication Window when the controller is first connected in the ApossIDE.

If the curve profile is intended for an older controller, then the ☑ **Interpolation enabled** checkbox should be selected. This can be done either when a new curve profile is added or by using the **Set** button. It cannot be done directly from this tab.

Older controllers require many more fixed points than newer controllers. If interpolation is enabled, then the CAM Editor will do this automatically for the user by adding interpolation points (i.e. “virtual” fixed points) on a uniformly spaced “interpolation grid”. The size of the interpolation interval (i.e. the distance between interpolation points) determines the accuracy with which the controller can follow the curve profile and must be chosen by the user based on the accuracy requirements of the application. Smaller interpolation intervals will give higher accuracy but will result in more interpolation points and larger arrays in the controller. Very small intervals can also cause the controller to have performance problems as a result of the large number of short segments. The “Interval Time” (found at the bottom on this tab) should not be smaller than 20-30 ms.

In order to avoid a fractional number of interpolation points, the **Master Length** is required to be an integral multiple of the interpolation interval. To ensure this, the CAM Editor will always snap the last fixed point to the interpolation grid. When the **Set** button is used, the CAM Editor will also ensure that the new **Master Length** is a multiple of the interpolation interval. As well, when the **Master Length** is changed with the **Set** button, the number of interpolation points is changed automatically so that the interpolation interval (i.e. the curve accuracy) remains unchanged. It is recommended that the master cycle length be at least 1000 in order to allow for an adequate resolution of the interpolation grid.

Although not strictly necessary, it is recommended that user fixed points also be placed exactly on the interpolation grid. The **Snap all to grid** toolbar button and the ☑ **Snap on Grid** checkbox option can be used to assist the user in placing fixed points on the interpolation grid. If fixed points are not placed on the interpolation grid, then the actual curve profile followed by the controller will pass near the off-grid fixed points but may not pass exactly through them.

Cycles/min Master

This specifies the number of cycles of the master per minute. In most cases, this will be the (maximum) number of products that are processed per minute.

Maximum Actual Velocity

Max actual Velocity indicates the maximum velocity of the slave required by this curve profile in user units per master unit (uu/MU). This is also shown as a percentage of the **Slave Velocity Limit**. If the velocity exceeds the limit, then it is shown in red.

You can graphically display the velocity in the curve profile diagram by enabling ☑ **Velocity**. The velocity curve of the slave will be shown in blue and the axis will be displayed on the right side of the chart (as long as acceleration and/or jerk are not also being displayed). Note that it may be necessary to press the **Reset Zoom** toolbar button for the velocity curve to be displayed with the proper scale within the existing chart boundaries.

Slave Velocity Limit

Slave Velocity Limit is the maximum velocity of which the slave is capable, given the current value of **Cycle / min Master** and the maximum speed rating of the slave motor. As **Cycle / min Master** increases, **Slave Velocity Limit** will decrease so that the actual velocity of the slave does not exceed the maximum speed rating of the slave motor. If the velocity required by this curve (**Max actual Velocity**) exceeds this limit, then the slave will not be able to follow the master.

If ☑ **Vel. Limit** is enabled, then the velocity limit will be displayed as horizontal dark-blue lines in the curve profile diagram. Note that both positive and negative velocity limits are displayed.

Maximum Actual Acceleration

Max actual Accel. indicates the maximum acceleration of the slave required by this curve profile in user units per master unit squared (UU/MU^2). This is also shown as a percentage of the **Slave Accel. Limit**. If the acceleration exceeds the limit, then it is shown in red.

You can graphically display the acceleration in the curve profile diagram, by enabling ☑ **Acceleration.** The curve of the slave will be shown in magenta and the axis will be displayed on the right side of the chart (as long as velocity and/or jerk are not also being displayed). Note that it may be necessary to press the **Reset Zoom** toolbar button for the acceleration curve to be displayed with the proper scale within the existing chart boundaries.

Slave Acceleration Limit

Slave Accel. Limit is the maximum acceleration of which the slave is capable, given the current value of **Cycle / min Master** and the maximum acceleration rating of the slave motor. As **Cycle / min Master** increases, **Slave Accel. Limit** will decrease so that the actual acceleration of the slave does not exceed the maximum acceleration rating of the slave motor. If the acceleration required by this curve (**Max actual Accel.**) exceeds this limit, then the slave will not be able to follow the master.

If ☑ **Accel. Limit** is enabled, then the acceleration limit will be displayed as horizontal dark-magenta lines in the curve profile diagram. Note that both positive and negative acceleration limits are displayed.

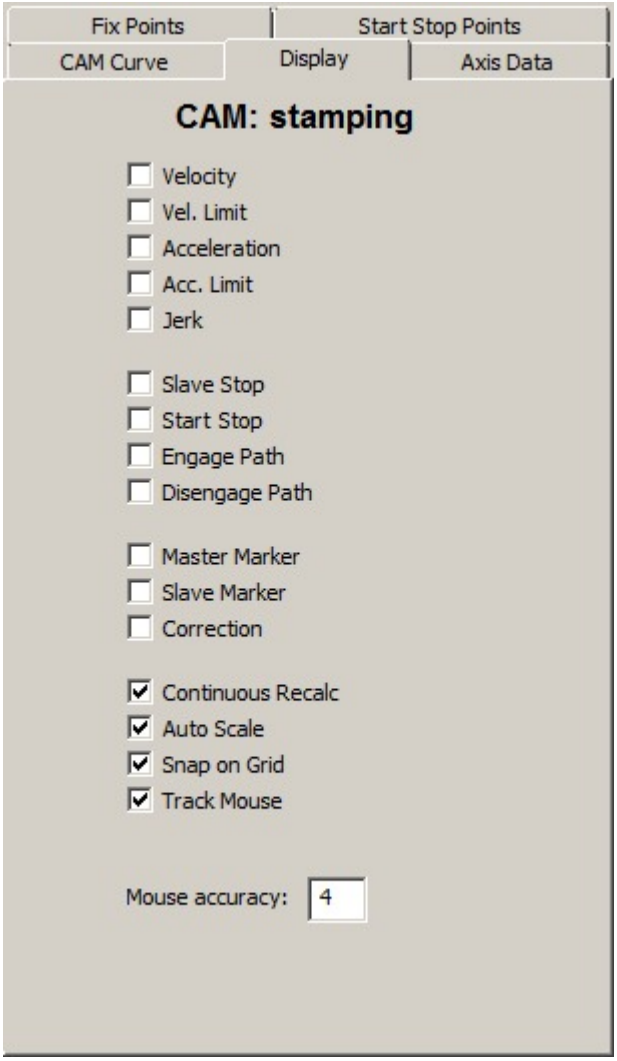
Interval Size and Interval Time (ms)

These values are only important if interpolation has been enabled. The interval size is the length of a single interpolation interval in master units. It is derived from the number of intervals and the length of the master cycle.

The interval time (in milliseconds) is derived from the interval size and the number of cycles of the master per minute. This time should not be smaller than 30 milliseconds. 30 to 100 ms are suitable values. Changes should be made by pressing the curve length **Set** button in order to ensure a reasonable value.

5.1.3.3 Display Tab

This tab contains checkboxes for displaying and hiding various attributes of the CAM Curve Diagram.



Velocity

Display the velocity curve corresponding to the curve profile. This is displayed as a blue line. The grid scale for this curve is displayed down the right side of the curve profile diagram (as long as this is the only grid to be displayed on the right side of the chart).

Vel. Limit

Display the velocity limits as two horizontal dark blue lines.

Acceleration

Display the acceleration curve corresponding to the curve profile. This is displayed as a magenta line. The grid scale for this curve is displayed down the right side of the curve profile diagram (as long as this is the only grid to be displayed on the right side of the chart).

Acc. Limit

Display the acceleration limits as two horizontal dark magenta lines.

Jerk

Display the "jerk" curve corresponding to the curve profile. The "jerk" is the impulse on the drive as a result of acceleration changes. This is displayed as a cyan (light blue) line. The grid scale for this curve is displayed down the right side of the curve profile diagram (as long as this is the only grid to be displayed on the right side of the chart).

Slave Stop

Display the slave stop position (defined in the CAM Curve Tab). This is displayed as a horizontal grey bar.

Start Stop

Display the positions of the Start Stop points. These are displayed as yellow or red flags on the curve profile.

Engage Path

Display the engage path for the currently selected Start Stop pair. This is displayed as a dark blue curve from the slave stop position to the curve profile between the Start Stop pair.

Note that a Start Stop pair must be selected in the Start Stop Points Tab before anything is displayed.

Disengage Path

Display the disengage path for the currently selected Start Stop pair. This is also displayed as a dark blue curve.

Master Marker

Display the master marker position (derived from the data in the CAM Curve Tab and the Axis Data Tab). This is displayed as a vertical green line. If a tolerance has been specified, then the tolerance area is displayed as a pale green rectangle underneath the master marker position.

Slave Marker

Display the slave marker position (derived from the data in the CAM Curve Tab and the Axis Data Tab). This is displayed as a horizontal green line. If a tolerance has been specified, then the tolerance area is displayed as a pale green rectangle underneath the slave marker position.

Correction

Display the positions where the master correction should begin and end (defined in the CAM Curve Tab). This is displayed as a blue-grey bar along the bottom axis of the curve profile diagram.

Continuous Recalc

If this flag is checked, then the curve profiles will be continuously recalculated and redisplayed as fixed points are dragged in the curve profile diagram. This redrawing of the curve may require a significant

amount of processing power. If this redrawing behaves erratically or creates a disruption on your computer, then you can deactivate it; the curve will then be recalculated and displayed as soon as you release the mouse button.

Auto Scale

If this flag is checked, then the curve profile diagram will be automatically re-scaled so that the entire curve is always visible.

Note that if the diagram has been manually zoomed or scrolled, then this flag will be automatically cleared so that the user's area of interest will remain visible.

Snap on Grid

If this flag is checked, then fixed points will be snapped to the nearest interpolation grid point whenever they are moved. It is recommended that this flag always be checked. Fixed points which do not fall on interpolation points, will not lie on the curve profile.

Note that the last fixed point is *always* snapped to the interpolation grid, regardless of this flag, so that the proper relationship can be maintained between the master cycle length and the number of interpolation points.

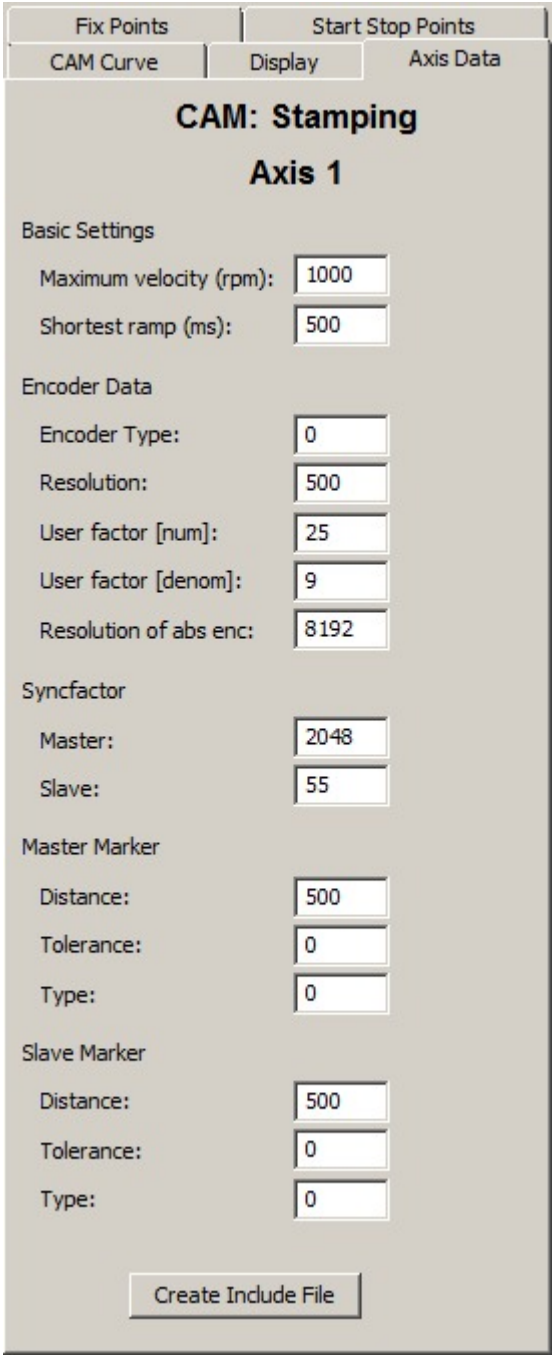
Mouse accuracy

If the right mouse button is pressed while the mouse is pointing at a fixed point, then the fixed point popup menu is displayed. Otherwise, the normal popup menu is displayed. This value specifies how near to a fixed point the mouse must be before the CAM Editor will recognize that the mouse is pointing at that fixed point. Note that a "hand" mouse icon (rather than the normal arrow mouse icon) will be displayed when the CAM Editor has determined that the mouse is pointing at a fixed point. That fixed point will also be highlighted in the Fix Points Tabso that the user can positively identify the point.

5.1.3.4 Axis Data Tab

This tab allows various axis parameters that are important to CAM curves to be changed. These axis parameters will be downloaded to the controller along with the curve profiles when the configuration file is downloaded. The axis synchronization parameters are of particular interest when working with curve profiles since they affect the synchronization of the slave to the master. The relevant parameters are described below.

Please see Parameter Reference for a detailed description of the axis parameters. A detailed help description of any parameter in the Axis Data tab may be displayed by placing the text cursor in the parameter field (e.g. by clicking on the field) and then pressing **F1**.



Syncfactor Master and Slave

The two parameters SYNCFACTM and SYNCFACTS are used to determine the MU units in the CAM control.

Distance

This is the distance of the sensor to the processing point. SYNCMPULSM is the master marker distance and SYNCMPULSS is the slave marker distance. The position of the curve where the marker is detected is calculated from the master marker position and the marker distance.

If ☑ **Master Marker** or **Slave Marker** is enabled, then these positions will be shown as a green line in the curve profile diagram.

Tolerance

This is the tolerance window for the appearance of the master marker SYNCMWINM or the slave marker SYNCMWINS (Marker monitoring).

If ☑ **Master Marker** or **Slave Marker** is enabled, then the tolerance window is shown as a green area in the curve profile diagram.

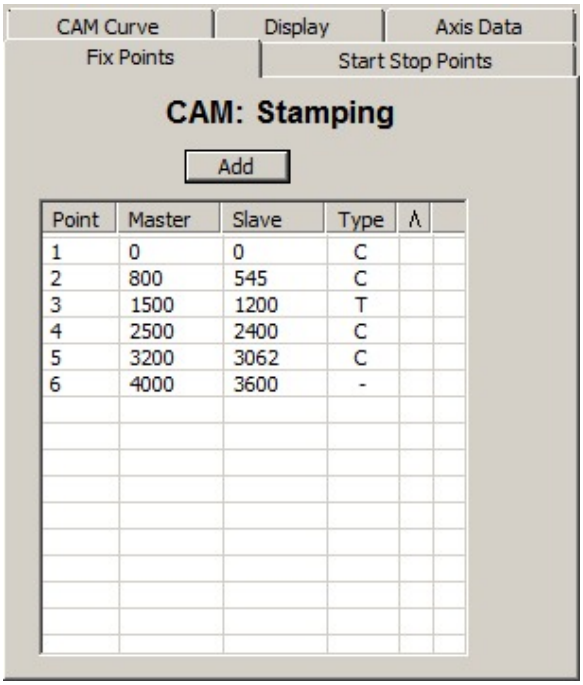
Create Include File

For MACS5 and newer controllers, axis parameters are *not* maintained permanently in flash memory. Axis parameters are saved in the configuration file but these axis parameters are *not* restored to the controller by **Controller** → **Parameters** → **Restore from file**. For these controllers, **Create Include File** can be used to create an include file that can be used with application programs. Please see Initializing Temporary Parameters for a more detailed description of axis parameter handling in these controllers.

5.1.3.5 Fix Points Tab

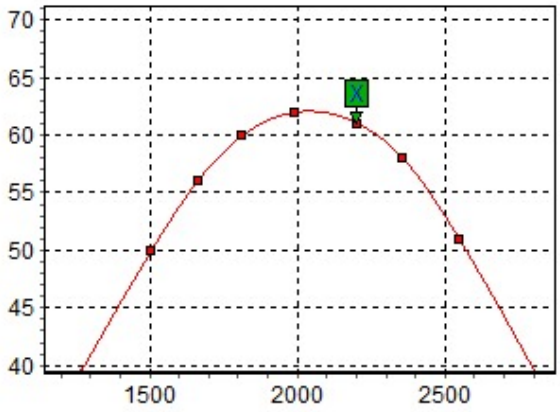
Fixed points are used to define the basic shape of the curve over the length of the master cycle. The CAM Editor will fit a mathematical spline through these fixed points in order to create a smooth curve. Note that you should not define more fixed points than are necessary to adequately produce the final desired curve profile. Adding more points than necessary will consume controller resources without a corresponding increase in benefit.

Note that any change made to fixed points using the **Fix Points Tab**, can be undone using the **Undo** toolbar button.



Identifying Fixed Points

Any fixed point may be identified in the curve by clicking on it with the left mouse button. This will cause a green flag to be displayed identifying the point in the CAM Curve Diagram as is shown below:



Editing Fixed Points

Please see "Using the Mouse" in CAM Curve Diagram for a description of how to add, delete, and modify fixed points using the mouse in the curve profile diagram.

Adding Fixed points

To add a new fixed point, click the **Add** button. This will display the dialog for adding points. Points must be specified as pairs of master and slave values and multiple pairs can be defined at the same time. The fixed points can be entered in any order; they will be automatically ordered as they are being added to the curve. If **Snap on Grid** is enabled, then the master values will be snapped onto the interpolation grid as they are being added.

If a new fixed point is added after the last fixed point, then this will change the master cycle length. When this happens, the number of interpolation points is changed automatically so that the interpolation interval (i.e. the distance between interpolation points) remains unchanged. The last fixed point is *always* snapped to the interpolation grid in order to avoid a fractional number of interpolation points.

Deleting Fixed Points

To delete a fixed point, click on the fixed point and then press the **Del** key.

Modifying Fixed Points

To modify a fixed point, double-click on the fixed point in the **Fix Points Tab**. If the “Master” or “Slave” column is clicked, then the fixed point position can be changed. If the “Type” column is clicked, then the fixed point type can be changed. If the "Λ" (Corner) column is clicked, then the fixed point will be toggled between a "corner" point and a "non-corner" point. If necessary, pressing the **Esc** key will restore the original value. If the value has already been committed, then the **Undo** toolbar button can be used to return to the previous value.

Note that a fixed point cannot be moved beyond either of the two adjacent fixed points. The last fixed point can be moved as far as desired towards the right. If this is done, then the master cycle length is automatically changed. The number of interpolation points is also changed automatically so that the interpolation interval (i.e. the distance between interpolation points) remains unchanged. The last fixed point is always snapped to the interpolation grid in order to avoid a fractional number of interpolation points.

Fixed Points Types

The fixed point type determines the shape of the single curve profile segment that *follows* the fixed point. Segments can either be straight (i.e. tangent) or curved. The CAM Editor will always try to make a smooth transition from one segment to the next. For this reason, at least one curve segment is required between adjacent tangent segments.

In the curve profile diagram, tangent points are displayed as small green squares and all other types of points are displayed as small red squares.

The following fixed point types are supported:

Curve (C)

The following segment is curved. If it connects adjacent tangent segments, then it will be calculated using a single 5'th order curve. This provides the smoothest possible transition between the segments since both the velocity and acceleration will match exactly at both ends of the segment. However, it may require higher velocities and accelerations than other curve types. If there is more than one consecutive curve segment, then the series of curve segments will be calculated together using a single cubic spline.

Tangent (T)

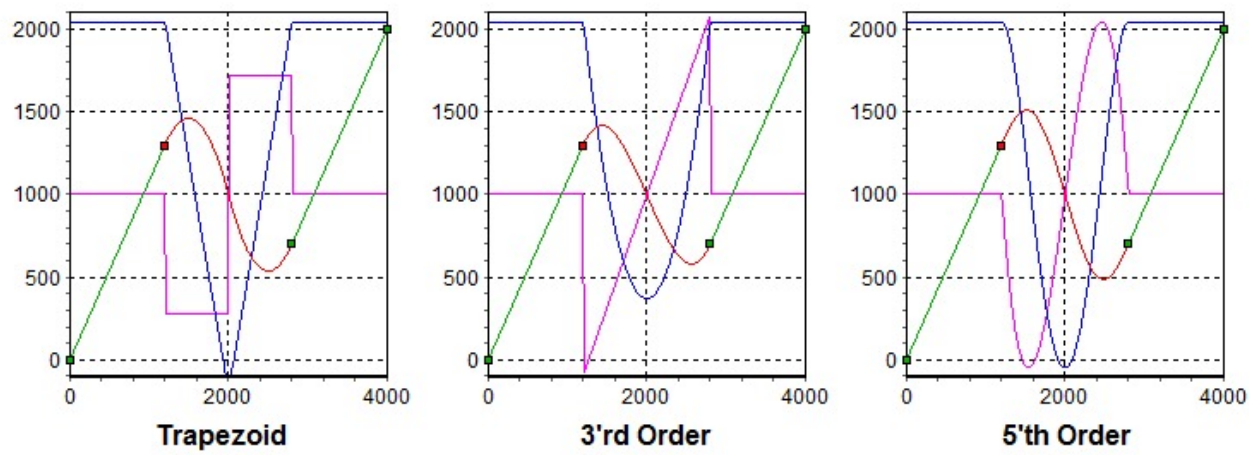
The following segment is straight. The velocity will be constant and the acceleration will be zero.

Trapezoid (CZ)

The following segment will be calculated using a trapezoidal velocity profile (i.e. two segments of constant acceleration). This curve type can only be used between adjacent tangent segments.

3'rd order (C3)

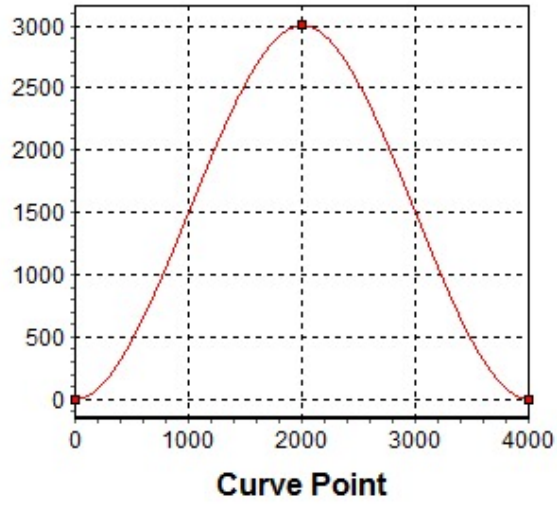
The following segment will be calculated using a single 3'rd order curve. This curve type can only be used between adjacent tangent segments.

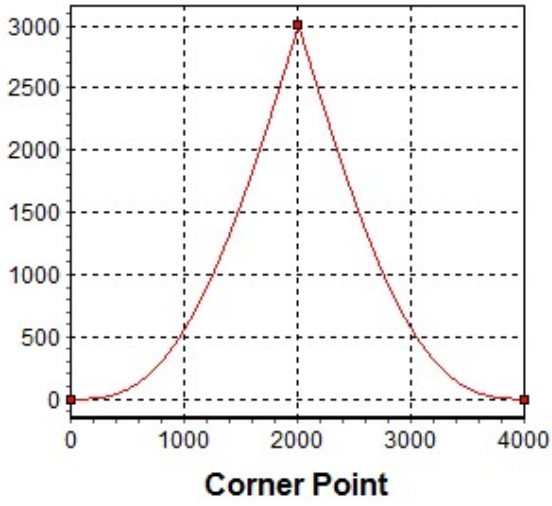


In the trapezoid curve, the two segments of constant acceleration (magenta line) can be seen. This produces a sharp change in velocity (blue line) midway through the curve. In the 3'rd order curve, the velocity change is smoother but there are still sharp changes in the acceleration. In the 5'th order curve, the changes in both the velocity and acceleration are smoother.

Corner Points

Normally, CAM curves will be "smooth"; the velocity will match before and after each fixed point. However, some specialized applications may require a sharp change in position at one or more specific fixed points. This is shown below.





This is done by setting the appropriate fixed points to be "corner points". Corner points will be identified by the "Λ" symbol in the "Corner" column of the **Fix Points Tab**. Any fixed point *except* the two ends of Trapezoid and 3'rd order curve segments, may be designated as a corner point. Trapezoid and 3'rd order curves do not support corners since these curves are specifically designed to match velocities, whereas corners specifically indicate that velocities do *not* match.

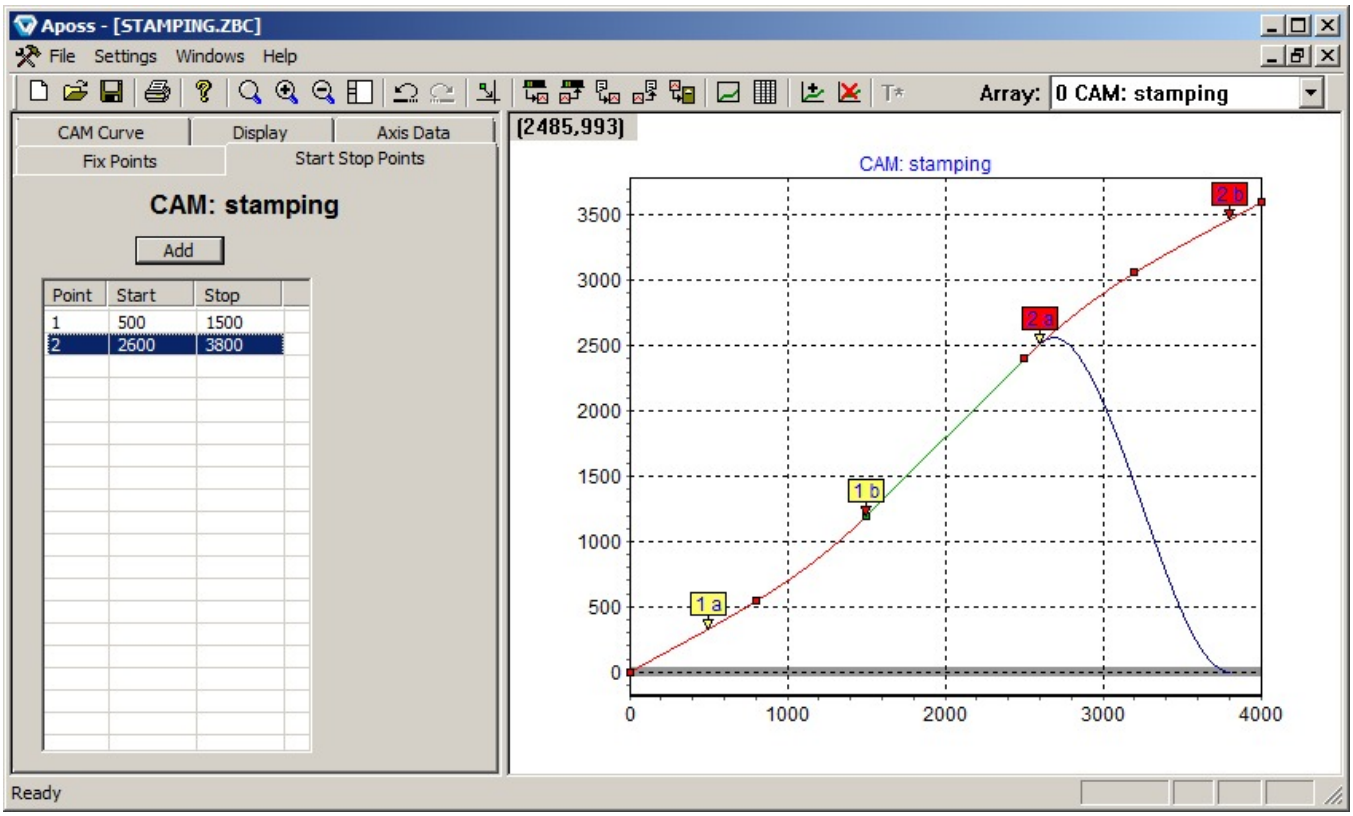
Mathematically, the shape of the curve at the corner point will be determined by setting the accelerations immediately before and after the corner point to 0 and allowing the velocities to "float". This will result in the velocity being discontinuous at the corner point (i.e. the velocity will "jump"). The acceleration will effectively be "infinite".

> Corner points should *ONLY* be used by applications that are specifically designed to handle them. Generally, this means stopping the motor at the corner point in order to avoid the "infinite" acceleration. Running through a corner point at non-0 velocity, will cause the motor to experience a severe jolt as it attempts to instantaneously change velocity.

> Normally, the CAM Editor will prevent CAM curves from containing adjacent tangents since the resulting curve cannot be guaranteed to be smooth at the point where the tangents join. However, if that point is made a corner point, then the CAM Editor will allow adjacent tangents since the user has specified that a sharp change is expected. To do this, ensure that the type of the appropriate fixed point is set to "Curve", then toggle the point to be a "corner", and finally change the fixed point type to "Tangent".

5.1.3.6 Start Stop Points Tab

Start Stop Points are used to define point pairs for engaging and disengaging the slave during synchronization. You will need one point pair to specify the master position where the synchronization should start and where the engaging should take place. You can use an additional point pair to specify the master position where the disengaging should start and where the synchronization should be stopped.



If ☑ **Start Stop** is enabled, then Start Stop points will be displayed with yellow flags. The Start point will be labeled with “a” and have a yellow arrow. The Stop point will be labeled with “b” and have a red arrow. If the Start and Stop points are the same (see below), then the point will be labeled with "ab". Selecting a Start Stop pair by clicking on it in the Start Stop Points Tab will cause the flags to be displayed in red. This will allow you to identify which pair is which.

Enable ☑ **Engage Path** and/or ☑ **Disengage Path** to display the engage and/or disengage curves. Then select the Start Stop pair of interest by clicking on it in the Start Stop Points table. Only one pair of engage and disengage curves can be displayed at a time.

Enable ☑ **Slave Stop** to display the slave stop position. This is the position where the engage curve starts and the disengage curve ends. It will be displayed as a horizontal gray bar.

> When the master moves forward engaging is shown by the engage path and disengaging is shown by the disengage path. When the master moves *backward*, engaging is shown by the *disengage* path and disengaging is shown by the *engage* path!

The order of the points in each pair is important and will automatically be taken into account by the run direction: When moving forward, the synchronization begins at the Start point and ends at the Stop point. When moving backward, it begins at the Stop point and ends at the Start point.

Up to 25 Start Stop point pairs can be defined. This is useful, for example, to define multiple starts and stops in a cycle in order to account for different situations during the start. Using the commands “SyncCamStart pnum” and “SyncCamStop pnum slavepos”, you can determine in your program which point pair is to be used.

If **Start points = Stop points**, the slave will be engaged with the set maximum velocity (i.e. without a curve) as soon as the master has reached this point.

If **Start point > Stop point**, then the engaging/disengaging operation will “wrap around” the master cycle.

If *no* **Start Stop points** have been defined, then SYNCCSTART will cause the slave will be engaged with the set

maximum velocity.

If the program is closed without the explicit command “SYNCCSTOP pnum slavepos”, then the second point pair will always be used for disengaging.

Editing Start Stop points

Start-Stop points are edited in the Start Stop table:

Adding Start Stop points

To add a new pair of Start Stop points, click the **Add** button. This will display the dialog for adding point pairs. Points must be specified as pairs of start and stop values and multiple pairs can be defined at the same time. Unlike fixed points which must be ordered, Start Stop pairs can be entered in any order.

Deleting Start Stop points

To delete a pair of Start Stop points, click on the Start Stop point pair in the table and then press the **Del** key.

Modifying Start Stop points

To modify a pair of Start Stop points, double-click on the Start Stop point pair in the table. This will display a dialog allowing the pair to be changed.

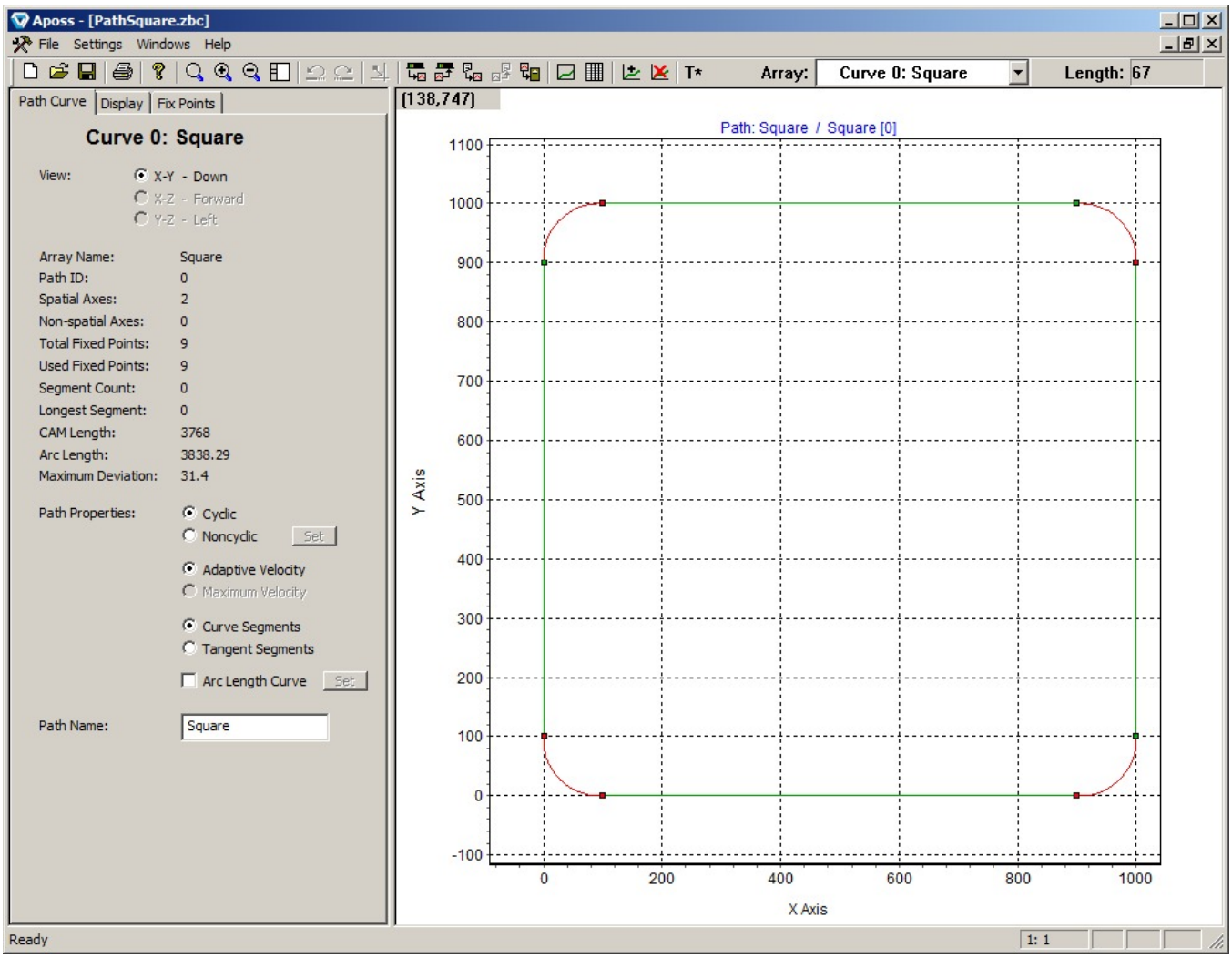
5.1.4 Editing Path Curves

If internet access is available, then a short online tutorial can be viewed here:

Creating a Path Curve

When a Path curve array is selected in the **Array** box in the toolbar, the CAM Editor window will appear similar to that shown below. The path used in this example is available as a sample called “ZBC Configuration Files/PathSquare.zbc” (see Sample).

> Unlike CAM curve arrays that support only a single CAM curve per array, Path curve arrays support multiple sub-path curves per array. For more information, see Multiple Paths.

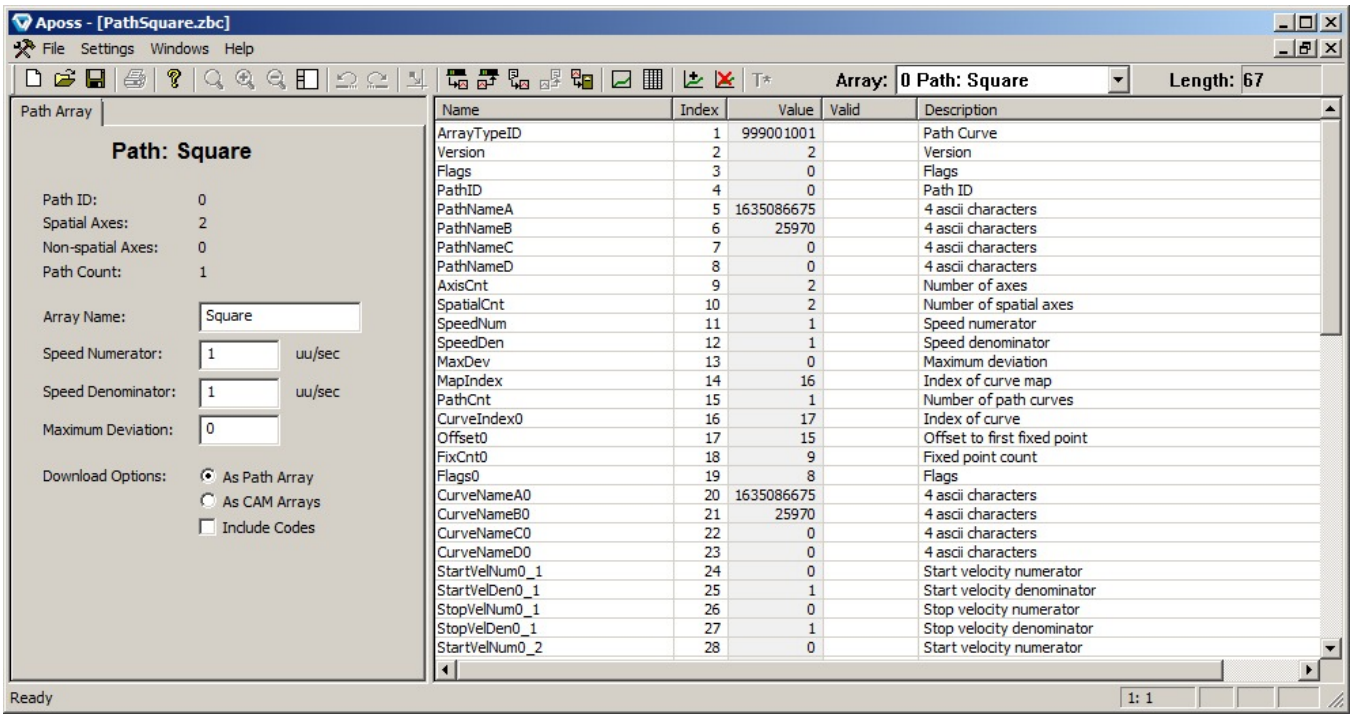


The window is divided into two sections:

- Path Curve Diagram
- Tabs:
  - Path Curve Tab for setting various curve-related parameter values.
  - Display Tab for displaying and hiding various attributes of the curve profile diagram.
  - Fix Points Tab for editing the fixed points belonging to the curve.

The window can be resized in the standard way. As well, the splitter bar between the two window sections can be adjusted using the mouse in the standard way. The splitter bar can be restored to its default positions by using the **Reset splitters** toolbar button.

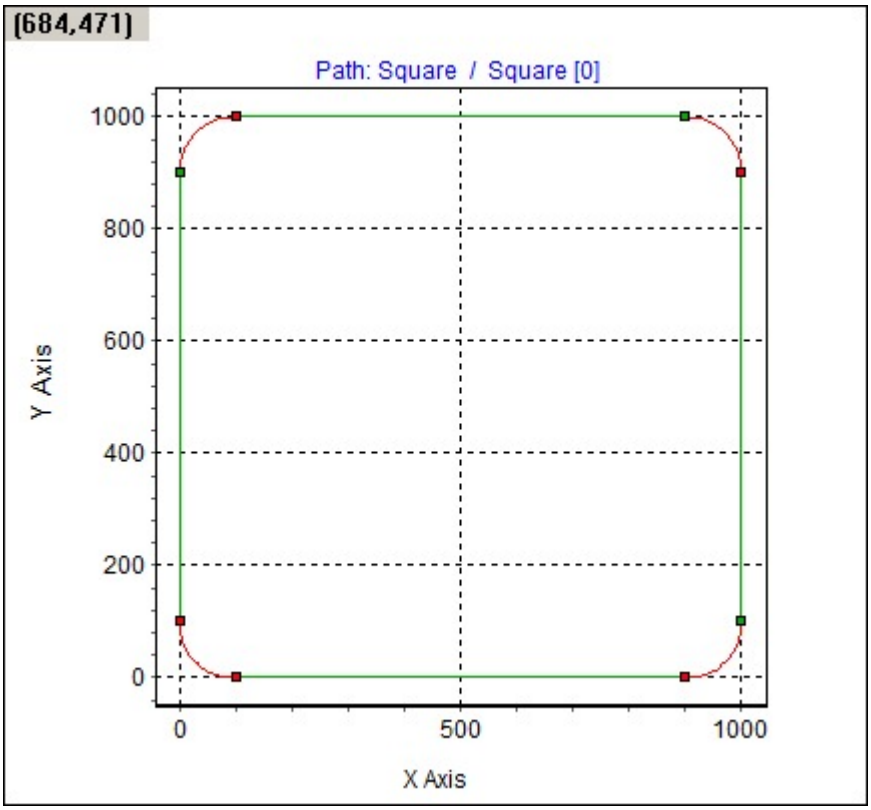
The actual Path array that will be downloaded to the controller can be seen if the display is switched to text mode using the **Text mode** toolbar button and selecting the Path array itself rather than the Path curve within the array. In this case, the CAM Editor window will appear similar to that shown below. Note that the array cannot be modified when text mode is being used.



5.1.4.1 Path Curve Diagram

The path curve diagram graphically displays the curve. It also allows fixed points to be added, deleted, and moved using the mouse.

Path curves use a *right-handed* coordinate system. For a 3-dimensional path, the X axis will point towards the right, the Y axis will point "away" from the user, and the Z axis will point "up". The default view of the path is a view showing the projection onto the XY plane. If available, other views can be chosen simply by selecting a different **View** at the top of any of the tab windows.



Please note the following about the path curve diagram:

- The name of the curve, if any, is shown at the top of the diagram. The Path array name is shown first, followed by the name of the sub-path. The index of the currently selected sub-path is shown in brackets at the end. In the example above, both the array and the curve are named "Square".
- If mouse tracking is enabled (see Display Tab), then the position of the mouse will be shown in the top-left corner of the diagram.
- Fixed points are displayed as small red, green, or yellow squares. “Curve” points (i.e. fixed points that begin a curved segment) will be red and “tangent” points (i.e. fixed points that begin a tangent segment) will be green. If points have been thinned (see Thinning Fixed Points), then they will be yellow.
- Curved segments of the profile will be drawn red and tangent segments will be drawn green. Note that segments that are straight will be drawn green even though they may be “curve segments” (i.e. segments that begin with a “curve” point).

Zoom and Pan

The mouse buttons can be used in the curve profile diagram for the following functions:

Zoom

Zoom in to an area of interest in the diagram by pressing the left mouse button, moving the mouse across the area of interest, and then releasing the mouse button.

> Unlike the CAM Curve Diagram, the selected rectangle will *always* be stretched so that it is exactly displayed in the window. The **Ctrl** key cannot be used to deform the coordinate system.

Use the **Zoom Out** or **Zoom Reset** toolbar buttons to zoom back out again.

Zooming the diagram will automatically clear the ☑ **Auto Scale** setting.

Pan

Pan across the diagram (i.e. move the diagram in the window) by pressing the right mouse button, moving the mouse, and then releasing the mouse button.

Panning the diagram will automatically clear the ☑ **Auto Scale** setting.

Use **Zoom Reset** to reset both the pan and zoom so that the entire diagram is displayed again.

Using the Mouse to Edit Fixed Points

Fixed points can be edited interactively using the mouse within the diagram. They can also be input directly in the Fix Points Tab.

Identify Fixed Point

Fixed points can be identified simply by moving the mouse and pausing over the fixed point in the diagram. The mouse cursor icon will change from the normal arrow icon to a hand icon and the corresponding fixed point will automatically be highlighted in the Fix Points Tab. The required "nearness" of the mouse to the fixed point can be adjusted using **Mouse accuracy** in the Display Tab.

Move Fixed Point

A fixed point can be moved by dragging it with the left mouse button. Make sure that the hand icon is displayed before trying to drag the fixed point. The point's position will be automatically updated in the Fix Points Tab as the point is being dragged.

If ☑ **Continuous Recalc** is enabled in the Display Tab, then the curve profile will also be automatically recalculated and the Path Curve Tab will be updated as the point is dragged.

Add Fixed Point

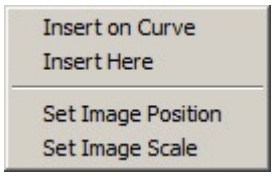
A new fixed point can be added by moving the mouse to the point where the fixed point is to be placed and then double-clicking the left mouse button. Please see Fix Points Tab for more information on adding fixed points.

If **Append Fixpoint** in selected in the Display Tab, then the new point will always be appended to the end of the existing path. If **Insert Fixpoint** is selected, then an attempt is made to intelligently insert the point into the existing path.

> Inserting an new point may not always produce the desired result and the new point may be inserted in the wrong position in the point sequence. If this happens, then delete the new point, re-insert a new point close to the existing curve, and then move the new point to its correct position.

Using Popup Menus

If the right mouse button is clicked in an empty part of the diagram, then the following popup menu will be displayed:



Insert on Curve

A new fixed point is inserted on the existing curve at the point closest to where the mouse button was clicked.

> The selection of **Append Fixpoint** or **Insert Fixpoint** is ignored by **Insert on Curve**. The new point will

> *always* be inserted into the existing path.

Insert Here

A new fixed point is inserted at the position where the mouse was clicked. This is the same as double-clicking with the left mouse button. The path will be recalculated.

> As with double-clicking, the selection of **Append Fixpoint** or **Insert Fixpoint** will determine whether the new point is appended to the path or inserted into the existing path.

Set Image Position

Set the position of the background image. See Background Images.

Set Image Scale

Set the scale of the background image. See Background Images.

Change Fixed Point Type

If the right mouse button is clicked on a fixed point, then the following popup menu will be displayed to change this fixed point type:

Select the type to which the identified point should be changed: **Curve**, **Tangent**, **Trapezoid**, or **3'rd Order**. See Fix Points Tab.

Corner

Toggle the identified fixed point between a "corner" point and a "non-corner" point. See Fix Points Tab.

Stop

Toggle the identified fixed point between a "stop" point and a normal point. See Fix Points Tab.

Delete

Delete the identified fixed point.

Set Image Position

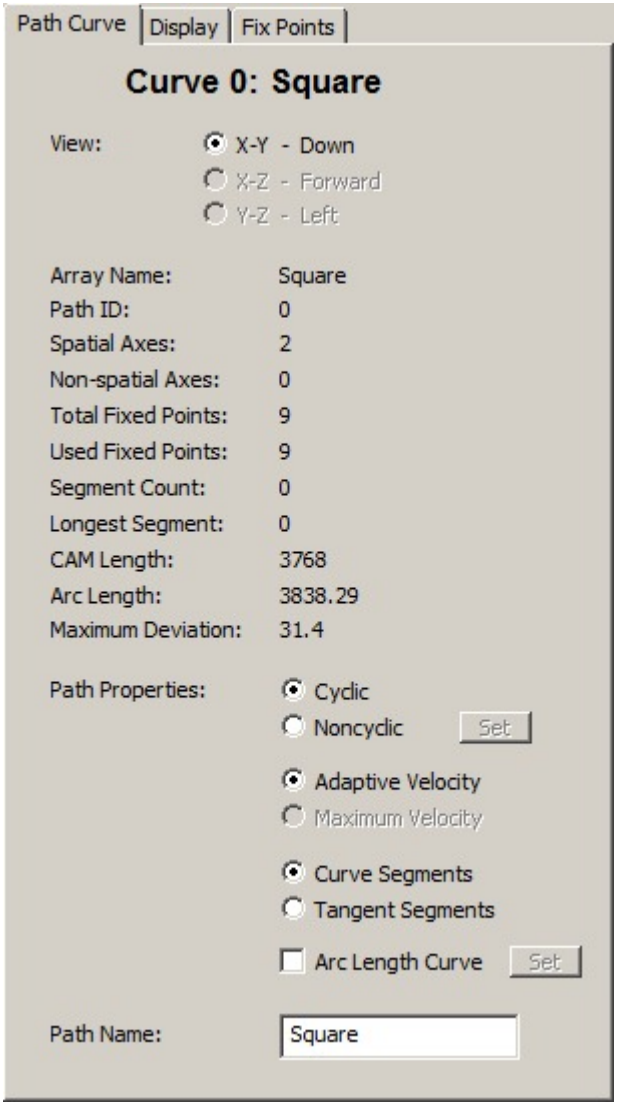
Set the position of the background image. See Background Images.

Set Image Scale

Set the scale of the background image. See Background Images.

5.1.4.2 Path Curve Tab

This tab is used for setting various curve-related parameter values and attributes.



Array Name

This is the name of the Path array containing this curve.

Path ID

Each Path curve array is given a unique user-assigned path ID when the path curve is first created. The ApossIDE assigns no meaning to this number. The number is used by the CAM Editor to reconstruct a complete path curve when it has been downloaded to a controller only as component CAM curves and then subsequently uploaded again into the CAM Editor. Application programs are also free to use this number to identify path curves.

Spatial Axes

The number of spatial dimensions defined in the path curve.

Non-spatial Axes

The number of non-spatial dimensions defined in the curve.

Total Fixed Points

The total number of fixed points defined in the path curve. This number includes any thinned fixed points.

Used Fixed Points

The number of fixed points that will be used in the final path curve. This number excludes any thinned fixed points.

Segment Count

Some applications use the fixed point "Code" value (see Path Fixed Points) to specify sequences of "pun up" and "pen down" points. For the purpose of **Segment Count** and **Longest Segment** only, the CAM Editor will treat a value of 0 as "pen down" and a sequence of consecutive "pen down" fixed points as a single "pen down segment". **Segment Count** will indicate the number of "pen down segments" embedded within the path curve.

Longest Segment

This is the number of "pen down" fixed points in the "pen down segment" with the most fixed points. It is *not* the arc length of the longest segment.

CAM Length

This is the **Master Length** (see CAM Curve Tab) of the individual CAM curves that will be created for each dimension of the path curve. All CAM curves created for the path will have the same master length. This will be approximately the sum of the straight line point-to-point distances between each fixed point in the path curve.

Arc Length

The actual arc length of the path curve.

Maximum Deviation

For path curves with very dense fixed points, the CAM Editor can automatically remove unnecessary fixed points. The **Maximum Deviation** value determines how many fixed points can be removed. See Thinning Fixed Points for more information.

Path Properties

A path curve can be specified as being either **Cyclic** or **Noncyclic**. If the path is cyclic, then the CAM Editor will ensure that the start and end velocities of the path are identical. If the path is noncyclic, then the user must explicitly set the path's start and end velocities using the **Set** button. See Path Curves for more information on cyclic and noncyclic paths.

A path curve can be calculated as either an **Adaptive Velocity** path or a **Maximum Velocity** path. For an adaptive velocity path, the *average* speed along each individual segment of the path will be the same (although the speed may vary within each segment). For a maximum velocity path, the CAM Editor will adjust the master coordinates within each path segment in order to maximize the speed along the segment without exceeding the maximum velocity and acceleration ratings of each individual axis (see VELMAXQC and ACCMAXQC).

A path curve can be composed of either **Curve Segments** or **Tangent Segments**. Normally, paths will be composed of curve segments (which can include non-adjacent tangent segments) and the CAM Editor will create a smooth curve through all fixed points. However, certain specially designed applications expect paths to be specified only as a series of straight line (tangent) segments. For

these applications, the path can be specified as containing only tangent segments. In this case, the application is responsible for handling the velocity discontinuities at the fixed points.

> **Tangent Segment** paths are supported only on controllers with firmware 7.1.00 or newer.

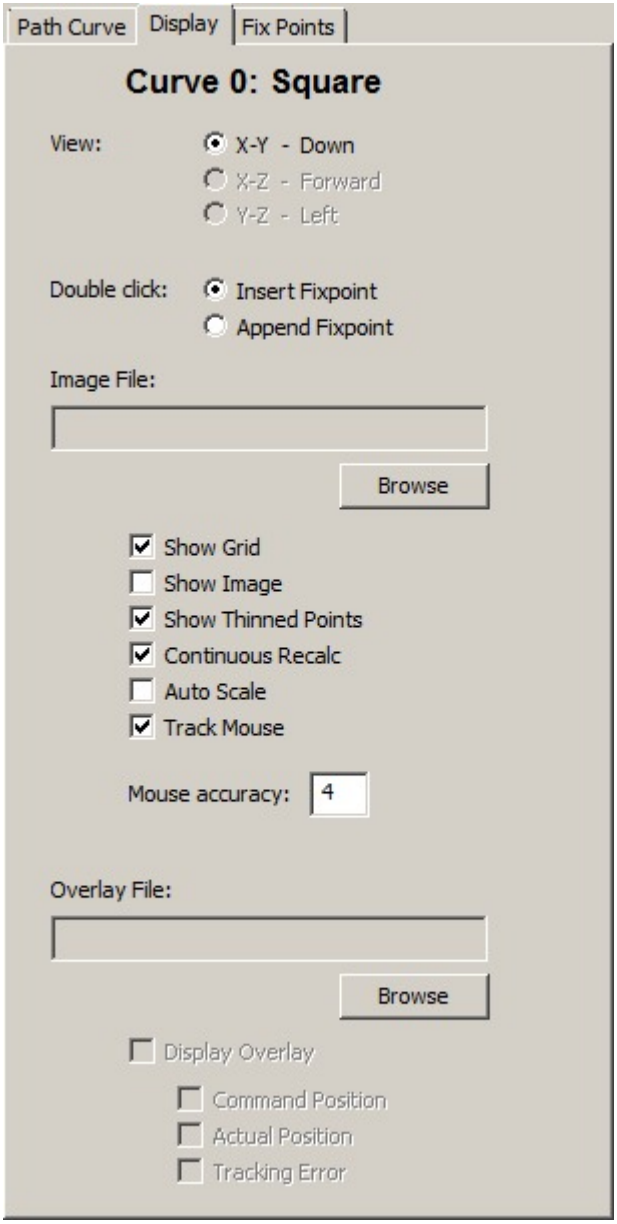
The basic CAM curves created to realize a path curve, will not produce a constant speed along the actual path. The speed will vary according to the mathematical curvature (sharpness) of the path. However, many applications require a *constant* speed along the entire length of the path. This can be achieved by enabling an **Arc Length Curve**. This is an auxiliary CAM curve that, when used to generate master coordinates for the other CAM curves, will produce a constant speed motion along the path. See Arc Length CAM Curves for more information.

Path Name

If the Path array is to contain several sub-path curves, you can give meaningful names to the curves. Names may be up to 16 characters long and will be displayed in the “Array” field in the CAM Editor toolbar. These names are for your own information; the controller does not use them when working with path curves.

5.1.4.3 Display Tab

This tab contains checkboxes for displaying and hiding various attributes of the Path Curve Diagram.



View

View allows the user to select which view of the path curve to show in the Path Curve Diagram.

Double click

This setting will determine how new fixed points are added to the path when the user double-clicks in the Path Curve Diagram.

Image File

The CAM Editor is able to display a ".jpg" file as a background image in the diagram. This can make it much easier for the user to maintain the orientation of the fixed points and path curve to the actual underlying design for which the path is to be used. See Background Images for more information.

Show Grid

This option will allow the user to show or hide the axes and grid lines in the display.

Show Image

If a background image has been selected (see above), then this option will allow the user to show or hide the image.

Show Thinned Points

If fixed points have been thinned (see Thinning Fixed Points), then this option will allow the user to show or hide the thinned points. If shown, then the fixed points will be displayed as small yellow squares.

Continuous Recalc

If this flag is checked, then the path curve will be continuously recalculated and redisplayed as fixed points are dragged in the curve profile diagram. This redrawing of the curve may require a significant amount of processing power. If this redrawing behaves erratically or creates a disruption on your computer, then you can deactivate it; the curve will then be recalculated and displayed as soon as you release the mouse button.

Auto Scale

If this flag is checked, then the curve diagram will be automatically re-scaled so that the entire curve is always visible.

Note that if the diagram has been manually zoomed or scrolled, then this flag will be automatically cleared so that the user's area of interest will remain visible.

Track Mouse

If mouse tracking is enabled, then the position of the mouse will be shown in the top-left corner of the Path Curve Diagram. Note that only the coordinates corresponding to the current **View** are shown.

Mouse accuracy

If the right mouse button is pressed while the mouse is pointing at a fixed point, then the fixed point popup menu is displayed. Otherwise, the normal popup menu is displayed. This value specifies how near to a fixed point the mouse must be before the CAM Editor will recognize that the mouse is pointing at that fixed point. Note that a "hand" mouse icon (rather than the normal arrow mouse icon) will be displayed when the CAM Editor has determined that the mouse is pointing at a fixed point. That fixed point will also be highlighted in the Fix Points Tab so that the user can positively identify the point.

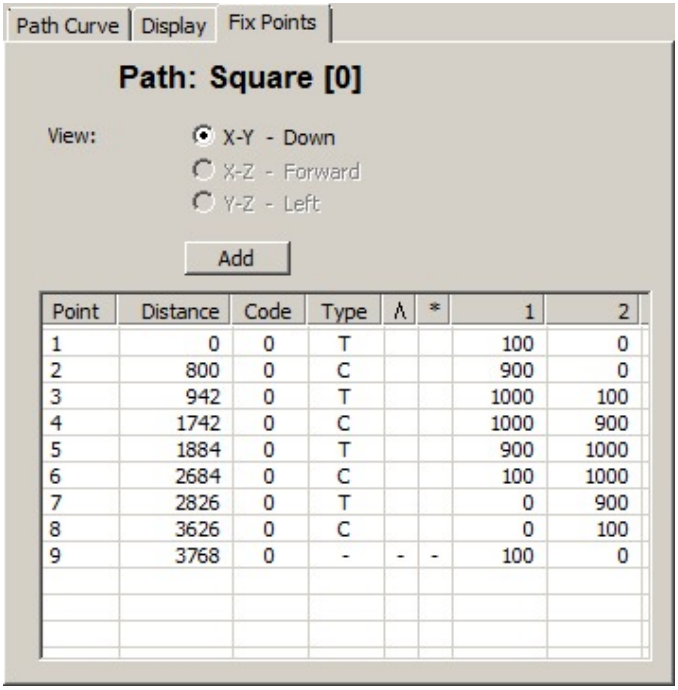
Overlay File

Overlay files are intended to be used during the debugging phase of the development of path curves and their associated application programs. If the Oscilloscope is used to collect position data while the path is being followed, then these data can be displayed as an overlay on the path in the diagram. This allows the user to visually see how closely the actual position followed the intended path. See Overlay Files for more information.

5.1.4.4 Fix Points Tab

The Fixed Point Tab can be used to view and to manually add, delete, and modify fixed points. Please see "Using the Mouse" in Path Curve Diagram for a description of how to add, delete, and modify fixed points using the mouse.

See Path Fixed Points for more information on fixed points.



Point	Distance	Code	Type	Λ	*	1	2
1	0	0	T			100	0
2	800	0	C			900	0
3	942	0	T			1000	100
4	1742	0	C			1000	900
5	1884	0	T			900	1000
6	2684	0	C			100	1000
7	2826	0	T			0	900
8	3626	0	C			0	100
9	3768	0	-	-	-	100	0

Identifying Fixed Points

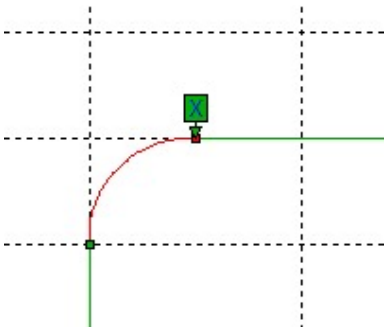
Fixed points are identified in the list using the following columns:

Point This column is simply the sequence number of the of the point. Sequence numbers are updated as points are added or deleted.

Distance This column gives the approximate arc length to the fixed point from the beginning of the path. These values are also are updated as points are added or deleted.

> Fixed points that have been thinned will be excluded from the final path and their **Distance** values will be blank. See Thinning Fixed Points for more information on thinned fixed points.

Any fixed point may also be identified in the curve by clicking on it with the left mouse button. This will cause a green flag to be displayed identifying the point in the Path Curve Diagram as is shown below:



Fixed Points Codes

The fixed point code is a 3-byte unsigned number in the range 0-0x00FFFFFF (16777215). The ApossIDE assigns no meaning to this number (other than to display a segment count in the Path Curve Tab). Application programs are free to use it in any way that is appropriate. For example, this is often used as a "pen up"/"pen down" code.

> Some older controllers do not support fixed point codes. For these controllers, the Path Curve Tab contains an option to suppress the downloading of fixed point codes.

Fixed Points Types

The fixed point type determines the shape of the single curve segment that *follows* the fixed point. Segments can either be straight (i.e. tangent) or curved. The CAM Editor will always try to make a smooth transition from one segment to the next. For this reason, at least one curve segment is required between adjacent tangent segments.

In the Path Curve Diagram, tangent points are displayed as small green squares and all other types of points are displayed as small red squares. Thinned fixed points will displayed as yellow squares, regardless of their type.

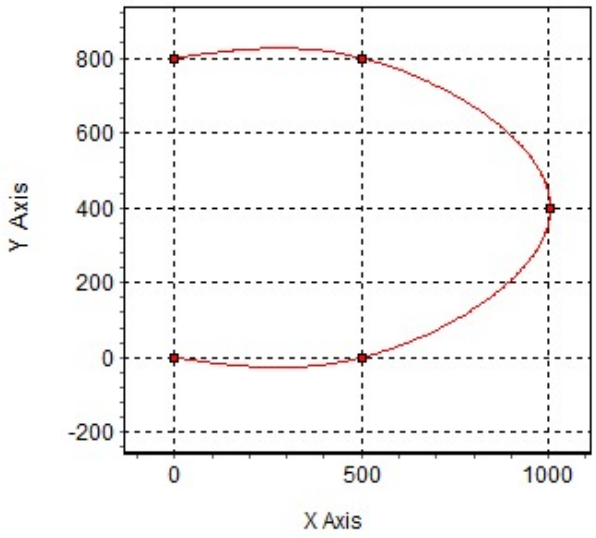
The following fixed point types are supported:

C	Curve segment.
T	Tangent (straight line) segment.
CZ	Trapezoidal curve segment.
C3	Third order (cubic) curve segment.

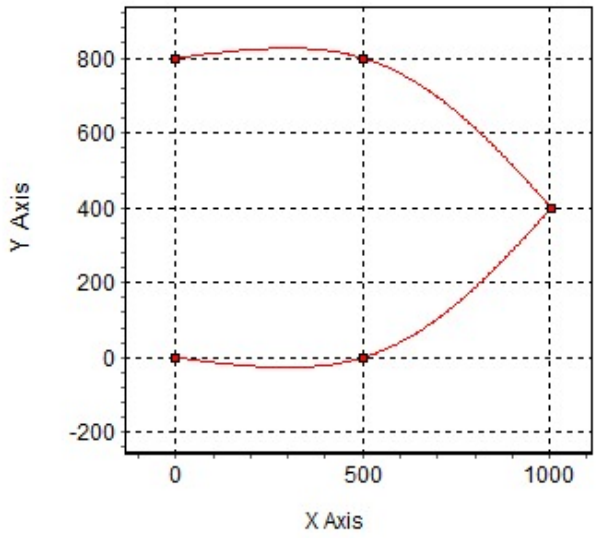
These fixed point types affect the shape of the path curve in exactly the same way as they affect the shape of CAM curves. See the CAM curve Fix Points Tab for a more detailed description of the effects of the different fixed point types. Also, see Path Fixed Points for a description of the restrictions on the sequence of fixed point types.

Corner Points

Normally, path curves will be "smooth"; the direction will match before and after each fixed point. However, some specialized applications may require a sharp change in direction at one or more specific fixed points. This is shown below.



Curve Point



Corner Point

This is done by setting the appropriate fixed points to be "corner points". Corner points will be identified by the "Λ" symbol in the "Corner" column of the **Fix Points Tab**. Any fixed point *except* the two ends of Trapezoid and 3'rd order curve segments, may be designated as a corner point. Trapezoid and 3'rd order curves do not support corners since these curves are specifically designed to match directions, whereas corners specifically indicate that directions do *not* match.

> Corner points should *ONLY* be used by applications that are specifically designed to handle them. Generally, this means stopping the motors at the corner point in order to avoid an "infinite" acceleration. Running through a corner point at a non-0 velocity, will cause the motors to experience a severe jolt as they attempt to instantaneously change velocity. Generally, corner points should also be designated as stop points (see below).

> Normally, the Path Editor will prevent path curves from containing adjacent tangents since the resulting curve cannot be guaranteed to be smooth at the point where the tangents join. However, if that point is made a corner point, then the Path Editor will allow adjacent tangents since the user has specified that a sharp change is expected. To do this, ensure that the type of the appropriate fixed point is set to "Curve", then toggle the point to be a "corner", and finally change the fixed point type to "Tangent".

Stop Points

Paths are often followed in descrete movements; the motors are initially stopped and then they are moved a certain way along the path at which point they are stopped again. For example, a pen plotter may put the pen down, draw a certain distance along the path, stop and put the pen up, and then move to the start of the next line. These points will be designated as stop points and will be identified by the "*" symbol in the "Stop" column of the **Fix Points Tab**.

Corner points should almost always be stop points. This will prevent the motors from attempting to run through corner points at a non-0 velocity.

Stop points do not affect the shape of a path. They only affect the behaviour of the movement as the controller follows the path (see PathStart and PathNext).

Adding Fixed points

To add one or more new fixed points, click one of the existing points in the list. The new points will be inserted into the path *before* the selected point. If no point is selected, then the new points will be appened

to the path. Then click the **Add** button. This will display the dialog for adding points. Points must be specified as sets of coordinate values, one value for each dimension in the path. Multiple sets can be defined at the same time. Fixed points must be entered in the order that they will appear in the path curve and will always be added with code 0 and type "Curve (C)". The user can modify these values after the points have been added.

Deleting Fixed Points

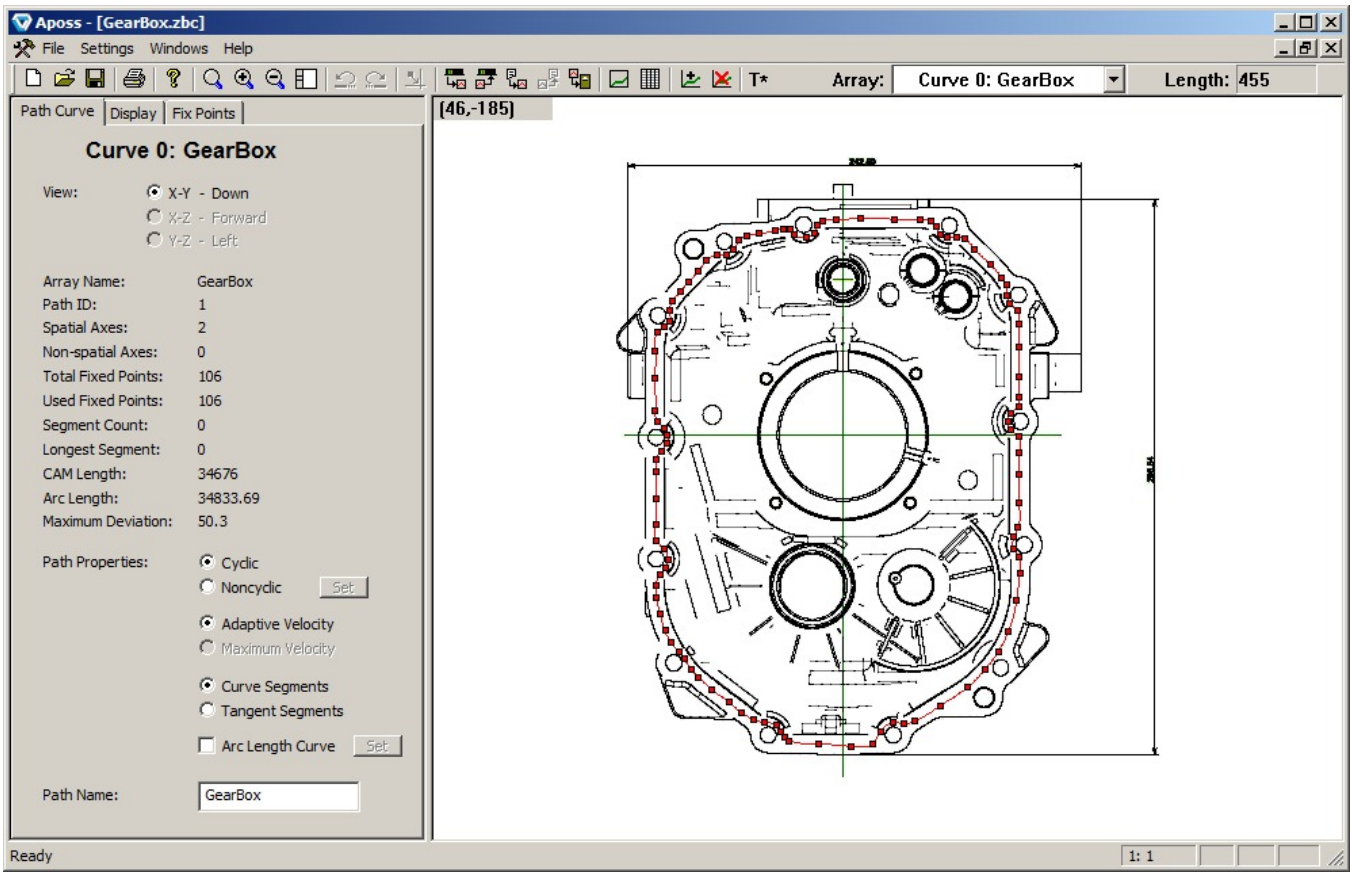
To delete a fixed point, click on the fixed point and then press the **Del** or **Delete** key.

Modifying Fixed Points

To modify a fixed point value, double-click on the appropriate value in the list of fixed points. If the “Type” column is clicked, then the fixed point type can be changed using a popup menu. If the "Λ" (Corner) column is clicked, then the fixed point will be toggled between a "corner" point and a "non-corner" point. Similarly, if the "*" (Stop) column is clicked, then the stop attribute will be toggled. If one of the other columns is clicked, then the corresponding value can be changed. Press **Enter** or click with the mouse to commit the new value. If necessary, pressing the **Esc** key will restore the original value if it has not yet been committed.

5.1.4.5 Background Images

The CAM Editor is able to display a ".jpg" file as a background image in the diagram. This can make it much easier for the user to maintain the orientation of the fixed points and path curve to the actual underlying design for which the path is to be used.



Currently, the CAM Editor only supports images that have a coordinate system where the horizontal

> coordinates increase from left to right and the vertical coordinates increase from bottom to top. As well, the image coordinate system cannot be rotated.

To add a background image:

1. Select the Display Tab.
2. Press the Image File **Browse** button and select the ".jpg" file that will be used as a background. This will then be displayed in the Path Curve Diagram (as shown above).
3. Pan and zoom the diagram very close to a position in the image that has a known coordinate. This could be (0,0) or some other reference position in the image.
4. Right-click on the known position and select the **Set Image Position** command from the popup menu. Enter the actual design coordinates of the position. This will reposition the image so that the coordinates of the known position are correctly displayed in the top left corner of the diagram.
5. Press the **Reset Zoom** toolbar button to redisplay the diagram.
6. Pan and zoom very close to a *different* position in the image that also has a known coordinate. This position should be as far from the first position as possible in order to maximize the accuracy of the background image scaling.
7. Right-click on the second know position and select the **Set Image Scale** command from the popup menu. Enter the actual design coordinates of the position. This will scale the image so that all coordinates displayed in the top left corner of the diagram should now be correct.
8. Press the **Reset Zoom** toolbar button to redisplay the diagram.

For a 3-dimensional path, a different background image may be selected for each view.

5.1.4.6 Overlay Files

As an aid to debugging and tuning applications that use path curves, the CAM Editor allows real motor position data, gathered during actual operation of the system, to be displayed over top of the path curve. This will help show how closely the system is following the originally designed path. This is done using "Overlay Files".

There are two steps to the procedure:

1. Collect the data and create the overlay file.
2. Display the overlay file in the CAM Editor.

Creating the Overlay File

An overlay file is simply an Oscilloscope (.zbo) file that contains curves for the actual and command positions of each axis that is of interest. Typically, this will be at least the axes representing the X axis and Y axis of the path. There are two alternate methods of creating the file:

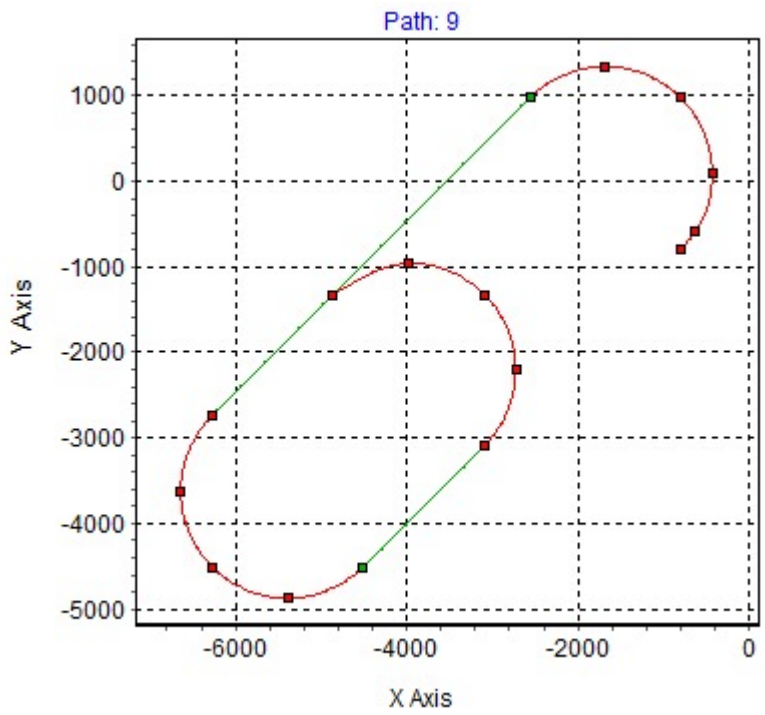
1. Configure and start the recording from within the application program. This is the preferred method since the data collection can be limited to exactly the time of interest.
    a. Add RecordIndex commands to the application program to configure the data the be recorded in the "Record" array. Both the command position (REG_COMPOS) and actual position (REG_ACTPOS) should be recorded for each axis of interest.
    b. Add the RecordStart command to the application program at the position where recording is to be started. Add the RecordStop command where recording is to be stopped.
    c. Run the application program.
    d. Using the Oscilloscope, open a Record Oscilloscope. This will load the recorded data from the controller.
    e. Save the Oscilloscope data. This will create the .zbo file that can then be used as the overlay file.
2. Configure and start the recording using the Oscilloscope. Then start the application running. This method is easier to use, but it is more dificult to limit the data collection to only the time that is of interest.

a. Using the Oscilloscope, open either a Free Run Oscilloscope or a Single Shot Oscilloscope.
b. Add curves for both the command position (REG_COMPOS) and actual position (REG_ACTPOS) of each axis of interest. See Adding Oscilloscope Curves.
c. Either start the Oscilloscope run and then start the application program or start the application program and then start the Oscilloscope run. The order will depend on how the application program is designed to operate and on exactly when then recording is to be done. See Starting and Ending the Test.
d. End the Oscilloscope run once enough data has been collected.
e. Save the Oscilloscope data. This will create the .zbo file that can then be used as the overlay file.

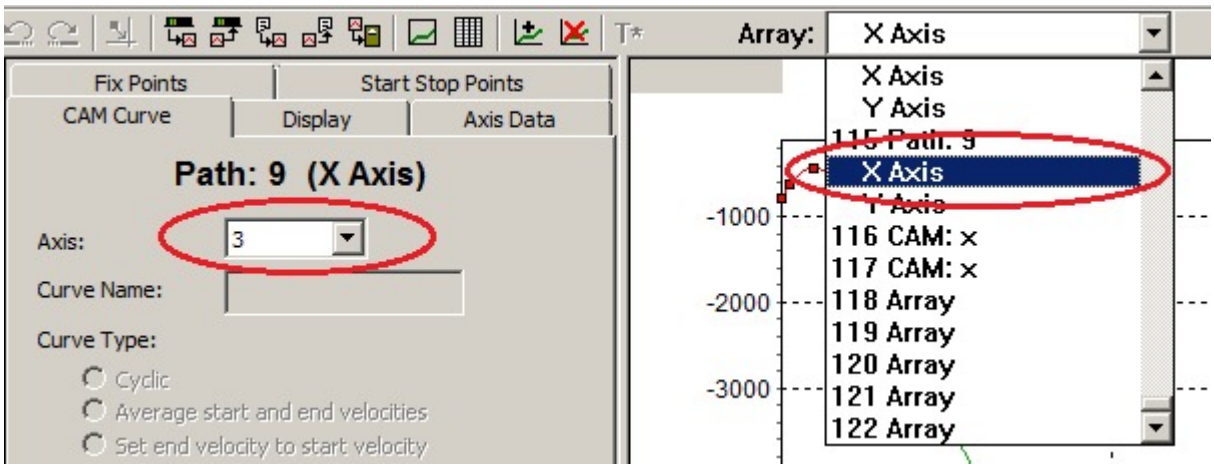
Displaying the Overlay File

Once the overlay file has been created, display it as follows:

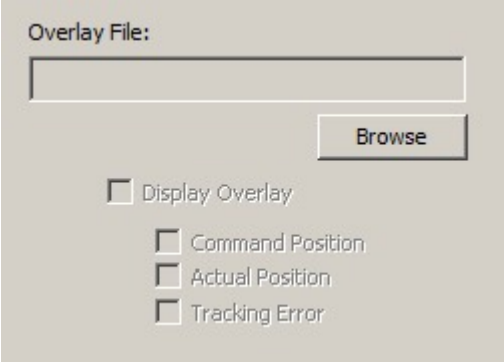
1. Open the CAM Editor and display the original path curve. The following illustration shows the example that will be used here.



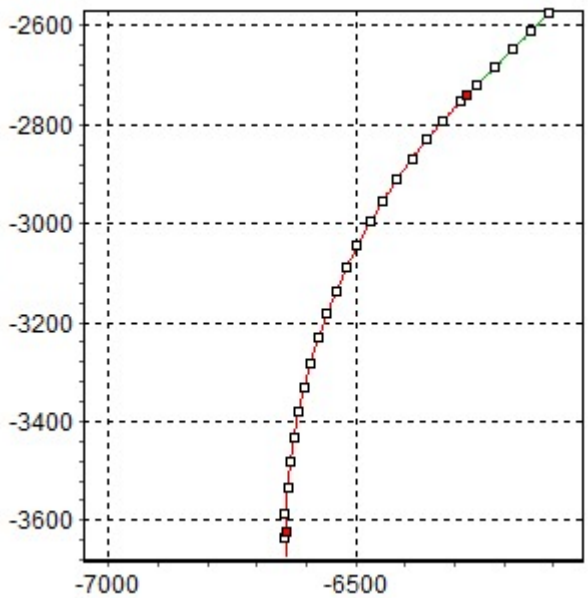
2. Using the **Array** list in the CAM Editor toolbar, select the X axis. This will display the CAM curve for the X axis of the path. In the **CAM Curve** tab, ensure that the correct controller axis is selected for the X axis. Repeat this for the Y axis and each additional axis.



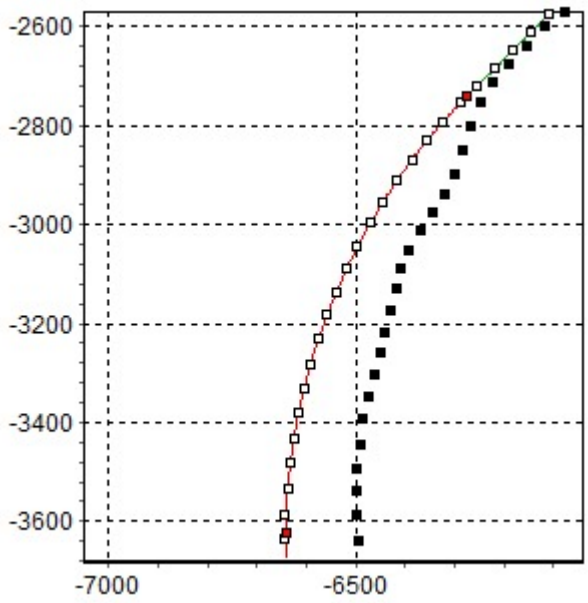
3. Using the **Array** list in the CAM Editor toolbar, switch back to the original path curve.
4. In the **Display** tab, use the **Browse** button to select the .zbo overlay file. Then enable the **Display Overlay** option. Use the options below this to select what is to be displayed.



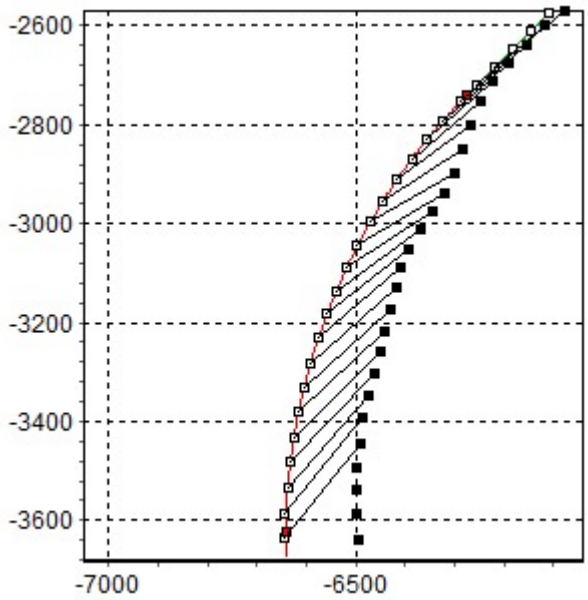
Displaying the "Command Position" will display each command position recorded in the overlay file as a small white square (see below). Note that if very dense data was collected, then it may be necessary to zoom in to a small portion of the graphics display to see how the command positions match the original curve.



Displaying the "Actual Position" will display each actual position recorded in the overlay file as a small black square (see below).



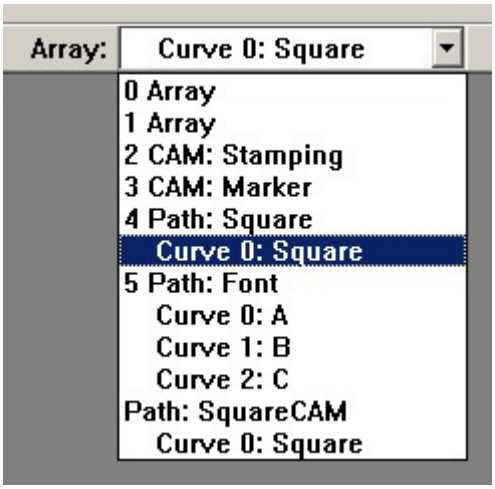
Displaying the "Tracking Error" will display a line connecting corresponding command and actual positions (see below). Both corresponding points must be visible within the graphics for the line to be drawn.



5.1.4.7 Multiple Paths

Unlike CAM arrays that support only a single CAM curve per array, Path arrays support multiple sub-path curves in each array. This is particularly useful for application that require multiple related path curves. For example, a character-drawing application may require a path curve for every possible character. Rather than consuming and managing perhaps 100 or more individual DIM arrays, the path curves for all individual characters can be put into a single DIM array. Note that this also simplifies the handling of multiple character fonts; each complete font can be placed in its own DIM array.

Sub-paths will appear in the **Array** field in the toolbar as shown below. In this example, DIM array 4 contains a Path array named "Square" that contains a single sub-path. DIM array 5 also contains a Path array (named "Font"), but this array conatins three sub-paths (named "A", "B", and "C").



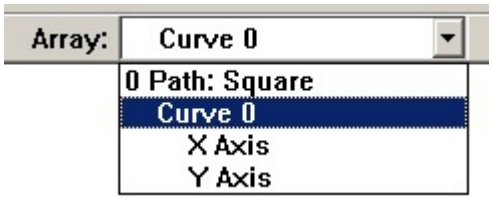
Additional sub-paths may be added to an existing Path array simply by using the **Add array** toolbar button and selecting the appropriate options. Individual sub-paths may be deleted by selecting the sub-path and using the **Delete array** toolbar button. The entire Path array may be deleted by selecting the main Path array item (e.g. "4 Path: Square") and using the **Delete array** toolbar button.

5.1.4.8 Viewing Individual Axes

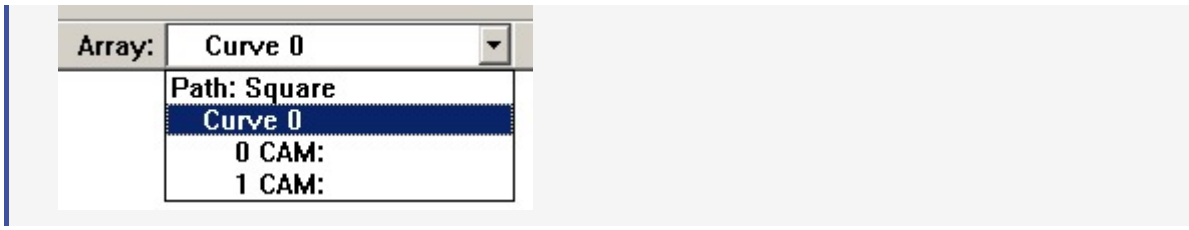
When a path curve is processed by a controller, each dimension (i.e. both spatial and non-spatial) of the curve is isolated and converted into a standard CAM curve. These CAM curves will then be driven by the same master signal which has the effect of keeping all axes synchronized.

At any time during the editing of a path curve, it is possible to view the CAM curves for the individual dimensions. This can be valuable because it allows the user to view the velocity and acceleration profiles of each axis and to check if any limits have been exceeded. Note that this is really only useful for Cartesian kinematics (see Working with Kinematics). For non-Cartesian kinematics, the relationship between the CAM curves and the final motion of the axes, is much more complicated and analyzing the CAM curve is not a good indication of whether or not the motor limits will be exceeded.

To view a CAM curve, click the dropdown button of the **Array** control in the window titlebar. The individual axes will be listed directly under the path curve to which they belong. Simply select the desired axis and it will be displayed as a CAM curve. To redisplay the original path curve, select the parent path item in the list again.



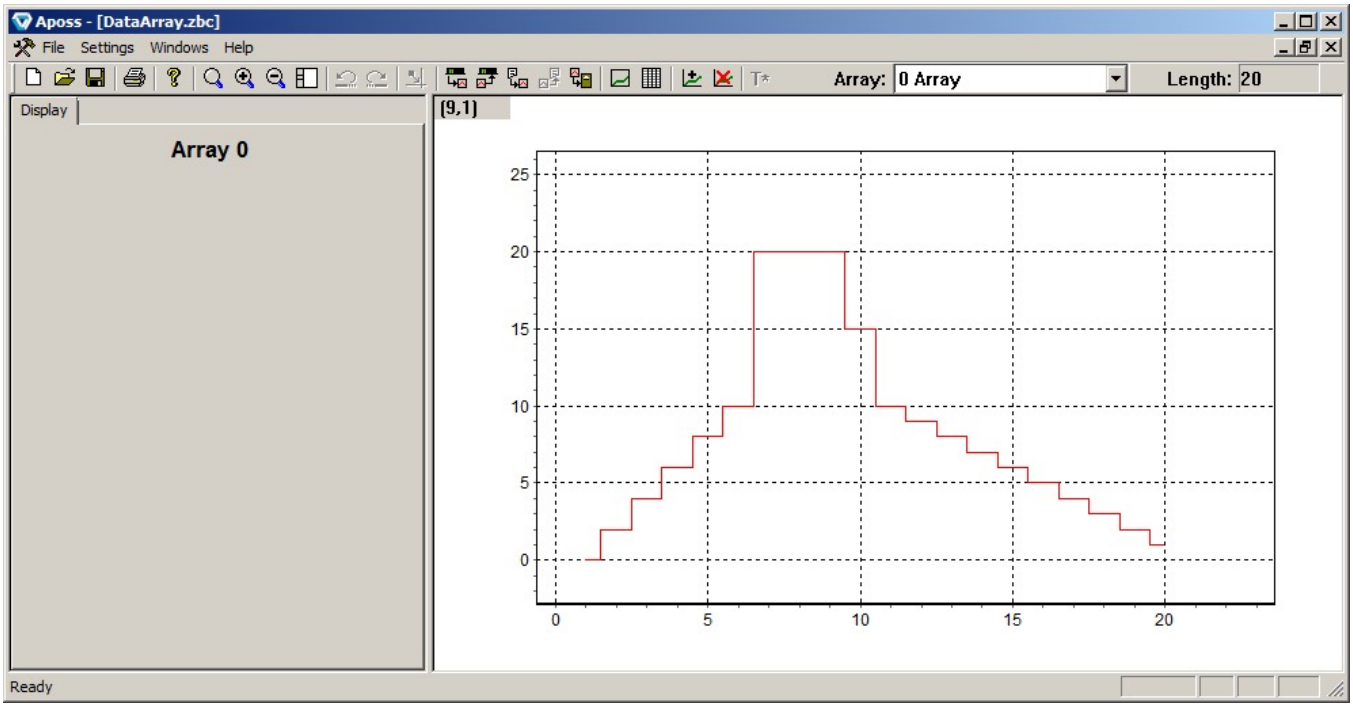
When a path curve is being downloaded as a **Path Array** (see Path Curve Tab), then the DIM array number of the array will be shown in the list as a prefix to the name of the path curve (as in the example above). If the path is being downloaded as **CAM Arrays**, then the DIM array numbers will be shown as prefixes to the individual CAM arrays (as in the example below).



When a CAM curve is being displayed, it is not possible to modify any value that directly relates to the path curve itself. For example, fixed points cannot be modified. However, any of the other CAM-related values *can* be modified. For example, axis parameters may be viewed and modified.

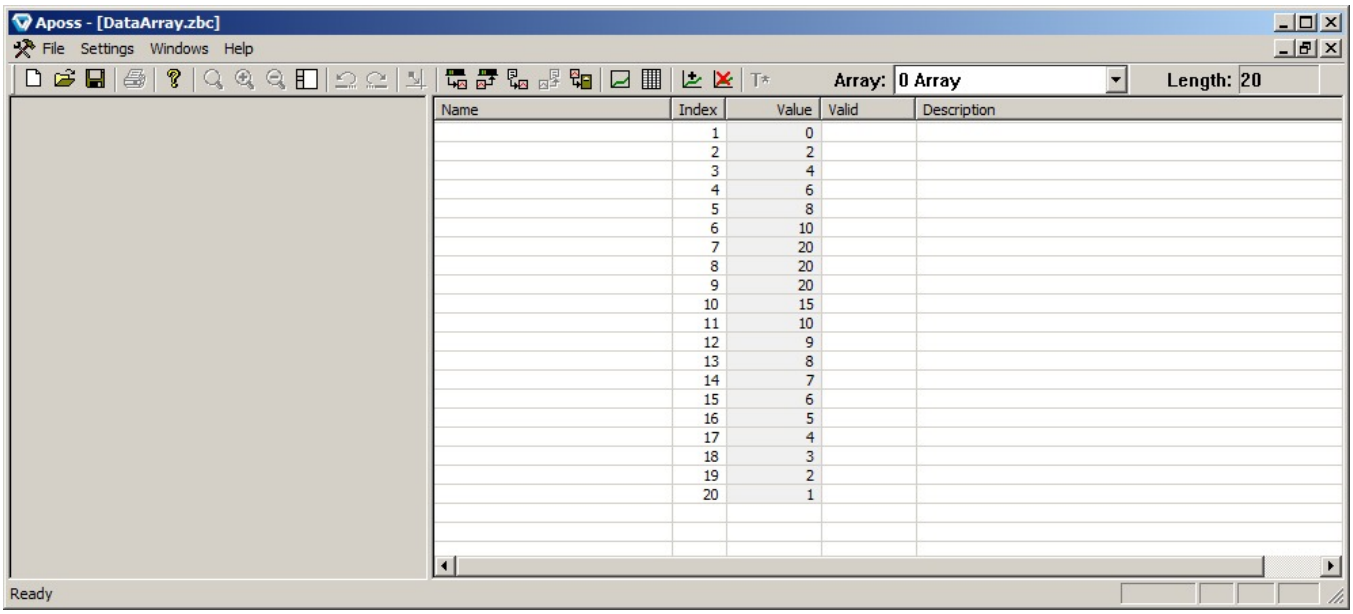
5.1.5 Editing Data Arrays

When a simple data array is selected in the **Array** box in the toolbar, the CAM Editor window will appear similar to that shown below:



This simply displays the data in the array as a bar chart. Note that the array cannot be modified when graphics mode is being used.

If the display is switched to text mode using the **Text mode** toolbar button, then the CAM Editor window will appear similar to that shown below. In this mode, array values can be modified simply by double-clicking on the value column.



5.2 Array Editor

The Array Editor provides a list-oriented interface that allows all controller arrays and parameters to be viewed and edited. This includes user parameters, global parameters, axis parameters, axis state variables, and system state variables.

In addition, an optional "**Array Definition file**" may be created as a template specifying such things as element names, minimum and maximum values, default values, etc. Using an array definition file allows much more user-friendly editing of arrays. The Array Editor behaves slightly differently depending on whether or not an array definition file is being used. This is described later. Array definition files will always use ".zba" as the file extension. They will be identified in Windows folders with the icon shown below. As with other types of files, array definition files may be opened, modified, and saved by the Array Editor at any later time.

The Array Editor supports two modes: an **Expert mode** and a **User mode**. Expert mode allows full access to all arrays and parameters and to all editing features. Expert mode is intended to be used to create the array definition files that will subsequently be used in User mode. In User mode, access to arrays and parameters is restricted to only those elements pre-defined in an array definition file. An array definition file may be used but cannot be edited in User mode. The main intent of User mode is to provide a "configuration tool" for field personnel. In this case, the array definition file can be used to limit access to only those pre-defined items required by the configuration.

The "Array Editor parameter support" affects the Array Editor. It can be accessed using the **Settings** → **Options** command. If ☑ **Array Editor Parameter Support** is enabled, then all arrays and parameters (i.e. user, global, axis, etc.) are loaded into the Array Editor. If **Array Editor Parameter Support** is disabled, then only arrays and user parameters are loaded. Other parameters are not accessible even if they are specified in an array definition file.

5.2.1 Starting the Array Editor

The Array Editor can be started either from within the ApossIDE or as a stand-alone application.

When started from within the ApossIDE, the Array Editor always starts in Expert mode and allows full access to all arrays and parameters. The Array Editor window will behave in a similar way to the other ApossIDE windows (i.e. edit window,

CAM Editor, oscilloscope, etc.).

When started as a stand-alone application, the Array Editor can be started in either Expert mode or User mode. The stand-alone Array Editor window will be a Windows dialog.

Only one instance of the ApossIDE may be running at a time so if the ApossIDE is already running, then a message is displayed and the stand-alone Array Editor instance will terminate.

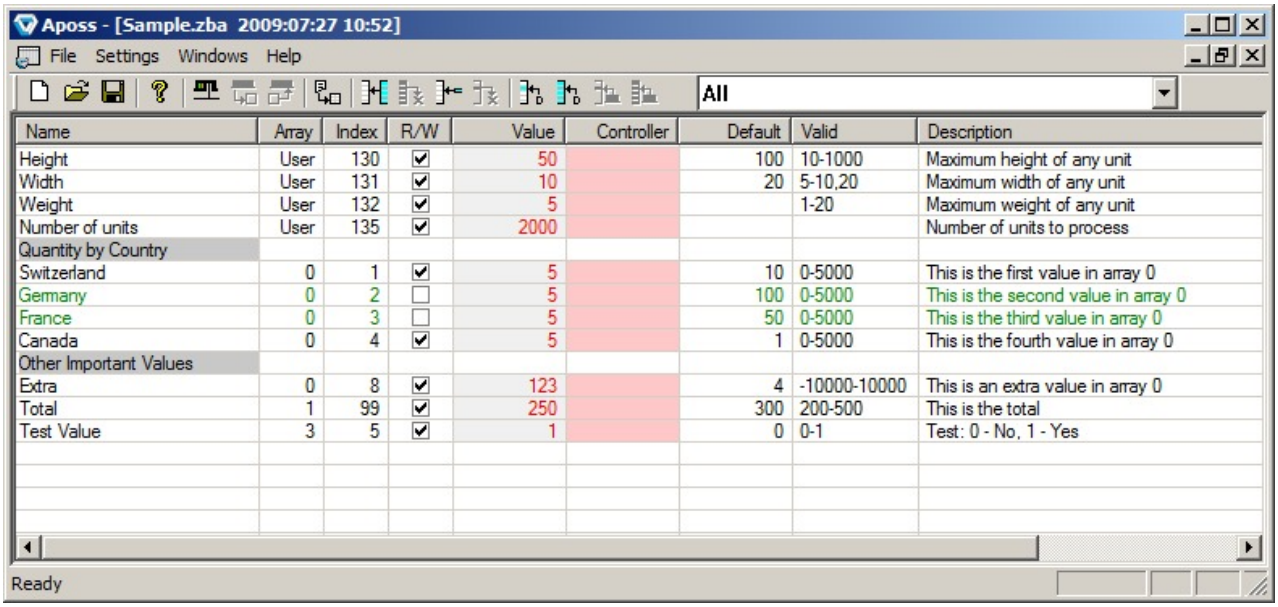
Within the ApossIDE

The Array Editor can be started from within the ApossIDE using any of the following methods:

1. Using the **Tools** → **Array Editor** menu item or the Array Editor toolbar button.
2. Using the **File** → **New** menu item or the File New toolbar button and then selecting "Create Array .zba file".
3. Using the **File** → **Open** menu item and then selecting an existing ".zba" file.
4. Using any of the other Windows file open mechanisms such as double-clicking on a ".zba" file or dragging a ".zba" file onto an open ApossIDE window.

In all of these cases, if a controller is already connected in the currently active window, then the arrays and parameters are automatically loaded from that controller.

The window title will include the controller ID and name (if a controller is connected) and the date when the values were read from the controller.



Name	Array	Index	R/W	Value	Controller	Default	Valid	Description
Height	User	130	☑	50		100	10-1000	Maximum height of any unit
Width	User	131	☑	10		20	5-10,20	Maximum width of any unit
Weight	User	132	☑	5			1-20	Maximum weight of any unit
Number of units	User	135	☑	2000				Number of units to process
Quantity by Country								
Switzerland	0	1	☑	5		10	0-5000	This is the first value in array 0
Germany	0	2	☐	5		100	0-5000	This is the second value in array 0
France	0	3	☐	5		50	0-5000	This is the third value in array 0
Canada	0	4	☑	5		1	0-5000	This is the fourth value in array 0
Other Important Values								
Extra	0	8	☑	123		4	-10000-10000	This is an extra value in array 0
Total	1	99	☑	250		300	200-500	This is the total
Test Value	3	5	☑	1		0	0-1	Test: 0 - No, 1 - Yes

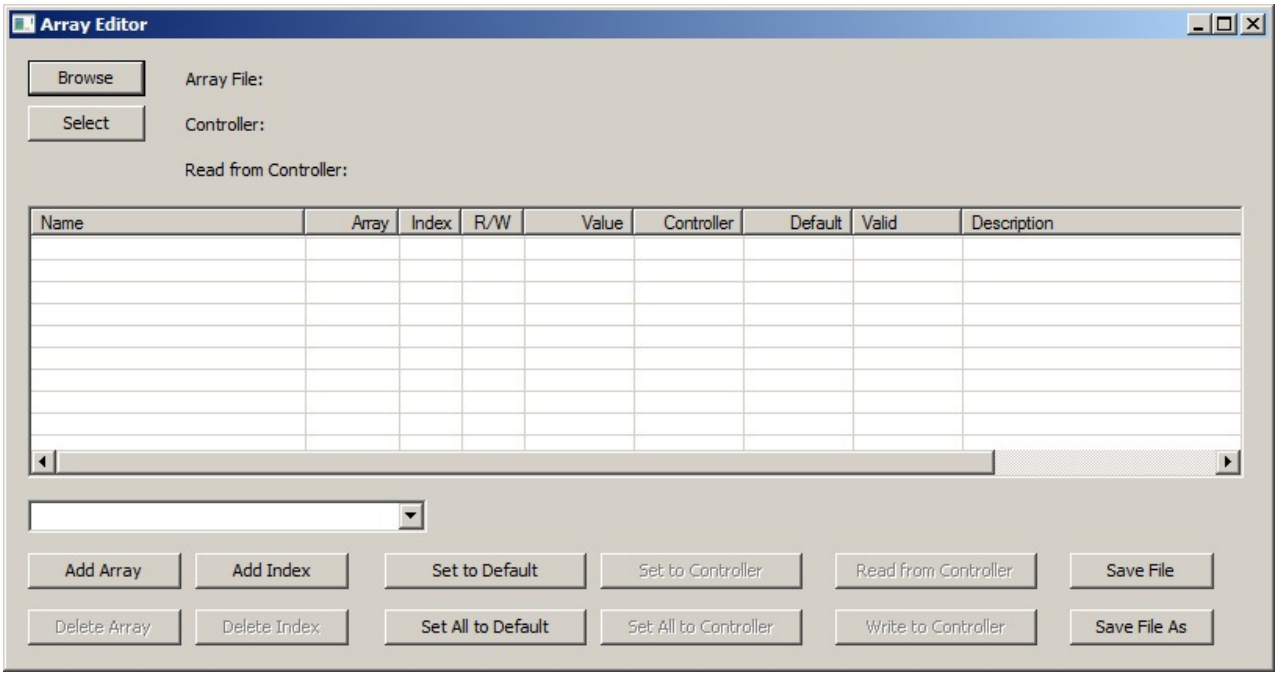
Stand-alone application

The Array Editor can be started stand-alone by adding one of the following arguments to the Windows command line:

/a Start the Array Editor in User mode.

/ae Start the Array Editor in Expert mode.

In stand-alone application, the Array Editor will always start without a controller being connected and without an array definition file being loaded. The user must explicitly connect to a controller and explicitly load an array definition file.



5.2.2 Array Editor List

Controller arrays and parameters are displayed in a list similar to that shown below.

Name	Array	Index	R/W	Value	Controller	Default	Valid	Description
Height	User	130	☑	50	50	100	10-1000	Maximum height of any unit
Width	User	131	☑	**1000**	10	20	5-10,20	Maximum width of any unit
Weight	User	132	☑	5	5		1-20	Maximum weight of any unit
Number of units	User	135	☑	2000	3000			Number of units to process
Quantity by Country								
Switzerland	0	1	☑	5	5	10	0-5000	This is the first value in array 0
Germany	0	2	☐	5	0	100	0-5000	This is the second value in array 0
France	0	3	☐	5	0	50	0-5000	This is the third value in array 0
Canada	0	4	☑	5	5	1	0-5000	This is the fourth value in array 0
Other Important Values								
Extra	0	8	☑	123	123	4	-10000-10000	This is an extra value in array 0
Total	1	99	☑	250		300	200-500	This is the total
Test Value	3	5	☑	1		0	0-1	Test: 0 - No, 1 - Yes

One or more rows in the list can be selected using any of the normal Windows list selection methods: Clicking, **Shift**-clicking, or **Ctrl**-clicking.

Values can be edited (as far as allowed) by double-clicking on them. All changed values will be displayed using a bold character font. For example, in the above illustration, the "Width" value has been changed to 1000.

Name

If the array index or parameter has been given a name, then it is displayed in this column. It will be blank if there is no name. Items in the list can be organized into “groups” and given a title (see **Add Index**). These titles will be displayed with a dark background. In Expert mode, this field can be edited by double-clicking on it.

Array

This column identifies the array number or parameter type.

Index

This column identifies the array index or parameter number.

R/W

If a list item can be written to the controller, then this column will be checked. If the item is read-only, then the column will not be checked. The entire row in the list will be displayed in green if the item is read-only. This column can only be modified in Expert mode.

Value

This column contains the editable “user” values for items. Values will be displayed with a light background so that the column is clearly identifiable. If a value differs from the value currently stored on the controller, then the value will be displayed in red. The value may be blank if there is no known value. If the current value is invalid, then it will be displayed with a light red background (e.g. the "Width" value in the above illustration). Pausing the mouse on the invalid value will cause a yellow tooltip to be displayed giving the reason why the value is invalid.

Users can change these values simply by clicking on the value and entering a new value or by typing **Ctrl-V** to paste a value from the Windows clipboard.

Press the **Enter** key to accept the new value. Press the **Esc** key to discard the new value and keep the previous value. Clicking with the mouse anywhere *other* than within the small edit box will automatically be treated as if the **Enter** key had been pressed.

> Changed values are not written to the controller until the user explicitly requests that they be written.

Controller

If the Array Editor is currently connected to a controller, then this column will display the current controller values. If there is no controller, then this column will be blank. If a controller is connected but the value does not exist on the controller (for example, an array may not exist), then this column will be blank and the entire row will be displayed in green to indicate that the item is “read-only” and cannot be written to the controller. If the current value is invalid or does not exist on the controller, then it will be displayed with a light red background.

Default

The default value is the value that the controller will assign to global and axis parameters when the controller is reset to the factory settings. The Array Editor extends the concept to the other parameters (i.e. user parameters and arrays) for the template .zba file that is being created.

If a default value is defined for the list item, then it is displayed here. Otherwise, this column is blank. In Expert mode, this field can be edited by double-clicking on it.

Valid

If the list item has a set of valid values, then the valid values are displayed here. Individual values are comma-separated; valid ranges are specified using "-" (e.g. 1-3,5,10-20). If there are no validation rules for the list item, then this column is blank. In Expert mode, this fields can be edited by double-clicking on it.

Description

Each list item may be given an optional description which is displayed in this column. In Expert mode, this field can be edited by double-clicking on it.

5.2.3 Array Editor Functions

If the Array Editor has been started from with the ApossIDE, then all Array Editor functions will be available on the toolbar. The **Array Selection Box** is displayed rightmost.

If the Array Editor has been started stand-alone, then all functions will be available as normal dialog buttons. The **Array Selection Box** is displayed in the bottom left corner just above the buttons:

Connect/Select

Connect to a controller or switch an existing connection to another controller. This will cause the arrays and parameters to be automatically loaded from the new controller. The new values will be displayed in the "Controller" column.

In the stand-alone Array Editor, this function is provided by the **Select** button.

Read from Controller

This will re-read all arrays and parameters from the controller. The new values will be displayed in the "Controller" column.

Write to Controller

This will cause all the values from the "Value" column in all arrays (even values in arrays that are not currently being displayed), to be written to the controller. Items that have no value (i.e. a blank in the "Value" column) will retain their existing values. Once the values have been written to the controller, the "Controller" column will be updated to reflect the new values on the controller.

Overlay descriptions

This will allow list item names, descriptions, minimum values, maximum values, and default values to be copied from an existing array definition file into the Array Editor.

This function is not available in the stand-alone Array Editor.

Add Array

Add a new array. A dialog provides the definition of array number and size.

This function is not available in User mode. Arrays can only be added in Expert mode.

Delete Array

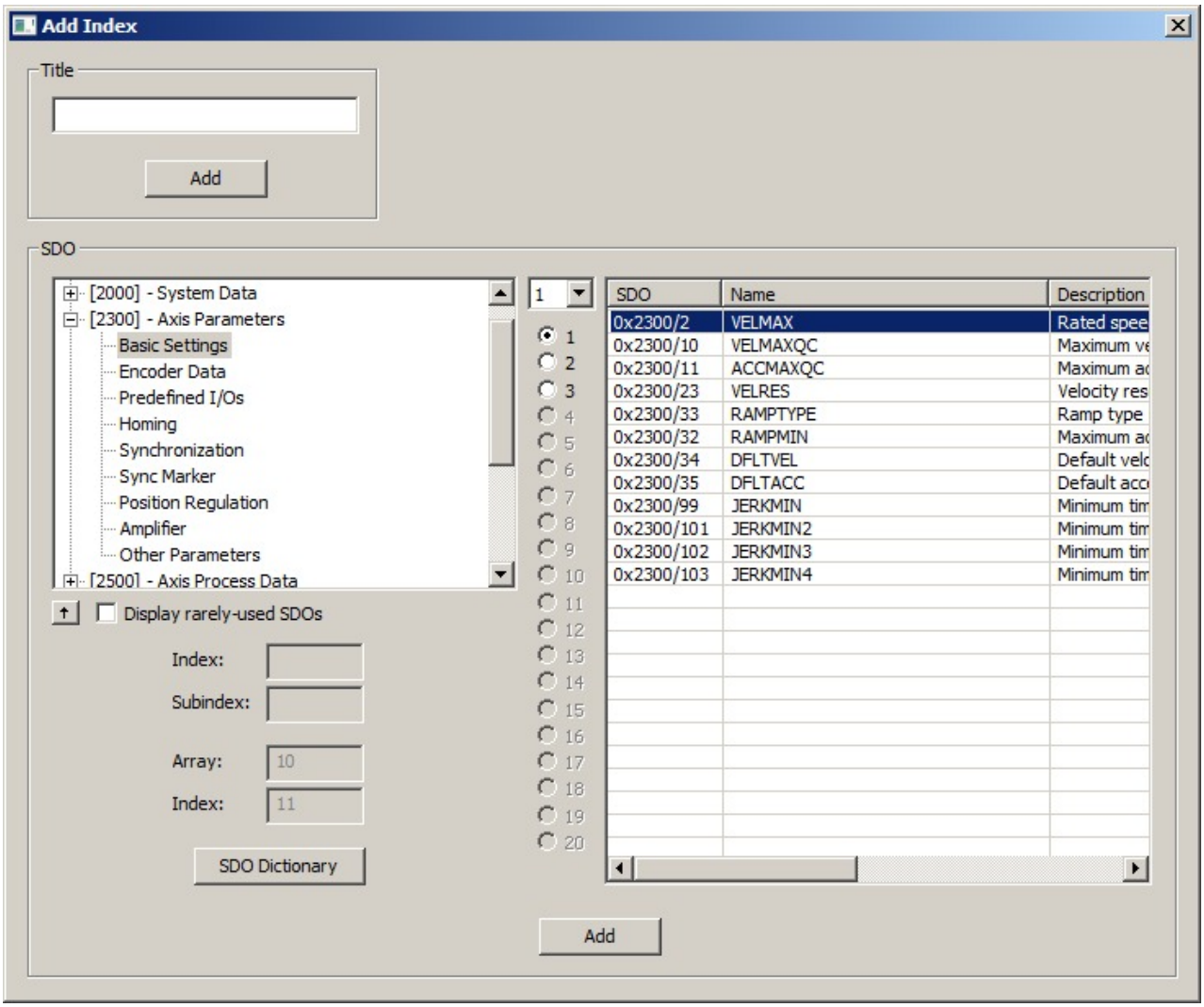
This will delete the currently selected array. The array must be selected in the "Array selection list".

This function is not available in User mode. Arrays can only be deleted in Expert mode.

Add Index

Insert a new item into the list. The item will be inserted before the currently selected list item. If the Array Editor

cannot automatically determine the type of the item to be inserted, then a dialog provides the available types. This dialog can be resized. See Read/Write SDO for a description of how to use this dialog to select SDO values.



This function is not available in User mode. New items can only be added in Expert mode.

Delete Index

This will delete the currently selected list item.

This function is not available in User mode. Items can only be deleted in Expert mode.

Set to Default

Highlight one or more array elements in the list and click this to set the "value" of all currently selected list items to the corresponding default values. Any previous values are overwritten.

Note that values are not changed for those items which have a blank default value.

Set All to Default

Set the "value" of *all* currently visible list items to the corresponding default values.Any previous values are overwritten. Values are also not changed for any array not selected in the "Array selection box".

Note that values are not changed for those items which have a blank default value.

Set to Controller

Set the "value" of all currently selected list items to the current controller values.

Note that values are not changed for those items which have a blank controller value.

Set All to Controller

Set the "value" of *all* list items to the current controller values. Note that values are not changed for those items which have a blank controller value. Values are also not changed for any array not selected in the "Array selection box".

Array selection box

The array selection box is displayed either in the Array Editor toolbar or in the bottom left corner of the stand-alone Array Editor dialog. This allows the user to select either all arrays or individual arrays. It may be easier for the user to select individual arrays in order to work with a smaller list.

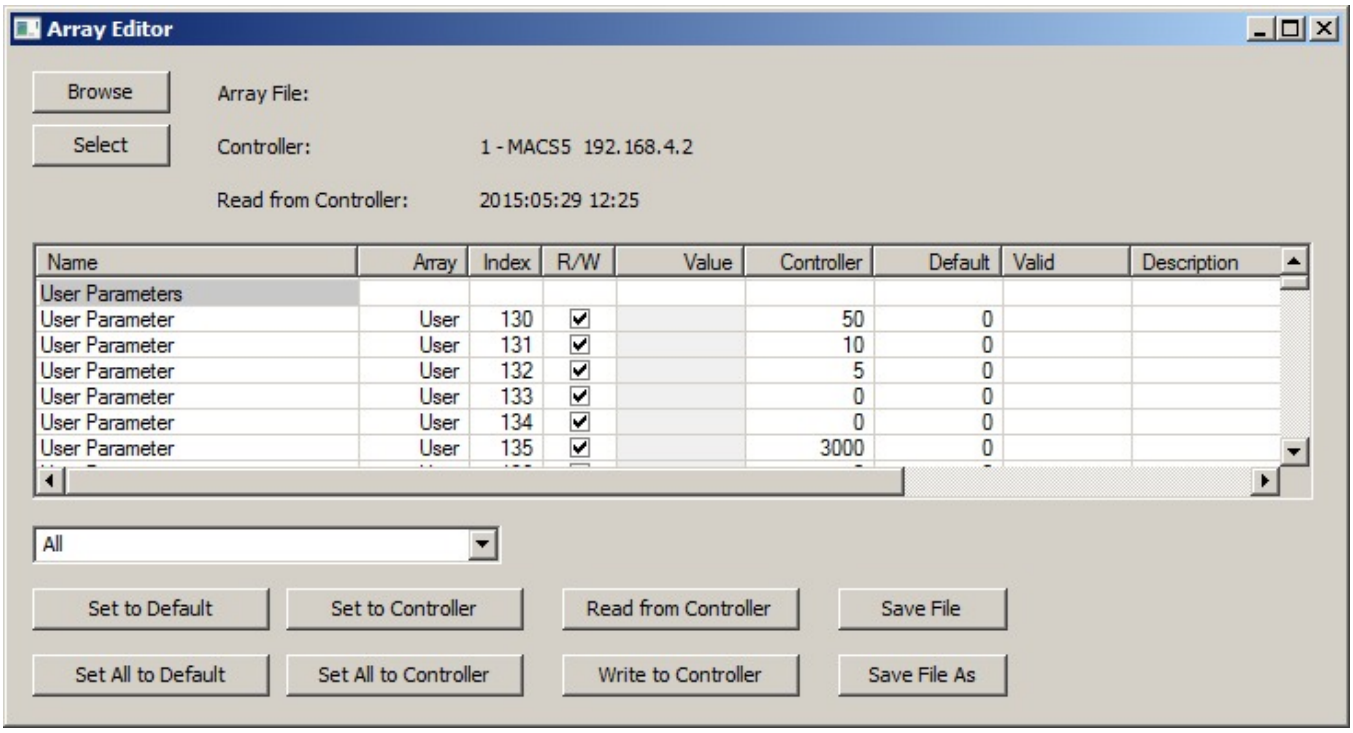
Save File / Save File As

Click on **File** → **Save** to save the current contents of the Array Editor as an array definition file; click on **File** → **Save as**, to save this file with a different name.

In the stand-alone Array Editor, the **Save File** and **Save File As** buttons allow the current contents of the Array Editor to be saved as an array definition file.

5.2.4 User mode WITHOUT an Array Definition File

The stand-alone Array Editor User mode dialog is shown below.



To use it WITHOUT an Array Definition File, follow these steps.

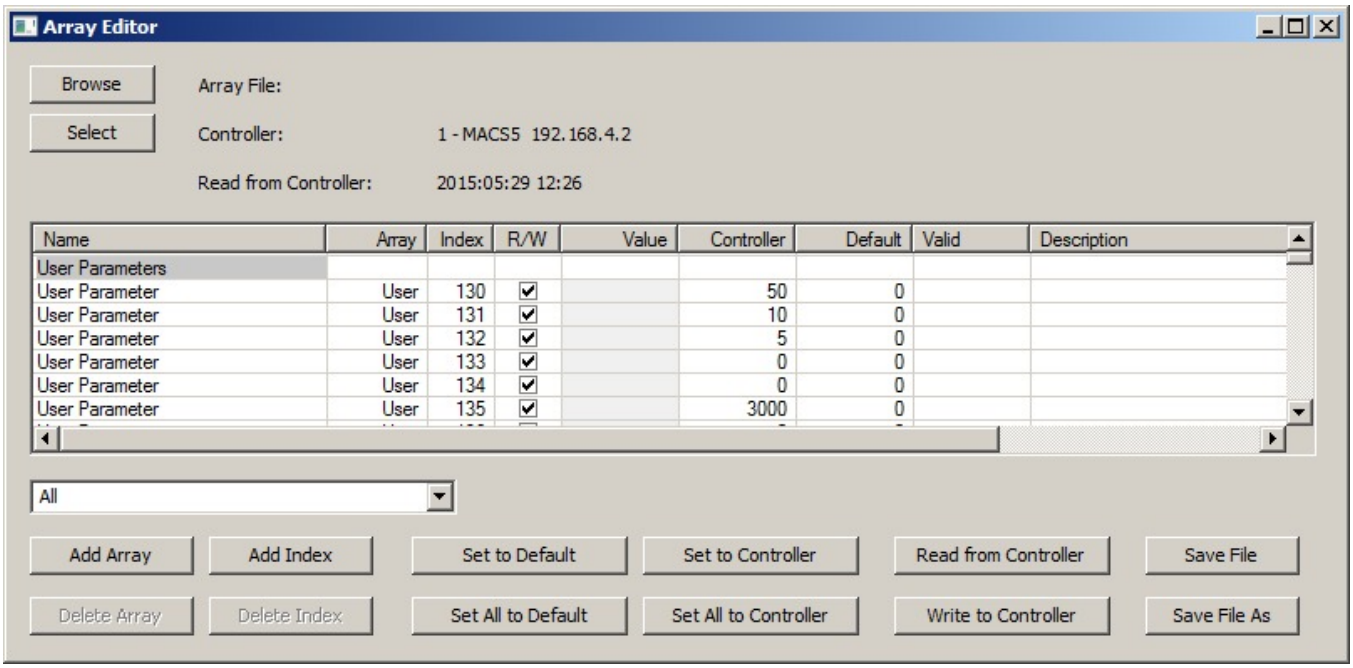
1. Press the **Select** button. This will display the normal **Select Controller** dialog that will allow the user to set interface parameters, open interfaces, and choose a controller. Once the connection to the controller has

been opened, the controller name will appear in the Array Editor dialog. The “Read from Controller” string will show the date and time when the arrays were read from the controller.
All existing arrays in the controller (including the User array, Global parameter array, permanent and temporary Axis parameter arrays, Axis Process data arrays, etc.) will be read automatically and displayed in the dialog. The array number (or name for User, Global, and Axis arrays) is displayed in the “Array” column, the array index is displayed in the “Index” column, and the value is displayed in the “Controller” column. The array selection box in the bottom-left of the dialog will list all the arrays that were found on the controller.

2. At any time once a controller is open, pressing the **Read from Controller** button will re-read all the arrays from the controller and display the current values in the dialog. The “Read from Controller” string will be updated to show the date and time when the arrays were read from the controller.
3. Click on any array in the array selection box to select that array for editing.
4. To change the value of an array element, click on the “Value” column of that array element. A small edit box will then be opened allowing the user to enter a new value. A new value may be entered either with the keyboard or by typing **Ctrl-V** to paste a value from the Windows clipboard.
   Note that some array elements may be read-only. These will be displayed green and cannot be changed by the user.
   Press the **Enter** key to accept the new value. Press the **Esc** key to discard the new value and keep the previous value. Clicking with the mouse anywhere *other* than within the small edit box will automatically be treated as if the **Enter** key had been pressed.
   Note that the “Value” column will contain only those values that have been changed. The values will be displayed in red if they are different from the values currently stored on the controller.
5. When editing is complete, press **Write to Controller**. This will cause all the values from the “Value” column in all arrays, to be written to the controller. Array elements that have no value (i.e. a blank in the “Value” column) will retain their existing values. Once the arrays have been written to the controller, the “Controller” column will be updated to reflect the new values on the controller.
6. If desired, the **Select** button can be pressed again and a different controller selected.
7. When the Array Editor dialog is closed and any changes have been made, then the user will be prompted whether or not to save the changes in a file. Saving the changes will allow the changed values to be retrieved and reviewed at a later time (by using the **Browse** button).

5.2.5 Expert mode WITHOUT an Array Definition File

The stand-alone Array Editor Expert mode dialog is shown below. Use the toolbar icons, if the Array-Editor has been started within the ApossIDE.



This dialog is used in the same way as the User mode dialog. The only difference is that **Add** and **Delete** buttons are now available. However, since no Array Definition File has been loaded, the Array Editor expects to work directly with the controller. As a result, the add and delete functions behave slightly differently than they would if a file was loaded:

Add Array

Add a new array after the last existing array. The user will be prompted for the array size.

> This new array will not be added to the controller until **Write to Controller** is pressed.

Delete Array

Delete the currently selected array. Since “holes” (i.e. missing array numbers) are not allowed, all arrays following the deleted array will be “renumbered” in order to close the gap left by the deleted array. The array will not actually be deleted on the controller until **Write to Controller** is pressed.

Note that only user arrays can be deleted; parameters cannot be deleted.

Add Index

Insert a single array index before the currently selected array index. If more than one array index is selected, then the new index is inserted before the first selected index. If no index is selected, then the new index is added at the end of the array. All array elements following the new index will be “renumbered” to make room for the new index.

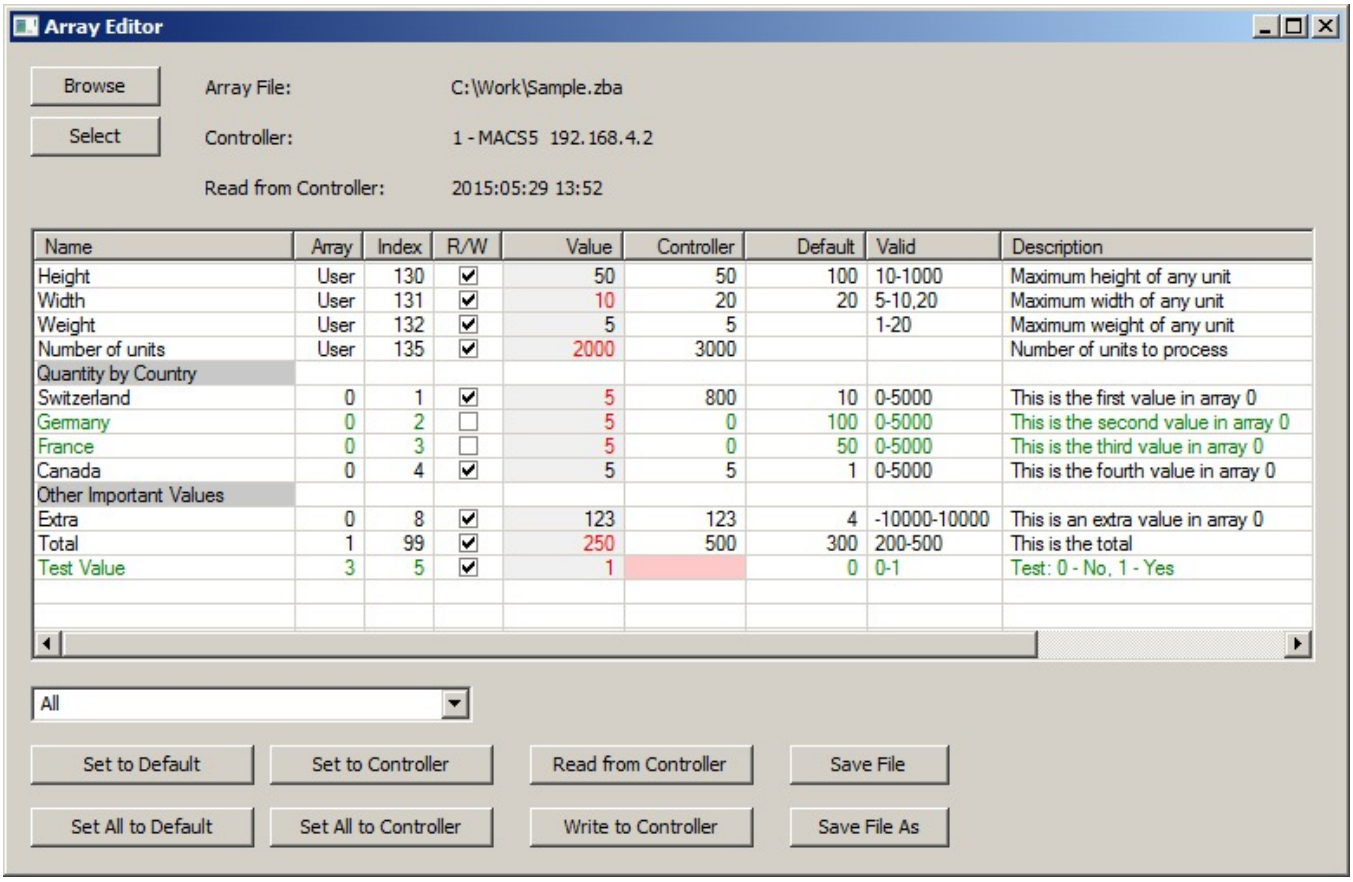
Note that new indices can only be added to user arrays.

Delete Index

This will delete all of the currently selected array elements. If all array elements are selected, then the array itself will be deleted. Note that only user array elements can be deleted. Since “holes” (i.e. missing array element numbers) are not allowed, all array elements following the deleted elements will be “renumbered” in order to close the gaps left by the deleted elements. The array elements will not actually be deleted on the controller until **Write to Controller** is pressed.

5.2.6 User mode WITH an Array Definition File

The stand-alone Array Editor User mode dialog when an array definition file is being used is shown below.



Name	Array	Index	R/W	Value	Controller	Default	Valid	Description
Height	User	130	☑	50	50	100	10-1000	Maximum height of any unit
Width	User	131	☑	10	20	20	5-10,20	Maximum width of any unit
Weight	User	132	☑	5	5		1-20	Maximum weight of any unit
Number of units	User	135	☑	2000	3000			Number of units to process
Quantity by Country								
Switzerland	0	1	☑	5	800	10	0-5000	This is the first value in array 0
Germany	0	2	☐	5	0	100	0-5000	This is the second value in array 0
France	0	3	☐	5	0	50	0-5000	This is the third value in array 0
Canada	0	4	☑	5	5	1	0-5000	This is the fourth value in array 0
Other Important Values								
Extra	0	8	☑	123	123	4	-10000-10000	This is an extra value in array 0
Total	1	99	☑	250	500	300	200-500	This is the total
Test Value	3	5	☑	1		0	0-1	Test: 0 - No, 1 - Yes

To use the Array Editor with an array definition file, proceed as follows:

1. Press the **Browse** button. This will display the normal Windows "**Open File**" dialog and allow the user to select an array definition file. Array definition files will have the file extension ".zba". The contents of the file will then be displayed in the dialog. At this point, the "Controller" column will still be empty (since no arrays have been read from a controller) but all other columns may or may not contain values.
2. Press the **Select** button and choose a controller in the same way as before. All arrays will then be read from the controller and the values will be "merged" into the list along with the existing values from the array definition file.
3. At any time once a controller is open, pressing **Read from Controller** button will re-read all the arrays from the controller and display the current values in the dialog. The "Read from Controller" string will be updated to show the date and time when the arrays were read from the controller.

When the Array Editor dialog is closed and any changes have been made, then the user will be prompted whether or not to save the changes in a file. Saving the changes will allow the changed values to be retrieved and reviewed at a later time (by using the "Browse" button).

Please note the following points when using the Array Editor with an array definition file:

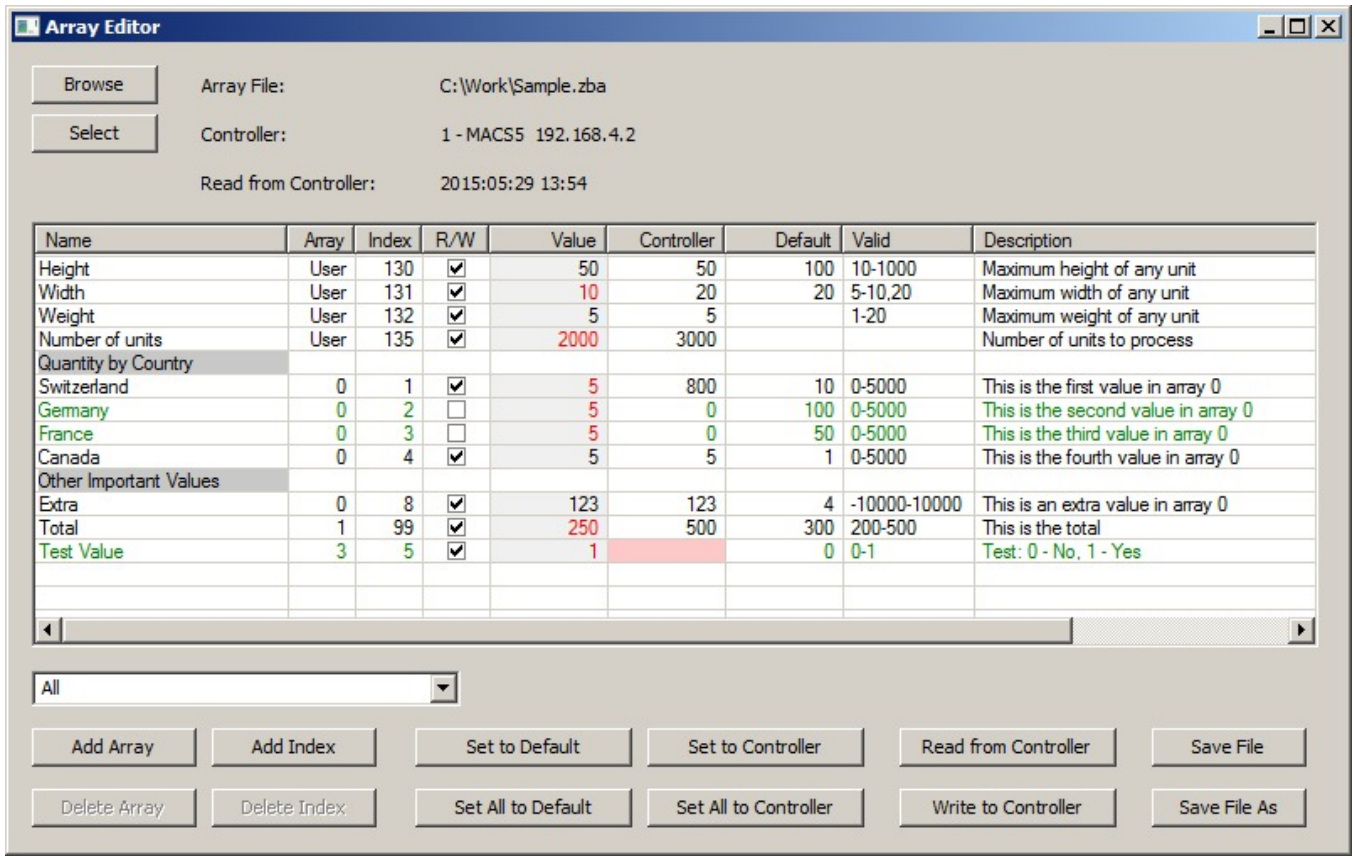
- Any array element (or any entire array) that does not have a definition within the array definition file, is not displayed and may not be accessed, even if it exists on the controller. The array definition file will allow access *only* to those array elements defined within the file.
- Any array element that is defined within the array definition file but that does not exist within the controller,

is displayed in the list but it is disabled and will be displayed in green. It cannot be selected or changed.

- Array elements may be flagged as “read-only” in the array definition file. In this case, the check box in the “R/W” (i.e. Read/Write) column will be clear and the array element will be displayed in green. Read-only array elements cannot be changed.
- Group headings will be displayed if they have been defined within the file. These are displayed with a gray background in the “Name” column. These rows in the list cannot be selected. The array element definitions within a group may correspond to different arrays; all definitions in a group need not come from the same array.

5.2.7 Expert mode WITH an Array Definition File

The stand-alone Array Editor Expert mode dialog when an array definition file is being used is shown below. Use the toolbar icons, if the Array-Editor has been started within the ApossIDE.



Name	Array	Index	R/W	Value	Controller	Default	Valid	Description
Height	User	130	☑	50	50	100	10-1000	Maximum height of any unit
Width	User	131	☑	10	20	20	5-10,20	Maximum width of any unit
Weight	User	132	☑	5	5		1-20	Maximum weight of any unit
Number of units	User	135	☑	2000	3000			Number of units to process
Quantity by Country								
Switzerland	0	1	☑	5	800	10	0-5000	This is the first value in array 0
Germany	0	2	☐	5	0	100	0-5000	This is the second value in array 0
France	0	3	☐	5	0	50	0-5000	This is the third value in array 0
Canada	0	4	☑	5	5	1	0-5000	This is the fourth value in array 0
Other Important Values								
Extra	0	8	☑	123	123	4	-10000-10000	This is an extra value in array 0
Total	1	99	☑	250	500	300	200-500	This is the total
Test Value	3	5	☑	1		0	0-1	Test: 0 - No, 1 - Yes

This dialog is used in the same way as the User mode dialog. The difference is that **Add** and **Delete** buttons are now available and that columns other than the “Value” column may be edited. As well, group headings and disabled list items (i.e. items that do not exist on the controller) may be selected. This allows those lines to be selected for the purposes of adding and deleting index lines. However, since an Array Definition File has been loaded, the Array Editor expects to work with the file rather than with the controller. As a result, the add and delete functions behave slightly differently than they would if a file was not loaded:

Add Array

Add a new array. The user will be prompted for the array number and array size. If no arrays have been read from the controller, then the new array elements will all be enabled. If arrays have been read from the controller, then the new arrays elements will be enabled or disabled according to whether or not they exist in the arrays read from the controller.

Note that adding arrays will add the arrays to the array definition file only; the arrays will *not* be added to the controller.

Delete Array

Delete the currently selected array from the array definition file. The array will *not* be deleted from the controller. Deleting an array in this case, is simply equivalent to removing user access to the array.

> When the array definition file is being edited, User, Global, Axis, etc., arrays *can* be deleted.

Add Index

Insert a single array element before the currently selected array element. If more than one array element is selected, then the new element is inserted before the first selected element. If no element is selected, then the new element is added at the end of the entire list. A dialog will be displayed allowing the user to specify exactly what kind of element is to be added.

Delete Index

Delete all of the currently selected array elements from the array definition file. The array elements on the controller will *not* be changed. Deleting array elements in this case, is simply equivalent to removing user access to those elements.

> When the array definition file is being edited, User, Global, Axis, etc., array elements *can* be deleted.

5.2.8 Creating an Array Definition File

Array definition files are simple text files with the file extension ".zba". Array definition files can be created directly using the Array Edit Expert mode. However, they are designed so that they may also be created using common spreadsheet programs. As well, they may be created and edited, with care, using simple text-editor programs.

To create a file using the Array Editor, follow these steps:

1. Start the Array Editor.
2. Click the **Select** button and connect to a controller. All arrays will then be automatically read from the controller.
3. Click the **Save As** button and write the array definition file to disk. This is important since it switches the Array Editor into "WITH File" mode.
4. Edit the array elements as required to add names, descriptions, minimum values, maximum values, etc.
5. Click the **Save** button when complete.

An example array definition file is shown below. The file will contain one line per definition. Each definition will consist of one or more "items", separated by either comma characters or semicolon characters. The first item in each line will be a "type" and can be one of the following values:

Date	The following item is the date that will be displayed in the dialog's "Read from Controller" heading.
Widths	The following items are the column widths to be used in the display. There should be 11 values for the following column order: Name, Array, Index, R/W, Value, Controller, Default, not used (0), not used (0), Description, Valid. Note that this is *not* the order shown by the Array Editor.

Title Defines a group title to be displayed in the list.

SDO index Defines a value where "SDO index" is the hexadecimal SDO index to be used to read/write the value (e.g. 0x2201 for user parameter values, 0x2300 for temporary axis parameter values, etc.). Note that DIM array values will *always* use 0x2100 regardless of the DIM array number.

Any line in the file that does not contain a recognized item type will be treated as a comment line and ignored.

For titles, the **Title** keyword must be followed by a definition for each of these items (in exactly this order):

Array, Index, Name, not used (empty), not used (empty), not used (empty), not used (empty), not used (empty), Read-only (0)

Array If the title is always to be displayed, regardless of the selected array, then this should be -1. Otherwise, it should be a hexadecimal SDO index. In this case, the title will be displayed only when either all arrays have been selected or that specific array has been selected.

Index This should be -1 *except* when **Array** is 0x2100 for a DIM array. In this case, it should be a DIM array number. The title will be displayed only when either all arrays have been selected or that specific DIM array has been selected.

Name The name to be displayed. This may contain commas, semicolons, and blanks but, if so, then the string *must* be enclosed in double quotes. These strings must *not* contain double quote characters.

For SDO values, the SDO index must be followed by a definition for each of these items (in exactly this order):

Array, Index, Name, Value, Default, not used (empty), not used (empty), Description, Read/Write, Valid

Array For DIM arrays, this value should be the array number. For all other values, it shoule be 0.

Index This should be the SDO subindex to be used to read/write the value.

Name The name to be displayed. This may contain commas, semicolons, and blanks but, if so, then the string *must* be enclosed in double quotes. These strings must *not* contain double quote characters.

Value The value to be displayed.

Default The default value.

Description The description to be displayed. This may contain commas, semicolons, and blanks but, if so, then the string *must* be enclosed in double quotes. These strings must *not* contain double quote characters.

Read/Write The Read/Write flag may be either 0 (for Read-only) or 1 (for Read/Write).

Valid An optional validation string (enclosed in double quotes). This string can be used to specify a comma-separated list of valid values and value ranges. Examples: "1,3,5", "1-10", or "1-5,11-15,21-25"

If an array definition file is being created using a spreadsheet, then the spreadsheet columns must be set up exactly as described above. The spreadsheet can then be saved as a "CVS (Comma delimited)" file type. This will generate the correct comma-separated or semicolon-separated file. The file should be then be renamed to give it the ".zba" file extension.

Some older array definition files used a different syntax to define the arrays. The Array Editor is still capable of reading these files. However, if users encounter any of these old file, then the files can easily be converted to the updated syntax simply by opening the file with the Array Editor and resaving it.

Example of an Array Definition File

```
Date,2009:07:27 10:52
Widths,150,40,40,40,70,70,70,0,0,365,78
This is a comment line.
0x2201,0,1,Height,50,100,,,"Maximum height of any unit",1,"10-1000"
0x2201,0,2,Width,10,20,,,"Maximum width of any unit",1,"5-10,20"
0x2201,0,3,Weight,5,,,,"Maximum weight of any unit",1,"1-20"
0x2201,0,6,"Number of units",2000,,,,"Number of units to process",1
***** Another comment line *****
Title,0x2100,0,"Quantity by Country",,,,,,0
0x2100,0,0,Switzerland,5,10,,,"This is the first value in array 0",1,"0-5000"
0x2100,0,1,Germany,5,100,,,"This is the second value in array 0",0,"0-5000"
0x2100,0,2,France,5,50,,,"This is the third value in array 0",0,"0-5000"
0x2100,0,3,Canada,5,1,,,"This is the fourth value in array 0",1,"0-5000"
Title,-1,-1,"Other Important Values",,,,,,0
0x2100,0,7,Extra,123,4,,,"This is an extra value in array 0",1,"-10000-10000"
0x2100,1,98,Total,250,300,,,"This is the total",1,"200-500"
0x2100,3,4,"Test Value",1,0,,,"Test: 0 - No, 1 - Yes",1,"0-1"
```

5.3 Oscilloscope

The Oscilloscope is a tool for recording and graphically displaying values from one or more controllers while those controllers are operational and executing programs. It is intended to be used as an aid in debugging prototype applications, diagnosing operational problems, and tuning the internal controller parameters for optimal system performance.

Operating the Oscilloscope basically involves the following steps:

1. Deciding what type of Oscilloscope is to be used.
2. Starting the Oscilloscope (except the Record Oscilloscope).
3. Specifying the "curves" (i.e. the values, variables, states, etc.) that are of interest.
4. Starting the test (i.e. executing the application on the controller). The data for those curves will then be collected automatically.
5. Analyzing the resulting curves.
6. Repeating the test after making whatever changes are appropriate (i.e. to the curve definitions, to the controller parameters, to the application program, etc.).

The exact details of steps 3 and 4 depend on the type of oscilloscope being used (see Oscilloscope Versions). The last two steps are the responsibility of the user.

The following generic toolbar buttons are provided to assist the user in steps 3 and 4 (defining curves and starting tests). See Chart Window for a description of the other toolbar buttons.

Close all - Close the connections to all controllers on all interfaces.

Note that this will also close the connections to controllers in other windows (e.g. in any open ApossIDE window).

Reconnect all curves defined in the Oscilloscope to the currently defined default controller. (See the **Settings** → **Interface** menu command.)

If the curves were previously associated with different controllers, then the connection will be changed to the new controller.

Switch all - Select a new controller and switch all curves in the Oscilloscope to this new controller.

Add curve - Add a new curve to the Oscilloscope (see Adding Oscilloscope Curves).

Start an oscilloscope "run". This will start the recording of curve data.

Pause an oscilloscope "run". This will pause the recording of curve data. The run can be started again using the **Start** toolbar button.

Stop an oscilloscope "run". This will stop the recording of curve data. If curve data is being saved on the controller, then it will be uploaded into the Oscilloscope at this time.

Delete all the data saved for all the curves. This includes curves that are not currently being displayed.

5.3.1 Oscilloscope Versions

The following versions of the Oscilloscope available to users. Each version is designed for a different use but all versions basically operate in a similar manner and have a similar appearance.

Free Run Oscilloscope	Designed for debugging and diagnostics using PDO messages.
Single Shot Oscilloscope	Designed for debugging and diagnostics using sample data saved in the controller memory.
Record Oscilloscope	Designed for debugging and diagnostics using user-defined "Record" arrays.

5.3.1.1 Free Run Oscilloscope

The **Free Run Oscilloscope** is a tool designed specifically for diagnosing operational problems in running systems. However, it can also be used for debugging prototype applications. It is referred to as "Free Run" because the application on the controller can run freely without regard to how the Oscilloscope is being used or that it is being used at all. The impact of the existence and usage of the Oscilloscope on the running application has been minimized as much as possible. This allows the controller to behave as it would in actual operational situations.

Values are recorded and displayed in real time during the actual test. This information is normally collected using PDO messages from the controller.

PDO messages are not supported by some older controllers and by some connection interface types.

Free Run Oscilloscope in Polling Mode

In certain circumstances, the user may be unable or unwilling to allow the Oscilloscope to use PDO messages. Reasons for this include the following:

- The running application may already be using all available PDO message types. In this case, there will be no free PDO message type that the Oscilloscope can use to receive curve data from the controller.
- The controller may have an older version of firmware installed that does not support the dynamic configuration of PDO messages. The Oscilloscope will then be unable to configure the required PDO messages.
- The controller may be connected using a connection interface type that does not support PDO messages. For example, the EtherCAT interface does not currently support PDO messages.

In these cases, the Free Run Oscilloscope either cannot be used or must be used in the slower "**polling**" mode. Note that using polling can have a significant impact on the performance and response of the controller and the communication network.

If the Free Run Oscilloscope cannot be used, then it may be possible to use the Single Shot Oscilloscope as an alternative.

5.3.1.2 Single Shot Oscilloscope

The **Single Shot Oscilloscope** is designed more for debugging prototype applications than it is for diagnosing operational problems in running systems. However, it can be used for both. It can also be used in some situations where the Free Run Oscilloscope cannot be used.

It is referred to as "Single Shot" because the curve data is recorded on the controller itself and then uploaded to the Oscilloscope after the recording is complete.

Values are recorded on the controller in a "Record" array. The allocation of this array can be done either manually by the user (i.e. by defining a user array for the purpose) or automatically using available free memory in the controller. In either case, the **Single Shot Oscilloscope** may affect the memory management of the application program. The "Record" array will be automatically configured by the **Single Shot Oscilloscope** when recording begins.

5.3.1.3 Record Oscilloscope

The **Record Oscilloscope** is designed to display user-defined Record arrays and recordings in the DYNMEM created on the controller (i.e. with the RecordStart command). As with the Single Shot Oscilloscope, values are recorded using a Record array and uploaded to the Oscilloscope after the recording is done. However, the Record Oscilloscope relies on users configuring their own Record arrays from within their application programs. The arrays are not automatically configured by the Oscilloscope; the Oscilloscope simply reads already existing Record arrays that have been created by the user's application.

The **Record Oscilloscope** is useful in the following situations:

- Applications where the user has created a customized Record array from within the application program. This includes the use of multiple recordings within a single Record array.
- Testing controllers that contain firmware that is too old to support the automatic configuration of Record arrays. In this case, the Single Shot Oscilloscope cannot be used.

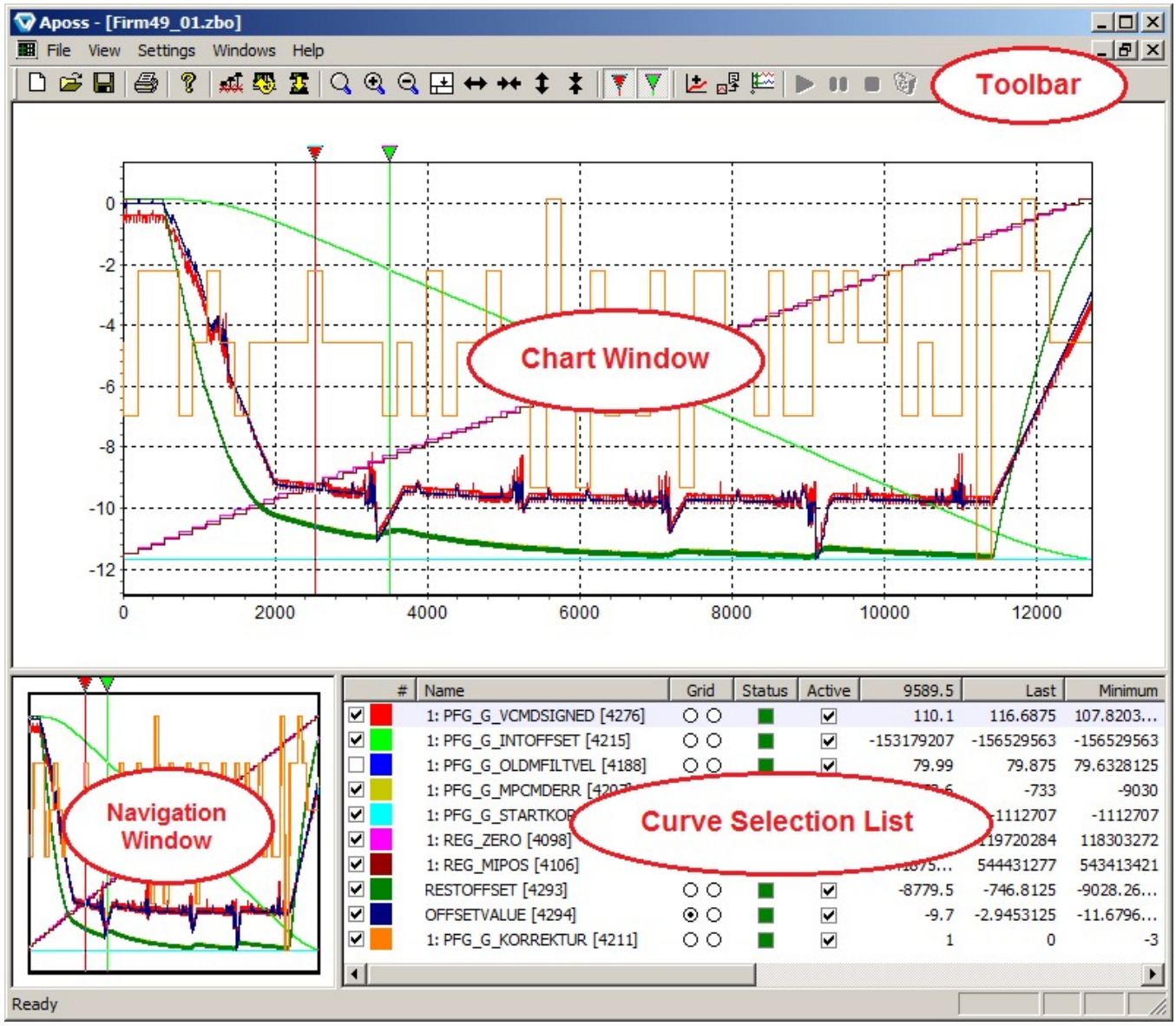
Since the **Record Oscilloscope** simply reads existing arrays, there is no need to start and stop recording. Hence,

the Record Oscilloscope operates slightly differently from the Single Shot Oscilloscope and the Free Run Oscilloscope. The steps are as follows:

1. *Before the Record Oscilloscope is started*, the user must execute an application program that records the desired data using the various Record commands. This program can be started in the normal way by using **Development** → **Execute**.
2. *After the test has been completed*, the Record Oscilloscope can be started. The Record Oscilloscope will then automatically attempt to read the Record array from the controller.

> The Record functionality that existed in ApossIDE versions prior to version 6.8, allowed the results to be saved in ".txt" files. These old ".txt" files can still be read and displayed by newer versions of the ApossIDE. See Reading Old Testrun Files.

5.3.2 Oscilloscope Window



The layout of the Oscilloscope Window is shown above. Note that when an Oscilloscope Window is active, then

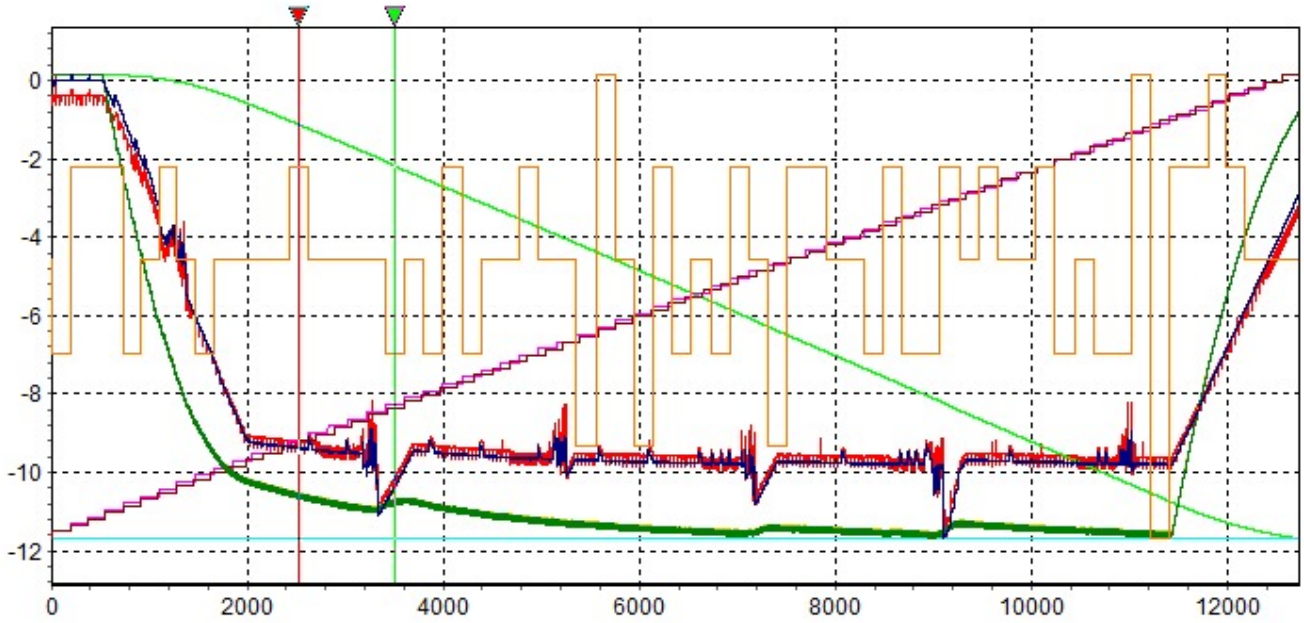
the normal ApossIDE menu and toolbar are replaced by a custom menu and toolbar for the Oscilloscope.

The Oscilloscope Window as a whole can be resized using the standard Microsoft Windows mechanisms. As well, the individual sub-windows can be resized by using the left mouse button to drag the “splitter” bars that separate the windows.

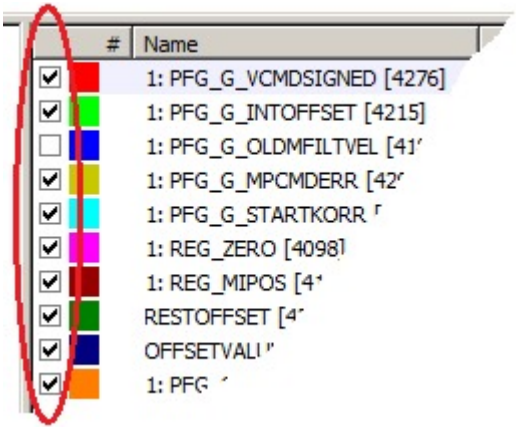
Note that if the windows are made too small, then the Oscilloscope will stop displaying the charts rather than continue to “shrink” the charts past the point where they become unusable.

5.3.2.1 Chart Window

The **Chart Window** is the main graphical display window for the Oscilloscope. The curves representing the collected data values are displayed here. The chart displayed in this window can be zoomed in and out, panned, and modified in various ways in order to see more or less detail in the curves.



The figure above shows a sample chart. Each colored curve in the chart represents one value, measured over time, during the execution of a “test run”. The X-axis of the chart represents time and is usually measured in milliseconds. Time increases from left to right in the chart. Note that the time is derived from the controller's internal clock. As a result, test runs will not start at time 0. The **Toggle Absolute/Relative origin** toolbar button can be used to reset the start time to 0, making the chart more readable.



Individual curves can be shown or hidden simply by clicking on the checkboxes in the Curve Selection List (see the figure to the left). Other properties of the curves (e.g. color, scale, etc.) can also be set using the Curve Selection List.

Using the Mouse in the Chart Window

The mouse buttons can be used as follows in the Chart Window:

Left button (Zoom) - Zoom in to an area of interest in the chart by pressing the left mouse button, moving the mouse across the area of interest, and then releasing the mouse button.

> The selected rectangle will be stretched so that it is exactly displayed in the window. If stretching is not desired, then press and hold the **Ctrl** key before pressing the left mouse button.

Use the **Zoom Out** or **Zoom Reset** toolbar buttons to zoom back out again.

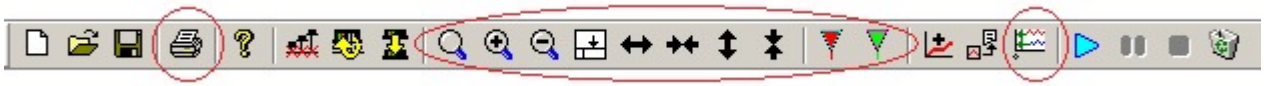
Right button (Pan) - Pan across the chart (i.e. move the chart in the window) by pressing the right mouse button, moving the mouse, and then releasing the mouse button.

Shift-Left button (Shift) - If a curve has been selected in the Curve Selection List, then pressing the Shift key before pressing the left mouse button will allow the selected curve to be shifted up or down. This can be useful if two curves need to be visually compared with each other. The mouse icon will change to an up/down arrow while shifting is active. If the selected curve cannot be shifted, then the mouse icon will not change. For more information, see Popup Menu Functions for Name Column → **Shift to Match**.

Mouse wheel (Shift) - If the mouse has a wheel, then this can be used as an alternative to the Shift-Left button to shift the selected curve up or down. This function can be enabled or disabled in the oscilloscope settings (see Oscilloscope Settings).

Toolbar Buttons

The toolbar buttons associated with the Chart Window are shown in the following figure:



Print - Print the chart.

Zoom Reset - Redisplay the entire chart.

Zoom In - Zoom in so that less of the chart (and more detail) is displayed.

Zoom Out - Zoom out so that more of the chart is displayed.

Splitter adjust - Adjust the splitter so that all of the curves are visible in the Curve Selection List.

Stretch - Stretch the time axis (X-axis) so that less time (and more detail) is displayed.

Shrink - Shrink the time axis (X-axis) so that more time is displayed.

Stretch - Stretch the vertical axis (Y-axis) so that more detail is displayed.

Shrink - Shrink the vertical axis (Y-axis) so that less detail is displayed.

Cursor A - Display or hide cursor "A".

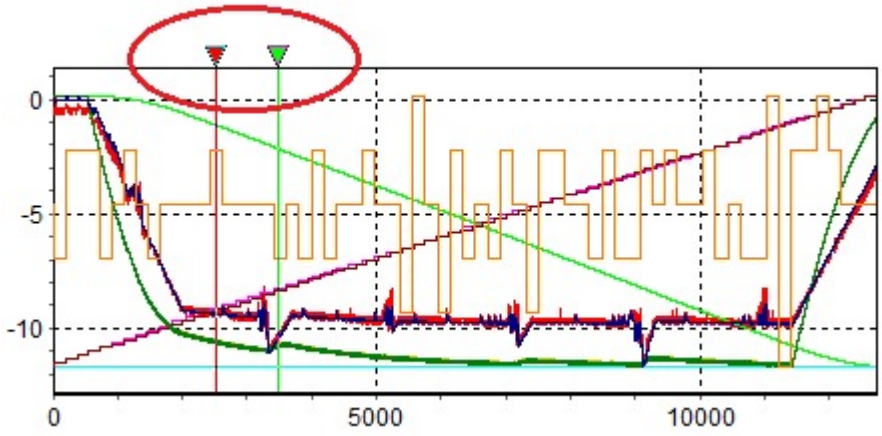
Cursor B - Display or hide cursor "B".

Toggle Absolute/Relative origin - Shift the time axis (X-axis) so that curves start at time 0. Clicking again will toggle the start time back to its original value.

Using A and B Cursors

The Oscilloscope provides two "cursors": A and B. These can be used as a convenience by the user to mark particular times along the X-axis. They can be shown or hidden by clicking on the **Cursor A** and **Cursor B** toolbar buttons or by using the **View** → **Cursor A** and **View** → **Cursor B** menu commands.

The cursors will appear as vertical lines through the chart and each will have a small triangular "handle" at the top end. This is shown in the figure below:



The cursors can be moved left or right by clicking on the triangular handle with the *left* mouse button and dragging the cursor left or right. Both cursors can be moved at the same time by using the *right* mouse button.

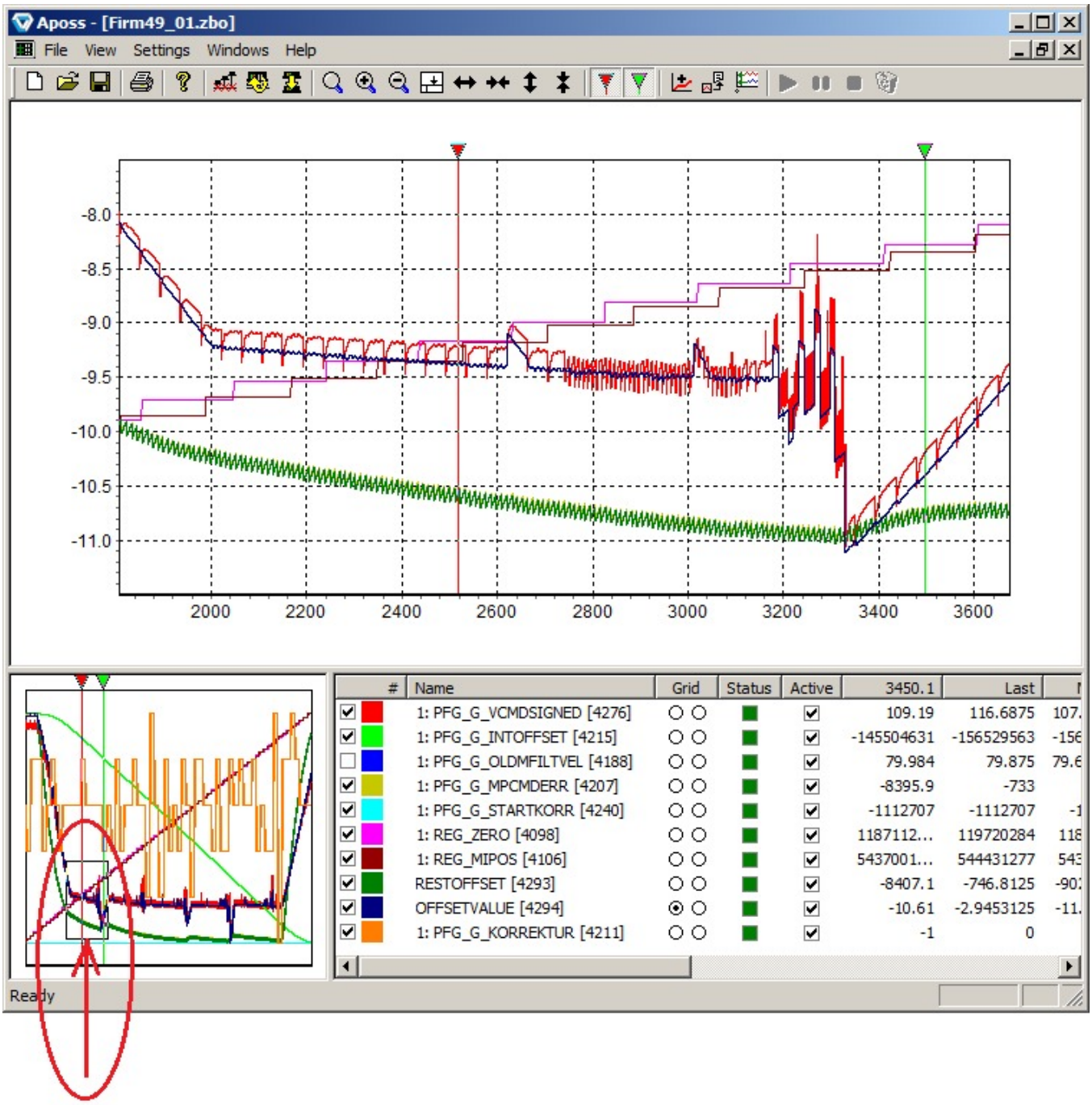
When the cursors are used, the value of each curve at the position of the cursor is shown in the **Curve Selection List**. The column headings will contain the X coordinate (time) of the position of each cursor. As well, there will be a column containing the difference between the curve values at the cursor positions (i.e. the value at cursor B minus the value at cursor A). This is shown in the figure below. If a curve is being drawn as a digital signal curve, then the difference column can be set to display the number of transitions between cursor A and cursor B. This can be done using the curve Properties dialog. See Modifying a Curve's Properties.

A 2517	B 3498	B-A 980.1
110.4	109.4	-1.02344
-144350851	-145569083	-1218232
79.99	79.99	0
-8171.5	-8325.1	-153.568
-1112707	-1112707	0
1186038...	1187112...	107349
5436141...	5437001...	86016
-8188.0	-8331.6	-143.563
-9.4	-10.4	-1.01563
1	-1	-2

5.3.2.2 Navigation Window

The **Navigation Window** displays an overview of the entire chart. If the main Chart Window has been zoomed, then this window allows the user to see and move the position of the zoomed area relative to the entire chart.

The zoomed area of the main chart will appear as a black outline rectangle in the Navigation Window. This is shown in the figure below. If Cursor A and Cursor B are shown, then they will also be visible in the Navigation Window.



Using the Mouse in the Navigation Window

Left button (Zoom) - Zoom in to an area of interest in the chart by pressing the left mouse button, moving the mouse across the area of interest, and then releasing the mouse button. The selected area will then be displayed in the Chart Window.

> The selected rectangle will be stretched so that it is exactly displayed in the window. If stretching is not desired, then press and hold the **Ctrl** key before pressing the left mouse button.

Right button (Pan) - Pan across the chart (i.e. move the chart in the window) by placing the mouse anywhere *within* the black outline rectangle, pressing the right mouse button, moving the mouse, and then

releasing the mouse button. The chart in the Chart Window will follow the movement of the black outline rectangle.

5.3.3 Curve Selection List

The **Curve Selection List** contains a list of all the curves that the user has defined and for which data have been collected, along with various details (i.e. values, variables, states, etc.) about each curve. All the properties of each individual curve, such as displaying, hiding, enabling, disabling, scaling, color, etc., are controlled from this window.

Curves can be moved within the list by clicking on the curve to select it and then using either the **Shift-Up** or **Shift-Down** arrow key. Curves can only be moved within their own group.

The setting of all display properties, data collection properties, etc., is done using this list. If the list is too long or too wide to fit within the window, then the associated scrollbars can be used to view other parts of the list. Column widths can also be changed by dragging the column header separators with the left mouse button. The figure below shows an example of the entire list.

#	Name	Grid	Status	Active	5754.5	Last	Minimum	Maximum	A 2780	B 3179	B-A 398.4	Units	Controller	Interface
	Test Group (Free Run)			☑									1 - MACS5 192...	Network (*.*.*.*)
	SYS_CLOCK	○ ○		☑	22037679	22041679	22031929	22041679	22034709	22035109	400	ms	1 - MACS5 192...	Network (*.*.*.*)
1	REG_ACTPOS	⦿ ○		☑	3633	824	-1	5000	71	792	721	cm	1 - MACS5 192...	Network (*.*.*.*)
1	REG_COMPOS	⊛ ○		☑	3639	829	-1	5000	70	798	728		1 - MACS5 192...	Network (*.*.*.*)
1	REG_TRACKERR	○ ○		☑	6	5	-9	9	-1	6	7	uu/ms	1 - MACS5 192...	Network (*.*.*.*)
	Other Curves													
1	VELMAX	○ ○		☑	1000	1000	1000	1000	1000	1000	0		1 - MACS5 192...	Network (*.*.*.*)
1	VELMAXQC	○ ○		☑	2184533	2184533	2184533	2184533	2184533	2184533	0		1 - MACS5 192...	Network (*.*.*.*)

5.3.3.1 Name Column

#	Name	Grid	Status	Active	5754.5	Last	Minimum	Maximum
	Test Group (Free Run)			☑				
	SYS_CLOCK	○ ○		☑	22037679	22041679	22031929	22041679
1	REG_ACTPOS	⦿ ○		☑	3633	824	-1	5000
1	REG_COMPOS	⊛ ○		☑	3639	829	-1	5000
1	REG_TRACKERR	○ ○		☑	6	5	-9	9
	Other Curves							
1	VELMAX	○ ○		☑	1000	1000	1000	1000
1	VELMAXQC	○ ○		☑	2184533	2184533	2184533	2184533

The "#" and "Name" columns consist of four parts:

Checkbox - If this is checked, then this curve will be displayed in the Chart and Navigation Windows. Otherwise, the curve is hidden. Left-clicking on the checkbox will toggle the checkbox.

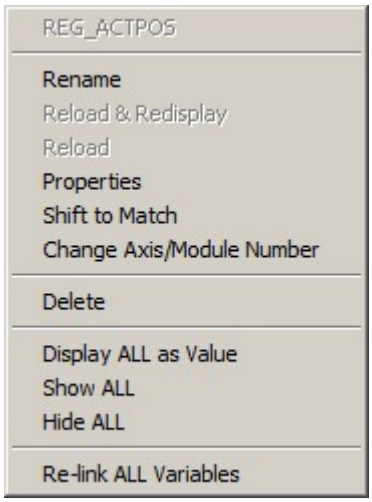
Color patch - This shows the color being used for the curve in the Chart and Navigation Windows. The color is one of the curve properties that can be changed by the user.

- The axis/module number. If the curve is an SDO curve that supports multiple instances (e.g. axis process data, etc.), then this column will contain the axis/module number. In all other cases, the column will be blank.

Name - The name of the curve. This can also be changed by the user. Names with a grey background will be headings that group together the following curves in the list. Note that "Other Curves" is a special group and its name cannot be changed.

Popup Menu Functions for Name Column

Right-clicking anywhere in the Name column will display the popup menu:



The top item in the list is simply the curve name for reference only. It will always be grayed. The other commands are as follows:

Rename - This command allows the user to rename the curve.

Reload & Redisplay - This command will "reload" the curve data from the controller and also redisplay the data in the Chart Window. Note that this will not always be possible, depending on the type of curve. The command will be grayed if reloading is not possible.

Reload - This command will "reload" the curve data from the controller. Note that this will not always be possible, depending on the type of curve. The command will be grayed if reloading is not possible.

Properties - This command is used to view and/or modify the properties of a curve. See Modifying a Curve's Properties.

Shift to Match - It is sometimes useful to be able to visually compare two curves. This is much easier to do if one curve is moved vertically so that it lies "on top" of the other curve. This command can be used to automatically shift the selected curve so that it has a "best fit" to another curve. The curve can be restored to its original position using the curve properties dialog (see Modifying a Curve's Properties).

Change Axis/Module Number - This command provides a quick way to modify the SDO axis/module numbers of all SDOs in a group. For example, all axis SDOs that reference axis 1 can be changed to reference axis 2. This command will be disabled if the selected curve is not an SDO curve or references an SDO that does not support multiple instances.

Delete - This command will delete the selected curve.

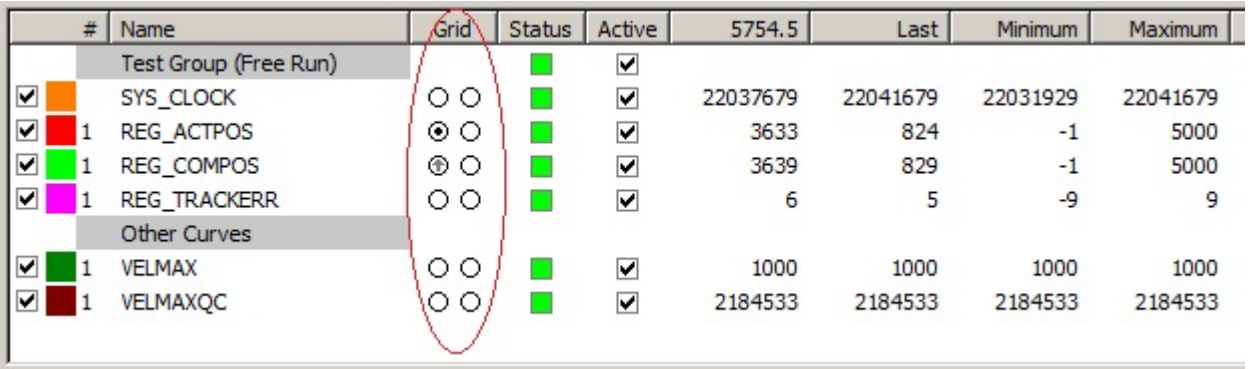
Display ALL as Value - This command will change the display interpolation style of *all* curves to "Value". See Modifying a Curve's Properties.

Show ALL - This command will display all curves in the chart. It can be used to avoid displaying all curves one-by-one.

Hide ALL - This command will hide all curves in the chart. It can be used to avoid hiding all curves one-by-one.

Re-link ALL Variables - When curves displaying program variables are being used, then the oscilloscope must maintain a link to the exact location of that variable in the controller's memory. However, if the user modifies the program source and downloads it to the controller, then the location of variables may change and the oscilloscope links will be invalidated. This command will cause the oscilloscope to "re-link" the variables using the current program source.

5.3.3.2 Grid Column



#	Name	Grid	Status	Active	5754.5	Last	Minimum	Maximum
	Test Group (Free Run)			☑				
☑	SYS_CLOCK	○ ○		☑	22037679	22041679	22031929	22041679
☑ 1	REG_ACTPOS	◉ ○		☑	3633	824	-1	5000
☑ 1	REG_COMPOS	⇧ ○		☑	3639	829	-1	5000
☑ 1	REG_TRACKERR	○ ○		☑	6	5	-9	9
	Other Curves							
☑ 1	VELMAX	○ ○		☑	1000	1000	1000	1000
☑ 1	VELMAXQC	○ ○		☑	2184533	2184533	2184533	2184533

The “Grid” column consists of two columns of radio buttons.

Clicking on the left radio button will cause the Y-axis of the Chart Window to be labeled on the *left* side with a scale corresponding to the selected curve. In the above example, the scale for “REG_ACTPOS” will be displayed. Clicking on the right radio button will be identical except that the Y-axis of the Chart Window will be labeled on the *right* side. The chart can be labeled on both the left and right sides at the same time.

> All the other curves will still be displayed using a generic “top-to-bottom” scale so that the entire curve will be visible. It is important to remember that the scale of these other curves will NOT correspond to the scale drawn on the Y-axis.

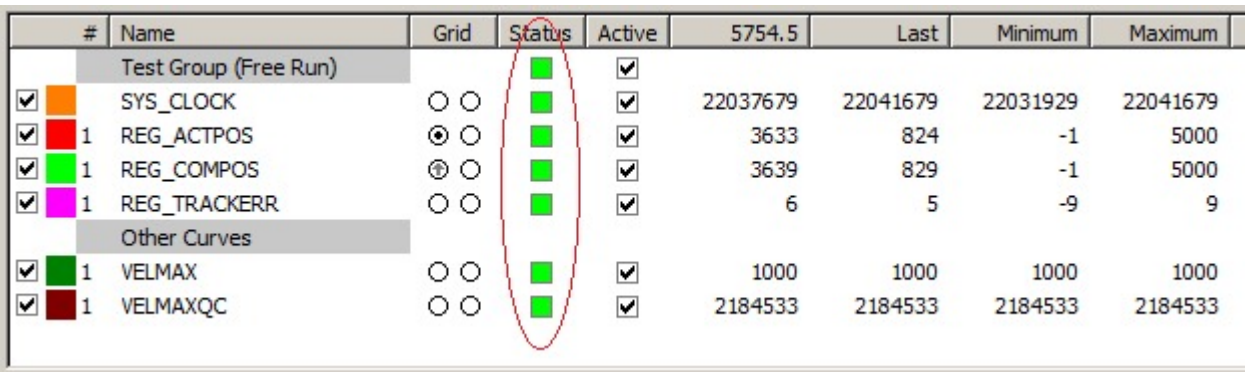
Since only one scale at a time can be drawn on either side of the chart, selecting a curve will cause any curve previously selected for that side to be deselected.

Sharing the Y-axis

In some cases, it is desirable to have two or more curves “share” the same Y-axis. For example, an “actual position” curve and a “commanded position” curve will be measured using the same units and should be displayed using the same scale in the Chart Window.

This can be done by right-clicking on the radio button. This will cause the radio button to display a small grey upward arrow. The selected curve will then share the same Y-axis as the curve that is currently being displayed on the chart's Y-axis. In the above example, “REG_COMPOS” shares the same Y-axis as “REG_ACTPOS”. Right-clicking again will toggle the “share” attribute.

5.3.3.3 Status Column



#	Name	Grid	Status	Active	5754.5	Last	Minimum	Maximum
	Test Group (Free Run)			☑				
☑	SYS_CLOCK	○ ○		☑	22037679	22041679	22031929	22041679
☑ 1	REG_ACTPOS	◉ ○		☑	3633	824	-1	5000
☑ 1	REG_COMPOS	⇧ ○		☑	3639	829	-1	5000
☑ 1	REG_TRACKERR	○ ○		☑	6	5	-9	9
	Other Curves							
☑ 1	VELMAX	○ ○		☑	1000	1000	1000	1000
☑ 1	VELMAXQC	○ ○		☑	2184533	2184533	2184533	2184533

The “Status” column shows the current status of the controller associated with the curve.

Note that not all curves need to be associated with the same controller; different curves can be associated with different controllers.

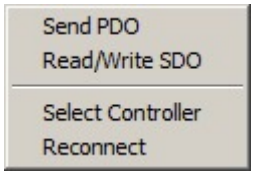
The Status column can have any of the following colors:

- Controller is executing a program.
- Controller is not executing a program.
- Controller is in an error state.
- Controller connection has been lost.
- Controller has been associated with this curve but is not currently connected.

This column will be empty for any curve that is not explicitly associated with a controller. For example, "Derived" curves are associated with other curves rather than with controllers.

Popup Menu Functions for Status Column

Right-clicking in this column will display the following popup menu of commands relating to the controller associated with the selected curve:



Send PDO - This command will allow the user to send a PDO message to the controller.

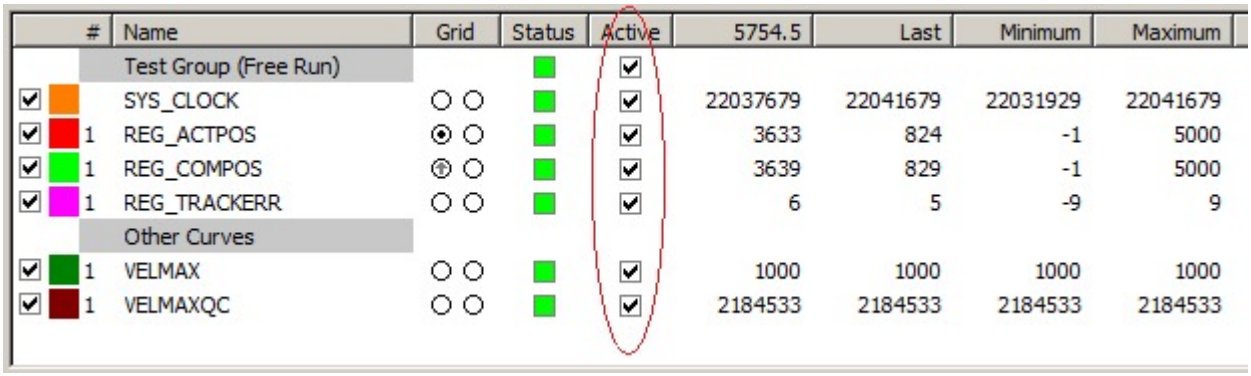
Read/Write SDO - This command will allow the user to read or write an SDO on the controller.

Select Controller - This command allows the user to select a different controller to be associated with the selected curve. Other curves that are associated with this controller, or other controllers, are not affected.

Note that in some cases, multiple curves *must* be connected to the same controller. For example, all curves that originate from the same recorded array *must* be associated with the same controller since the array comes from a single controller! In this case, the entire group of curves will be changed when one curve is changed. The Controller and Interface columns will show the controller with which the curve is currently associated.

Reconnect - If the controller is not currently connected or the connection has been lost, then this command will try to reconnect to the controller.

5.3.3.4 Active Column



#	Name	Grid	Status	Active	5754.5	Last	Minimum	Maximum
	Test Group (Free Run)			☑				
	SYS_CLOCK			☑	22037679	22041679	22031929	22041679
1	REG_ACTPOS			☑	3633	824	-1	5000
1	REG_COMPOS			☑	3639	829	-1	5000
1	REG_TRACKERR			☑	6	5	-9	9
	Other Curves							
1	VELMAX			☑	1000	1000	1000	1000
1	VELMAXQC			☑	2184533	2184533	2184533	2184533

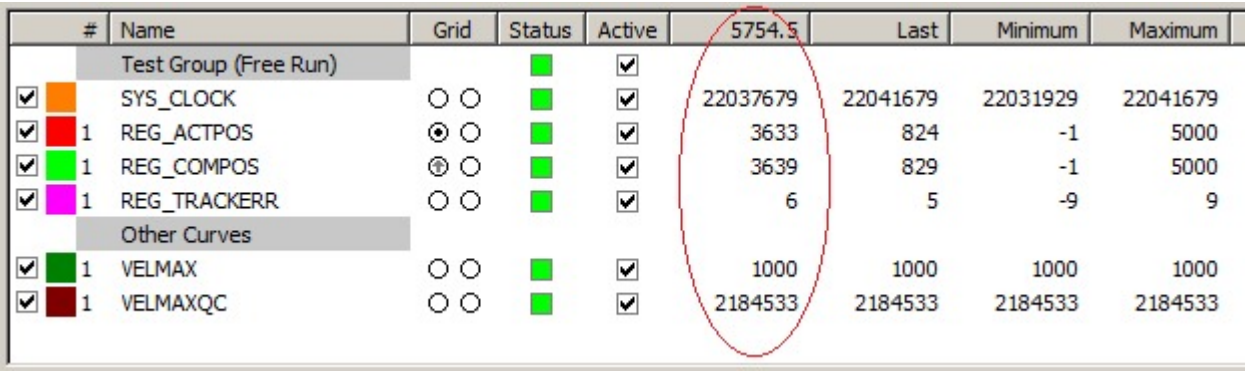
The "Active" column will contain a checkbox for those curves that are explicitly associated with a controller.

When the box is checked, then the Oscilloscope will enable data collection for that curve. Otherwise, data will not be collected for that curve. Clicking in the column will toggle the status of the checkbox.

If the box for a group curve is unchecked, then the boxes for all curves within that group will be grayed and data collection will be disabled for all these curves (even if the curve itself is still checked).

Note that data is only collected by the Oscilloscope when an Oscilloscope "run" has been started. See Starting and Ending the Test.

5.3.3.5 Mouse Position Column

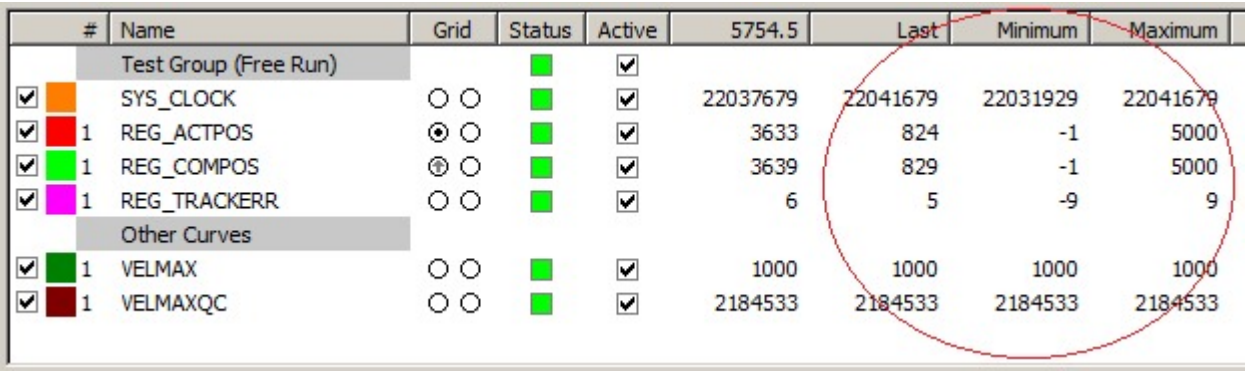


#	Name	Grid	Status	Active	5754.5	Last	Minimum	Maximum
	Test Group (Free Run)			☑				
☑	SYS_CLOCK	○ ○		☑	22037679	22041679	22031929	22041679
☑ 1	REG_ACTPOS	⦿ ○		☑	3633	824	-1	5000
☑ 1	REG_COMPOS	⦿ ○		☑	3639	829	-1	5000
☑ 1	REG_TRACKERR	○ ○		☑	6	5	-9	9
	Other Curves							
☑ 1	VELMAX	○ ○		☑	1000	1000	1000	1000
☑ 1	VELMAXQC	○ ○		☑	2184533	2184533	2184533	2184533

When the mouse is moved across the Chart Window, the value of each curve at the position of the mouse is shown in the Mouse Position column.

The column heading will contain the X coordinate (time) of the position of the mouse. If a value is shown as "*", then the column is too narrow to display the entire value. In this case, the column can be widened by dragging the column separator in the header with the left mouse button.

5.3.3.6 Last, Minimum, and Maximum Columns



#	Name	Grid	Status	Active	5754.5	Last	Minimum	Maximum
	Test Group (Free Run)			☑				
☑	SYS_CLOCK	○ ○		☑	22037679	22041679	22031929	22041679
☑ 1	REG_ACTPOS	⦿ ○		☑	3633	824	-1	5000
☑ 1	REG_COMPOS	⦿ ○		☑	3639	829	-1	5000
☑ 1	REG_TRACKERR	○ ○		☑	6	5	-9	9
	Other Curves							
☑ 1	VELMAX	○ ○		☑	1000	1000	1000	1000
☑ 1	VELMAXQC	○ ○		☑	2184533	2184533	2184533	2184533

These columns will show the values listed below.

Last - The last recorded value of each curve.

Minimum - The minimum recorded value of each curve.

Maximum - The maximum recorded value of each curve.

If a value is shown as "*", then the column is too narrow to display the entire value. In this case, the column can be widened by dragging the column separator in the header with the left mouse button.

5.3.3.7 Cursor A/B Columns

Maximum	A 2780	B 3179	B-A 398.4	Units	Controller	Interface
					1 - MACS5 192...	Network (*.*.*.*)
22041679	22034709	22035109	400	ms	1 - MACS5 192...	Network (*.*.*.*)
5000	71	792	721	cm	1 - MACS5 192...	Network (*.*.*.*)
5000	70	798	728		1 - MACS5 192...	Network (*.*.*.*)
9	-1	6	7	uu/ms	1 - MACS5 192...	Network (*.*.*.*)
1000	1000	1000	0		1 - MACS5 192...	Network (*.*.*.*)
2184533	2184533	2184533	0		1 - MACS5 192...	Network (*.*.*.*)

The column headings will contain the X coordinate (time) of the position of each cursor. When the cursors are used, the value of each curve at the position of the cursor is shown in the columns A an B: As well, there will be a column containing the difference between the curve values at the cursor positions (i.e. the value at cursor “B” minus the value at cursor “A”).

As well, the If a value is shown as “*”, then the column is too narrow to display the entire value. In this case, the column can be widened by dragging the column separator in the header with the left mouse button.

5.3.3.8 Units Column

Maximum	A 2780	B 3179	B-A 398.4	Units	Controller	Interface
					1 - MACS5 192...	Network (*.*.*.*)
22041679	22034709	22035109	400	ms	1 - MACS5 192...	Network (*.*.*.*)
5000	71	792	721	cm	1 - MACS5 192...	Network (*.*.*.*)
5000	70	798	728		1 - MACS5 192...	Network (*.*.*.*)
9	-1	6	7	uu/ms	1 - MACS5 192...	Network (*.*.*.*)
1000	1000	1000	0		1 - MACS5 192...	Network (*.*.*.*)
2184533	2184533	2184533	0		1 - MACS5 192...	Network (*.*.*.*)

An optional “Units” string can be defined by the user to each curve. This is described in Modifying a Curve's Properties. If defined, then the Units string is displayed in this column. This string is solely for information; it is not used by the Oscilloscope.

5.3.3.9 Controller and Interface Columns

Maximum	A 2780	B 3179	B-A 398.4	Units	Controller	Interface
					1 - MACS5 192...	Network (*.*.*.*)
22041679	22034709	22035109	400	ms	1 - MACS5 192...	Network (*.*.*.*)
5000	71	792	721	cm	1 - MACS5 192...	Network (*.*.*.*)
5000	70	798	728		1 - MACS5 192...	Network (*.*.*.*)
9	-1	6	7	uu/ms	1 - MACS5 192...	Network (*.*.*.*)
1000	1000	1000	0		1 - MACS5 192...	Network (*.*.*.*)
2184533	2184533	2184533	0		1 - MACS5 192...	Network (*.*.*.*)

If a controller is connected, then the “Controller” column will list the ID number (usually the CAN ID) and user-assigned name of the controller associated with each curve. If a controller is not connected, then only the ID number is shown.

The “Interface” column will list the connection interface description associated with each curve.

5.3.3.10 Modifying a Curve's Properties

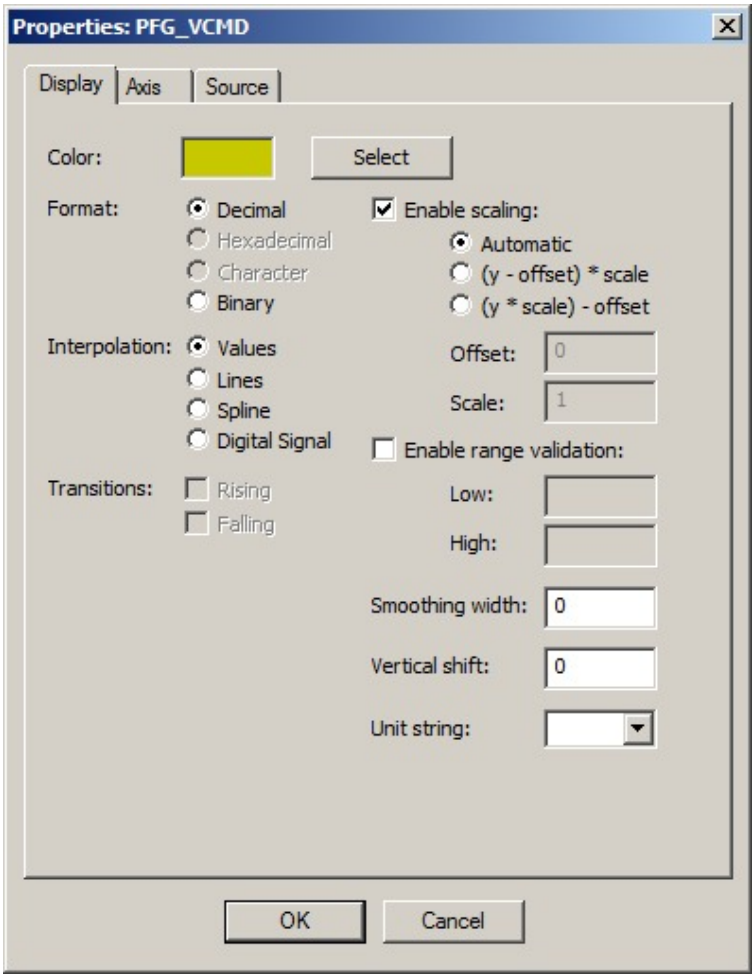
Curve Properties can be viewed and modified by right-clicking on the name of the curve in the “Name” column of the Curve Selection List and then selecting the Properties command. This will display a dialog containing the following tabs:

Display - Shows the various attributes affecting the display of the curve in the chart and the display of values in the list. This tab will not be present if the curve is not graphical (i.e. it cannot be drawn in the chart). For example, group headings contain no actual curve data and are never drawn.

Axis - Shows the various attributes affecting the Y-axis of the curve. This tab will also not be present if the curve is not graphical (e.g. group headings).

Source - Shows details of where and how the data for the curve is collected.

5.3.3.11 Curve Properties: Display



Color

Click on the **Select** button to change the color of a curve in the Chart Window.

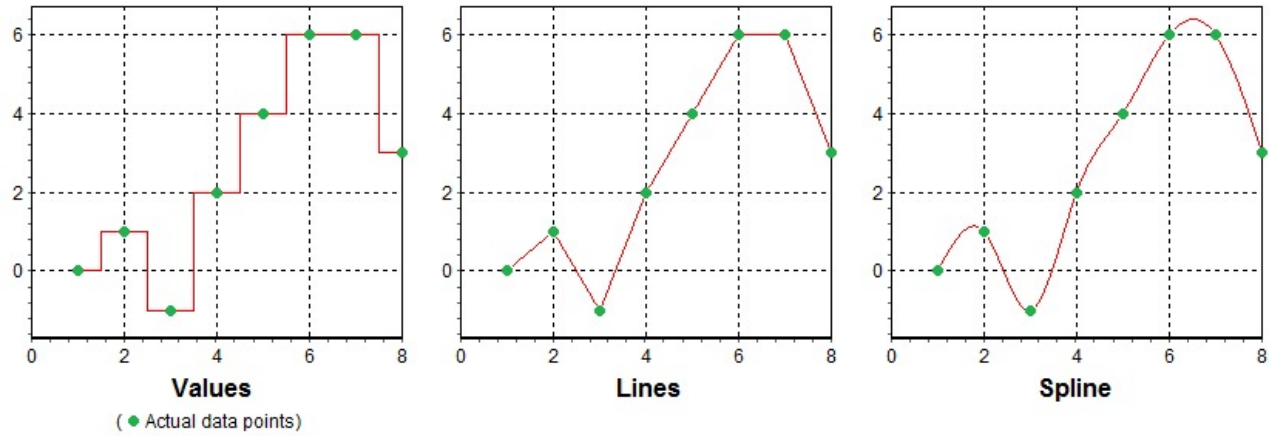
Format

This setting specifies how the curve values (i.e. mouse position, last, minimum, maximum, etc.) are to be displayed in the Curve Selection List. Note that the setting of **Interpolation** will also affect how curve values are displayed.

> Values should not be displayed as characters unless the controller is actually generating character values for that curve. If not, then blanks will usually be displayed.

Interpolation

Data are recorded for a curve as a series of discrete sample points. This setting specifies how the Oscilloscope will interpolate values between the actual sample points. The settings are illustrated below:

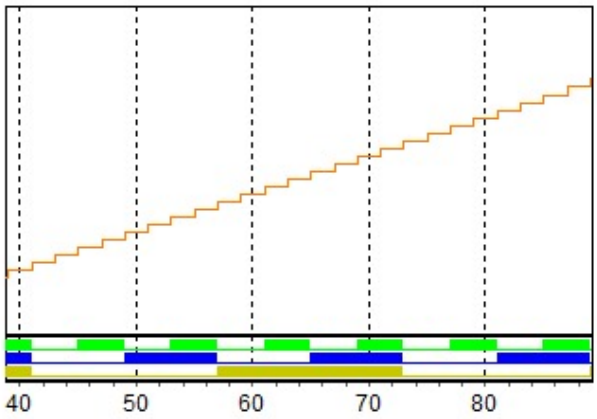


Typically, the choice of interpolation depends on the type of value being recorded. For example, any sort of "state" value will normally be displayed using **Values** since states are usually discrete in nature (i.e. they are one value or another and never have "in between" values). Values that are continuous in nature, such as position, velocity, etc., will normally be displayed using **Lines**.

Spline interpolation should only be used in very exceptional cases where a curve contains only a very limited number of sample points and is specifically intended to represent a spline (e.g. CAM Editor curves). In general, **Spline** should not be used when doing normal time-based sampling of data values. The reason for this is that splines are somewhat "expensive" in terms of CPU resource consumption.

> This setting may override the **Display** setting for some columns in the Curve Selection List. If **Values** is selected, then the values in the Curve Selection List will be displayed according to the **Display** setting. However, if **Lines** or **Spline** is selected, then the values for the mouse position column and all the cursor columns will be *interpolated* values and will be displayed as **Decimal** regardless of the **Display** setting.

The fourth option for "Interpolation" is **Digital Signal**. This is intended to be used for values (e.g. bit flags) that represent only an "on/off" state. When this option is chosen, then the curve is displayed at the bottom of the oscilloscope chart as shown below:



The curve is displayed as a thin line when the value is 0 and as a thick line when the value is non-0. The thickness of the line can be set using the **Settings** → **Oscilloscope** menu command.

Transitions

If "Interpolation" is set to "Digital Signal", then either "Rising", "Falling", or both can be selected. If selected, then the cursor difference column in the Curve Selection List will show the number of transitions of the specified type between the two cursors.

Enable scaling

For convenience, values recorded from the controller can be scaled in the Oscilloscope. For example, the user might want a value that is recorded in millimeters to be displayed in centimeters. This affects both the curve in the Chart Window and the displayed values in the Curve Selection List. Two choices of scaling algorithms are available: “(y-offset)*scale” and “(y*scale)-offset”. Usually, one or the other will be more natural in any particular application. In the millimeter-to-centimeter example, **Offset** would be set to 0 and **Scale** would be set to 0.1.

For curves displaying SDO values (see SDO Curve and Record Curve), an additional "Automatic" selection may be available. This will be available for SDO values that are defined to include a user-configurable scaling factor (e.g. HOME_VEL or PFG_VCMD). When selected, the scaling parameters will be automatically updated from the controller each time an Oscilloscope run is started.

Enable range validation

Often, it is useful to know if a curve value has gone “out of bounds”. This can be done by enabling range validation and setting the low and high bounds of the valid range of values. Any value that is not within the valid range will then be displayed in the Curve Selection List with a red background.

Note that if only one bound is appropriate, then the other bound can be left blank. An example is shown below where the maximum value for the second curve has been set to 4500.

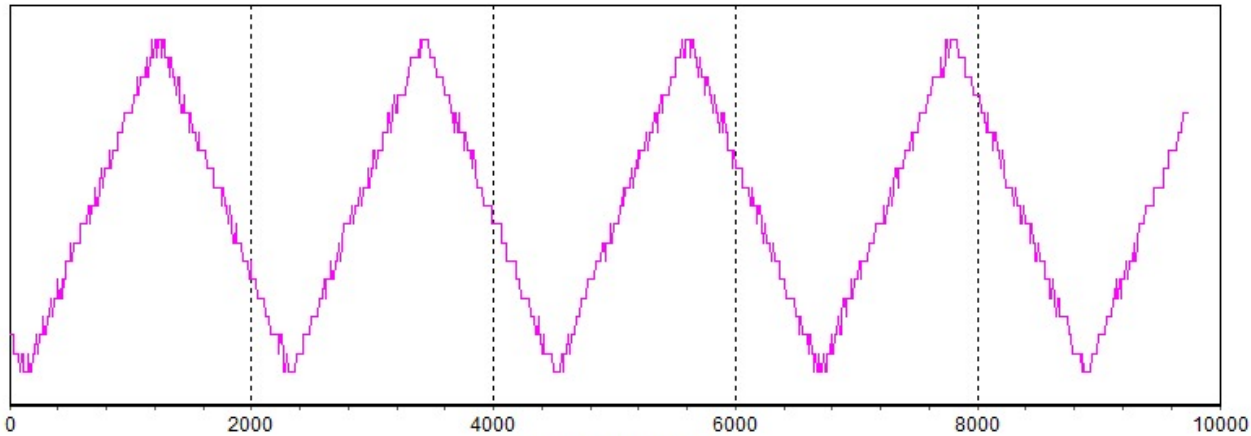
3857.4	Last	Minimum	Maximum	A 3766	B 5594
22035789	22041679	22031929	22041679	22035699	22037519
4905	824	-1	5000	4676	2344
4906	829	-1	5000	4680	2353
1	5	-9	9	4	9

Smoothing width

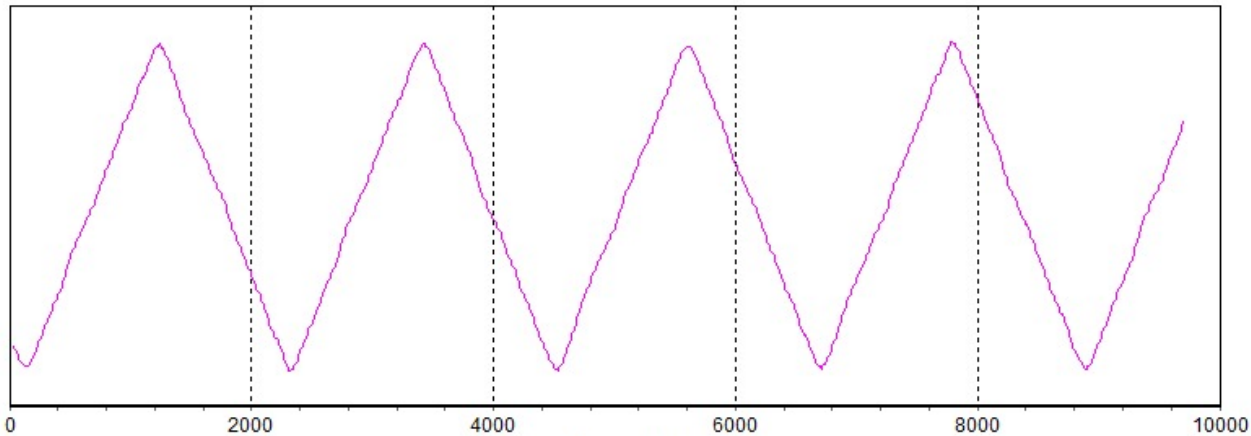
In some cases, measured values can contain a certain amount of “jitter” (i.e. oscillations that are fast, somewhat random, and usually small). This can be distracting in the chart. An example is shown below. Setting **Smoothing width** will reduce the appearance of this jitter by averaging the values of adjacent sample points.

Note that this is visual in the Oscilloscope only; it does NOT affect any of the actual values in the controller.

The width will specify the number of points on either side of each point that will be used in the averaging. For example, a width of 2 means 2 points on the left and 2 points on the right of each point for a total of 5 points in each averaged point.



Smoothing width = 0



Smoothing width = 4

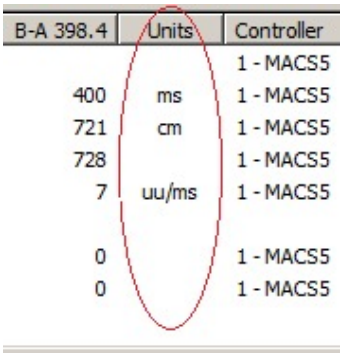
Warning: Care must be taken to avoid setting the smoothing width any higher than necessary. For example, if a width of 2 gives satisfactory results, then do not use a width of 4. As the width is set higher and higher, "real" information in the curve will begin to be "averaged away" and lost.

Vertical shift

A curve can be shifted vertically in the chart. This is usually done in order to visually compare one curve with another. This value specifies the amount of vertical shifting. Setting this value to 0 will restore the curve to its original position. Note that shifting a curve also affects the values that are displayed in the Curve Selection List.

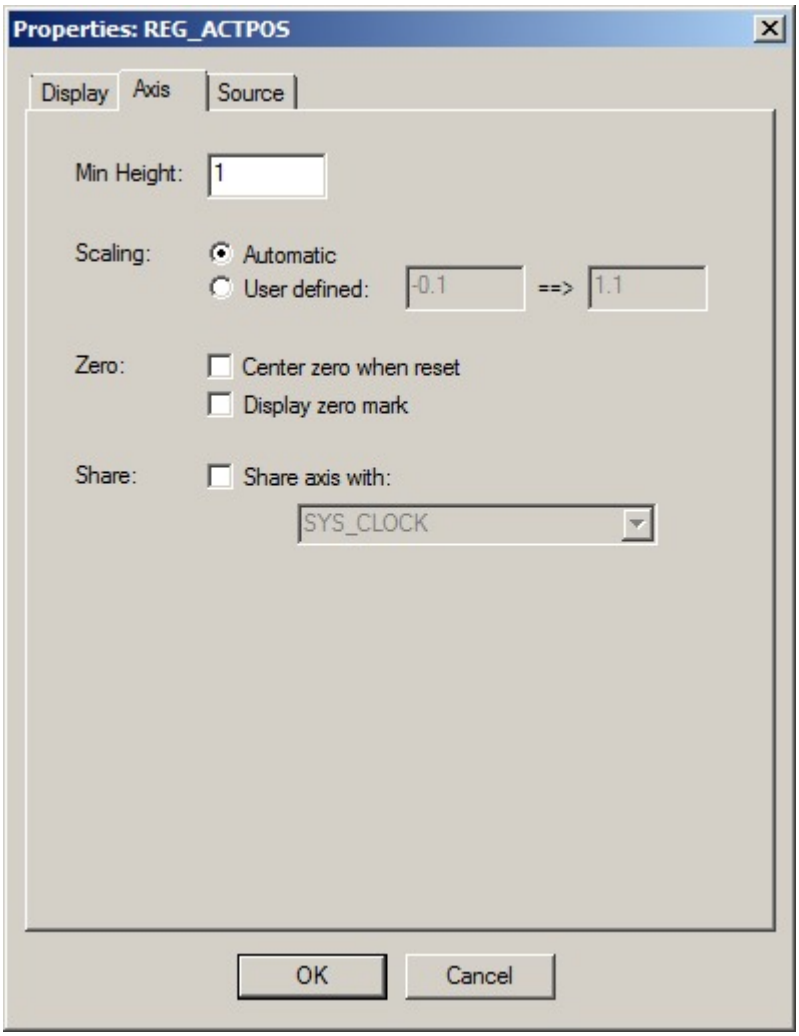
Unit string

An optional "Units" string can be set for each curve. Users can either enter their own string or use the drop-down box to select one of the predefined strings. The string will appear in the Units column in the Curve Selection List.



B-A 398.4	Units	Controller
		1 - MACS5
400	ms	1 - MACS5
721	cm	1 - MACS5
728		1 - MACS5
7	uu/ms	1 - MACS5
0		1 - MACS5
0		1 - MACS5

5.3.3.12 Curve Properties: Axis



Min Height

The Y-axis of the chart is normally scaled so that the entire curve can be drawn on the chart. However, this scaling cannot be determined in a way that will produce easily readable grid labels when the curve has a constant value or changes by only a very small amount. In these cases, the **Min Height** value is used as the minimum height of the Y-axis (i.e. the magnitude of the range of the Y-axis).

Scaling

If scaling is set to **Automatic**, then the Y-axis of the chart will be automatically scaled so that the entire curve can be drawn on the chart with maximum detail. The Y-axis range will be set to approximately 10% higher than the maximum value of the curve and 10% lower than the minimum value of the curve (and adjusted for the **Minimum Height** described above).

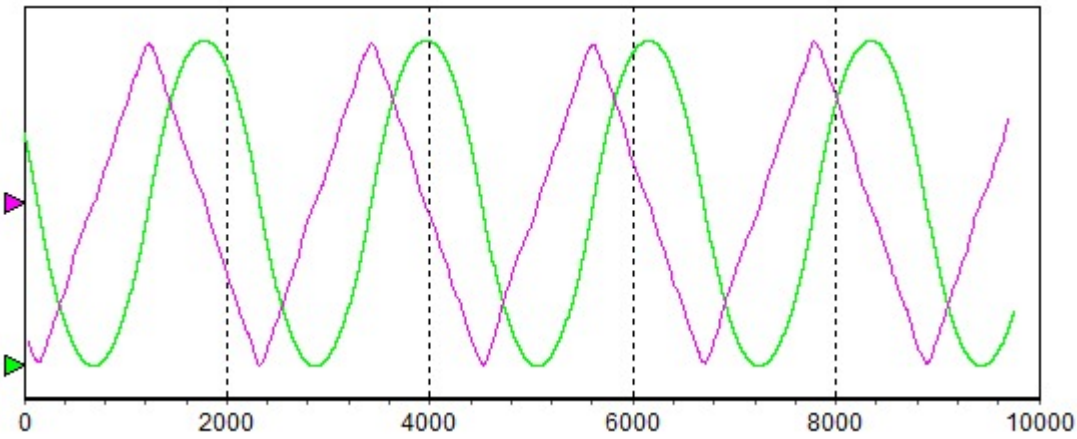
If scaling is set to **User defined**, then the user must specify the range of the Y-axis in the supplied boxes. Note that this will be automatically adjusted if the **Center zero when reset** option is enabled (see below).

The Y-axis scale defined here (i.e. either Automatic or User defined) will be the scale used for the curve when **Reset zoom** toolbar button is pressed. Note that if the scaling is changed, then the new scaling will not take effect until the *next* time that the user presses the **Reset zoom** toolbar button.

Zero

If **Center zero when reset** is enabled, then the Y-axis range for the curve will be adjusted so that it is symmetric about 0. This will result in the "0" grid line being drawn through the center of the chart.

If **Display zero mark** is enabled, then a small triangular mark will be drawn on the left side of the chart at the position of the 0 grid line for that curve. It will be drawn in the same color as the curve so that marks can be distinguished if marks are being drawn for multiple curves. The following figure illustrates these zero marks.



Share

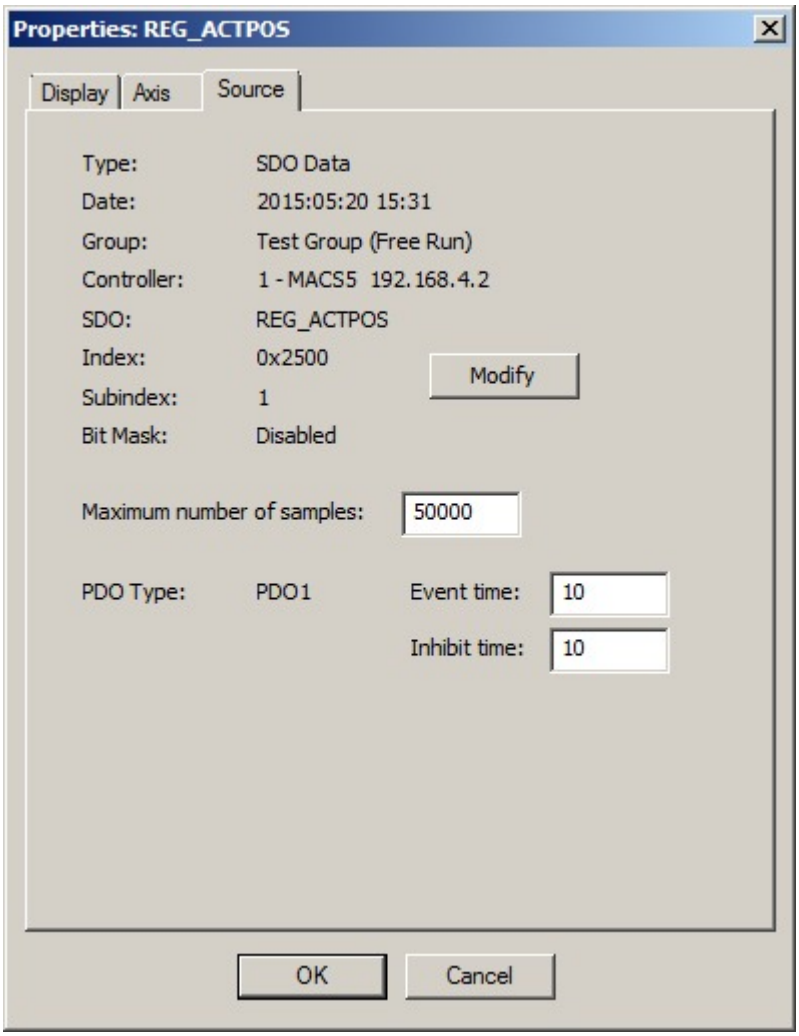
In some cases, it is desirable to have two or more curves "share" the same Y-axis. For example, an "actual position" curve and a "commanded position" curve will be measured using the same units and should be displayed using the same scale in the Chart Window.

This can be done by enabling **Share** and choosing the "other" curve from the list of curves provided. This curve will then share the same Y-axis as the chosen "other" curve.

Sharing a Y-axis can also be done using the radio buttons in the Grid column of the Curve Selection List.

> As well as *explicit* sharing (e.g. curve B shares the Y-axis of curve A and curve C also shares the Y-axis of curve A), *implicit* chains can also be created (e.g. curve C shares the Y-axis of curve B and curve B shares the Y-axis of curve A ... implying that curve C also shares the Y-axis of curve A). Changing how a curve shares the Y-axis can make or break these implicit chains.

5.3.3.13 Curve Properties: Source



The Source tab will display various information about the type of the curve and where the data for the curve came from. The exact content of this dialog will vary depending on the type of the curve. In some cases, attributes related to the collection of the data can also be modified.

5.3.4 Starting the Oscilloscope

Starting the Oscilloscope involves following steps:

1. Opening an application program in an ApossIDE window.
2. Connecting to the controller.
3. If the Record Oscilloscope is being used, then before starting the Oscilloscope, execute the application program which creates the "Record" array.
4. Starting the selected Oscilloscope.

Opening an application program

Begin by opening the application program (.mc or .m) corresponding to the program currently loaded or executing in the controller. This will open a normal ApossIDE window. Note that it is not strictly necessary to open the corresponding program. Any program, or even a new program, can be opened. For example, the user may not have the actual program readily available. However, opening the correct program can make the diagnostics effort easier since the user can refer to the program while trying to understand the results of the test.

Connecting to the controller

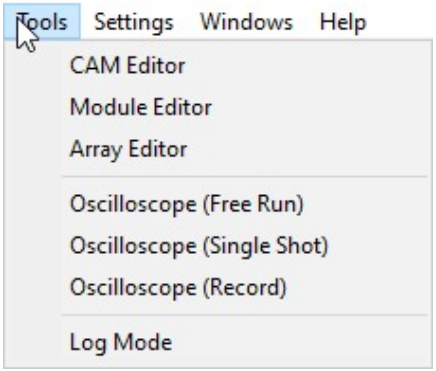
In the ApossIDE window, connect to the appropriate controller. Use any of the available methods (e.g. the **Esc** key to connect to the default controller, the **Development** → **Select Controller** menu command, the toolbar button, etc.).

Before using the Record Oscilloscope

The Record Oscilloscope expects a "Record" array to exist prior to the Oscilloscope being started. The Oscilloscope will automatically read this array when it is started. Hence, the user must first execute an application program that records the desired data using the various Record commands. This program can be started in the normal way by using **Development** → **Execute**.

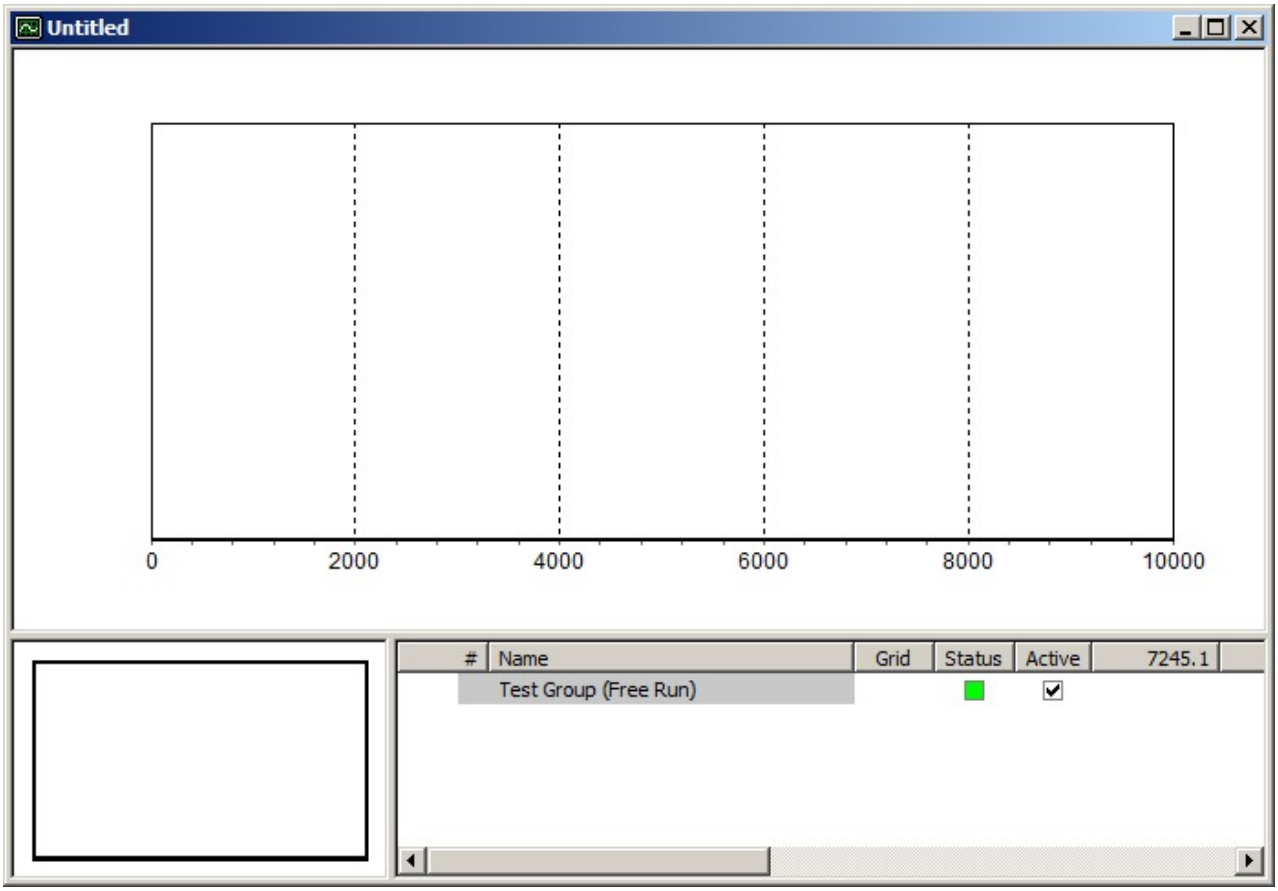
Starting the Oscilloscope

Each of the different Oscilloscope versions can be started from the **Tools** menu or by clicking on one of the Oscilloscope toolbar buttons. The Oscilloscope will also be started if a previously saved oscilloscope file is re-opened. Oscilloscope files are described below.

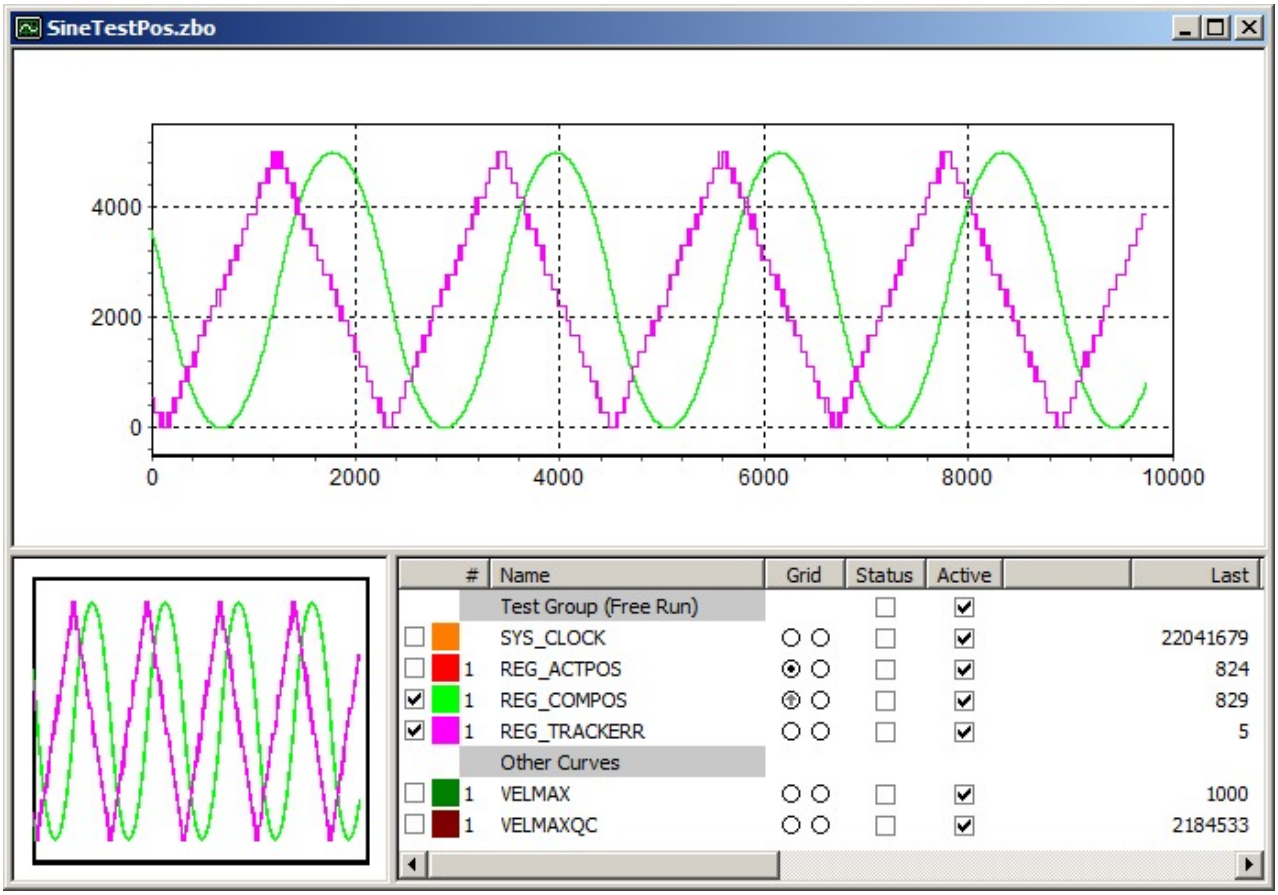


The Oscilloscope will automatically be connected to the controller that was connected in the ApossIDE window. Note that if a controller was not already connected in the ApossIDE window, then starting the Oscilloscope will cause the default controller to be connected.

An Oscilloscope window similar to the following will be opened. The connected controller can be seen at the far right side of the Curve Selection List. The green box in the Status Column in this example indicates that the controller is connected, is executing a program, and is not in an error state.



An Oscilloscope can also be started by opening an Oscilloscope file that was previously saved from an Oscilloscope. This is done simply by opening the Oscilloscope ".zbo" file using any of the normal methods for opening files. An Oscilloscope window similar to the following will be opened.

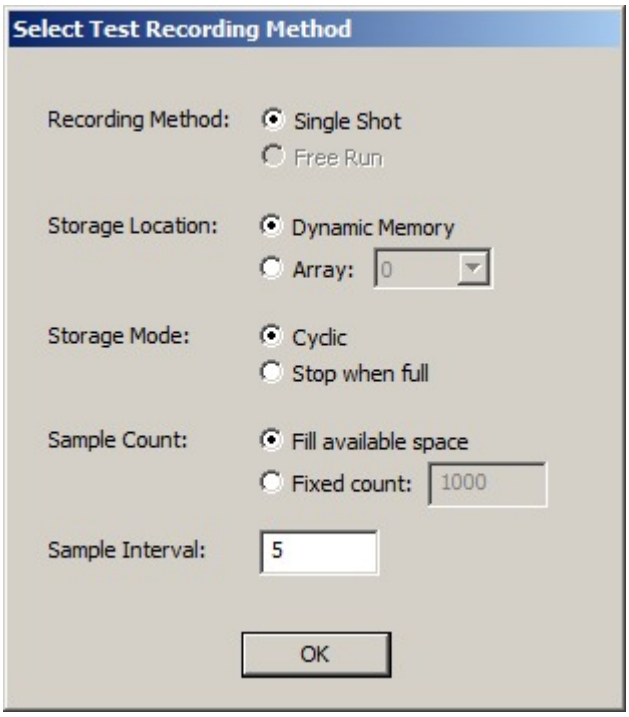


The white boxes in the Status column indicate that controllers are not connected. However, the controller ID and connection interface can be seen at the far right side of the Curve Selection List.

When Oscilloscope files are opened, controllers are never connected automatically. This allows Oscilloscope files to be opened and analyzed offline. Controllers can be reconnected either using the toolbar buttons or using the right mouse button in the Status Column in Curve Selection List.

5.3.4.1 Usage Notes for the Single Shot Oscilloscope

When the **Single Shot Oscilloscope** starts, it will automatically add a single group in the same way that the Free Run Oscilloscope does. However, the user is required to specify some characteristics of the "Record" array that will be used to record the curve data. The following window will be displayed:



Recording Method

For the Single Shot Oscilloscope, the recording method must be **Single Shot**.

Storage Location

This specifies the location where the "Record" array will be stored. If set the **Dynamic Memory**, then the "Record" array will be placed in available free memory in the controller. Note that the amount of free memory will depend on the size and number of programs and other data already stored in the controller. If set to **Array**, then the "Record" array will be placed in the specified user-defined array in the controller. This array must already have been defined by the user's application program.

Storage Mode

This determines what happens when the "Record" array becomes full. If it is set to **Cyclic**, then new data will start to overwrite old data and the old data will be lost. If set to **Stop when full**, then recording stops once the "Record" array is full and new data will be lost.

Sample Count

This specifies the maximum number of points that will be saved in the "Record" array. If set to **Fill available space**, then as many points as possible will be saved in the array. If set to **Fixed count**, then not more than that many sets of points (i.e. one point per curve) will be saved.

Sample Interval

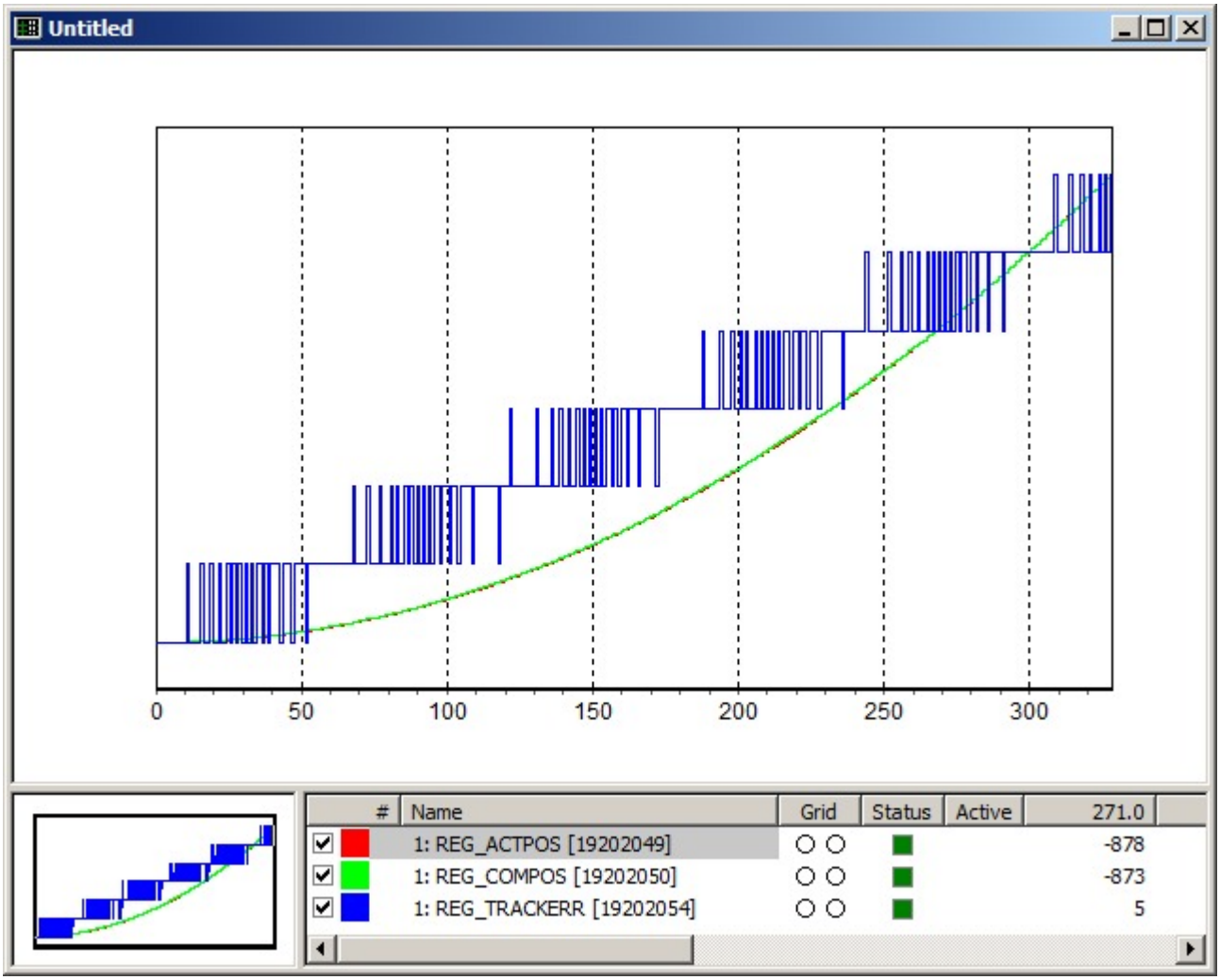
This specifies the time interval (in milliseconds) between samples.

5.3.4.2 Usage Notes for the Record Oscilloscope

When the **Record Oscilloscope** starts, it will automatically try to locate Record arrays on the controller. If only one Record array is found, then that array will be automatically uploaded and displayed. If more than one Record

array is found, then the user is requested to identify which array is to be uploaded. The other Record arrays can be uploaded once the Oscilloscope starts.

The Record Oscilloscope window will appear similar to that shown below.



This example was created with the ApossC commands:

```
RecordIndex(0x1250001 0x1250002 0x1250006);
RecordStart(0);
```

Please note the following with regard to the Curve Selection List:

- The command RecordIndex defined three values so three curves are automatically added to the Oscilloscope. The Oscilloscope will always add one curve for each curve defined with RecordIndex.
- The command RecordDest defined the location of the recorded data in the controller. By default the dynamic memory is used.
- The name of first curve in the set of Record curves will always be drawn with a grey background. This will let the user identify the sets of Record curves if multiple Record arrays have been added to the Oscilloscope.
- The Oscilloscope will try to derive curve names based on the indices specified in the RecordIndex command. These names will have the format “Axis: Name [Index]” where “Index” is the index from the command.
- The Oscilloscope can upload the Record array when the controller is in almost any state. The controller need not be executing a test program. So the Status column will not necessarily be bright green.

- Since the Oscilloscope does not control which values are recorded or the starting and stopping of the recording, the Active column will always be empty for Record curves.
- Also note that the **Start** toolbar button will be disabled in the Record Oscilloscope since starting and stopping of the recording is not done by commands RecordStart and RecordStop inside the application.

5.3.5 Saving and Opening Oscilloscope Files

At any time, the complete set of curves, their data and properties, and all the oscilloscope settings, can be saved as an "oscilloscope file" on the PC. This is done in the standard way by using the **File → Save** and **File → Save As** commands or by using the corresponding toolbar buttons. Oscilloscope files will always have a ".zbo" file extension. They will be identified in Windows folders with the icon .

Any of the standard methods of opening files (i.e. using the **File → Open** menu command or toolbar button, double-clicking the file, dragging the file onto the ApossIDE application icon or onto an open ApossIDE window, etc.) can be used to re-open an oscilloscope file. When an oscilloscope file is re-opened, then the Oscilloscope will be started and the curves re-displayed. Note that Record Import files created using the MemoryDump command, can be opened in the same way as normal oscilloscope files.

The Oscilloscope also has a special **File → Save without Data** command in the File menu. This command can be used to save the complete oscilloscope configuration (i.e. all curve definitions and settings, etc.) but without including any of the actual recorded curve data. The intent of this is to provide the user with a method of creating "template" oscilloscope setups that can be opened and re-used at a later time.

5.3.6 Updating from Record Files

For those controllers that support a microSD card slot, debug data can be saved in Record arrays and then written to files on microSD cards using the MemoryDump command. These files can then be analyzed at a later date using the Oscilloscope. The Oscilloscope provides two commands to support this:

File → Refresh from Record File

This command will update the currently open Oscilloscope file using the data saved in a user-specified Record file. The command will match existing curves in the Oscilloscope with curve data from the Record file. The data in all matching Oscilloscope curves will be replaced with the corresponding data from the file. Curves in the Oscilloscope that do not match any curve in the Record file will not be changed. Similarly, any curve data in the Record file that does not match any existing curve in the Oscilloscope, will not be loaded into the Oscilloscope. This command is expected to be used with Oscilloscope "template" files (see Saving and Opening Oscilloscope Files) although this is not necessary; the command may be used with any Oscilloscope file.

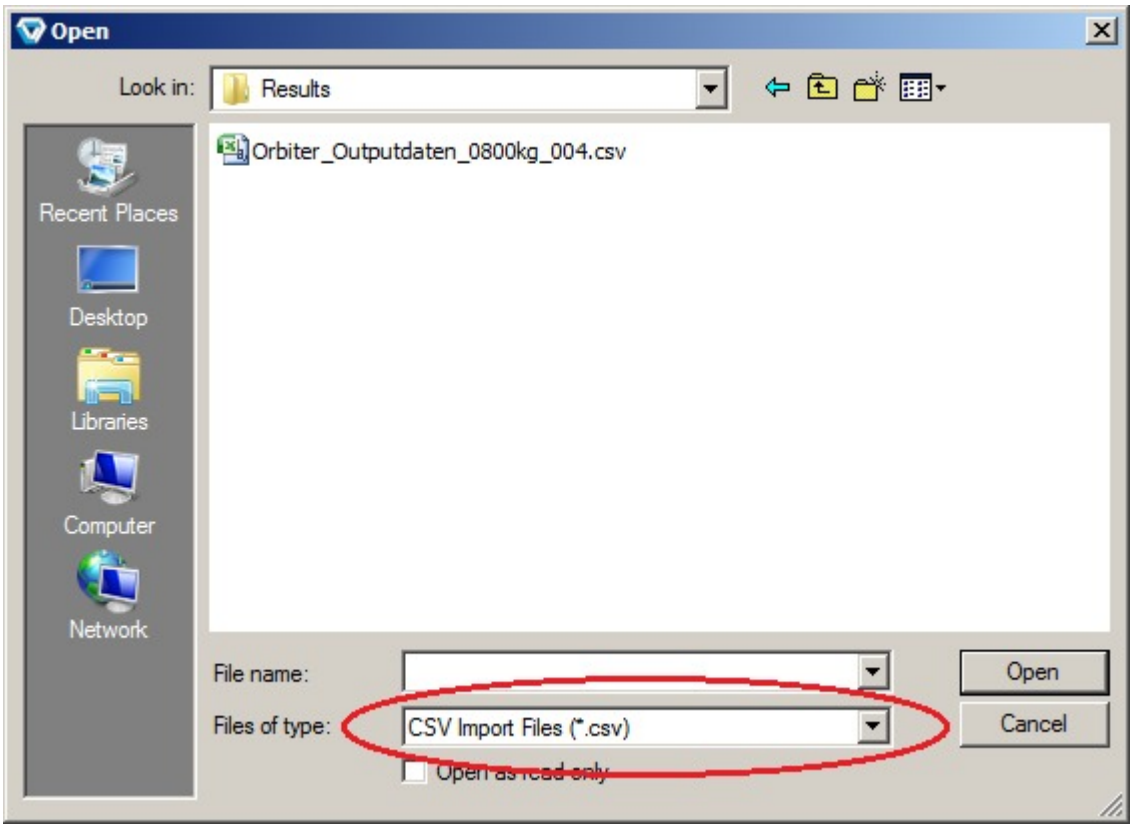
File → Batch Record File Conversion

If multiple Record files have been saved on a microSD card, then this command can be used to convert all of the files into Oscilloscope files. The currently open Oscilloscope file will be used as a "template" and processing will be done on each individual Record file in the same way it is done by the **Refresh from Record File** command. The result will be one Oscilloscope file for each Record file. The Oscilloscope files can then be opened in the normal way with the Oscilloscope.

> Typically, all of the Record files should be created containing the same set of curves and a corresponding Oscilloscope "template" file should be created containg the matching curves.

5.3.7 Reading .csv Files

The Oscilloscope is able to import curve data that has been saved in ".csv" files (e.g. files that have been created using Microsoft Excel). To do this, simply open the file with the ApossIDE using any of the standard Windows mechanisms (e.g. drag-and-drop the file onto an open Editpr window, use the **File** → **Open** menu command, etc.). Note that double-clicking on a ".csv" file will not open the file with the ApossIDE; it is more likely that the file will be opened with an application such as a spreadsheet application. To make the **File** → **Open** command easier to use, the **Files of type** field in the Open window can be set to **CSV Import Files** as is shown below.



When the file is opened, then a Single Shot Oscilloscope window will be opened to display the curves. The file can then be resaved as a normal Oscilloscope ".zbo" file if desired. It cannot be resaved as a ".csv" file.

Please note the following regarding the ".csv" file:

1. Only ',' or ';' are supported as separators.
2. The last column in each line may have an optional separator character at the end.
3. The first row must specify curve names (also separated in the normal way), one per column of data. ',' or ';' characters may not be used in curve names. Quotes are also not allowed.
4. All values will be treated as double precision floating point values (even if they are specified as integers).
5. Each line must specify one value for each curve; values may not be omitted. Empty lines are not supported.
6. The very last line may end in an optional carriage return.
7. The first column will be treated as time values and will be used for the X axis of the Oscilloscope. These values must increase.

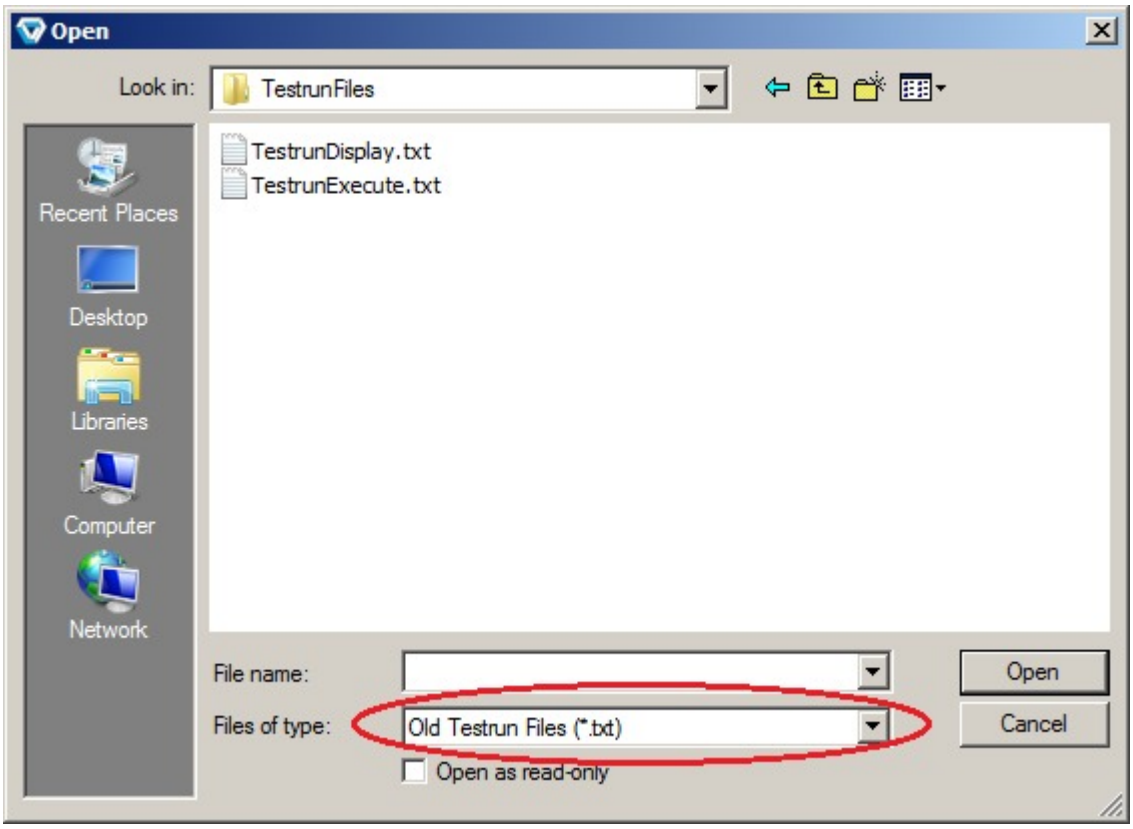
Example:

```
TimeStamp [ms], VPRI_MEAN, CAP1CUR_MEAN, CAP2CUR_MEAN, CAP1VOLT_MEAN,
CAP2VOLT_MEAN,
810336, 3.15, -3.81, -3.22, 14.77, 14.73,
810346, 3.15, -3.52, -3.12, 14.77, 14.73,
810356, 3.15, -4.20, -3.32, 14.77, 14.73,
810366, 2.94, -4.30, -3.42, 14.75, 14.73,
```

```
810376, 2.94, -4.10, -3.32, 14.77, 14.73,
810386, 2.94, -4.30, -3.52, 14.75, 14.73,
810396, 2.94, -4.39, -3.61, 14.75, 14.73,
```

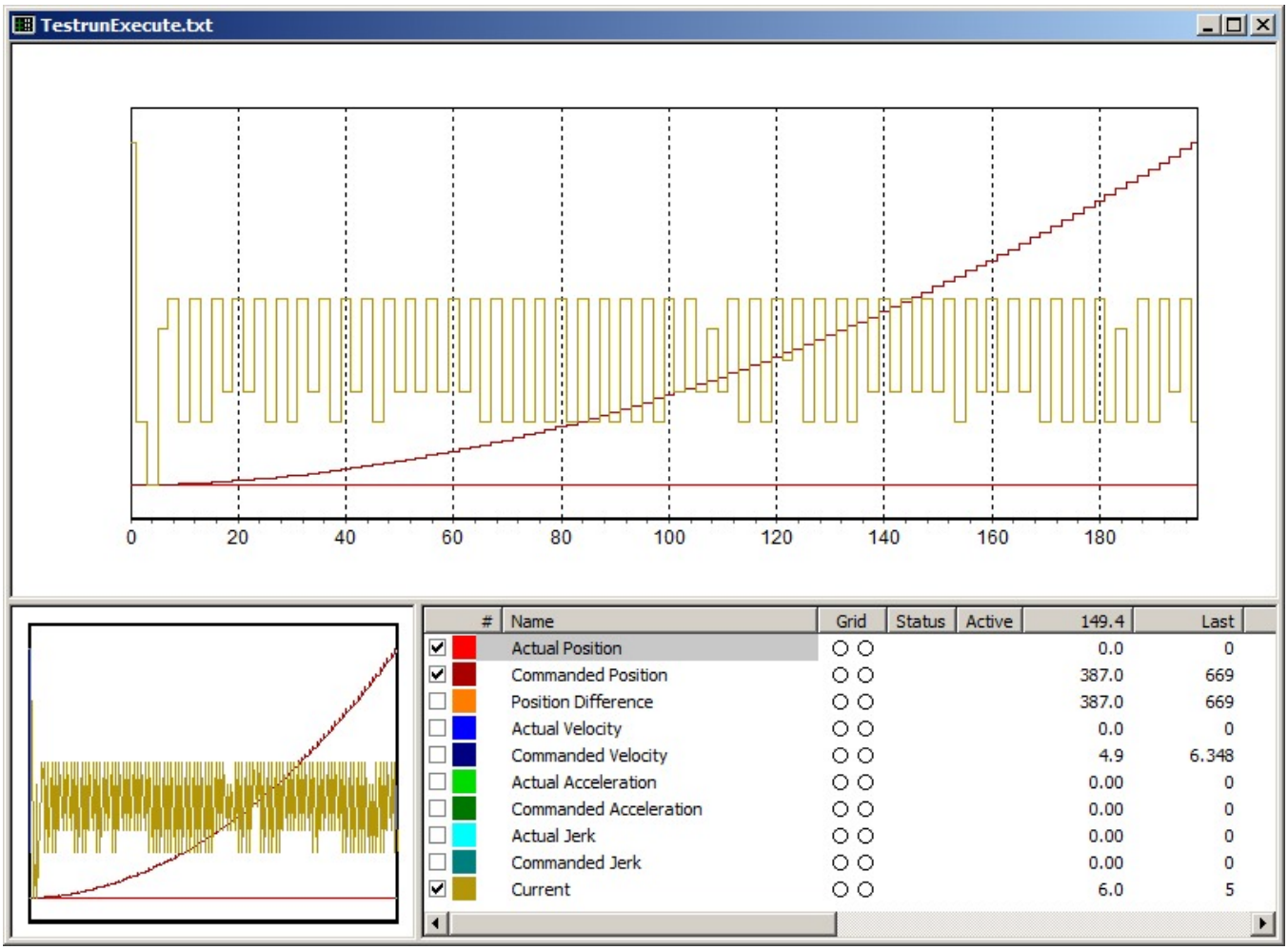
5.3.8 Reading Old Testrun Files

The TESTRUN functionality that existed in ApossIDE versions prior to version 6.8, allowed the results of **Execute testrun** and **Display recording** commands to be saved in ".txt" files. These old ".txt" files can still be read and displayed by newer versions of the ApossIDE. To do this, simply open the ".txt" file using the standard file open mechanism. To make this easier, the **Files of type** field in the Open window can be set to **Old Testrun Files** as is shown below.



When the file is opened, then a Record Oscilloscope window will be opened to display the curves. An example is shown below. The file can then be resaved as a normal Oscilloscope ".zbo" file if desired.

> If the ApossIDE does not recognize the file as containing old Testrun data, then the file will be opened in a normal Editor Window rather than in an Oscilloscope Window.



Old ".txt" files do not contain enough information about the original controller to reliably reconnect to the controller. Hence, the Status, Controller, and Interface columns are left empty. However, information about the original test parameters can be viewed using the curve **Properties** → **Source** window.

5.3.9 Adding Oscilloscope Curves

Curves are added to the Oscilloscope using the **Add curve** toolbar button. The types of curves that can be added will depend of the type of Oscilloscope being used.

The Oscilloscope uses "Groups" to manage sets of curves. When a test is started, the Oscilloscope will configure all the curves in a group into a PDO mapping on the controller. Then each time the controller generates a PDO message, the entire set of curve values will be transmitted from the controller to the Oscilloscope and displayed.

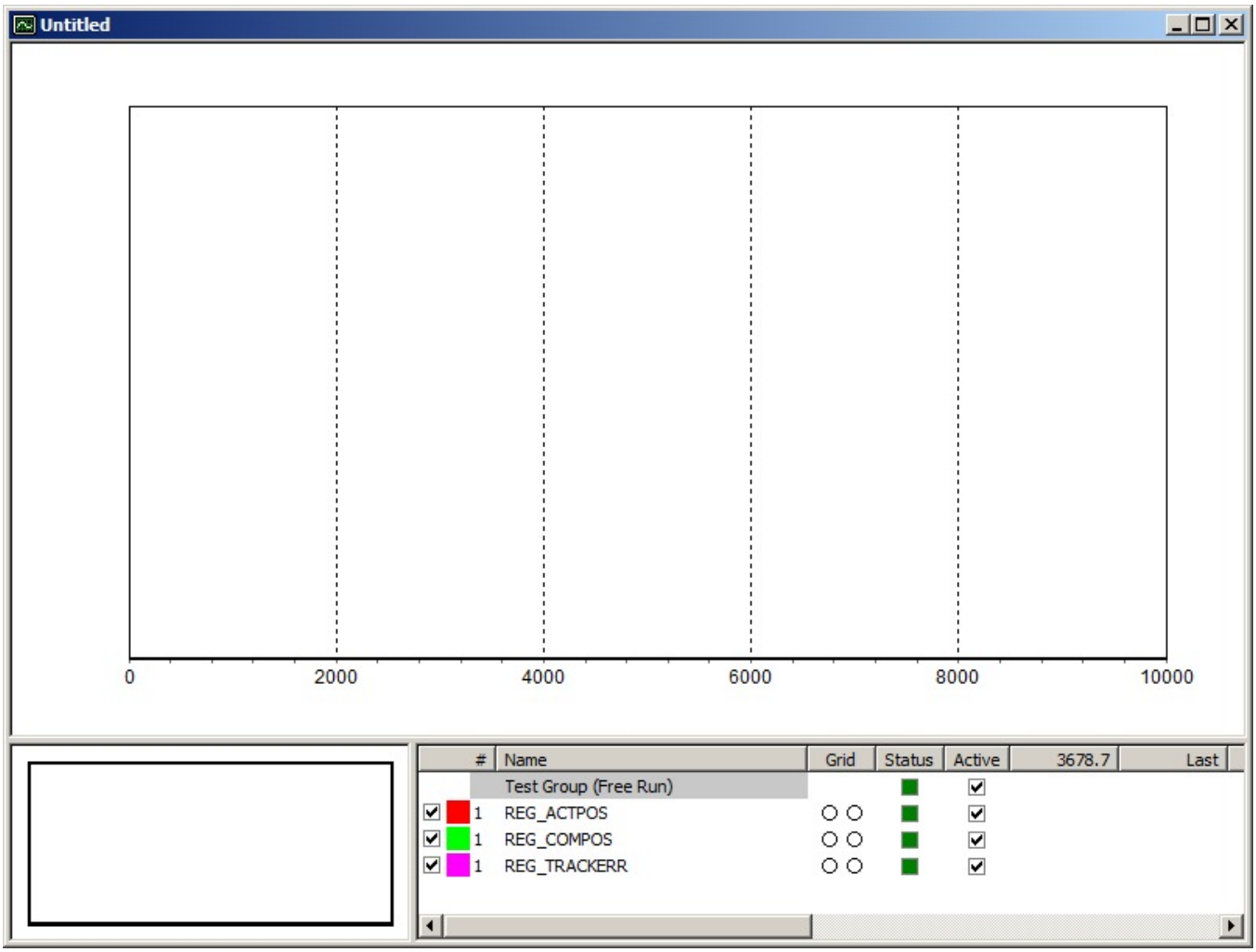
When the Oscilloscope is started, a single Group is added automatically. The user is then free to add any curves of interest to this group. The following types of curves may be of particular interest when diagnosing problems:

- Axis process data (SDO 0x2500). For example, axis position, velocity, tracking error, etc.
- System process data (SDO 0x2202). For example, states of the digital I/O pins.
- User parameters (SDO 0x2201) … if used by the application.
- Program variables. If the correct application program is known, then any variable used in the program can

be recorded.

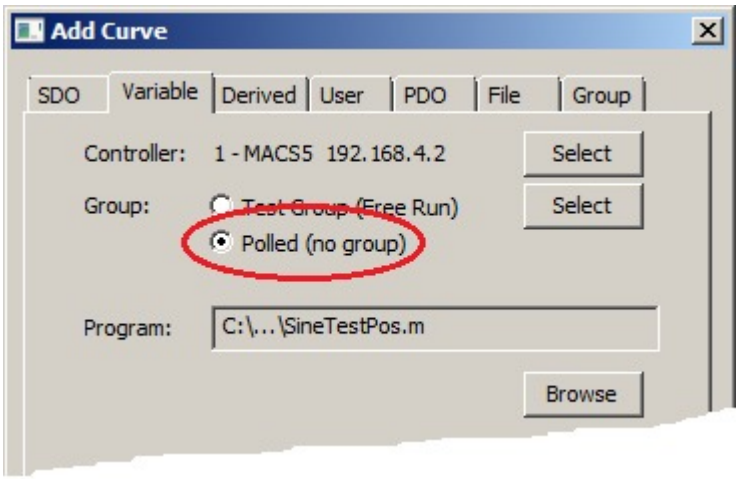
A single group is normally adequate for most purposes. However, the Oscilloscope will also support multiple groups. Multiple groups may be necessary, for example, if multiple controllers are being tested simultaneously. Since all curves in a group must be associated with the same controller, if two controllers are being used, then two groups must be defined, one for each controller.

Once the curves are added, then the Oscilloscope Window will look something like that shown below. In this example, three curves have been added but no data has yet been recorded so the chart is empty. Also, the dark green Status column indicates that the controller is connected but that no program is currently executing.

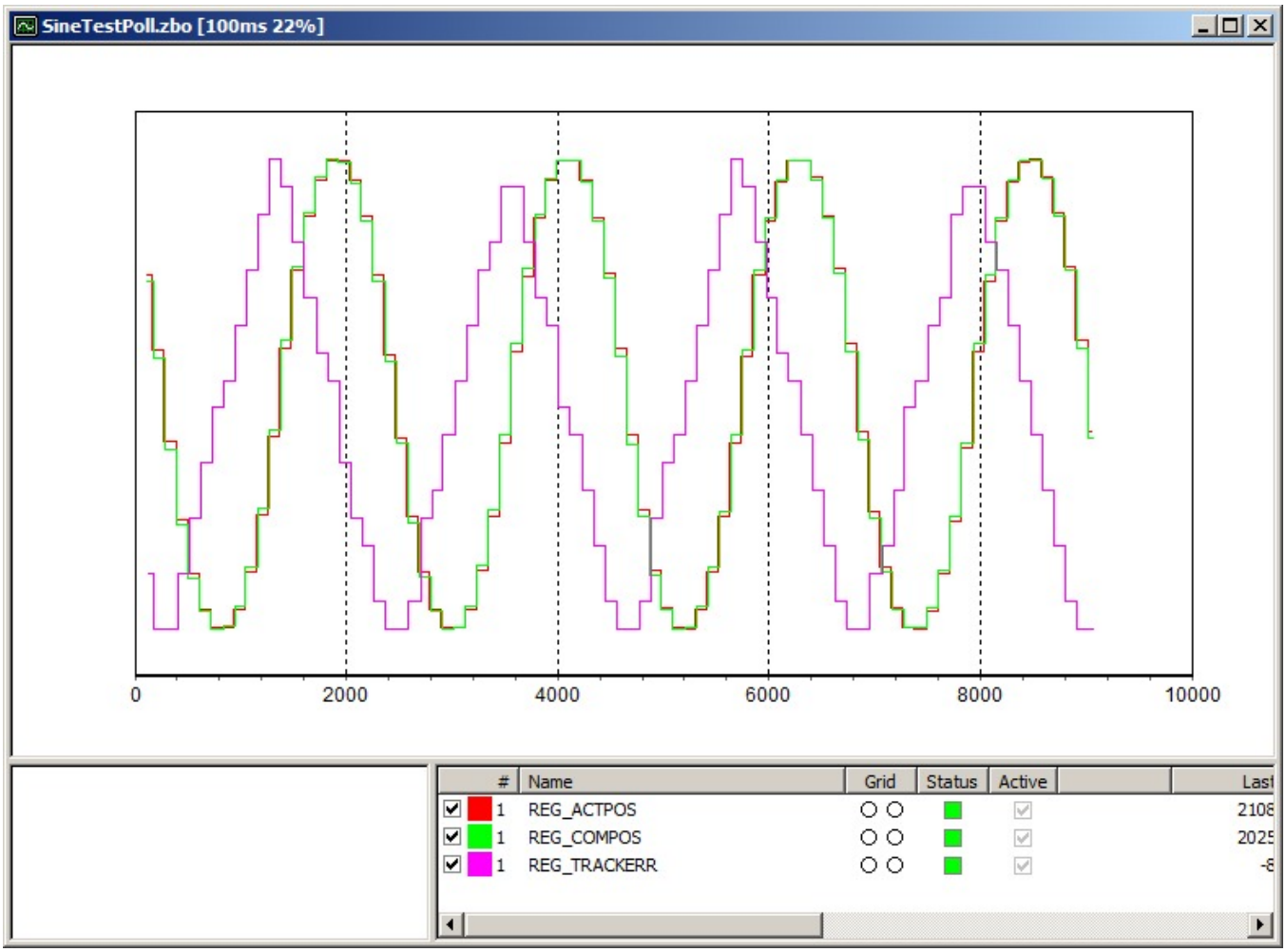


5.3.9.1 Usage Notes for Polling Mode

If the Free Run Oscilloscope is being used in Polling Mode, then add the curves as usual but instead of selecting a group for the curve, the **Polled (no group)** option should be selected.



Recording is then started in the normal way using the **Start** toolbar button. The Oscilloscope will retrieve curve data by "polling" each curve on a regular time interval. Each curve is polled by sending an explicit SDO request for its value. The user can set the polling time interval (i.e. the polling rate) using the **Settings** → **Oscilloscope** menu command. The Oscilloscope Window will look similar to the following:



When the Oscilloscope is actively recording data, the window title (top left in the above example) will contain a string similar to "[100ms 22%]". The "100ms" is the polling rate. In this example, each curve will be polled once every 100 milliseconds. The "22%" gives an estimate of how much of the communications network is being

consumed. This percentage will go up as more and more curves are being polled. A percentage higher than 100% indicates that the Oscilloscope can no longer poll all curves in each time interval. If this happens, the user should either reduce the number of curves being polled or increase the poll time interval.

The number of curves that can be polled and the rate at which they can be polled, cause a significant limitation to the Oscilloscope when running in polling mode. Many fewer curves can be polled, and much less often, than is the case if PDO messages are used. The result is that the recorded data is much less detailed.

5.3.9.2 SDO Curve

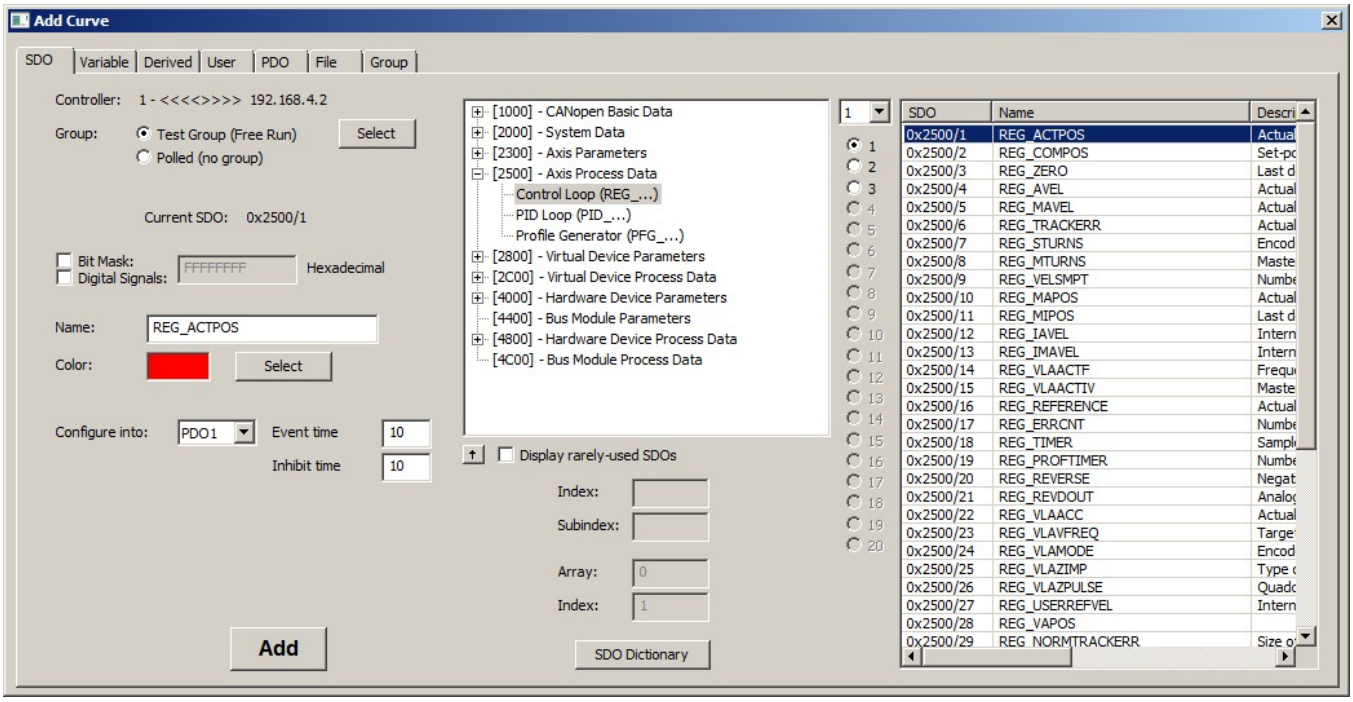
A curve can be added for any SDO that can be read from the controller.

There are three methods of recording the data for the curve: as a “polled” value, through a “Free Run” group curve, and through a “Single Shot” group curve. “Free Run” groups can only be used in the Free Run Oscilloscope and “Single Shot” groups can only be used in the Single Shot Oscilloscope. Recording the data through a group curve is the preferred method since this greatly reduces the utilization of both the controller processor and the communications network.

Polled value - When an oscilloscope run is started, the Oscilloscope will regularly “poll” the SDO by sending an “SDO read request” to the controller. Polled SDO curves will be added to the Curve Selection List under the “Other Curves” heading.

Free Run group - When an oscilloscope run is started, all SDO's in the group will be configured into a single PDO message. The controller will then automatically assemble and send the PDO message to the Oscilloscope on a regular cycle. These SDO curves will be added to the Curve Selection List under the associated group heading.

Single Shot group - When an oscilloscope run is started, all SDO's in the group will be configured into a "Record" array. When the run ends, the Oscilloscope will upload the "Record" array and extract the SDO data. These SDO curves will be added to the Curve Selection List under the associated group heading.



Controller

This is the controller that will be associated with the curve. If a polled curve is being added, then the user will be able to select a different controller for the curve. Each polled curve can be associated with a different

controller. If the curve is being added to a group, then the curve must be associated with the controller that the group is associated with.

Group

The SDO can either be polled or added to an existing group. If more than one group exists, then the user will be able to select the group to which the SDO is to be added.

SDO list

Use this list to select the SDO of interest. See Read/Write SDO for more information of how to select an SDO.

> Double-clicking on an SDO will automatically add the SDO. It is not necessary to also press the **Add** button. Also, this dialog can be resized to make more SDO information visible.

Bit Mask

If the SDO value has an internal coding (e.g. multiple values packed into a single SDO value), then the mask can be used to extract the appropriate bits. For example, a value of 000000FF will extract the low-order 8 bits from the 32-bit SDO value.

Digital Signals

If the SDO value consists of a set of 1-bit flags, then the mask can be used to select the bits to be displayed. The SDO value itself will be added as a normal curve and then an additional "Digital Signal" curve will be added for each bit in the mask. For example, a value of 01010101 will create 5 curves; the SDO curve and 4 digital signal curves (1 curve for the first bit in each byte).

Name

This is the name that will appear in the Curve Selection List. A default name will be generated but the user is free to change this to any other desired name. The default name will be formed as "Subindex - Axis: Name".

Color

This is the color that will be used to draw the curve. The Select button can be used to choose a different color.

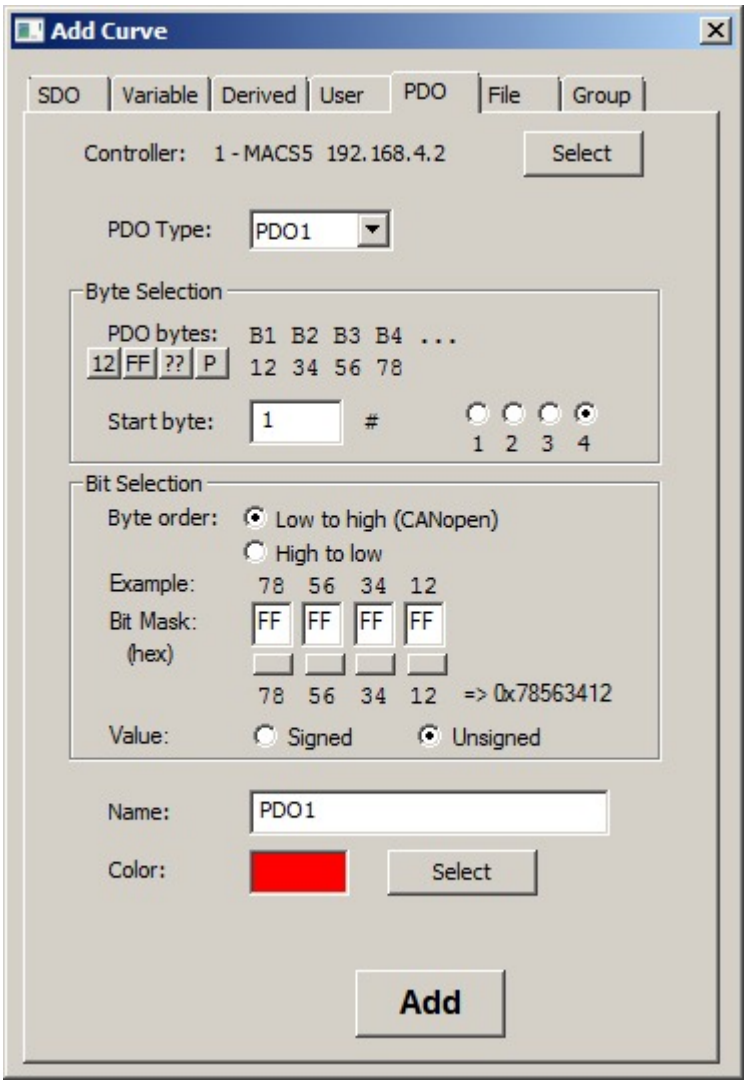
Configure into

If the SDO is being added to a Free Run group, then the user is able to select the PDO into which the SDO will be configured. The PDO event and inhibit times can also be set. (See SDO 1800 in SDO Object Dictionary Transmit PDO Parameters.)

5.3.9.3 PDO Curve

If the application running on the controller is sending PDO messages, then a PDO curve can be used to extract data from the PDO message. The data can be extracted from any byte position within the PDO.

Note that PDO curves are not polled and cannot be added to groups. Data is recorded asynchronously by the Oscilloscope whenever a PDO message is received. PDO curves will be added to the Curve Selection List under the "Other Curves" heading.



Controller

This is the controller that will be associated with the curve. The user will be able to select a different controller for the PDO. Each PDO curve can be associated with a different controller.

PDO Type

This is the type of the PDO.

Note that not all connection interface types support all PDO types and some interface types do not support any PDO messages at all.

Byte Selection

This section is used to select which bytes in the PDO message will contain the data of interest. Bytes are numbered starting at 1, which is the first data byte in the PDO message. The number of bytes required can be from 1 to 4. The "B1 B2..." string with the "12 34..." string underneath it, will give the user an indication of which bytes have currently been selected.

The four small buttons can be used to generate test PDO data that the user can then use to help verify the correct byte and bit specification.

12 - The byte string 0x12 0x34 0x56 0x78 will be used.

FF - The byte string 0xFF 0xFF 0xFF 0xFF will be used.
?? - A string of random bytes will be generated.
P - The Oscilloscope will listen for an actual incoming PDO. If found, the bytes will be extracted from the PDO.

Bit Selection

This section is used to specify how the value is to be extracted from the selected bytes. **Byte order** can be either **Low to high** (which is the standard CANopen byte order and is used for commands such as PDO and LINKSDO) or **High to low** (which should be used for low-level commands such as CANIN and CANOUT).

The **Example** bytes are exactly the test PDO bytes from the Byte Selection section above. The example bytes in this section are always displayed in the standard high-to-low byte order for a single multi-byte value.

Changing **Byte order** will also change the order of the example so that the user can tell exactly how the incoming PDO bytes will be handled.

The **Bit Mask** fields contain hexadecimal values that will be used as a mask to extract the curve data from the selected bytes. The small button under each field can be used to toggle the mask between 0xFF and 0x00. Under these buttons, the same example bytes are displayed again, this time incorporating the bit mask. And finally, the user can specify whether the value is to be treated as a signed or unsigned value.

Name

This is the name that will appear in the Curve Selection List. A default name will be generated but the user is free to change this to any other desired name.

Color

This is the color that will be used to draw the curve. The Select button can be used to choose a different color.

5.3.9.4 Variable Curve

If the application currently executing on the controller is known and has been compiled by the ApossIDE, then curves can be added to the Oscilloscope for any variable used within the program. As with SDO curves, there are three methods of recording the data for the curve:

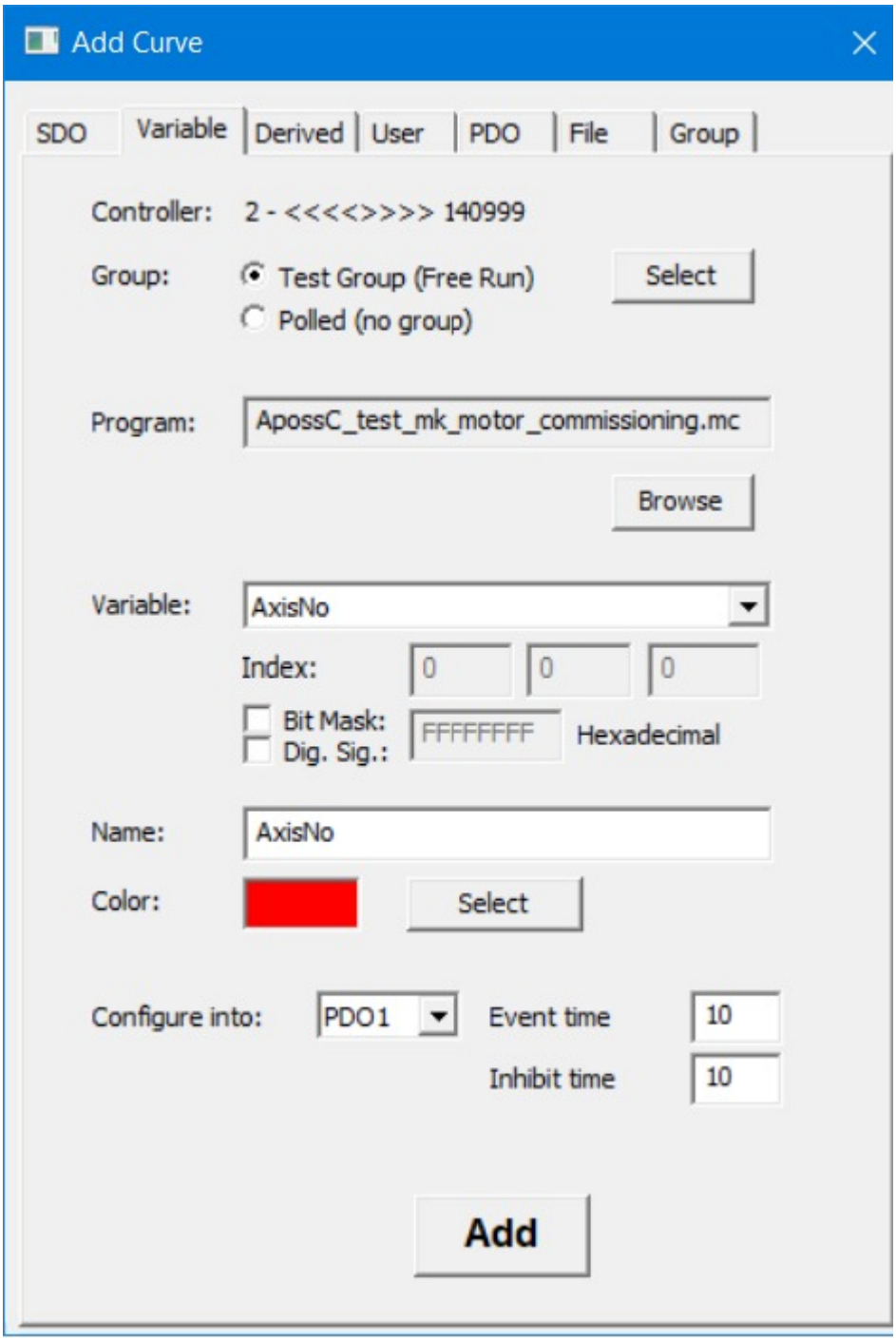
Polled value - When an oscilloscope run is started, the Oscilloscope will regularly “poll” the variable by sending an “SDO read request” to the controller. Polled variable curves will be added to the Curve Selection List under the “Other Curves” heading.

Free Run group - When an oscilloscope run is started, all variables (and SDO’s) in the group will be configured into a single PDO message. The controller will then automatically assemble and send the PDO message to the Oscilloscope on a regular cycle. These variable curves will be added to the Curve Selection List under the associated group heading.

Single Shot group - When an oscilloscope run is started, all variables (and SDO's) in the group will be configured into a "Record" array. When the run ends, the Oscilloscope will upload the "Record" array and extract the variable data. These variable curves will be added to the Curve Selection List under the associated group heading.

The Oscilloscope cannot verify that the program running on the controller is identical to the program selected by the user when adding a **Variable curve**. It is the user's responsibility to ensure that this is true. A mismatch

will not cause the Oscilloscope to fail but it also will not generate the correct results for the curve.



Controller

This is the controller that will be associated with the curve. If a polled curve is being added, then the user will be able to select a different controller for the curve. Each polled curve can be associated with a different controller. If the curve is being added to a group, then the curve must be associated with the controller that the group is associated with.

Group

The variable can either be polled or added to an existing group. If more than one group exists, then the user will be able to select the group to which the variable is to be added.

Program

This is the program containing the variable of interest. *It is the responsibility of the user to ensure that this is the program that is executing on the controller*.

The Browse button can be used to select a different program file.

Variable

This will be a list of variables found in the program. Use this list to select the variable of interest. If an array (at most three dimensional) is selected, then **Index** must be used to specify exactly which array element is to be recorded by the curve.

If the variable value has an internal coding (e.g. multiple values packed into a single value), then the **Bit Mask** can be used to extract the appropriate bits. For example, a value of 000000FF will extract the low-order 8 bits from a 32-bit variable value.

> **Bit Mask** cannot be used with "double" variables.

Name

This is the name that will appear in the Curve Selection List. A default name will be generated but the user is free to change this to any other desired name.

Color

This is the color that will be used to draw the curve. The Select button can be used to choose a different color.

Configure into

If the variable is being added to a Free Run group, then the user is able to select the PDO into which the variable will be configured. The PDO event and inhibit times can also be set. (See SDO 1800 in SDO Object Dictionary Transmit PDO Parameters.)

5.3.9.5 Derived Curve

If other curves have already been added, then curves can be added that are derived from those curves. The derived curve will always be added to the Curve Selection List after the curve from which it is derived.

The following forms of derived curves can be added:

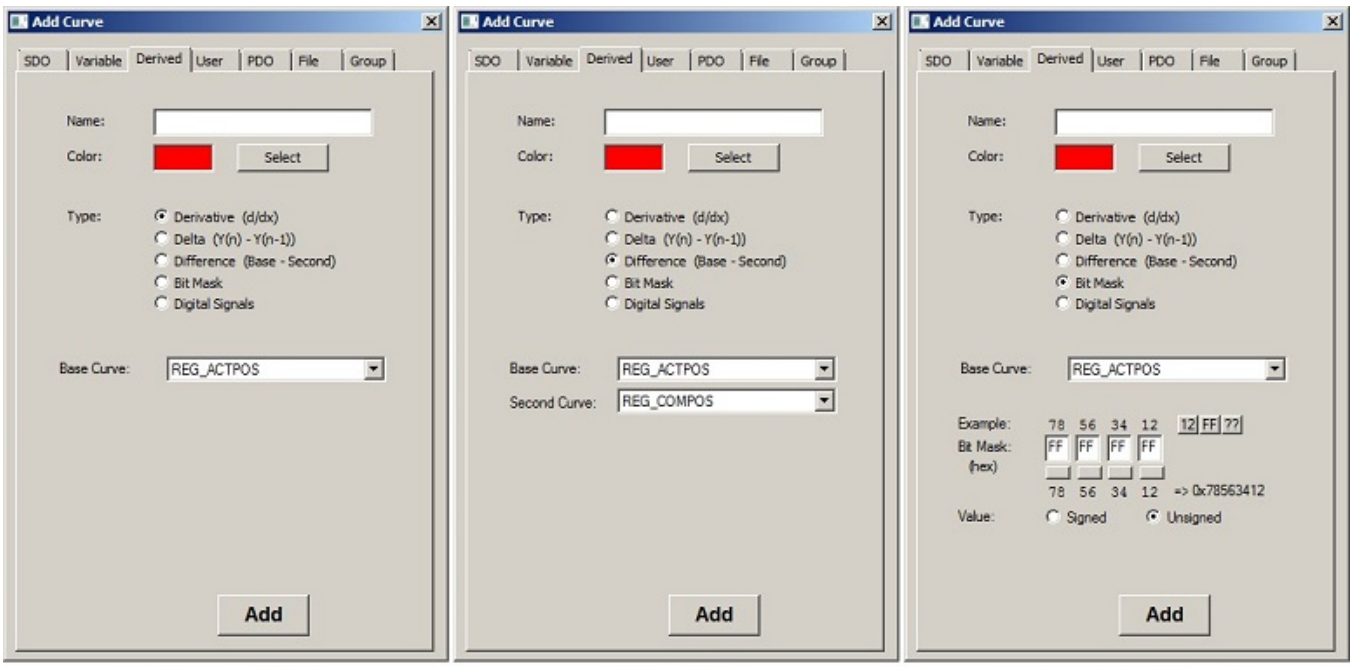
Derivative - This is the mathematical derivative (i.e. d/dx) of the original curve.

Delta - A "delta" curve will show the incremental differences along the original curve. Each point on the curve will be the difference between the corresponding point on the original curve and the previous point on the original curve.

Difference - A "difference" curve will be the simple difference (i.e. subtraction) between the values of two original curves.

Bit Mask - If a curve value has an internal coding (e.g. multiple values packed into a single value), then this curve will derive its data from bits extracted from the values of the original curve. For example, a mask of 000000FF will extract the low-order 8 bits from a 32-bit value.

Digital Signals - If a curve value actually contains a set of single bit digital signals, then selecting “Digital Signals” and setting a bit mask will cause one digital signal curve to be added for each bit in the bit mask.



Derivative / Delta | Difference | Bit Mask / Digital Signals

Name

This is the name that will appear in the Curve Selection List. The user should set this to some appropriate name.

Color

This is the color that will be used to draw the curve. Use **Select** to choose a different color.

Type

This is the type of derived curve to be added.

Base Curve(s)

Base Curve is the curve from which the new curve will be derived.

Second Curve - If the new curve is derived from two curves (e.g. a “difference” curve), then this will be the second curve.

Align Curves

When adding a "difference" curve, it is often the case where the starting times of the two curves are not the same. For example, this will happen if two curves from two different tests are being compared. Enabling this option will cause the start time of the second curve to be shifted so that it aligns with the first curve.

Bit Mask

This is used to specify how a Bit Mask value is to be extracted from the original value (which is always a 32-bit value). The Example bytes can be used by the user to help verify the correct bit mask

specification. The three small buttons can be used to generate different test values for the example.

12 The value 0x78563412 will be used.
FF The value 0xFFFFFFFF will be used.
?? A random value will be generated.

The **Bit Mask** fields contain hexadecimal values that will be used as a mask to extract the curve data from the selected bytes. The small button under each field can be used to toggle the mask between 0xFF and 0x00. Under these buttons, the same example bytes are displayed again, this time incorporating the bit mask.

Value

Finally, the user can specify whether the value is to be treated as a signed or unsigned value.

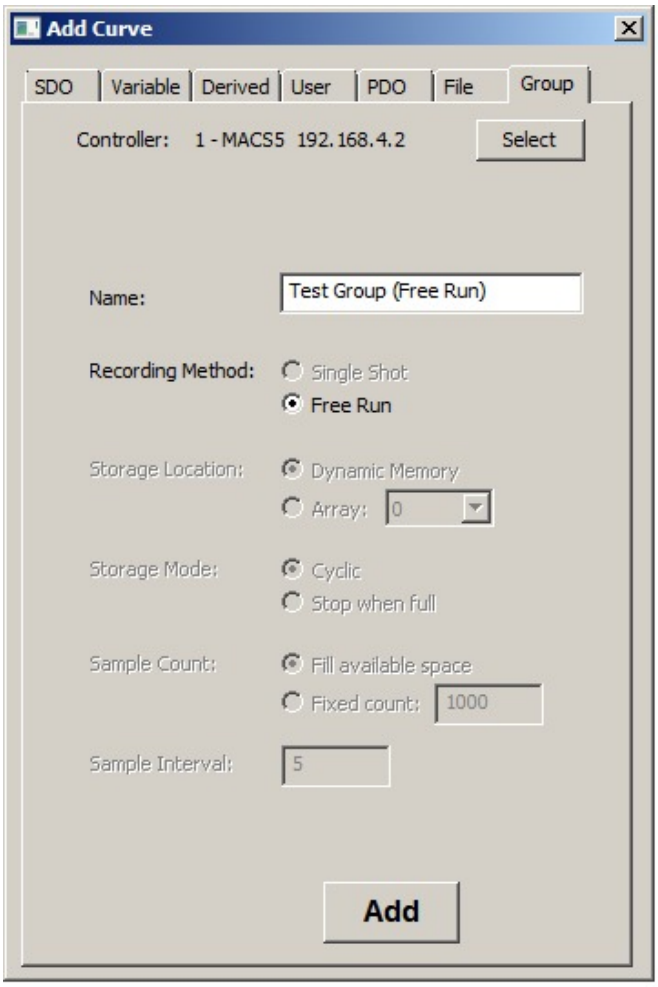
5.3.9.6 Group Curve

A Group curve is not an actual curve and it contains no recorded information. Instead, it simply contains a collection of other curves and allows that entire set of curves to be handled together as a unit. For example, all the curves in a group will be configured by the Free Run Oscilloscope into a single PDO message for transferring values from the controller to the Oscilloscope. Or all the curves in a group will be configured by the Single Shot Oscilloscope into a single "Record" array for data collection.

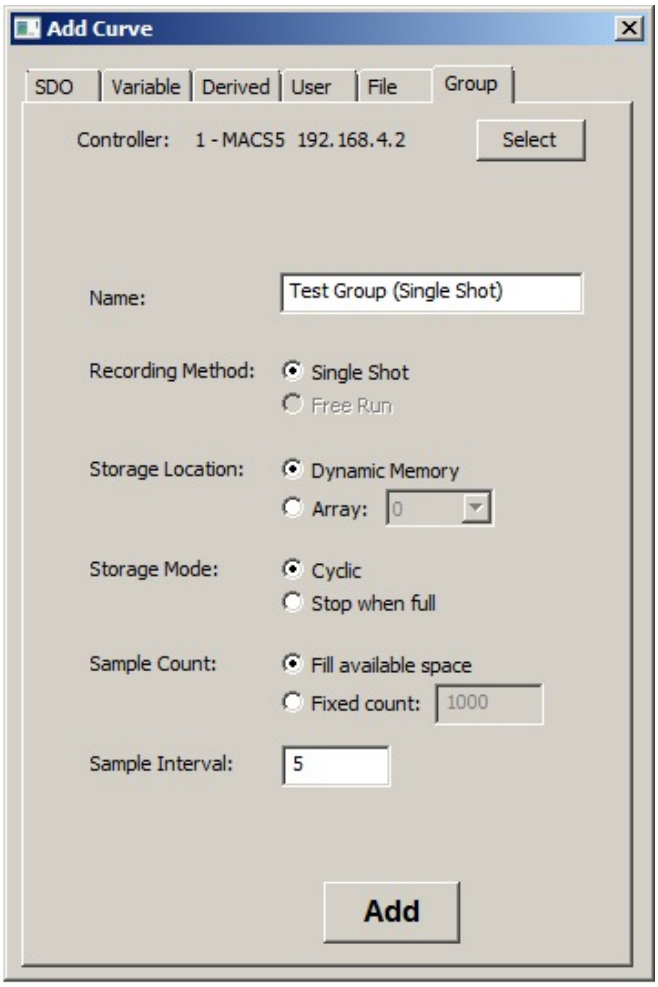
> One consequence of this is that all curves in a group must be associated with the same controller.

The preferred method of using the Oscilloscope is to add a Group curve and then add other curves to that group. A single Group curve is adequate for most uses of the Oscilloscope. However, in some cases (e.g. testing with multiple controllers), adding multiple Group curves can be useful.

Group curves will be displayed in the Curve Selection List with a grey background. Groups will always be added to the list after all previously existing groups and before the “Other Curves” heading if it exists.



Free Run Group



Single Shot Group

Name

This is the name that will appear in the Curve Selection List. A default name will be given but the user should set this to some appropriate name.

Recording Method

If the Free Run Oscilloscope is being used, then this will be set to **Free Run**. In this case, sample data is transferred from the controller to the Oscilloscope in real time using PDO messages. If the Single Shot Oscilloscope is being used, then this will be set to **Single Shot**. In this case, sample data is recorded in a "Record" array and transferred to the Oscilloscope at the end of the test.

All the following attributes apply only to **Single Shot Groups**. They affect various aspects of the "Record" array that is configured to record the curve data during the test.

Storage Location

This specifies where the "Record" array will be placed. If set to **Dynamic Memory**, then the "Record" array will be saved in the remaining unused memory in the controller. Note that the amount of unused memory will vary depending on such things as the size of the loaded program, the number and sizes of any pre-allocated user arrays, the number of saved programs, etc. If **Array** is selected, then the "Record" array will be placed in the specified array. This array must be pre-allocated in the

controller (e.g. by defining it in an application program or by using the Array Editor, etc.).

Storage Mode

This specifies what happens when the "Record" array becomes full.

If set to **Cyclic**, then the oldest existing data point will be lost in order to make room for the newest data point (i.e. the "Record" array will contain only the *most recent* set of points recorded).

If set to **Stop when full**, then new data points are discarded (i.e. the "Record" array will contain only the first and oldest set of points recorded).

Sample Count

This specifies the number of sets of sample points that can be saved (i.e. it determines the size of the "Record" array).

Note that it refers to the number of *sets* of points (i.e. one sample point for each curve in the group) and not to the total number of individual sample points.

If set to **Fill available space**, then the "Record" array will be made as large as possible while still fitting within either the dynamic memory or the predefined user array. Alternatively, **Fixed count** can be used if the user wants a specific number of points only.

Sample Interval

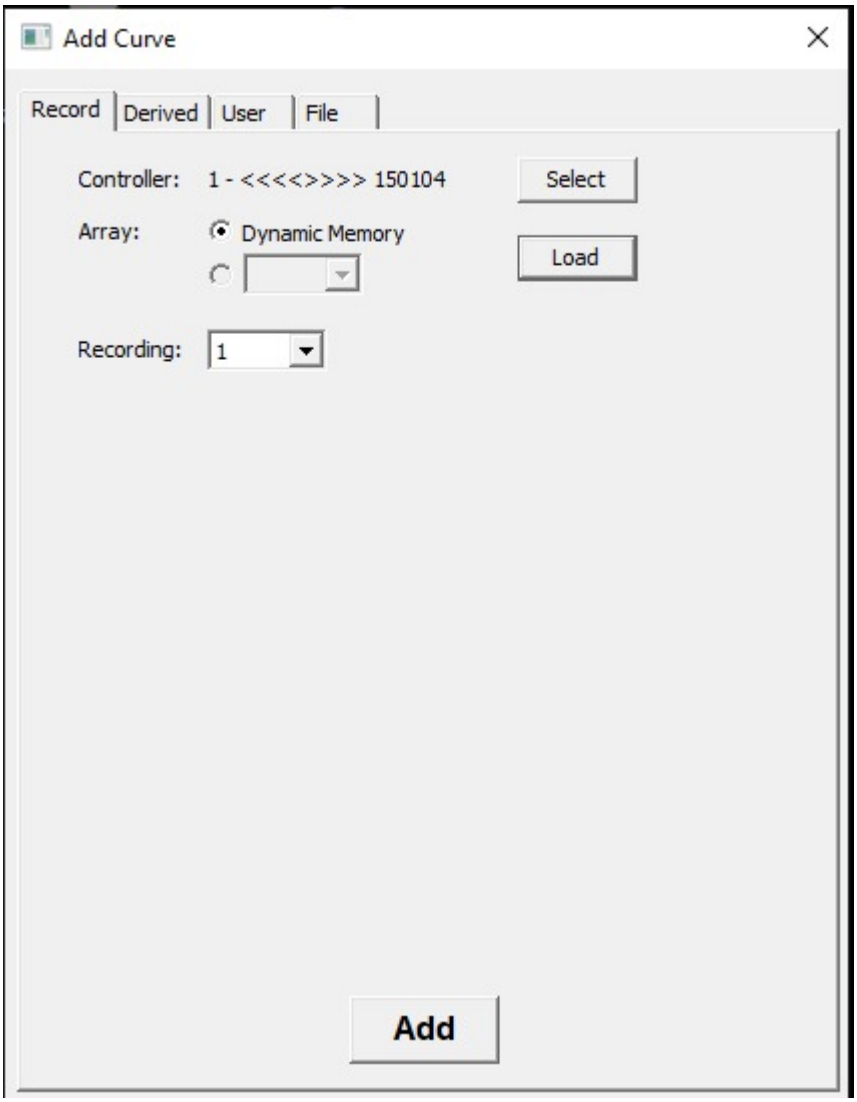
This specifies the time interval, in milliseconds, between each saved set of sample points.

5.3.9.7 Record Curve

If a user configures a custom "Record" array in an application program (i.e. rather than using the Single Shot Oscilloscope to configure the array automatically) then a "Record" curve must be added to the Oscilloscope to store the data. This will be necessary, for example, with older controllers that do not support the automatic configuration of "Record" arrays. When the "Record" curve is added, the Oscilloscope will immediately try to read the "Record" array from the controller. Hence, the test must already have been run and completed and the "Record" array created on the controller *before* the "Record" curve is added to the Oscilloscope.

When a "Record" curve is added, the Oscilloscope will actually add a separate curve for each curve defined in the "Record" array. The first of these curves will always be drawn with a grey background in the Curve Selection List.

"Record" curves can only be added when the "Record" Oscilloscope is being used.



Controller

This is the controller containing the "Record" array. The **Select** button can be used to choose a different controller.

Array

The array can be loaded from either dynamic memory or from a predefined user array. Once the array location has been specified, then the **Load** button must be pressed. This will cause the Oscilloscope to upload the "Record" array from the controller.

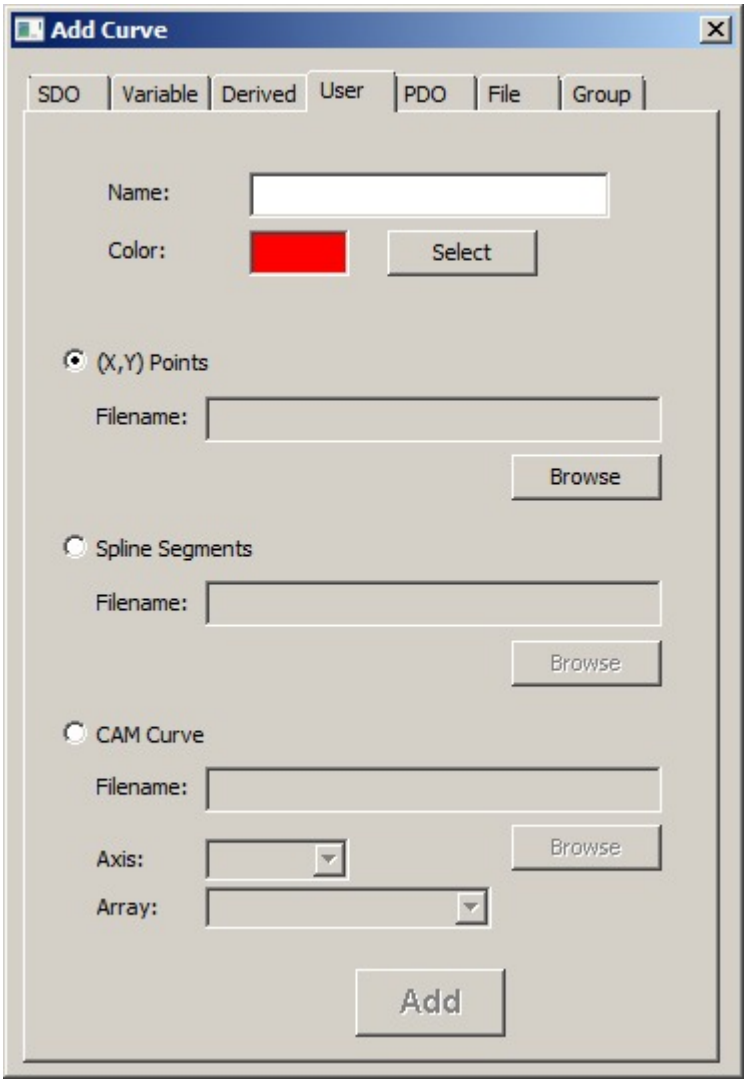
> **Dynamic Memory** will be disabled if the controller does not support dynamic memory or if dynamic memory does *not contain* a "Record" array.

Recording

If the "Record" array contains multiple recordings, then one of the recordings must be chosen. More than one recording can be added to the Oscilloscope but a separate "Record" curve must be added for each one.

5.3.9.8 User Curves

In some cases, it is useful to be able to compare a theoretical “model” or “prototype” curve with an actual curve that has been recorded from the controller. User curves allow the theoretical curve to be defined. The shift function (see “Shift-Left button” in “Using the Mouse in the Chart Window”) or difference derived curves (see “Derived Curve”) can then be used to do the actual comparison.



Name

This is the name that will appear in the Curve Selection List. The user should set this to some appropriate name.

Color

This is the color that will be used to draw the curve. Use **Select** to choose a different color.

(X,Y) Points

Use this selection if the theoretical curve is defined by a series of (X,Y) points. Use **Browse** to select the file containing the points. The points should be defined in the file as space-separated or comma-separated pairs of numbers. The points must be ordered by increasing X values.

Spline Segments

Use this selection if the theoretical curve is defined by a spline. Use **Browse** to select the file containing the spline. The spline segments should be defined in the file as space-separated or comma-separated sets of numbers. Each set must consist of 6 numbers defining a 5th order spline segment:

- the X coordinate of the start of the segment,
- the 0th degree coefficient of the spline function,
- the 1st degree coefficient, the 2nd degree coefficient, etc.

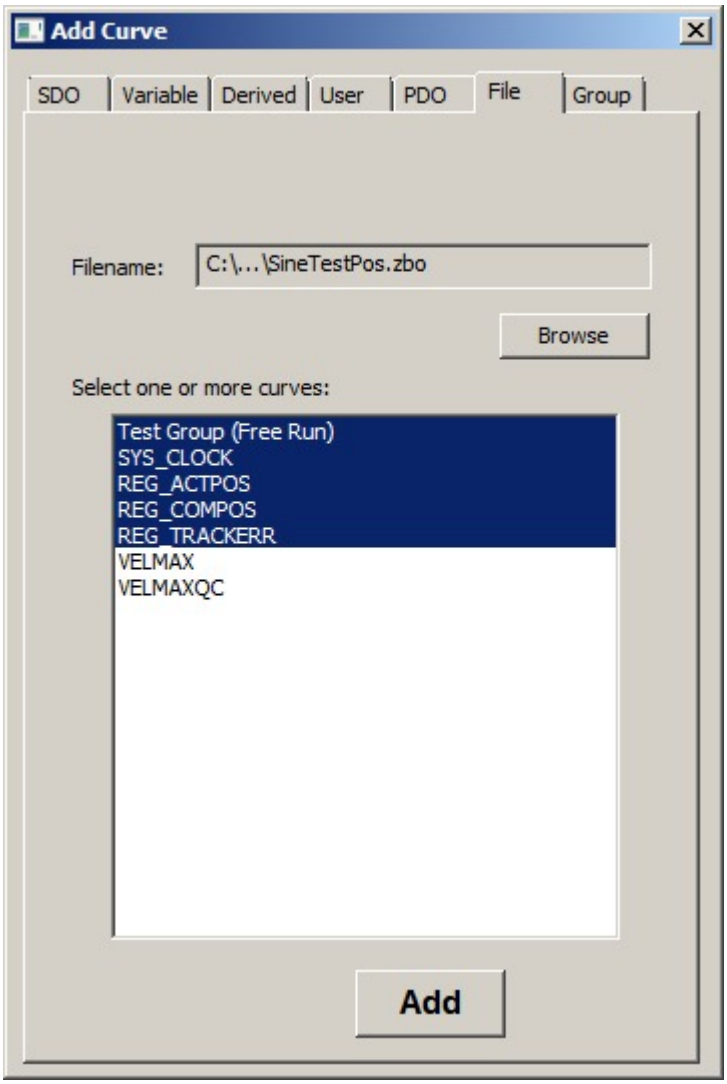
The segments must be ordered by increasing X values.

CAM Curve

Use this selection if the theoretical curve is defined by a CAM curve. Use **Browse** to select the file containing the curve. Note that CAM curves are axis-dependent so a specific axis will need to be selected of the controller supports multiple axes.

5.3.9.9 Adding Curves from other Oscilloscope Files

Curves defined in other Oscilloscope ".zbo" files can be copied into the current Oscilloscope by selecting the File tab.



Filename

This is the Oscilloscope file containing the curves that will be copied. The **Browse** button can be used to select the Oscilloscope file.

Curve list

This is a list of the curves found in the selected Oscilloscope file.

Select the curve to be copied by clicking on it with the left mouse button. Multiple curves can be selected simply by clicking on the curves. Some curves rely on the presence of other curves and cannot be copied as stand-alone curves. For example, a derivative curve cannot be copied without also copying the curve from which it is derived. Similarly, all curves in a group must be copied together. If the user selects one of these curves, then the other required curves will be automatically selected as well.

> The list will contain only those curves in the selected Oscilloscope file that are compatible with the current Oscilloscope. For example, if the selected Oscilloscope file contains Single Shot groups, then these groups will not be in the list if the current Oscilloscope is a Free Run Oscilloscope.

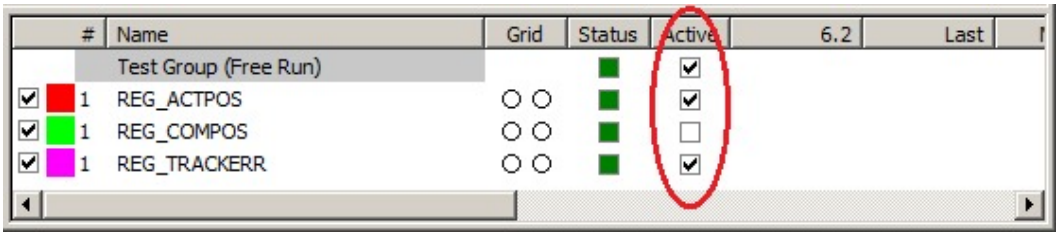
5.3.10 Starting and Ending the Test

Once all the curves have been added, then the test itself can be started. This involves three steps:

1. Selecting which curves are to be recorded.
2. Starting the application program running on the controller.
3. Starting the Oscilloscope recording.

Selecting which curves are to be recorded

The user can define many curves in the Oscilloscope but it is not necessary to record data for all curves during each test. Use the checkbox in the "Active" column in the Curve Selection List to select which curves will be recorded. In the example below, data will be recorded only for the "REG_ACTPOS" (actual position) and "REG_TRACKERR" (tracking error) curves. Data for "REG_COMPOS" will not be recorded.

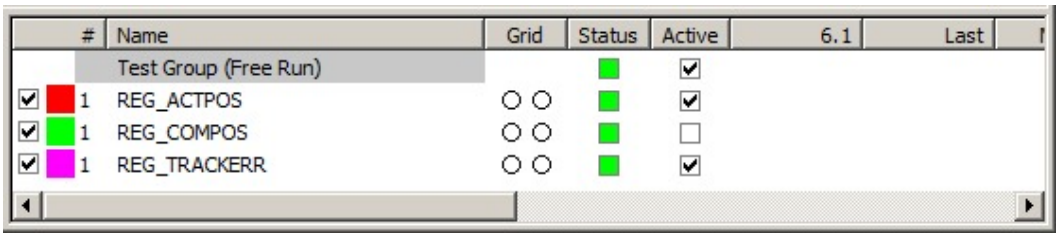


Note that groups as a whole can be selected or deselected. If selected, then data for the selected curves within that group will be recorded. If deselected, then the checkboxes for all curves within that group will be grey and no data will be recorded for any of those curves.

When multiple groups have been defined, then there is a restriction as to which groups may be selected at the same time. No two groups may be selected at the same time if they are associated with the same controller *and* use the same PDO message. If groups have been defined in this way, then selecting one group will automatically cause the other group to be deselected.

Starting the application program

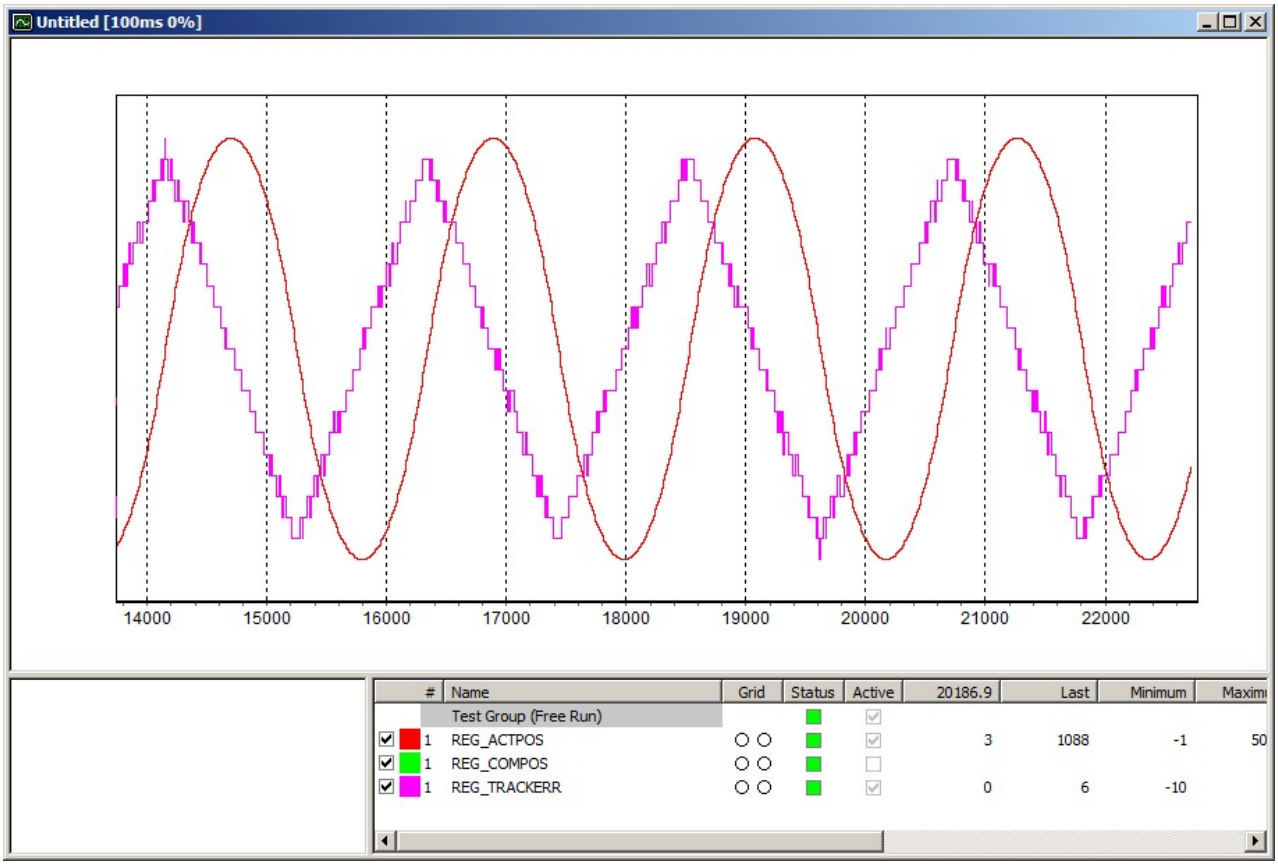
The next step is to start the application program running on the controller if it is not already running. Do this by switching to the original ApossIDE window and starting the application in any of the normal ways. Then switch back to the Oscilloscope Window. The Curve Selection List should now look something like that below. Note that the bright green Status column indicates that the controller is now executing a program.



Starting the Oscilloscope recording

The final step is to start the Oscilloscope recording. This is done used the **Start** toolbar button.

At this point, curve data should start to be recorded and displayed in the Chart Window. This will be done in real time as the test progresses (for the Free Run Oscilloscope) or after the end of the test (for the Single Shot Oscilloscope). The Oscilloscope Window should now look similar to that shown below.



The order in which the test program execution is started and the Oscilloscope recording is started, is not fixed. The user is free to start the program execution and then the recording or start the recording and then the program execution, whichever is appropriate.

At any time during the recording, any of the **Zoom** toolbar buttons can be used, any of the curves can be hidden or shown in the Chart Window using the checkbox in the Name column, and the Chart Y-axis can be set using the Grid column. Note that the Navigation window is not drawn and cannot be used while recording is in progress.

Each curve has a fixed maximum number of points that can be recorded. Once this number has been reached, then the oldest points are discarded to make room for new points. The user can change this maximum number in the **Properties** → **Source** window.

The recording will continue until the user stops it by pressing the **Stop** toolbar button.

> This does *not* abort the executing program; the executing program will continue to run. The **Stop** button only stops the recording of new data.

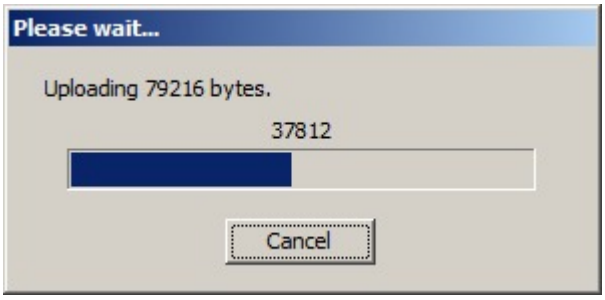
> The user can abort the executing program at any time simply by switching to the Editor Window and pressing the **Esc** key to break the program. Aborting the program does *not* stop the Oscilloscope from recording. The Oscilloscope will continue to listen for incoming PDO messages until the **Stop** toolbar button is pressed.

5.3.10.1 Usage Notes for the Single Shot Oscilloscope

If multiple groups are used, then only one group per controller can be selected for recording. If a second group is selected for the same controller, then the first group will be automatically deselected.

Start the recording as described above with the **Start** toolbar button. However, while data is being recorded, no curves will be displayed in the Chart Window. Once the recording is stopped, then a window similar to the

following will be displayed as the "Record" array is uploaded to the Oscilloscope. The curves will be displayed in the Chart Window once the upload completes.



5.3.11 Analyzing the Results

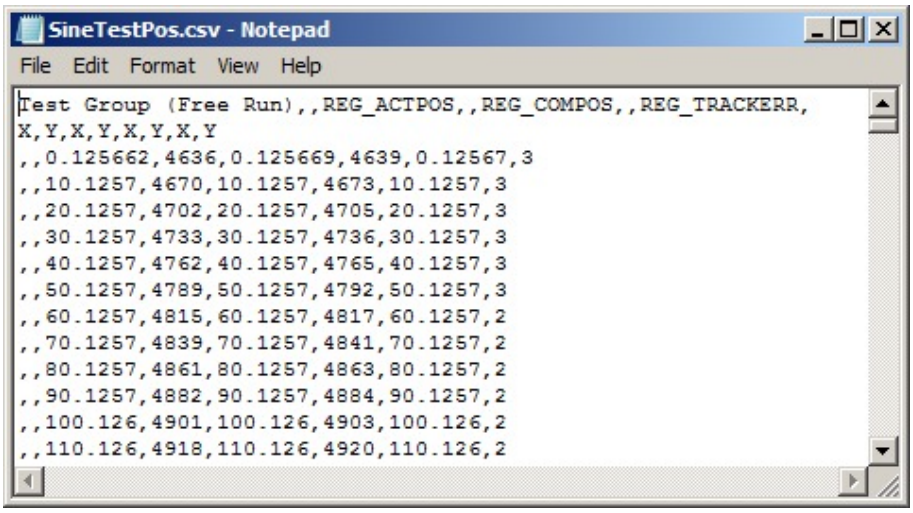
Any of the following functions and features of the Oscilloscope can be used to help analyze the recorded data:

1. The curves in the Chart Window may show correlations or anomalies. Use the **Zoom** toolbar buttons and the Navigation Window to display more detailed portions of the curves.
2. The Last, Minimum, and Maximum columns in the Curve Selection List may provide useful information.
3. Use the mouse and the Cursors to scan back and forth across the curves. Curve values will be displayed in the Curve Selection List.
4. Use the **Properties** → **Display** window to enable **range validation** for curves. This may make it easier to locate "out-of-bounds" problems.
5. Create "**derived**" Curves. For example, a derived curve can be added that shows the difference between two curves or the rate at which a curve is changing.

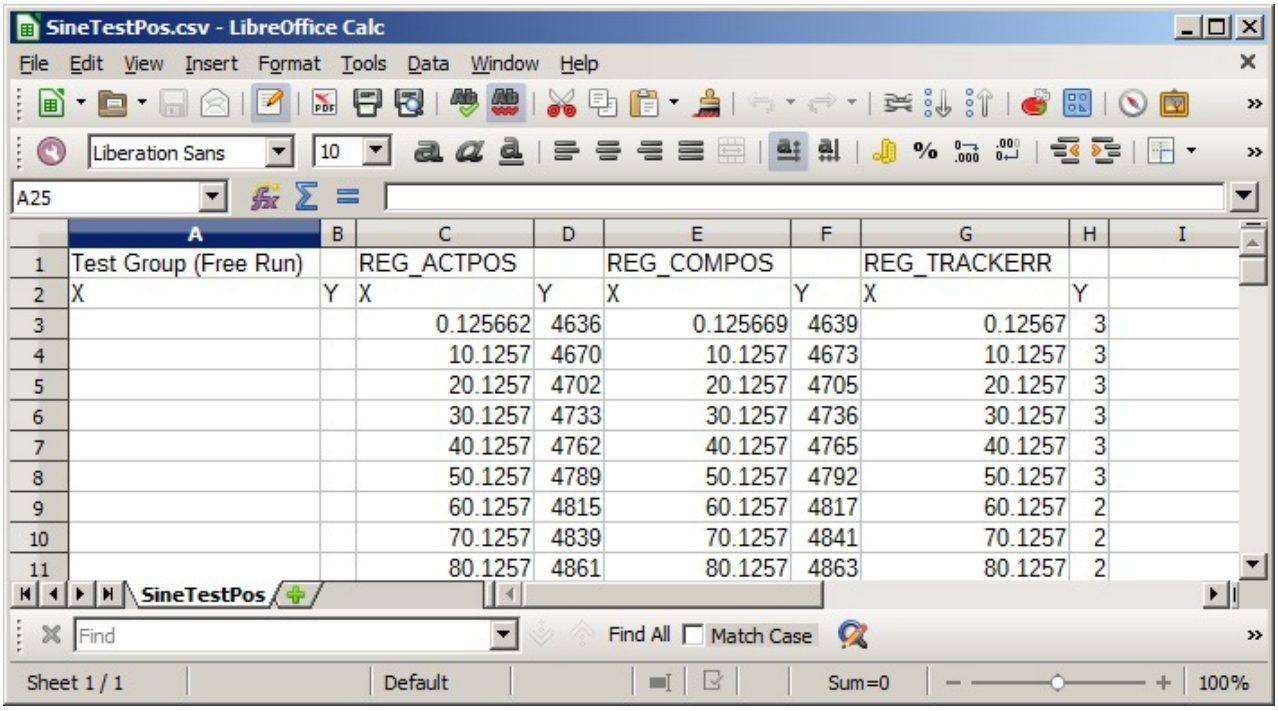
Export Curve Data into a txt file

Use the **Export** toolbar button to export the curve data to a standard text file. This file can then be used by other programs to analyze the curve.

This will export all the curve data to a "comma-separated value" (.csv) file. This is a standard text file that can be imported into spreadsheet programs and opened with normal text editors. The file will appear similar to the following example:



Once imported into a spreadsheet application such as LibreOffice, it will appear similar to this:



	A	B	C	D	E	F	G	H	I
1	Test Group (Free Run)		REG_ACTPOS		REG_COMPOS		REG_TRACKERR		
2	X	Y	X	Y	X	Y	X	Y	
3			0.125662	4636	0.125669	4639	0.12567	3	
4			10.1257	4670	10.1257	4673	10.1257	3	
5			20.1257	4702	20.1257	4705	20.1257	3	
6			30.1257	4733	30.1257	4736	30.1257	3	
7			40.1257	4762	40.1257	4765	40.1257	3	
8			50.1257	4789	50.1257	4792	50.1257	3	
9			60.1257	4815	60.1257	4817	60.1257	2	
10			70.1257	4839	70.1257	4841	70.1257	2	
11			80.1257	4861	80.1257	4863	80.1257	2	

5.3.12 Repeating the Test

Once Oscilloscope recording has been stopped, then the user is free to make any necessary changes, additions, and deletions to the curves. Recording can be restarted simply by pressing the **Start** button again.

> This will delete any previously recorded data. Previously recorded data will not be deleted if the **Pause** toolbar button is used instead to the **Stop** button. In this case, a vertical red bar is drawn in the chart at the position where the recording was paused.

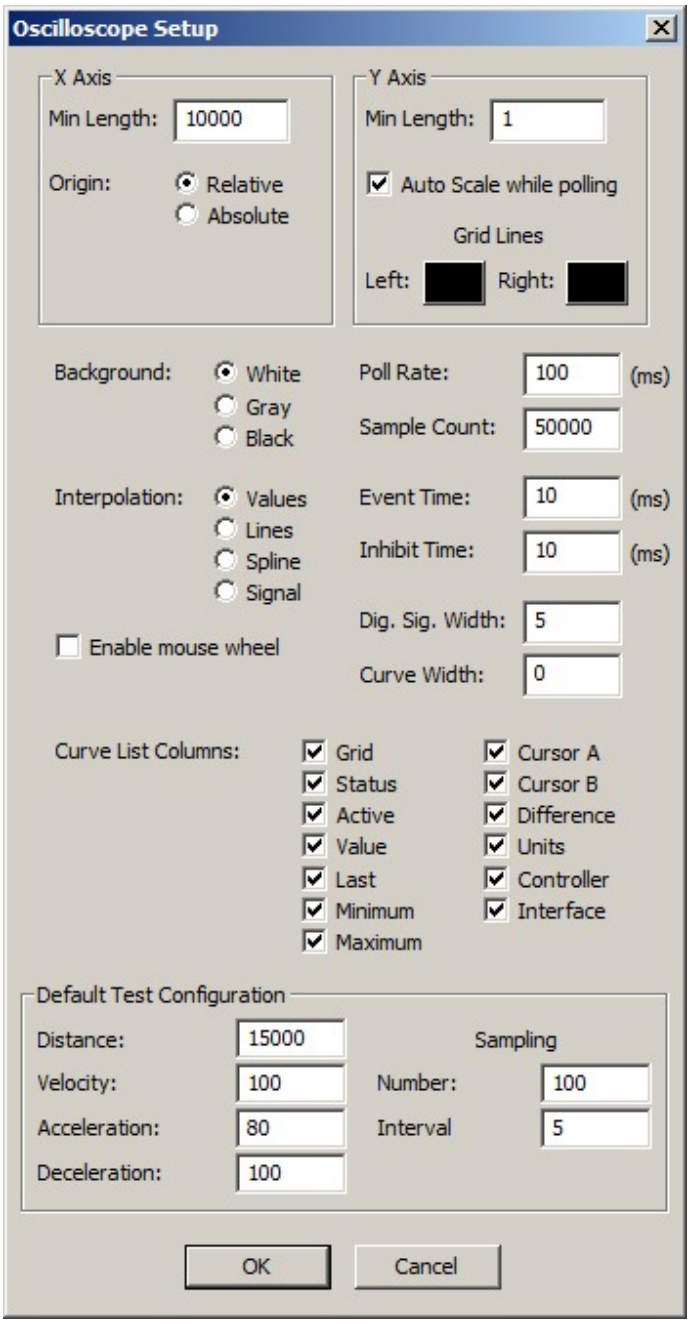
5.3.12.1 Usage Notes for the Record Oscilloscope

Repeating a Record test is done entirely by the user outside the Record Oscilloscope. Once completed, then the user has three options:

1. Open an entirely new Record Oscilloscope.
2. If the previous values are no longer needed, then the existing curves can be “reloaded”. Do this by right-clicking on the curve name in the Curve Selection List and selecting the **Reload** or **Reload & Redisplay** command. The old values will be discarded and the new values will be loaded.
3. Add new Record curves using the **Add curve** toolbar button. This has the advantage that it allows the new curves to be compared with the old curves. This can be done either visually or by adding derived curves.

5.3.13 Oscilloscope Settings

Various global Oscilloscope attributes and defaults can be set using the **Settings** → **Oscilloscope** menu command.



X axis

Min Length specifies the minimum length of the X-axis in the Chart Window when the Freerun Oscilloscope is being used. This value is used when the chart is empty. For other types of Oscilloscopes, the X-axis length is based on the range required by the curves being displayed.

The **Origin** setting determines the initial state of the **Shift to 0** toggle button. If set to **Relative**, then by default the X-axis (time) will be shifted so that curves start at time 0. If set to **Absolute**, then the X-axis is not shifted. This setting will affect all curves.

Y axis

Min Length specifies the default minimum length to be used for the Y-axis when new curves are being added. After a curve has been added, then this value can be changed by setting the curve **Properties** → **Axis** in the

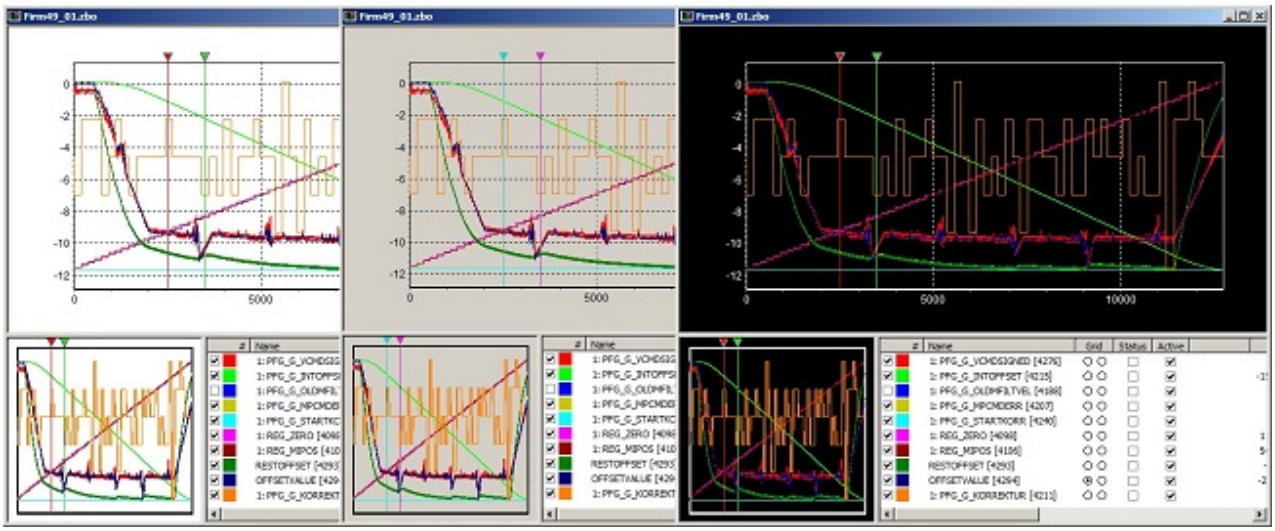
Curve Selection List.

If **Auto Scale while polling** is enabled then the Y-axis used in the Chart Window will be recalculated while an Oscilloscope "run" is active. This prevents curves from being drawn "outside" the Chart Window if Y values go outside the range of the current Y-axis. If this happens, then the Y-axis of the Chart Window will be scaled so that the curve always remains entirely visible in the Chart Window.

The color of the horizontal grid lines can be set using the colored **Left** and **Right** buttons. In some cases, changing the grid line color can help to better distinguish the grid lines on the chart.

Background

This selects the background color of the Chart and Navigation Windows. The different backgrounds are shown below.



Interpolation

This is the default display style to be used when new curves are added. After a curve has been added, its display style can be changed using **Properties** → **Display** in the Curve Selection List.

Poll Rate

The poll rate specifies how often the Oscilloscope will try to read data for polled curves from the controller when an Oscilloscope "run" start been started. Care should be taken not to set this time shorter than necessary as it can significantly increase demands on the PC processor, the controller processor, and the communication network.

Sample Count

This is the default sample count to be used when new curves are added. The sample count is the maximum number of points that the Oscilloscope will save for a curve before old points are discarded to make room for new points. See also **Properties** → **Source**.

Event Time / Inhibit Time

This is the default PDO event time and the default PDO inhibit time to be used when new curves are being added.

After a curve has been added, these values can be changed by using **Properties** → **Source** in the Curve Selection List.

Dig. Sig. Width

If any curves are being displayed using the digital signal style, then this value specifies the width (thickness) of the displayed curves.

Curve Width

This values specifies the width (thickness) of displayed curves. If it is set to 0, then a thin line is used.

Enable mouse wheel

The mouse wheel can be used to shift a selected curve up or down. This option allows this function to be enabled or disabled (see Using the Mouse in the Chart Window).

Curve List Columns

These boxes allow the user to select which data columns will be displayed in the Curve Selection List.

Default Test Configuration

This set of parameters specifies the default values to be used by the Tune Oscilloscope.

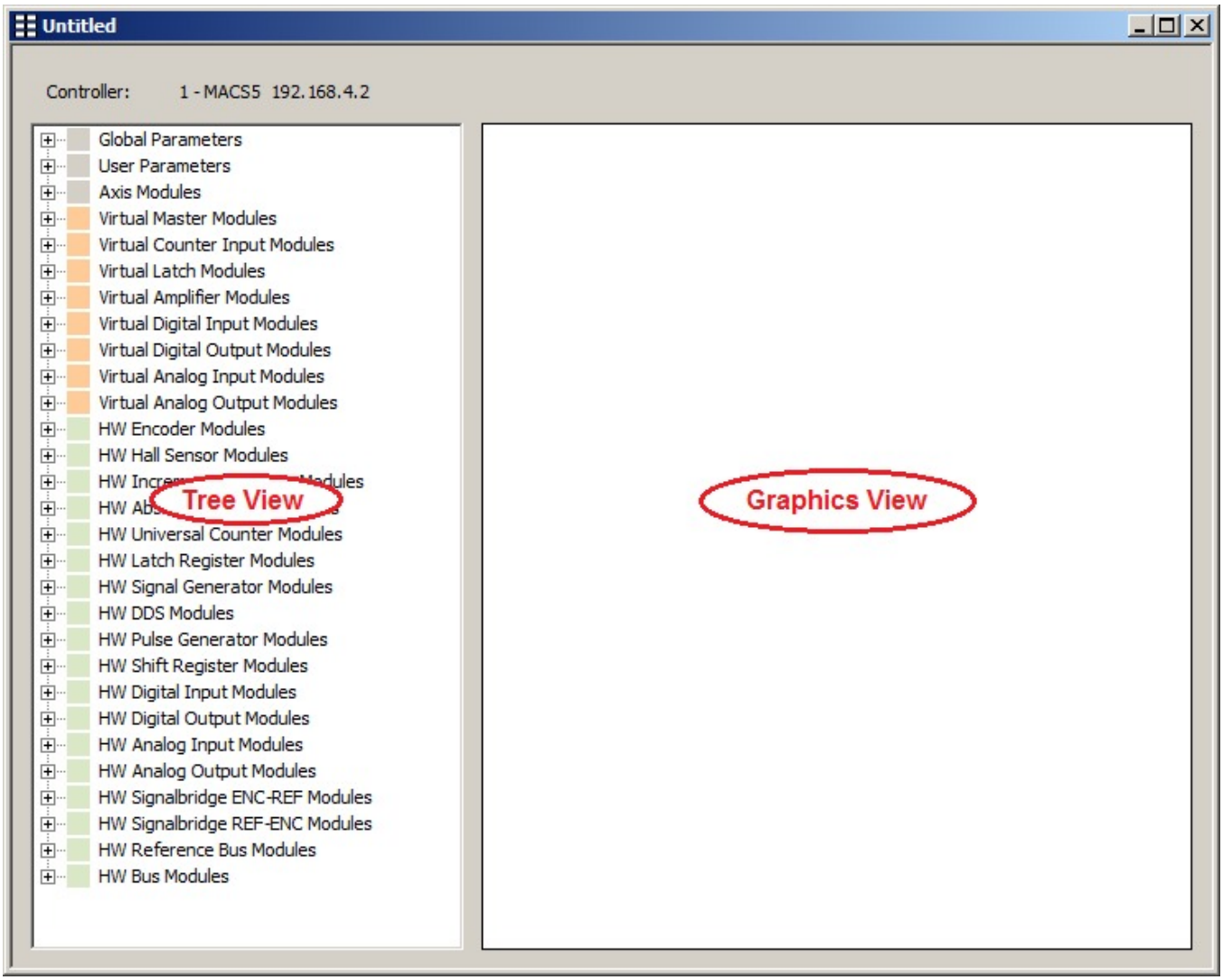
5.4 Module Editor

The Module Editor is a tool allowing users to configure and debug Function Modules (see System Architecture). Function Modules provide application programs with access to hardware and/or software functions such as encoders, counters, latches, pulse generators, etc. Function Modules are particularly important in MACS5 and later controllers.

When the Module Editor is started, it will upload the configuration from the currently connected controller. Two options are available:

1. **Upload CURRENT configuation** will upload the configuration values as they currently exist on the controller. This can be done while an application program is executing.
2. **Upload INITIAL configuration** will upload the default configuration values. These are the values to which the configuration is set when an application program is initially started. Note that these values cannot be uploaded while an application program is running.

After the configuration is uploaded, a Module Editor window similar to the following will be opened. The window on the left is the Module Tree View and will show all modules found on the controller. The window on the right is the Module Graphics View. The graphics view will show all modules that are currently "open" for viewing. Modules are opened using the Tree View window.

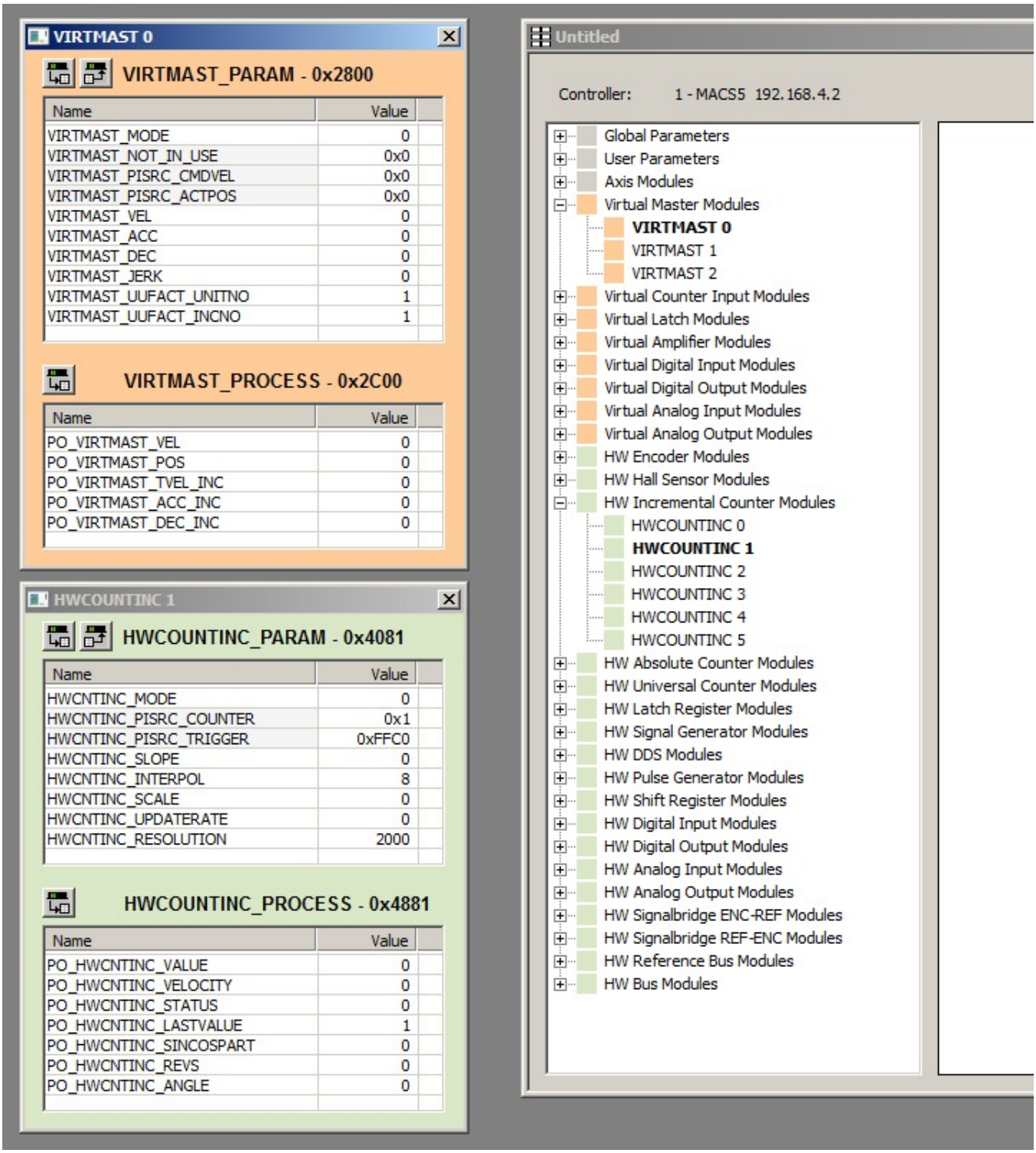


5.4.1 Module Tree View

The Module Tree View shows all the function modules available on the controller. The color patches are used to identify the types of the modules as follows:

- Virtual module.
- Hardware module.
- All other modules.

Double-clicking on a module in the Tree View will "open" that module. When a module is opened, then a Module Window for that module will be opened (e.g. "VIRTMAST 0" and "HWCOUNTINC 1" below left) and an icon will be added to the Module Graphics View. Module names in the Tree View will be displayed using bold text for modules that are open. The Module Window shows all the configuration parameters available for that module. As an aid to identifying the module type, the background color of the window will match the color patch in the Tree View. If a module is already open, then double-clicking on it in the Tree View will switch the Windows focus to that Module Window.



As with other ApossIDE windows, the Module Editor window may be resized and moved anywhere within the main ApossIDE window. Individual Module Windows may also be resized, but they may be moved anywhere on the computer monitor. They are not restricted to the ApossIDE window. This allows the user freedom to create a window layout that is more natural to work with.

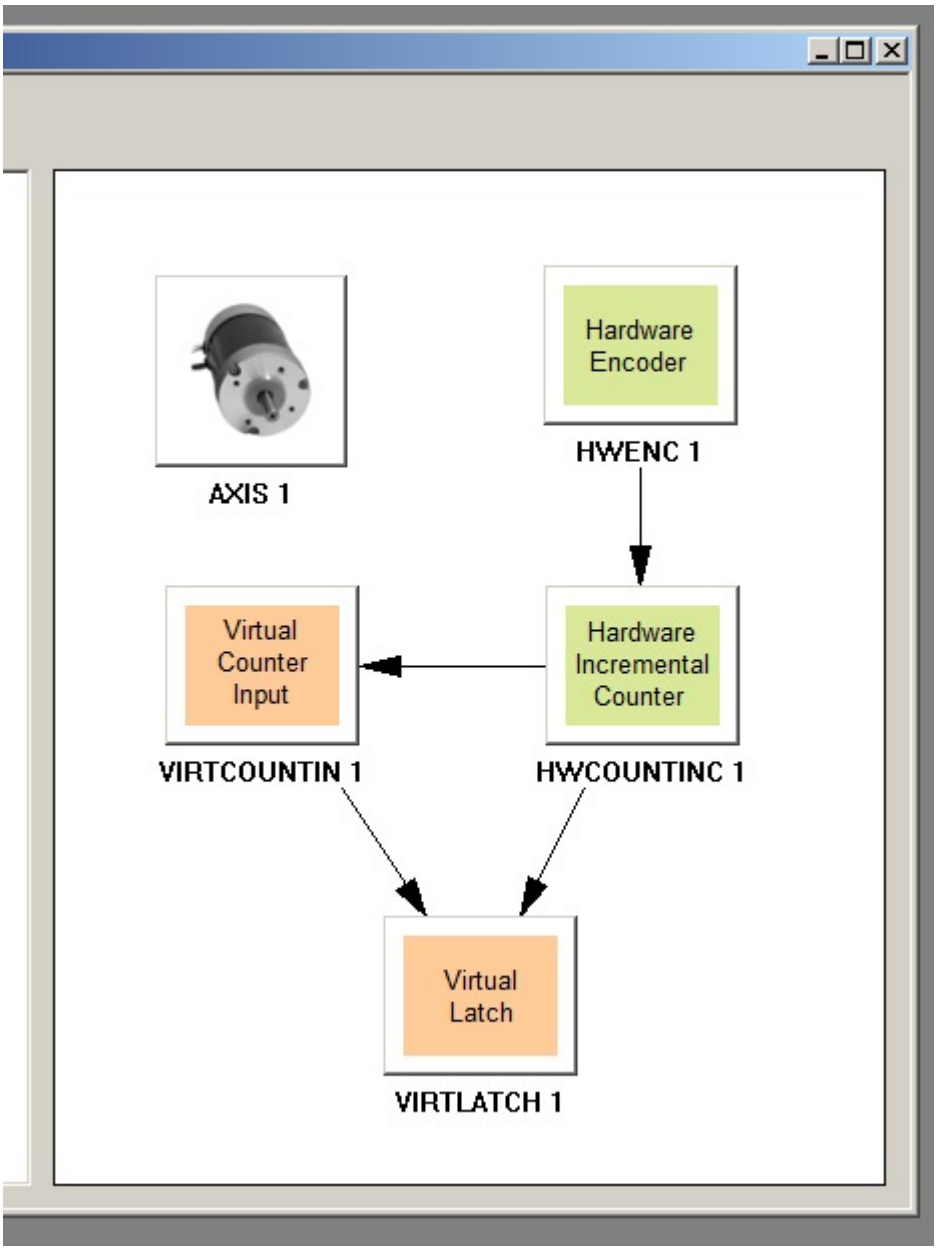
The Module Editor allows the current window layout to be saved a file type of ".zbw". These files can be re-opened at a later time to restore the saved layout. This is done using the normal **File** → **Save** command. As with other types of ApossIDE files, all the normal Windows "file open" mechanisms can be used to re-open the file (e.g. drag-

and-drop, double-clicking on the file, etc.). Note that when a file is re-opened, the Module Editor will attempt to reconnect to the controller and re-read the configuration. Module Editor files will be identified in Windows folders with the icon shown below.

5.4.2 Module Graphics View

The Module Graphics View provides a way for users to organize function modules into a visual arrangement that makes them easier to work with. When a function module is opened using the Module Tree View, then in addition to a Module Window being opened, an icon for that module is automatically added to the Graphics View. Users can then arrange these icons into a layout that is appropriate to the application that they are developing.

Function modules are normally linked together; one module will provide output values that are then used as input values to other functions. This flow of data is shown in the Graphics View using arrows that point from the "source" module to the "target" module. See Module Linking for more information on linking modules.

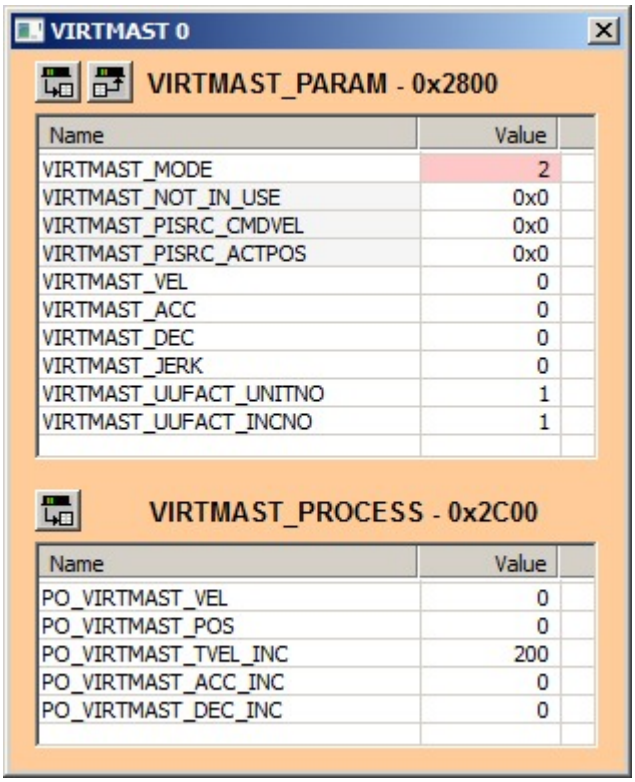


Module Graphics Views will appear similar to that shown above. Note the following about these views:

1. Module icons can be moved simply by clicking and dragging them with the left mouse button.
2. The module name and module number are displayed under each icon.
3. The background color of the icon is set to help the user identify the type of the module (see Module Tree View).
4. Double-clicking on an icon will switch the Windows focus to the Module Window for that module.
5. Right-clicking on an icon will display a popup menu containing commands appropriate to that module.

5.4.3 Module Window

Module Windows display module parameters and process data values from the controller. A Module Window can be opened for any function module available on the controller. Module Windows are opened using the Module Tree View.



Module Windows will appear similar to that shown above. Note the following about these windows:

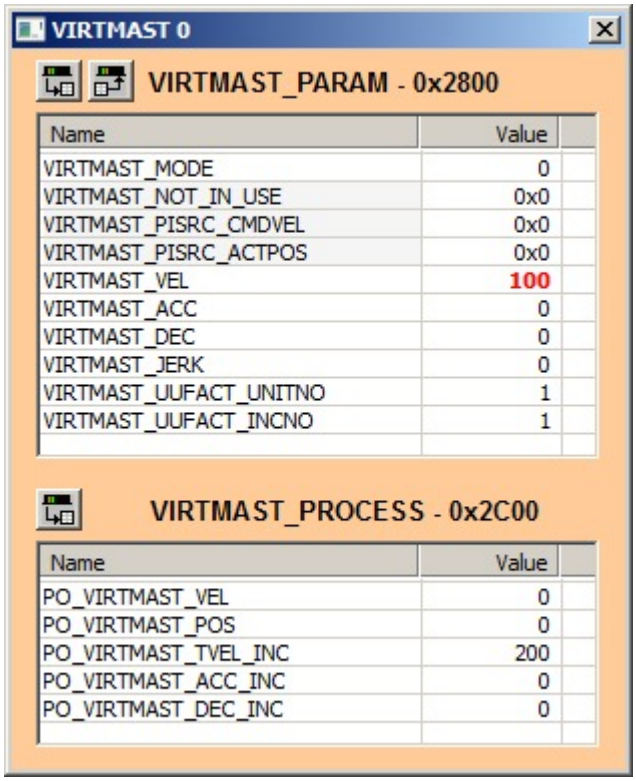
1. The windows can be moved and resized in the same way as other windows.
2. Newly opened windows will always be opened in the same position (i.e. at the top-left corner of the ApossIDE window). Users should move the windows to more appropriate positions when they are opened.
3. The module name and module number are given in the window title (e.g. "VIRTMAST 0" in the above illustration).
4. The background color of the window is set to help the user identify the type of the module (see Module Tree View).
5. Each module usually consists of a set of "Parameter" values (shown in the upper list) and a set of "Process Data" values (shown in the lower list). The corresponding SDO name and index number are displayed above each list. Each individual value corresponds to an SDO subindex and the name of the subindex is given in the list.
6. The user can configure the behaviour of a function module setting the "Parameter" values. The current state of the function module is shown by the "Process Data" values.
7. Pausing the mouse on a subindex name, will display a yellow popup window with a short description of that subindex (if a description exists). Pausing on a subindex value, will display a yellow popup window describing the valid values for that subindex (if this information exists). More detailed help is often available by clicking on the particular subindex of interest and then pressing **F1**.
8. Values that have invalid values will be displayed with a pale red background (e.g. "VIRTMAST_MODE" above).
9. Some values will contain links to other function modules. These will be displayed with a pale grey background (e.g. "VIRTMAST_PISRC_CMDVEL" above). See Module Linking for more information on linking function modules.
10. Some function modules have Parameter or Process Data values that are used only rarely. These values are normally not included in the lists. However, if this is the case, then the window will include a "Display rarely-used SDOs" option that allows the user to display all values.
11. At any time, the button can be used to upload current values from the controller. Note that these are "current" values and not "initial" values (see Module Editor for the distinction between current and initial

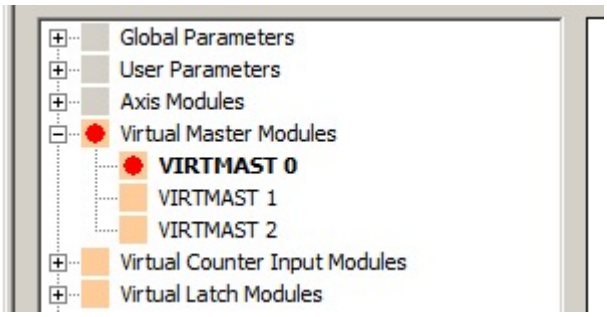
values).

12. At any time, the button can be used to download the "Parameter" values in the window to the controller.

Modifying Parameter Values

Parameter values may be modified by double-clicking in the "Value" column. Modified values will be displayed using bold red text (e.g. "VIRTMAST_VEL" below). In addition, any module that has been modified will be highlighted in the Tree View with a red dot (e.g. "VIRTMAST 0" below). If Parameters are downloaded to the controller (with either the window button or the toolbar button), then modified values will be shown in black again.





If a value is currently being modified (i.e. the value was double-clicked), then pressing **Esc** will abort the edit and restore the previous value. If a modified value has already been accepted (i.e. is displayed red), then the original unchanged value can be restored by right-clicking on the value and selecting the **Clear Change** command from the popup menu.

Polling Process Data Values

The Module Editor provides a polling feature to help developers with debugging application

programs. When polling is active, then "Process Data" values in *all* Module Windows are polled at regular intervals and the Module Windows are updated. This allows developers to see in real time whether or not module values are changing.

Polling is started using the toolbar button and stopped using the toolbar button. The interval at which polling occurs, can be set using the **Settings** → **Module Editor** command.

Copy/Paste

To assist developers that are writing application programs, the Module Window provides a **Copy (Ctrl-C)** function. By clicking on any "Parameter" or "Process Data" item and then pressing **Ctrl-C**, a short piece of code will be copied to the Windows clipboard. This can then be pasted into the application program.

If **Ctrl-C** is used in the "Parameter" list, then an "assignment" statement will be copied. This will have a form similar to this:

```
VIRTMAST_PARAM(0,VIRTMAST_VEL) = 0
```

Note that using this code will require the application program to include the "sysdef.mi" file (see SysDef Defines). This file contains the #define statements needed by this statement. If for some reason, the parameter is missing from the "sysdef.mi" file, then code similar to the following will be copied to the clipboard:

```
sysvar[0x01nnnnss] = 0
```

If **Ctrl-C** is used in the "Process Data" list, then code similar to the following (without the assignment) is copied to the clipboard:

```
VIRTMAST_PROCESS(0,PO_VIRTMAST_TVEL_INC)
```

Note that the assignment is not created since "Process Data" values are read-only. The Module Editor expects "Process Data" values to be used in expressions, rather than being assigned a value.

5.4.4 Module Editor Toolbar

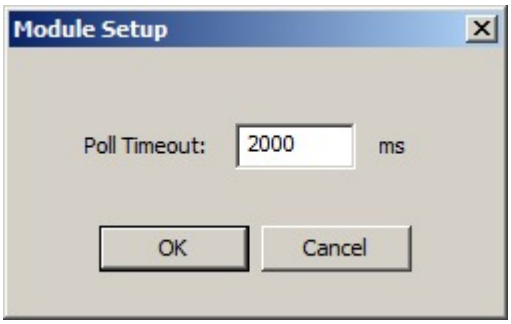
The Module Editor toolbar contains the following toolbar buttons that are specific to the Module Editor:

- **Connect** - Select a controller, connect it to the current window, and upload the configuration from the controller. This will replace the existing configuration values in the Module Editor.
- **Read from controller** - Upload the configuration from the currently connected controller. This will replace the existing configuration values in the Module Editor.
- **Write to controller** - Download the configuration values to the currently connected controller.
- **Read Include File** - Read a "Parameter Include File" (see Initializing Temporary Parameters). This will replace the existing configuration values in the Module Editor.
- **Write Include File** - Write the existing configuration values in the Module Editor to a "Parameter Include File" (see Initializing Temporary Parameters).
- **Start Polling** - Start polling for "Process Data" values (see Module Window).

- **Stop Polling** - Stop polling for "Process Data" values (see Module Window).

5.4.5 Module Editor Settings

The **Settings** → **Module Editor** command allows the user to specify the Module Editor settings. These settings are saved in Module Editor ".zbw" so they may be set on a per-file basis.



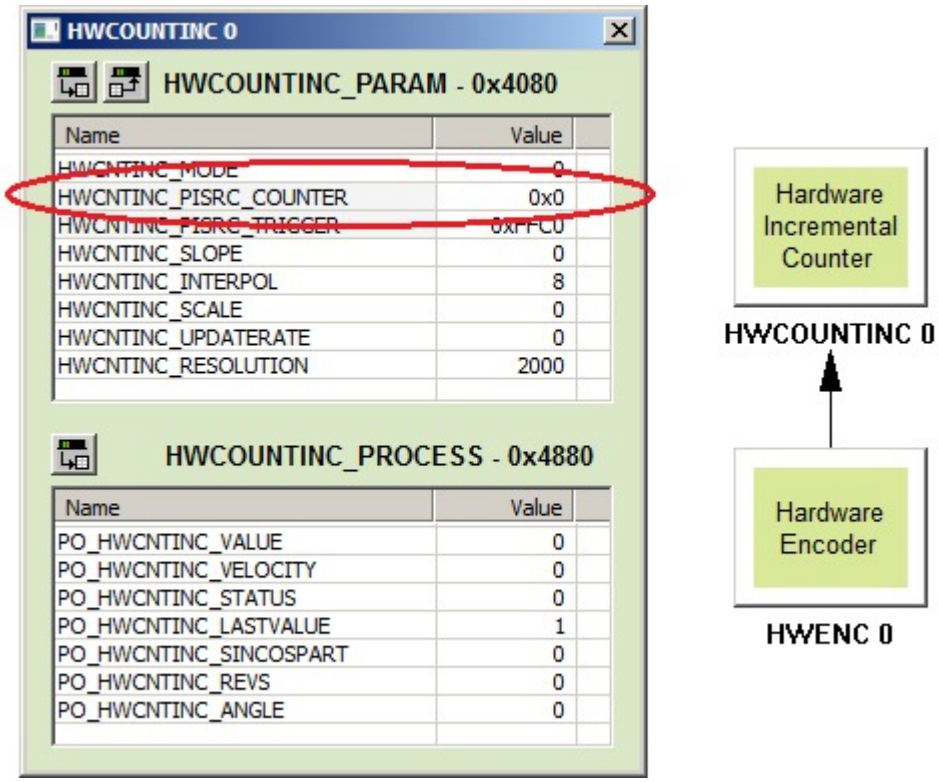
Poll Timeout

This value specifies the time interval at which process data values are polled when polling is active. See Module Window for more information about polling.

5.4.6 Module Linking

Function modules are normally linked together; one module will provide output values that are then used as input values to other functions. For example, a hardware counter is normally configured to receive input from a hardware encoder, a virtual counter will receive input from a hardware counter, and so forth. This is displayed graphically in the Module Graphics View.

This linking is done using special "link" SDO subindices in modules. For example, a value of 0x0 for "HWCNTINC_PISRC_COUNTER" (shown below) indicates that this hardware incremental counter module receives its input from hardware encoder module 0. Link subindices will always be displayed using a pale grey background.



Link Parameter Values

Several different formats are used when specifying values for "link" subindices:

0x00nnnnss	This is a link to a specific SDO index (0xnnnn) and subindex (0xss). For example, 0x250002 links to the REG_CMDPOS value for axis 1.
0xnnnnssll	This is a link to a specific SDO index (0xnnnn) and subindex (0xss) and it include a length (0xll).
0xvvvv	This is a link to a function module. The actual function module is determined by the coded 0xvvvv value.

The format required depends on the specific subindex. Click on the subindex and press **F1** for information about the appropriate format for that subindex.

Modifying Link Values

The Module Editor provides several different ways for users to modify link values:

1. Double-clicking on the link value in the "Value" column of the Module Window, will allow the value to be edited directly.
2. Right-clicking anywhere on the link subindex in the Module Window, will display a popup menu. The "Select SDO" command will display a dialog allowing the user to choose the SDO from a list.
3. The link can be made graphically using the Module Graphics View. To do this, place the mouse on the "source" module (i.e. the module that will providing the input), press the **Ctrl** key, click with the left mouse button, and drag to the "target" module. If the Module Editor needs additional information to complete the link, then the user will be prompted.
4. The link can also be made graphically by right-clicking on the "source" module in the Module

Graphics View and selecting **Link to...** from the popup menu. Then click with the left mouse button on the "target" module. If the Module Editor needs additional information to complete the link, then the user will be prompted.

Opening Module Chains

The Module Editor provides several methods for following "links" and opening the "source" module:

1. Double-clicking on the name of the "link" subindex in the Module Window will open the module to which this subindex is pointing. If the module is already open, then the Windows focus is moved to the "source" module.
2. Right-clicking anywhere on the link subindex in the Module Window, will display a popup menu. The "Open Module" command will open the module to which this subindex is pointing. If the module is already open, then the Windows focus is moved to the "source" module.
3. Right-clicking and selecting the "Open Chain" command will open all "source" modules *plus* all the modules to which those "source" modules point. This is done recursively so that all modules in the chain are opened.

5.5 State Machines

The ApossIDE and zub motor controllers provide built-in support for hierarchical state machines. This section describes how to program such a state machine using the ApossC programming languages. It assumes that the reader has a basic understanding of the structure, functioning, and use of state machines in general and of hierarchical state machines in particular.

> To program a state machine in ApossC, a zub motor controller with firmware version 9.0.0 (or later) is required.

The basic steps required to program a state machine are as follows:

1. Define the signals and event types that the state machine will be using. See Defining Signals and Event Types.
2. Define the state hierarchy of the state machine. This is the organization of states and substates within the state machine. See Defining the State Hierarchy.
3. Specify the event types that each state and substate will recognize. See Specifying the Event Types to be Handled by a State.
4. Define the state machine variable. This variable will contain the dynamic memory that is used by the state machine while it is running. See Defining State Machine Variables.
5. Specify the state machine configuration parameters. These are global properties that the controller uses while state machines are running. See Specifying the State Machine Configuration.
6. Code the actual handlers and state functions for each state and substate. See Defining Handlers for Events.
7. Code the application main program so that state machine execution is started. See Starting and Stopping State Machines.

Overview of all state machine commands:

Command	Syntax
SmClear	`SmClear(machine, sig-expr);`
SmConfig	`#pragma SmConfig { runflags, poolsize, timercnt, subsize, prmsize, possize }`

SmEvent	`SmEvent signal { parameter1, parameter2, ... }`
SmInput	`SmInput(input-expr, test-expr, machine, sig-expr);`
SmMachine	`SmMachine machine;` `SmMachine machine { id, initfunc, topstate, queuesize, datasize }`
SmParam	`SmParam(param-expr, mask-expr, test-expr, machine, sig-expr);`
SmPeriod	`SmPeriod(timeout, machine, sig-expr);`

SmPosition	`SmPosition(sign postype, axis, position, machine, sig-expr);`
	`SmPosition(sign postype, axis, position, machine, sig-expr, ONCE);`
	
	`SmPosition(sign postype, axis, position, SETOUT, output);`
	`SmPosition(sign postype, axis, position, SETOUT, output, ONCE);`
	
	`SmPosition(sign postype, axis, position, SETOUT, output, input);`
	`SmPosition(sign postype, axis, position, SETOUT, output, input, ONCE);`

SmPost	`SmPost(machine, sig-expr, param-expr, param-expr,...);`
SmPublish	`SmPublish(sig-expr, param-expr, param-expr,...);`
SmRun	`res = SmRun(machine1, machine2,...);`
SmState	`SmState state { ... }`
SmStop	`SmStop(expr);`
SmSubscribe	`SmSubscribe(machine, sig-expr, param-expr, param-expr,...);`
SmSystem	`SmSystem(keyword, param-expr, param-expr,..., machine, sig-expr);`
SmTimer	`SmTimer(timeout, machine, sig-expr);`
SmUnsubscribe	`SmUnsubscribe(machine, sig-expr);`
SmUrgent	`SmUrgent(machine, sig-expr, param-expr, param-expr,...);`

5.5.1 Defining Signals and Event Types

State machine programs may define an arbitrary number of event types. Each event type will consist of a user-defined signal and an arbitrary number of event parameters. The meanings of the signal and each of the parameters are entirely up to the user. For example, the user may decide to implement an "Article complete" event type to indicate that article processing is complete and that contains two parameters: the article serial number and an article count.

An event is defined in using the statement:

```
SmEvent signal { parameter1, parameter2, ... }
```

where:	*signal*	The name of the signal.
	parameter	The names of 0 or more event parameters. The braces are required even if there are no parameters.

Both signal and parameter names can be any names chosen by the user as long as they conform to the same naming convention as normal variables (i.e. letters, digits, etc.). These names, as well as the `SmEvent` keyword itself, are not case-sensitive. Signal and parameter names share the same "name space" as variables and functions so all of these names must be unique. The signal name and braces are mandatory, even if no parameters are specified. After being defined, the signal and parameter names will behave like `long` constants and can be used anywhere in an application program where a constant can be used. See State Functions for more details.

`SmEvent` statements may be placed anywhere within a program where it is valid to place a statement, except within functions. However, `SmEvent` statements must be placed before any other statement that references either the signal name or the parameter names. The order in which the statements are placed is irrelevant.

Examples:

```
SmEvent ARTICLE_COMPLETE { ARTICLE_SERIAL, ARTICLE_COUNT }
SmEvent ARTICLE_MISSING  { ARTICLE_SERIAL }
SmEvent SHUTDOWN_REQUEST { }
```

The same parameter name may be used in multiple event types (as is "`ARTICLE_SERIAL`" in the above example). However, if this is done, then the parameter *must* be placed in the *same* position in each event type in which it is used. For example, the following is invalid and will produce a compiler error:

```
SmEvent ARTICLE_COMPLETE { ARTICLE_COUNT, ARTICLE_SERIAL }
SmEvent ARTICLE_MISSING  { ARTICLE_SERIAL }
```

ApossC predefine the following event types that are related to the general operation of hierarchical state machines:

```
SmEvent SIG_ENTRY { }
SmEvent SIG_EXIT  { }
SmEvent SIG_INIT  { }
SmEvent SIG_START { }
SmEvent SIG_IDLE  { }
```

Events of these types are generated automatically and sent to the state machine in the following situations:

`SIG_ENTRY`	Generated immediately after the state machine transitions into a state. The state will receive this event *before* any `SIG_INIT` event.
`SIG_EXIT`	Generated immediately before the state machine transitions out of a state.
`SIG_INIT`	Generated each time the state machine enters the target (i.e. final)

state of a transition *and the target state is entered from the target state's superstate*. The event is *not* generated if a state simply transitions to its superstate. This behaviour can be modified using the SM_RUN_SIGINIT flag. The target state will receive this event *after* the `SIG_ENTRY` event.

`SIG_START`	Generated once each time the state machine is about to start processing its event queue.
`SIG_IDLE`	Generated once each time the state machine finishes processing its event queue.

ApossC also predefines the following event type that relates to the general operation of the motor controller:

`SIG_ERROR`	The controller has detected an error.

See Handling General Controller Errors for a detailed description of the `SIG_ERROR` event type.

5.5.2 Defining the State Hierarchy

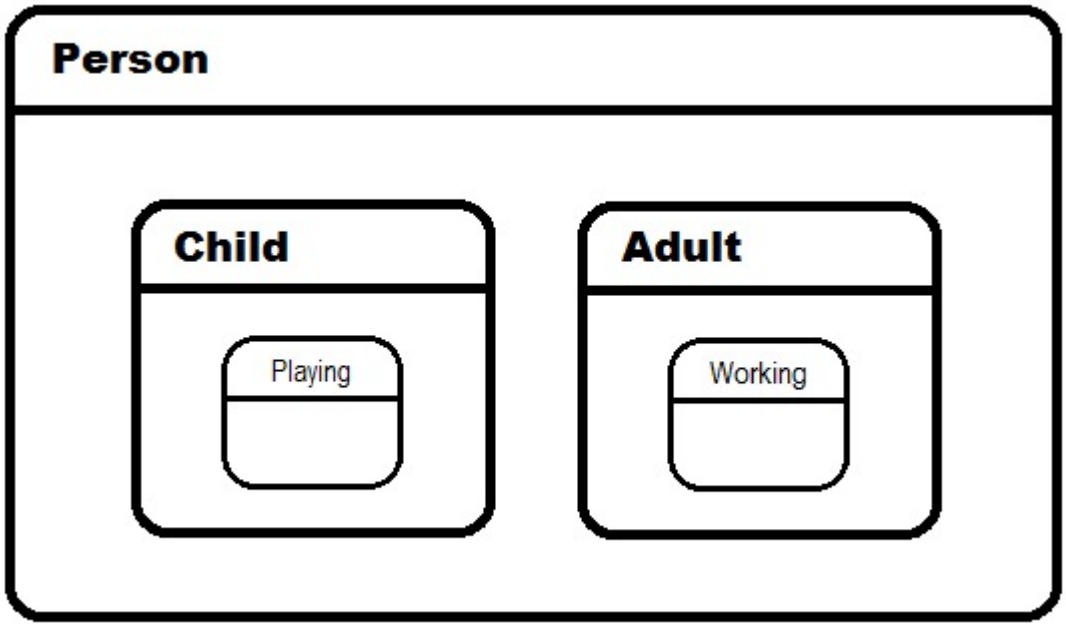
State machine programs may define an arbitrary number of states and an arbitrary number of substates within each state and substate. This is done using the SmState statement as follows:

```
SmState state {
              SmState substate1 { ... }
              SmState substate2 { ... }
              ...
              }
```

where:	*state*	The name of the state.
	substate	The names of 0 or more substates.

The state and substate names can be any names chosen by the user as long as they conform to the same naming convention as normal variables (i.e. letters, digits, etc.). These names are not case-sensitive. The names of all immediate (i.e. sibling) substates within a state must be unique. However, names need not be unique if substates are within different states. As well, state names and variable names need not be unique; the same name may be used for both a state and a variable but this is not recommended. The state name and braces are mandatory, even if the state contains no substates. The order in which states and substates are defined is irrelevant.

For example, this state machine:



would be defined in an application program like this:

```
SmState Person {
                SmState Child {
                               SmState Playing { }
                               }
                SmState Adult {
                               SmState Working { }
                               }
                }
```

5.5.3 Specifying the Event Types to be Handled by a State

Any state may handle an arbitrary number of different event types. This is done by specifying the signals to be handled within the SmState statement braces as follows:

```
SmState state {
               signal1 = handler
               signal2 = handler
               ...
               }
```

where: *signal* A signal name specified in any of the `SmEvent` statements or any of the predefined event types.

handler Defines what to do when the state receives that particular event. See Defining Handlers for Events for a detailed description of handlers.

These signal statements may be placed anywhere within the state's braces (i.e. before, between, or after the `SmState` statements of substates). They need not be placed in a contiguous block; they may be split and interspersed between substates if that provides a better organization for the developer creating the application program. Signals may only be specified once within a state; multiple handlers for the same signal are not allowed

(although the same signal *can* be specified within the substates of the state). Signal names must be defined in `SmEvent` statements before they are referenced in `SmState` statements. Note that `SmEvent` statements themselves may be placed between top-level `SmState` statements; they may not be placed within a state's braces. The order in which the signal statements are listed, may have a very small influence on performance. If many signals are specified and some of those signals are generated very frequently, then those signals should be placed near the beginning of the list. This prevents the entire signal list from being scanned each time a frequent event is received.

As is normal for hierarchical state machines, if a state receives an event for a signal that is not specified in that state's list of signals, then that event is automatically forwarded to the state's superstate. If the superstate also does not handle the event, then it is forwarded again up the chain. If the final top-most state does not handle the event, then the event will be discarded. There are three exceptions to this rule: the predefined signals `SIG_ENTRY`, `SIG_EXIT`, and `SIG_INIT`. These three events are never forwarded to a superstate. If they are not handled by the state that receives them, then they are discarded.

5.5.4 Defining State Machine Variables

Multiple state machines may be executed simultaneously. Each state machine must be defined using the `SmMachine` statement. The `SmMachine` statement has the following syntax:

```
SmMachine machine { id, initfunc, topstate, queuesize, datasize }
```

where:	*machine*	The name of the state machine.
	id	A user-defined ID number.
	initfunc	The name of an initialization function (or *).
	topstate	The name of the top-most state of the state machine.
	queuesize	The maximum number of events in the state machine's event queue.
	datasize	The number of elements in the private data array for the state machine.

The statement will define a "state machine variable" with the name "*machine*". This variable will behave similarly to other global variables. It may be accessed from anywhere in the application program and its name must be unique from all other global variable names. However, a state machine variable cannot be assigned values, passed as a function argument, etc. It may only be used in state machine commands. These commands are described later.

> The statement:
>
> ```
> SmMachine machine;
> ```
>
> may be used. This only declares "*machine*" to be a state machine name. The full SmMachine statement defining the state machine attributes, must follow at some point later in the program. This allows the actual state machine name (rather than just a state machine "*id*" number) to be used in other state machine statements, even before the state machine has been fully defined.
>
> This is important since it allows state machine names to be declared *before* the state heirarchy of a state machine is defined. Doing this then allows inline state functions (see State Functions) the opportunity to use state machine names, rather than just "*id*" numbers, in other state machine commands. Specifying the state machines name rather than "*id*" numbers, has some performance benefits.

The "*id*" can be any `long` value (or expression that evaluates to an integer at compile time), however its value must be unique among all state machine variables. This value is intended to allow generic initialization functions and state functions to identify the state machine for which they are being called. For example, in a multi-axis system, the axis number might be a good choice for the ID number. Although any value may be used for a state machine ID, error logs currently support only ID numbers in the range 0-255. If ID numbers outside this range are used, then only the low-order byte of the ID number will be recorded in error logs (see State Machine Errors).

"*initfunc*" is an optional "initialization function". This will be called automatically before actual state machine execution starts and is intended to allow the user to do whatever initializations may be necessary before execution starts. If no initialization is necessary, then this parameter may be set to "*". An initialization function is a normal application function, but it must have the following prototype:

```
long initfunc(long id, long data[]);
```

where:	*initfunc*	The name of the initialization function.
	id	The state machine ID number.
	data	The private data array of the state machine.

If this function already has a prototype, then the existing prototype is checked to ensure that it matches the above prototype. If a prototype does not yet exist, then a prototype will be automatically created. The initialization function may be defined anywhere in an application program that it is valid to define a function. Multiple state machines can specify the same initialization function. When the execution of one or more state machines is started (i.e. by the `SmRun` command), then the initialization function is called for each state machine being executed *before* any state machine begins to receive events. There is no guarantee as to the order in which the initialization functions will be called. Initialization functions must return 0 for success. If any initialization function returns non-0, then state machine execution is not started (and the `SmRun` command will return with that non-0 value as a return code). This is intended to allow the application program to abort state machine execution if initialization fails for some reason.

The "*topstate*" specifies the top-most state of the state machine. This state must be previously defined by an `SmState` statement and it cannot be a substate of any other state. Multiple state machines can have the same top-most state. For example, in a multi-axis system, there may be one state machine for each axis and each state machine may be in a different state, but each state machine will use the same basic state structure.

The "*queuesize*" value is an integer (or expression that evaluates to an integer at compile time) that defines the maximum number of pending events waiting to be handled by the state machine. There is a minimum queue size which is currently set to 10. Care must be taken when determining the queue size for a state machine. A larger queue will consume more system memory so a queue that is larger than necessary will waste memory. However, if the queue is too small (i.e. events are added to the queue faster than the state machine can handle the events), then the controller will enter an error condition and program execution will stop (see State Machine Errors). Determining an appropriate queue size requires a careful analysis of the expected behaviour of the system as a whole (which may include multiple interacting state machines).

"*datasize*" is an integer (or expression that evaluates to an integer at compile time) that specifies the size of a private data array of `long` values that will be allocated for this particular state machine. Each state machine will have its own private data array accessible only to itself. This array is intended to allow the state machine to save whatever private data it may need. For example, it may save information such as target position, CAM curve index, etc. This data array is passed as an argument to initialization functions and state functions.

5.5.5 Specifying the State Machine Configuration

State machines require a few global parameters to be set before state machines can be run. These are set using the `SmConfig` statement as follows:

```
#pragma SmConfig { runflags, poolsize, timercnt, subsize, prmsize, possize, syssize }
```

where:	*runflags*	Runtime flags.
	poolsize	The total number of events available in the system event pool.
	timercnt	The total number of timers required by all running state machines.
	subsize	The total number of records available in the system subscribe pool.
	prmsize	The total number of records available in the system parameter pool.
	possize	The total number of records available in the system position pool.
	syssize	The total number of records available in the system signal pool.

Note that this is a #pragma statement so all of these parameters must either be integers or expressions that will evaluate to integers at *compile time*. It is used to preallocate the memory required by the pools. Also, "*runflags*" and "*poolsize*" are mandatory parameters. All other parameters are optional and will default to 0. If an optional parameter is specified, then all previous optional parameters must also be specified (e.g. "`#pragma SmConfig { 0, 20, 0, 0, 5 }`").

The breakdown of the memory required by the various pools can be found in the [POOLMEMORY] section of the ".ad$" cross reference file that can be generated by the ApossIDE compiler. The generation of this file can be enabled using the ApossIDE **Settings** → **Compiler** menu command.

The `SmConfig` statement may be placed anywhere within an application program where it is valid to place a `#pragma` statement. It is expected that the statement be placed before the state machine is defined but this is not strictly necessary. It may even be placed after all other state machine statements. It is also expected that there be a single `SmConfig` statement for the entire application program. However, it there are multiple statements, then the values specified in the last statement will be the values used. If a program contains no `SmConfig` statement, then the event pool size will default to 10 and all other sizes will all default to 0.

Any combination of the following bits (defined in "`sysvar.mi`") may be set in "*runflags*":

SM_RUN_DELAY 0x0001	If this flag is set, then the state machine will allow commands that require any type of "wait" (e.g. the various `DELAY` and `WAIT` commands, as well as the `HOME` command) to be executed. Otherwise, a state machine error is generated if any "wait" command is executed. In general, state machines should not execute commands requiring delays since this will halt all state machine execution (n.b. especially when multiple state machines are running) until the delay is complete. No event queue processing will be done which causes a risk of the event queues becoming full. Note that this flag is *not* active during the execution of the state

	machine initialization routines. It only becomes active when event queue processing is started.
SM_RUN_INTERRUPT 0x0002	If this flag is set, then the state machine will allow interrupt handling to be processed. Otherwise, all interrupts are disabled during state machine execution. The controller firmware will automatically generate state machine events for all cases covered by interrupts. In general, state machines should be written to handle these events and not rely on interrupt handlers. Note that interrupt handler is *always* disabled when state machines are running. See Handling General Controller Errors for a description of how errors are handled.
SM_RUN_DELDISABLE 0x0004	Normally, if a state machine attempts to delete an `SmParam`, `SmInput`, or `SmSystem` record that does not exist, then the controller will be put into an error condition with error #57 and state machine Error 18. If this flag is set, then the state machine will disable this test. This is useful in cases where a state machine is in a situation where it attempts to delete the same record multiple times. It is recommended that this flag *not* be set unless necessary, since deleting an unknown record will often indicate a programming error.
SM_RUN_SIGINIT 0x0008	Normally, the last state at the end of the transition chain will receive a `SIG_INIT` event *if it is being entered from its superstate*. The `SIG_INIT` event is *not* received if the last state is being entered from a substate. This behaviour can be modified by setting this flag. If set, then the `SIG_INIT` event will *always* be received by the last state at the end of the transition chain.
SM_RUN_NO_DUPLICATE_EVENTS 0x0010	Normally, every new event is put to the event queue and then processed. If this flag is set and the new event is received, then event queue will be first checked if any such event already exists and if so the new event will be discarded. It is recommended that this flag *not* be set unless necessary, since this can cause certain events to be lost. This flag may be useful though in situations when commands requiring delays are to be executed.

The system event pool is a block of memory from which all event records are allocated. When an event is created and posted (see Posting Events to State Machines), the memory used to save the signal, event parameter values, etc., is allocated from the event pool. All events for all running state machines are allocated from the same event pool. When an event is completely processed and is no longer required, the system will automatically free the allocated memory and the memory will become available for a new event when one is created.

The size of the event pool must be determined by careful analysis of the expected behaviour of the state machines. The event pool must be large enough to hold all pending, unprocessed events that may exist at any time during the execution of the state machines. If the event pool is much larger than necessary, then system memory will be unnecessarily wasted. However, if the event pool is too small and an attempt is made to create a new event when the pool has no more free space, then the controller will enter an error condition and all program execution will stop (see State Machine Errors).

The ApossIDE has built-in support for timers that may be used by state machines if needed. There are two types of timers available: timeout timers that will send an event once when the timer expires and periodic timers that will send events at regular intervals. See State Machine Timers for more details. The controller needs to know the total number of timers that may be active at any one time during the execution of the state machines so that system memory can be reserved for the timers. As with the event pool, this number must be determined by analysis of the state machines: too large and memory will be wasted; too small and there will be no available timer when one is needed.

The system subscribe pool, parameter pool, position pool, and system signal pool are blocks of memory from which all subscribe records, parameter records, position records, and system signal records respectively are allocated. All records for all running state machines are allocated from the same pools. When records are no longer required, the system will automatically free the allocated memory and the memory will become available for a new record when one is needed. See Subscribing to Events and Publishing Events for more a more detailed description of subscribe records, Detecting Changes to Parameters for more information on parameter records, Detecting Motor Positions for more information on position records, and Detecting General Operation Events for more information on system signal records.

5.5.6 Defining Handlers for Events

There are three classes of handlers that may be used to define how a state will handle an event:

- Predefined Handlers
- State Transitions
- State Functions

Normally, any class may be used with any event type. However, there are some restrictions that will be described where appropriate.

5.5.6.1 Predefined Handlers

There are two predefined handlers: `SmHandled` and `SmNotHandled`. These are used as follows:

```
SmState state {
              signal1 = SmHandled;
              signal2 = SmNotHandled;
              ...
              }
```

The `SmHandled` keyword will cause the event to be treated as “handled” when no other action is required. The event will simply be discarded; it will not be forwarded to the state’s superstate and the state machine will remain in its current state. Note that the current state may be a substate of the state that actually handles the event if the event has been passed up the superstate chain.

The SmNotHandled keyword will cause the event to be explicitly passed up to the state's superstate when no other action is required. Note that this is not strictly necessary since the same end result will happen if the signal is simply not specified: the event will then be implicitly passed up to the superstate. In cases where a state specifies many signals and at the same time receives frequent events that are "not handled", then there may be a small performance benefit to explicitly specifying the "not handled" signals and placing them near the beginning of the list. This prevents the entire signal list from being scanned each time a frequent event is received.

The following default handlers are provided for every state. They need not be explicitly specified unless a different behaviour is required (even in the case of a state that has many signals that are generated frequently). Take special note of SIG_START and SIG_IDLE which are passed up to the superstate by default rather than being "handled" and discarded:

```
SmState state {
              SIG_ENTRY = SmHandled;
              SIG_EXIT  = SmHandled;
              SIG_INIT  = SmHandled;
              SIG_START = SmNotHandled;
              SIG_IDLE  = SmNotHandled;
              }
```

The SmNotHandled keyword should not be used with SIG_ENTRY, SIG_EXIT, or SIG_INIT events. If it is used with these events, then it will be ignored. These three events are never passed up to a state's superstate.

5.5.6.2 State Transitions

If the only action required in response to an event, is to transition to another state, then this is done using either of the following two syntax examples:

```
SmState state {
              signal = SmTrans(transition);
              ...
              }
SmState state {
              signal = (transition);
              ...
              }
```

where "*transition*" specifies the path from the current state to the target state. Note that the parentheses are required and indicate that this is a transition and not some other type of handler.

Transitions are specified by naming target states and using the arrow operators: <-, ->, and <-*->. These have the following meanings:

name	Transition from the current state to the sibling state "*name*". Note that "name" *must* be a sibling state of the current state (i.e. at the same "level" with the same superstate).
<-	Transition "up" one level from current state to the superstate. Transitions can be made up more than one level by specifying more than one "up" arrow.
<-*name*	Transition "up" from current state until the superstate "*name*" is found.
->*name*	Transition "down" from current state to the substate "*name*".
<-*->*name*	Transition "up" from current state until a superstate is found that

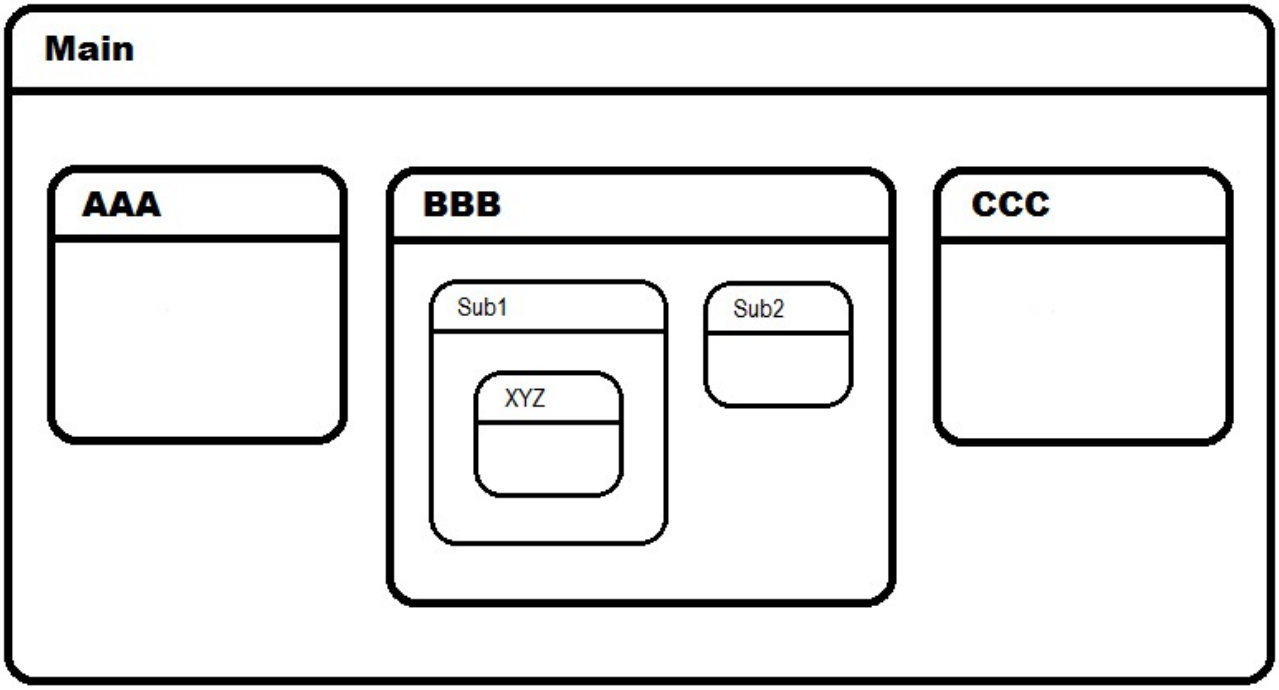
contains the substate "*name*". Then transition "down" to the substate "*name*".

Multiple operators may be concatenated to specify longer paths. However they are subject to the following rules:

1. Once the transition starts "down", it cannot go "up" again. This would be equivalent to going into a state and then back out again during the same transition (e.g. "`-><-`"). This is not a valid transition.
2. A transition cannot go "up" from the top-most state. This would be equivalent to exiting the state machine which is an undefined operation.

Both of these will produce compiler errors.

As an example, consider the following state machine:



All of the following are valid transitions:

Current state	Target state	Transition
BBB	AAA	(AAA)
BBB	CCC	(CCC)
BBB	Main	(<-)
BBB	Main	(<-Main)
BBB	Sub1	(->Sub1)
BBB	XYZ	(->Sub1->XYZ)
AAA	Sub1	(<-->BBB->Sub1)
AAA	Sub1	(BBB->Sub1)

```
AAA         XYZ         (BBB->Sub1->XYZ)
XYZ         Sub1        (<-)
XYZ         BBB         (<-<-)
XYZ         BBB         (<-BBB)
XYZ         Main        (<-<-<-)
XYZ         Main        (<-Main)
XYZ         Sub2        (<-<-->Sub2)
XYZ         Sub2        (<-BBB->Sub2)
XYZ         Sub2        (<-*->Sub2)
XYZ         AAA         (<-*->AAA)
Sub2        XYZ         (<-*->Sub1->XYZ)
```

Please note the following details:

1. The syntax "`<-name`" will transition up until the named superstate is found. The transition will not exit state "`name`". The search will start at the current state in the transition. For example, "`<-<-name`" will go up one level and *then* start the upward search for the named superstate. This can have a different result from "`<-name`" depending on the uniqueness of the state names.
2. The syntax "`<-*->name`" will transition up until the first superstate is found that contains the named substate and then the named state is entered. The transition will not exit the superstate. The search will start at the current state in the transition. For example, "`<-<-*->name`" will go up one level and *then* start the upward search for the named superstate. This can have a different result from "`<-*->name`" depending on the uniqueness of the state names.
3. The explicit "up" operators "`<-`" and "`<-name`" will force a transition "up" (and exit the current state) *even* if a subsequent transition re-enters the *same* state. For example, if the above state machine is in state "`Sub1`", then the transition "`(<-->Sub1)`" will force an exit and re-entry to the state even though the state machine is already in state "`Sub1`". Similarly, the transition "`(Sub1)`" contains an implicit "`<-->`" and so will also force an exit from "`Sub1`" and a re-entry back into "`Sub1`".
4. The operator "`<-*->name`" behaves in a slightly different way from the other "up" operators. If the state machine is not in state "`name`" or any substate of "`name`", then the transition exits all states up to, but not including, the superstate of "`name`" and then enters "`name`" as would be expected. However, if the state machine is *already* in state "`name`" or a substate of "`name`", then the transition will exit all states up to, but *not* including, state "`name`". This makes it possible to specify the top-most state as "`name`" without forcing an implicit exit and re-entry of the top-most state (which is not allowed). For example, if the above state machine is in state "`Sub1`", then the transition "`(<-*->Sub1)`" will *not* exit and re-enter "`Sub1`" whereas "`(<-->Sub1)`" *will* exit and re-enter the state.

> Transitions start in the state which contains the transition handler. For example, if the above state machine is in state "`XYZ`" and it receives an event that is not handled by that state, then the event is passed up to "`Sub1`" to be handled. If state "`Sub1`" handles the event with a "`<-`" transition, then the actual transition that is executed will be equivalent to "`<-<-`" from state "`XYZ`". All substates from the active state up to the state that handled the event are automatically exited. These states will be exited *even* if the final event handler makes a transition back down the substate chain again.

The following restrictions apply to transitions specified in the handlers of specific signals:

1. Transitions should not be specified as handlers for `SIG_ENTRY` and `SIG_EXIT` event types. These signals are not allowed to make transitions. If a transition is specified for these event types, then it will be ignored.
2. Transitions specified as handlers for `SIG_INIT` event types must not transition "up". This is forbidden in hierarchical state machines. Doing so will produce a compiler error (or an execution error if it is done during state machine execution).

Executing a transition will cause `SIG_EXIT`, `SIG_ENTRY`, and `SIG_INIT` events to be generated for the states being exited and entered. See State Machine Execution for details on exactly when these different events are generated.

5.5.6.3 State Functions

If a complex action is required in response to an event, then this is done using a "state function". State function handlers are specified using the following syntax:

```
SmState state {
              signal = statefunc;
              ...
              }
```

where "*statefunc*" specifies the name of the function to execute when this event type is received. State functions are normal application functions but they must have the following function prototype:

```
long statefunc(long id, long signal, long event[], long data[]);
```

If the state function already has a prototype when it is referenced in an `SmState` statement, then the existing prototype is checked to ensure that it matches the above prototype. If a prototype does not yet exist, then a prototype will be automatically created.

The "*id*" will be the ID value set for the state machine in the `SmMachine` statement. "*signal*" will be the event signal number, "*event*" will be the array of event parameters, and "*data*" will be the state machine's private data array. The length of the "*event*" array will be the maximum length of the parameter array for any event defined by the `SmEvent` statements. Any extra parameters not defined for a specific signal will be set to 0.

The signal names defined in `SmEvent` statements can be used as constants when dealing with "*signal*". In the same way, the "*event*" array can be accessed using the parameter names defined in the `SmEvent` statements. For example, if an event has been defined as:

```
SmEvent ARTICLE_COMPLETE { ARTICLE_SERIAL, ARTICLE_COUNT }
```

then a state function could contain statements such as:

```
if (signal == ARTICLE_COMPLETE) ...
if (event[ARTICLE_COUNT] == 100) ...
```

⚠ An event function can modify the event parameters passed to it in the "*event*" argument. Even the extra "unused" parameters (if any) can be modified. This may have the following intended or unintended side effects:

1. If the state function returns `SmNotHandled`, then the event is passed to the state's superstate for handling. The superstate will then receive the modified event parameters rather than the original parameters. This could be useful in certain cases. However, it is not necessarily good programming practice.
2. If the event was "published" (i.e. rather than being "posted"), then the *same* event may be posted to the event queues of multiple state machines. If one state machine's state function modifies the parameters, then the other state machines which have not yet processed the event, will receive the modified event parameters rather than the original parameters. There is no guarantee as to the order

> in which multiple state machines will process a published event. Modifying event parameters in this type of system architecture is almost certain to cause problems.

State functions are expected to return using any of the following statements:

```
return(SmHandled)
return(SmNotHandled)
return(SmTrans(transition))
```

where "*transition*" is any valid transition. However, state functions are simply functions as any other and can therefore return any value. The controller will treat all unknown return codes as "SmHandled". The above return statements are equivalent to the following:

```
return(0)    // SmHandled
return(1)    // SmNotHandled
return(2)    // SmTrans
```

with the exception of the "SmTrans" return. If "return(2)" is used in a state function, then this is equivalent to returning with an empty transition and it will be treated identically to an "SmHandled" return code.

There is nothing that prevents an application program from explicitly calling a state function as if it were a normal function. In this case, the state machine keyword return codes are treated like the integer return codes (i.e. "return(SmHandled)" will be executed as "return(0)", etc.). A "return(SmTrans(...))" statement will be executed as "return(2)" and any specified transition will simply be discarded.

Similarly, a state function can call another function which provides a state function return code. For example, code like the following will work as expected:

```
long StateFunc(long id, long signal, long event[], long data[])
{
    long result;
    result = ProcessFunc(...);
    return(result);
}

long ProcessFunc(...)
{
    if (...) return(SmHandled);
    if (...) return(SmTrans(<-));
    return(SmNotHandled);
}
```

Inline State Functions

In many cases, state functions are quite simple and small. To make these functions easier to implement, ApossC allows them to be coded "inline" by placing the body of the function, enclosed in braces, directly into the signal statement as shown below:

```
SmState state {
              signal = { function body }
              ...
              }
```

Note that no function name and no prototype are specified. The function will have an internally generated name. The prototype (including the argument names) will be exactly as follows:

```
long name(long id, long signal, long event[], long data[]);
```

Example:

```
SmEvent ARTICLE_COMPLETE { ARTICLE_SERIAL, ARTICLE_COUNT }
```

```
SmState Article {
                ARTICLE_COMPLETE = {
                                   data[1]; = event[ARTICLE_SERIAL];
                                   return(SmHandled);
                                   }
                }
```

Design Guidelines for State Functions

The entire state machine execution architecture is based on the idea of "run to completion" state functions. This means that when a state function is called to handle an event, then the entire state function is executed until it completes (i.e. returns) *before* any other state machine execution will take place. As a result, any state function that requires an excessively long execution time will block the handling of other events by the same state machine *as well as* by other state machines that are running simultaneously. State machines and state functions should be designed so as to avoid long execution times within state functions.

A consequence to this requirement for short execution times is that the various Delay and `WAIT` commands are not allowed within state functions. If an attempt is made to execute either of these commands while state machines are running, then this is deemed to be a user programming error in the application program and it will cause the controller to generate an error. See State Machine Errors for a description of how errors are handled.

5.5.7 Starting and Stopping State Machines

One or more state machines can be started using the `SmRun` command. The syntax is:

```
long result;
result = SmRun(machine1, machine2,...);
```

where "*machine*" is the name of a state machine defined in one of the `SmMachine` statements. Note that an individual state machine cannot be started more than once and it is not necessary to start all defined state machines. Currently, once state machines have been started, additional state machines *cannot* be started.

The initialization functions for *all* state machines specified in the `SmRun` command are executed before any event processing is started. The functions are executed in the order in which the state machines are specified. All initialization functions are expected to return a value of 0. If any initialization function returns a non-0 value, then subsequent initializations functions will *not* be executed and the `SmRun` command will return with that non-0 value. This will allow initialization functions to abort state machine execution if necessary. It also allows the main application program to take some recovery action based on the return code if the application program developer has anticipated such situations and if a recover action is available. Note that during initialization, all "delay" commands are valid and all normal interrupt handling is active (see Specifying the State Machine Configuration for more information on delays and interrupts during state machine execution).

Except as noted above, the `SmRun` command will *not* return and application program statements following the `SmRun` command will *not* be executed, until state machine execution is terminated. State machine execution can be terminated by executing the `SmStop` command inside any one of state machine's handler functions. The syntax of the `SmStop` command is:

```
SmStop(expr);
```

where "*expr*" is an expression that evaluates to a non-zero integer. If "*expr*" evaluates to 0, then state machine execution is *not* terminated. The value of the expression is the value that the `SmRun` command will return. No meaning is assigned to this return code; it may be used by the application program for whatever use is appropriate.

The `SmStop` command may be executed by *any* state machine and it will terminate the execution of *all* state machines. Note that all state machines will be terminated *immediately* without any further events being processed even if there are unprocessed events waiting in the event queues. The state function that executes the `SmStop`

command will also be terminated immediately without executing any further commands (i.e. without even completing the processing of the event that led to the `SmStop` command being executed).

Once state machine execution has been stopped, it is not possible to "continue" the execution. The state machines can be restarted only by re-executing the `SmRun` command which will restart the state machines from the beginning (i.e. clear all event queues, call the state machine initialization functions, etc.).

5.5.8 State Machine Execution

The concept of an "interrupt handler routine" does not exist in the state machine architecture. Events that would normally be handled by interrupt routines, are handled by state machines in the normal way by implementing appropriate event handlers. For more information, see Detecting General Operation Events. For the duration that state machines are executing, all interrupt handling will be automatically disabled and no interrupt routine will be called. The original state of interrupt handling will be restored when state machine execution stops.

State machine execution proceeds in a round-robin fashion, starting with the first state machine in the `SmRun` statement and continuing with each successive state machine specified in the command. Execution then starts again with the first state machine. For each individual state machine, execution proceeds as follows:

1. A single `SIG_START` event is queued at the *front* of the state machine's event queue.
2. An event is removed from the front of the event queue and sent to the state machine's currently active state for processing. This event is processed to completion (i.e. until processing ends with either an `SmHandled` or an `SmTrans` return code).
3. If the return code is `SmHandled`, then the event is discarded. If the return code is `SmTrans`, then the transition processed to completion and the event is discarded.
4. If a transition is processed, then each state being exited (if any) will receive a `SIG_EXIT` event. These will be processed *before* any other event is processed and they will be processed in the order in which the states are being exited. All return codes will be treated as `SmHandled`. After this, each state being entered (if any) will receive a `SIG_ENTRY` event. These will be processed *before* any other event is processed and they will be processed in the order in which the states are being entered. All return codes will be treated as `SmHandled`. The last state at the end of the transition chain will then receive a `SIG_INIT` event *if it is being entered from its superstate*. The `SIG_INIT` event is *not* received if the last state is being entered from a substate. Note that this behaviour can be modified using the SM_RUN_SIGINIT flag. The `SIG_INIT` event will also be processed before any other event is processed. If the `SIG_INIT` event processing ends in an `SmTrans` return code, then that transition will *also* be executed to completion before any other event is processed. All *other* return codes will be treated as `SmHandled`.
5. After the current event is completely processed, then the next event is removed from the front of the event queue and processed as above.
6. If the event queue is empty, then a single `SIG_IDLE` event is generated and sent to the currently active state. This event is then processed to completion. If it ends in a transition, then the transition is also processed.
7. Once the `SIG_IDLE` processing is complete, no further event processing is done even if the event queue is no longer empty. The controller will proceed on to the next state machine in the `SmRun` command.

5.5.9 Posting Events to State Machines

Any state machine can generate an event and "post" (i.e. send) it to another state machine. A state machine can also post events to itself. This is done using the following command:

```
SmPost(machine, sig-expr, param-expr, param-expr,...);
```

where: *machine* Tthe state machine to which the event is being sent. Note that

this can either be the name of a state machine or an integer expression that gives the ID of a state machine.

sig-expr An integer expression that gives a signal number.

param-expr 0 or more integer expressions that give event parameter values. Any event parameter not specified, will be set to 0. The braces are required even if there are no parameters.

> ⚠ Using any expression for the signal number (beyond simply assigning a signal number constant to a variable and then using the variable), is not recommended. Signal numbers are generated internally by the compiler and the user does not have control over which numbers are associated with each signal (i.e. one signal number cannot be derived from another signal number).

The event will be posted to the *end* of the specified state machine's event queue (i.e. all events already in the event queue will be processed *before* this new event). The event can be posted to the *front* of the specified state machine's event queue using this command:

```
SmUrgent(machine, sig-expr, param-expr, param-expr,...);
```

This new event will then be processed *before* any of the events that are already in the state machine's event queue. Posting events this way should only be done with care since it can easily compromise the proper functioning of the state machine. For example, there may be high priority events waiting in the queue which are then deferred until the processing of this new event is complete.

The controller will enter an error condition and all program execution will stop in the following cases:

1. An expression is used to specify a state machine ID and the expression evaluates to a value that does not correspond to a valid state machine ID.
2. An attempt is made to post an event to a state machine that is not currently executing.
3. The signal number expression evaluates to a value that is not a valid signal number.
4. The signal number expression evaluates to any of the values `SIG_ENTRY`, `SIG_EXIT`, `SIG_INIT`, `SIG_START`, or `SIG_IDLE`. These are state machine control events generated by the controller and state machines themselves are not allowed to generate them.
5. The event queue of the target state machine is full.
6. The system event pool is full.

5.5.10 Subscribing to Events and Publishing Events

In many cases, it is desirable to be able to generate events and "broadcast" them to any state machine that needs to know about the event. Often, the state machine generating the event will have no knowledge of what other state machines, if any, exist and whether they have an interest in the event. In these cases, it is not possible for the original state machine to post the event to the other state machines. This functionality is handled using a "publish/subscribe" mechanism that works like this:

1. If a state machine is interested in receiving events of a particular type, then that state machine will "subscribe" to that event type.
2. If a state machine needs to generate and broadcast an event, then that state machine will "publish" the event.
3. When an event is published, then the controller will automatically post that event to the *end* of the event queues of all state machines that have subscribed to the event type. Note that it is currently not possible to post an event to the *front* of state machine queues (i.e. the equivalent of `SmUrgent`).

A state machine can publish an event by using the following command:

```
SmPublish(sig-expr, param-expr, param-expr,...);
```

where: *sig-expr* An integer expression that gives a signal number.

param-expr Are integer expressions that give the event parameter values. There can be 0 or more event parameters. Any event parameter not specified, will be set to 0. The braces are required even if there are no parameters.

> ⚠ Using any expression for the signal number (beyond simply assigning a signal number constant to a variable and then using the variable), is not recommended. Signal numbers are generated internally by the compiler and the user does not have control over which numbers are associated with each signal (i.e. one signal number cannot be derived from another signal number).

The controller will enter an error condition and all program execution will stop in the following cases:

1. The signal number expression evaluates to a value that is not a valid signal number.
2. The signal number expression evaluates to any of the values `SIG_ENTRY`, `SIG_EXIT`, `SIG_INIT`, `SIG_START`, or `SIG_IDLE`. These are state machine control events generated by the controller and state machines themselves are not allowed to generate them.
3. The event queue of *any* target state machine is full.
4. The system event pool is full.

A state machine can subscribe to an event type using the following command:

```
SmSubscribe(machine, sig-expr, param-expr, param-expr,...);
```

where: *machine* The state machine which is subscribing to the event type. Note that this can either be the name of a state machine or an integer expression that gives the ID of a state machine. Usually, a state machine will only subscribe itself to an event type. Although it is possible for one state machine to subscribe a different state machine to an event type, this is not recommended as it can lead to programming errors later on.

sig-expr An integer expression that gives a signal number.

param-expr Are integer expressions that give test values. There can be 0 or more test values. The braces are required even if there are no test values. If no test values are specified, then the state machine will receive all published events of this event type. If one or more test values are specified, then the state machine will receive *only* those events where the event parameter is equal to the corresponding test value. For example, if an event was defined in such a way that the first parameter was an axis number, then this could be used to filter events and receive only those events related to a specific axis. Note that the test expressions are evaluated at the time the command is executed and *not* at the time that events are published.

It is possible to execute multiple `SmSubscribe` commands using different "*param-expr*" values. In this case, events will be received if *any* of the `SmSubscribe` commands match the event. However, note that state

machines cannot subscribe to "all" events and "selected" events at the same time. If the command is used with no parameters expressions to subscribe to all events, then any previous "selected" event definitions are cleared. Similarly, if a state machine has previously subscribed to "all" events and then subscribes to a "selected" event, then the state machine will no longer be subscribed to "all" events.

When multiple `SmSubscribe` commands using "*param-expr*" values are specified, then the commands are saved in a system "Subscribe Pool". Note that commands subscribing to "all" events are saved without using resources from the pool. The total number of simultaneous commands (for all state machines) that can be saved in the pool at any one time is defined by the user in the `SmConfig` command (see Specifying the State Machine Configuration). However, if state machines "unsubscribe" from events, then that space is returned to the pool and may be re-used later by subsequent `SmSubscribe` commands.

The controller will enter an error condition and all program execution will stop in the following cases:

1. An expression is used to specify a state machine ID and the expression evaluates to a value that does not correspond to a valid state machine ID.
2. The state machine that is being subscribed is not currently executing.
3. The signal number expression evaluates to a value that is not a valid signal number.
4. The signal number expression evaluates to any of the values `SIG_ENTRY`, `SIG_EXIT`, `SIG_INIT`, `SIG_START`, or `SIG_IDLE`. These are state machine control events generated by the controller and state machines themselves are not allowed to subscribe to them.
5. Test parameter values are specified but the "Subscribe pool" is empty.

A state machine can unsubscribe from an event type using the following command:

```
SmUnsubscribe(machine, sig-expr);
```

If multiple `SmSubscribe` commands have been executed using "*param-expr*" values, then this command will unsubscribe from all of them at the same time. It is not currently possible to unsubscribe from individual `SmSubscribe` commands. If the state machine is not currently subscribed to that event type, then the command will do nothing and no error will be generated.

All the same validation and error testing that is done by the `SmSubscribe` command is also done for this command (i.e. both the state machine and signal must be valid).

5.5.11 Clearing Pending Events

In some cases, it may be useful to clear pending events from a state machine's event queue before the events have been handled. This can be done with the following command:

```
SmClear(machine, sig-expr);
```

where:	*machine*	The state machine owning the event queue to be checked. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then events will be cleared from all state machines.
	sig-expr	An integer expression that gives the signal number.

All pending events in the state machine's queue that match the specified signal, will be cleared and removed from the queue without ever being processed. If there are no events with the specified signal, then the command will do nothing.

This command may not be used to clear events that are related to the general operation of hierarchical state

machines. These are events that use any of the following signals: SIG_ENTRY, SIG_EXIT, SIG_INIT, SIG_START, SIG_IDLE.

5.5.12 Handling General Controller Errors

When state machines are not executing, the normal error handling procedure is as follows:

1. If an ERROR interrupt routine has been defined, then it is called. The routine will continue to be called until the error has been cleared.
2. If an ERROR interrupt routine has *not* been defined, then application program execution is stopped, the motor is stopped, etc.

When state machines are executing, the error handling procedure is as follows:

1. If the error is a state machine error (see Error 57), then state machine execution will stop (and the SmRun command will return with a return code of -1). State machine execution is stopped because the running state machine normally cannot recover from a state machine error. Since state machines are no longer running, the error will get handled using the normal ERROR handling (see above).
2. For all other errors, the controller will "publish" an error event for the executing state machines to handle. This event will have the predefined signal SIG_ERROR and it will contain a single event parameter that specifies the error number of the error. The event will be published only once regardless of how long the error exists before being cleared. The event will be placed on the *front* of event queues (i.e. as if posted using the SmUrgent command). At least one state machine is expected to subscribe to the SIG_ERROR event and provide a handler function for it. Note that the ERROR handling is *not* done. If a *different* error occurs before the original error is cleared, then another error event will be published.
3. If state machine execution is stopped with the SmStop command and there is a residual error that has not been cleared, then the ERROR handling will be invoked in the normal way (i.e. since state machines are no longer running).
4. If the error event is published but no state machines have subscribed to the SIG_ERROR event, then the error event will be completely unhandled. The error event will also be unhandled if all states choose not to handle the event and the event is eventually discarded. In these cases, the controller will execute an implicit "SmStop -1" (or "SmStop(-1);") command to force the state machines to stop executing. The SmRun command will then return with a return code of -1 and the normal ERROR handling will be invoked to handle the error (see #3 above).

The SIG_ERROR event is predefined as follows:

```
SmEvent SIG_ERROR { PRM_ERRNO, PRM_ERRINFO, PRM_ERRAX }
```

where:	PRM_ERRNO	Error number. This will be one of the numbers listed in Error Messages. These errors can also be found in the "sysdef.mi" include file (see SysDef Defines). See also ErrorNo.
	PRM_ERRINFO	Error information value. This is an error-specific value that *may* provide additional information about the error that occurred. Note that not all errors provide additional information values and in these cases, the information value will be meaningless and should be ignored. See also ErrorInfo.
	PRM_ERRAX	Axis number associated with the error. If the error was associated with an individual axis, then this provides the axis number of that axis. Note that not all errors are associated

with an axis and in these cases, the axis value will be meaningless and should be ignored. See also ErrorAxis.

Example

An example of how to use `SIG_ERROR` has been provided in the following sample program:

File → Sample → State Machine Samples → Operational Events → ISR_Error.mc

5.5.13 State Machine Timers

Built-in timer functionality is provided for state machines to use if they require it. There are two types of timers available: timeout timers that will send an event once when the timer expires and periodic timers that will send events at regular intervals. Timers can be started using the commands:

```
SmTimer(timeout, machine, sig-expr);
SmPeriod(timeout, machine, sig-expr);
```

where:

timeout	An integer expression that gives the timeout value in milliseconds.
machine	The state machine which is to receive the event. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then timer events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.
sig-expr	An integer expression that gives the signal number of the event to be sent.

The events are expected to be defined by the user as follows:

```
SmEvent signal { clock }
```

where:

clock	The system clock (SYS_CLOCK) value at the time the event was generated.

The `SmTimer` command will create a timer that will post the specified event to the *end* of the specified state machine's event queue just once when the timeout expires. The event will contain a single parameter (regardless of whether or not a parameter was specified for the signal in its `SmEvent` statement) that contains the system clock value. After this, the timer is automatically freed and will become available to state machines if a new timer is required. If "*timeout*" > 0, then the timer is started. If "*timeout*" < 0, then any existing timeout timer referencing the same state machine and signal, will be deleted without ever posting an event (as long as the event has not already been posted). The command will do nothing and no error will be generated if an existing timer is not found (which may happen if the timer has already posted its event and been deleted). If "*timeout*" = 0, then the signal is posted immediately. Note that this differs from the TIME interrupt where "*timeout*" = 0 deletes the timer without calling the subprogram.

The `SmPeriod` command will create a timer that will post the specified event to the *end* of the specified state machine's event queue each time the timeout expires. The event will contain a single parameter (regardless of whether or not a parameter was specified for the signal in its `SmEvent` statement) that contains the system clock value. The timer will continue to do this until it is explicitly stopped. The timer will be freed and become available

again only after it is stopped. If "*timeout*" > 0, then the timer is started. If "*timeout*" <= 0, then any existing periodic timer referencing the same state machine and signal, will be deleted and no further events will be posted. Note that (unlike `SmTimer`) "*timeout*" = 0 will also delete the timer since a 0-period timer is not allowed. The command will do nothing and no error will be generated if an existing timer is not found.

There can be multiple timers running concurrently if different state machines and/or signals are used. As well, there can be an `SmTimer` and an `SmPeriod` timer running concurrently for the same state machine and signal. The maximum number of timers is specified by an application program in the `SmConfig` statement (see Specifying the State Machine Configuration). When either statement is executed, if there is an existing timer that matches the state machine, signal, and type of timer, then the timeout value is updated (i.e. the timer is effectively restarted with the new timeout value). If there is no existing timer that matches, then a new timer is created (as long as the command is not deleting a timer).

> ⚠ Explicitly deleting either an `SmTimer` or `SmPeriod` timer does *not* cause any events already posted by that timer to be deleted. Any timer events that are still unhandled will remain in state machines' event queues and will be handled doing the normal event processing. If an application requires unhandled timer events to be cleared when a timer is explicitly deleted, then the `SmClear` command must be executed immediately after deleting the timer (i.e. in the same state function).

> ⚠ If the controller is very busy, then timers may not be completely accurate. If the controller is too busy to react immediately to a timeout, then the timeout event will be posted as soon as possible after the timeout has expired. For very short periodic timeouts, this may take longer than two periods. In this case, one or more timeout events will be missed altogether (i.e. rather than several timeout events being posted simultaneously). Also, the period for a periodic timer is restarted when the event is posted and not when the event was originally scheduled to have been posted. For example, if a 5 ms timer is 1 ms late on its second period, then it will post events at times 5, 11, 16, 21 rather than at times 5, 11, 15, 20.

If an application program is paused (i.e. with **Development** → **Break**), then timer event processing is halted. When the program is continued again (i.e. with **Development** → **Continue**), then timer event processing is continued. The effect of this is that repeated timeout events are lost. If a timeout occurs, then a single event will be posted when the program is continued and preceding instances of that event will be discarded.

If multiple timers are triggered simultaneously, then there is no guarantee as to the order in which state machines will receive the timeout events.

> STOP Great care must be taken when using very short periodic timers. If event handling takes too long, then a short period can result in event handling consuming the entire controller and no available time will be left for the rest of the state machine to execute. This will generally result in the event pool being exhausted, causing the controller to enter an error condition and stop all program execution (see State Machine Errors).

The controller will enter an error condition and all program execution will stop in the following cases:

1. An expression is used to specify a state machine ID and the expression evaluates to a value that does not correspond to a valid state machine ID.
2. The state machine that is to receive the event is not currently executing.
3. The signal number expression evaluates to a value that is not a valid signal number.
4. The signal number expression evaluates to any of the values `SIG_ENTRY`, `SIG_EXIT`, `SIG_INIT`, `SIG_START`, or `SIG_IDLE`. These are state machine control events generated by the controller and state machines themselves are not allowed to subscribe to them.
5. There are no timers available.

> 💡 **Example**

> An example of how to use `SmTimer` and `SmPeriod` has been provided in the following sample program:
>
> **File → Sample → State Machine Samples → Operational Events → ISR_Period.mc**

5.5.14 Detecting General Operation Events

The controller can be configured to automatically publish various system-related events. For example, the arrival of a character on the serial interface, the arrival of a CAN message, etc. This is done using the `SmSystem` command. These events will always be placed at the *end* of the existing event queue.

The command has the following syntax pattern:

`SmSystem(keyword, ..., machine, sig-expr);`

where *keyword* specifies the type of event:

`SERIALRECV`	An RS485/RS232 message has been received (see RS485/RS232 Events).
`SERIALCHAR`	A key character has been received on the serial interface (see Serial Interface Events).
`CANRXPDO`	A PDO1 message has been received (see PDO Message Events).
`CANIN`	A CAN message has been received (see General CAN Message Events).
`PFGEVENT`	A Profile Generator event has been received (see Profile Generator Events).
`ETHERNET`	An ETHERNET telegram has been received (see Ethernet events).

When `SmSystem` commands are specified, then the commands are saved in a "System Signal Pool". The total number of simultaneous `SmSystem` commands (for all state machines) that can be saved in the pool at any one time is defined by the user in the `SmConfig` command (see Specifying the State Machine Configuration). However, if state machines "delete" existing commands, then that space is returned to the pool and may be re-used later by subsequent `SmSystem` commands.

In non-state machine application programs, these events are handled using Interrupt Handling routines. However, in state machine programs, no interrupt handlers are called. Instead, state machines must rely on receiving the published events. Note that, unlike normal interrupt handling, no interrupt `ENABLE` or `DISABLE` functionality exists for published events. State machine event handlers must either handle or not handle the events as appropriate.

> `PFGEVENT` events are not supported in non-state machine application programs. They are only available in state machine programs.

For each interrupt handler, there is a corresponding state machine event or command that duplicates the functionality. These are described below.

Interrupt handler	State machine equivalent
ERROR	`SIG_ERROR` event (see Handling General Controller Errors)
PERIOD	`SmPeriod` statement (see State Machine Timers)
TIME	`SmTimer` statement (see State Machine Timers)

PARAM	`SmParam` statement (see Detecting Changes to Parameters)
Position	`SmPosition` statement (see Detecting Motor Positions)
DIGINPUT	`SmInput` statement (see Detecting Digital Input Changes)
STATBIT	`SmParam` statement (see Detecting Changes to Parameters)
SERIALCHAR	`SmSystem` statement (see Serial Interface Events)
CANINPUT	`SmSystem` statement (see General CAN Message Events)
ETHERNET	`SmSystem` statement (see Ethernet events)

5.5.14.1 RS485/RS232 Events

To request the generation of events when RS485/RS232 telegrams arrive, use the following command:

```
SmSystem(SERIALRECV, machine, sig-expr);
```

where: *machine* The state machine which is to receive the event. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then the events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.

sig-expr An integer expression that gives the signal number of the event to be sent.

The events are expected to be defined as follows:

```
SmEvent signal { }
```

It is possible to execute multiple `SmSystem(SERIALRECV)` commands as long as each combination of *machine* and *sig-expr* is unique.

If the events are no longer needed, then they can be disabled using the command:

```
SmSystem(DELETE, SERIALRECV, machine, sig-expr);
```

Note that *machine* and *sig-expr* must *exactly* match the original values specified, in order to allow the correct command to be identified. An error is generated if the specified command cannot be found.

Applications must execute the SerialSetup command to initialize the RS485/RS232 interface handler *before* events will be generated. `SmSystem` commands may be executed either before or after the SerialSetup command has been executed.

When a state machine receives the event indicating that a telegram has been received, then the event handler function for that event is expected to execute the SerialReadComm command to read the message.

> ⚠ The SerialReadComm command may only be executed once, even if multiple `SmSystem(SERIALRECV)` commands have been executed. Executing the SerialReadComm command clears the latest telegram and the command cannot be used again until the next telegram is received.

> ⚠ Unlike some other interrupts, the CANINPUT -1 interrupt cannot be used simultaneously with the `SmSystem(SERIALRECV)` command, even if the SM_RUN_INTERRUPT flag is set (see Specifying the State

Machine Configuration). If `SmSystem(SERIALRECV)` events are enabled, then CANINPUT -1 interrupts will *not* be generated.

Example

An example of how to use `SmSystem(SERIALRECV)` has been provided in the following sample program:

File → Sample → State Machine Samples → Operational Events → ISR_Dcp.mc

5.5.14.2 Serial Interface Events

To request the generation of events when characters are received on the serial interface, use the following command:

```
SmSystem(SERIALCHAR, char, machine, sig-expr);
```

where:	*char*	If the received character matches this character, then an event is generated. Note that this can either be a constant (e.g. 'X'), an integer expression that gives the character, or the '*' character (without quotes). If '*' is used, then events will be generated for all characters.
	machine	The state machine which is to receive the event. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then the events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.
	sig-expr	An integer expression that gives the signal number of the event to be sent.

The events are expected to be defined as follows:

```
SmEvent signal { char }
```

where:	*char*	The key character that was pressed.

It is possible to execute multiple `SmSystem SERIALCHAR` commands as long as each combination of *char*, *machine*, and *sig-expr* is unique.

If the events are no longer needed, then they can be disabled using the command:

```
SmSystem(DELETE, SERIALCHAR, char, machine, sig-expr);
```

Note that *char*, *machine*, and *sig-expr* must *exactly* match the original values specified, in order to allow the correct command to be identified. An error is generated if the specified command cannot be found.

In non-state machine programs, the SerialReadChar command must be used to read the received character after the SERIALCHAR interrupt has been received. In state machine programs, SerialReadChar is not required and should not be used; the received character will be passed to the event handler function as an event parameter.

If the SM_RUN_INTERRUPT flag is set (see Specifying the State Machine Configuration), then `SmSystem`

> SERIALCHAR commands and SERIALCHAR interrupts can be used simultaneously.

> **Example**
>
> An example of how to use SmSystem SERIALCHAR has been provided in the following sample program:
>
> **File → Sample → State Machine Samples → Operational Events → ISR_Keypressed.mc**

5.5.14.3 PDO Message Events

To request the generation of events when PDO1 messages are received, use the following command:

```
SmSystem(CANRXPDO, pdo, node-id, machine, sig-expr);
```

where:	*pdo*	At the moment this value must always be 1. Only PDO1 messages can be received with the CANRXPDO command.
	node-id	If the node ID of the PDO1 matches this value, then an event is generated. Note that this can either be an integer expression that gives the value or the '*' character (without quotes). If '*' is used, then events will be generated when a message is received from any node ID.
	machine	The state machine which is to receive the event. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then the events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.
	sig-expr	An integer expression that gives the signal number of the event to be sent.

The events are expected to be defined as follows:

```
SmEvent signal { byte03, byte47, node-id }
```

where:	*byte03*	Bytes 0-3 of the received PDO message.
	byte47	Bytes 4-7 of the received PDO message.
	node-id	The node ID of the device sending the message.

It is possible to execute multiple SmSystem CANRXPDO commands as long as each combination of *pdo*, *node-id*, *machine*, and *sig-expr* is unique.

If the events are no longer needed, then they can be disabled using the command:

```
SmSystem(DELETE, CANRXPDO, pdo, node-id, machine, sig-expr);
```

Note that *pdo*, *node-id*, *machine*, and *sig-expr* must *exactly* match the original values specified, in order to allow the correct command to be identified. An error is generated if the specified command cannot be found.

Applications must execute the CanInit command to initialize the CAN interface handler *before* events will be generated. SmSystem commands may be executed either before or after the CanInit command has been executed.

In non-state machine programs, this corresponds to the CANINPUT 100*n command and the CanIn command may be used to read the received message after the interrupt has been received. In state machine programs, the CanIn command is not required; the received 32-bit values will be passed to the event handler function as event parameters.

If the SM_RUN_INTERRUPT flag is set (see Specifying the State Machine Configuration), then `SmSystem CANRXPDO` commands and CANINPUT interrupts can be used simultaneously.

Example

An example of how to use `SmSystem CANRXPDO` has been provided in the following sample program:

File → Sample → State Machine Samples → Operational Events → ISR_CanPdo.m

5.5.14.4 General CAN Message Events

To request the generation of events when non-global CAN messages are received, use the following command:

```
SmSystem(CANIN, objno, machine, sig-expr);
```

where:

objno	This is the object number returned by the DefCanIn command. Note that the '*' character (meaning "All") is not supported.
machine	The state machine which is to receive the event. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then the events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.
sig-expr	An integer expression that gives the signal number of the event to be sent.

The events are expected to be defined as follows:

```
SmEvent signal { byte03, byte47, objno }
```

where:

byte03	Bytes 0-3 of the received CAN message.
byte47	Bytes 4-7 of the received CAN message.
objno	The object number returned by the DefCanIn command.

It is possible to execute multiple `SmSystem CANIN` commands as long as each combination of *objno*, *machine*, and *sig-expr* is unique.

If the events are no longer needed, then they can be disabled using the command:

```
SmSystem(DELETE, CANIN, objno, machine, sig-expr);
```

Note that *objno*, *machine*, and *sig-expr* must *exactly* match the original values specified, in order to allow the correct command to be identified. An error is generated if the specified command cannot be found.

Applications must execute the DefCanIn command to create the handle used to identify messages *before* events will be generated. `SmSystem` commands must be executed after the DEFCANIN command has been executed since they require the handle value.

In non-state machine programs, this corresponds to the CanIn command. In state machine programs, the CANIN command is not required; the received 32-bit values will be passed to the event handler function as event parameters.

`SmSystem CANIN` should not be used concurrently with the LinkPdo command to receive PDO messages from the same source.

Example

An example of how to use `SmSystem CANIN` has been provided in the following sample program:

File → Sample → State Machine Samples → Operational Events → ISR_CanIn.mc

5.5.14.5 Profile Generator Events

To request the generation of events by the Profile Generator, use the following command:

```
SmSystem(PFGEVENT, axis-expr, type-expr, machine, sig-expr);
```

where:

- *axis-expr* — If the event occurs with this axis, then an event is generated. Note that this can either be an integer expression that gives an axis number or the '*' character (without quotes). If '*' is used, then events will be generated when the event occurs with any axis. For AposcC applications, axes must be specified using 0-based axis modules numbers.
- *type-expr* — An integer expression that specifies the type of event that is of interest (see below). Note that this can either be an integer expression that gives a type or the '*' character (without quotes). If '*' is used, then events will be generated when any Profile Generator event occurs.
- *machine* — The state machine which is to receive the event. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then the events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.
- *sig-expr* — An integer expression that gives the signal number of the event to be sent.

The events are expected to be defined as follows:

```
SmEvent signal { axis, type, value }
```

where:

- *axis* — The axis for which the event was generated. For AposcC applications, axes will be specified using 0-based axis modules numbers.
- *type* — The type of event (see above) that was generated.
- *value* — A value specific to the type of event that was generated (see below).

It is possible to execute multiple `SmSystem PFGEVENT` commands as long as each combination of *axis-expr*, *type-expr*, *machine*, and *sig-expr* is unique.

If the events are no longer needed, then they can be disabled using the command:

```
SmSystem(DELETE, PFGEVENT, axis-expr, type-expr, machine, sig-expr);
```

Note that *axis-expr*, *type-expr*, *machine*, and *sig-expr* must *exactly* match the original values specified, in order to allow the correct command to be identified. An error is generated if the specified command cannot be found.

Event Types

The event type is handled as follows. #defines have been provided for these values in the "SysDef.mi" include file.

Event Type	Meaning
1 - SM_PFG_PATHMOVEDONE	A PathMove movement has stopped. The event value will consist of one or more of the following flags: 0x01 - SM_PATHSTOP_USER — PathStop was executed. 0x02 - SM_PATHSTOP_MOVE — The PathMove command reached its target position. Note that if movement is *already* in the final deceleration phase towards the target position, then PATHSTOP cannot stop any sooner and the target position will be reached. In this case, both flags will be set.
2 - SM_PFG_PATHSTARTDONE	A PathStart movement has reached the end of the start curve. The event value is not used. It will be set to 0. If the end of the start curve corresponds to a fixed point, then this event is generated *before* the **SM_PFG_PATHFIXPNT** event.
3 - SM_PFG_PATHFIXPNT	A PathStart movement has arrived at a fixed point. The event value will give the 0-based fixed point number of the corresponding fixed point. If movement stops at this point, then this event is generated *before* the **SM_PFG_PATHSTOP** event. If movement stops because it reaches the end of a non-cyclic path, then this event is generated for the final fixed point in the path. For cyclic paths and cases where PathNext has defined a next path, this event is generated for the *first* fixed point of the *next* cycle; an event is *not* generated for the final fixed point of the current cycle. The fixed point number may be used to find the fixed point in the original Path Array with the help of the **PATH_CurveFixpointIndex** macro (see Program Access to Paths).
4 - SM_PFG_PATHSTOP	A PathStart movement has stopped. The event value will consist of one or more of the following flags: 0x01 - SM_PATHSTOP_USER — PathStop was executed. 0x04 - SM_PATHSTOP_SYNC — Path movement stopped at the synchronization point. This flag is generated *only* if movement stops at the synchronization point after reaching the end of the start curve. It is *not* generated if cyclic paths are being used and movement stops at the synchronization point on subsequent cycles. 0x08 - SM_PATHSTOP_FIX — Path movement stopped at a fixed point. The previous **SM_PFG_PATHFIXPNT** event will indicate the fixed point number at which the stop occurred. 0x10 - SM_PATHSTOP_END — Path movement stopped at the end of the current active path curve. This will happen when the curve is non-cyclic and there is no pending curve. If the PathStop command is executed and movement is already stopped (i.e. -1 is specified or the position has already been passed), then an **SM_PFG_PATHSTOP** event is generated immediately, even though a previous **SM_PFG_PATHSTOP**

event will *already* have been generated for this stop position. This allows applications to more easily handle situations where a stop is required but movement may or may not already be stopped. The application will always receive at least one **SM_PFG_PATHSTOP** event after PATHSTOP has been executed.

5 - `SM_PFG_PATHCONSTVEL` A PathStart movement has reached a constant velocity. The event value is not used. It will be set to 0. Note that this is generated only for movements that are expected to continue indefinitely (i.e. for cyclic paths that contain no stop points).

6 - `SM_PFG_PATHCYCLE` The current path curve is complete and the next path has been started. The event value will give the cycle count for the path cycle that has just started. This will be 1 for the first path after the PathStart command and it will increment for each subsequent path that is started. This event is generated *before* the first **SM_PFG_PATHFIXPNT** event of the new path curve.

> If the PathNext command is being used to set different path curves, then application applications can use this event to track when pending curves are activated. This is important if fixed point numbers from **SM_PFG_PATHFIXPNT** are being used since these numbers refer to the active path. The **SM_PFG_PATHFIXPNT** event is generated only *after* the corresponding path has been activated.

7 - `SM_PFG_PATHTERMINATE` Kinematics mode has now been terminated and the axes are now free to be moved individually (e.g. with the AxisPosAbsStart command). Kinematics mode is entered when the KinematicsSetup command is executed and terminated when the AxisControl command is used to either stop or turn off the motors.

> Event types 1-7 (i.e. the path-related events) are generated *only* for the first axis that is specified with the KinematicsSetup command.

> **Following paths using PathStart and PathNext**
>
> If PATHSTART is used to start a path *without stopping at stop points*, then events will be generated in the following order:
>
> 1. **SM_PFG_PATHSTARTDONE** - This is generated *before* the first SM_PFG_PATHFIXPNT event.
> 2. **SM_PFG_PATHFIXPNT** - One event is generated for each fixed point in the path when movement arrives at, or passes, the fixed point. Note that when movement cycles onto a new path, the **SM_PFG_PATHFIXPNT** event is *not* generated for the final fixed point in the current path. Instead, the event will be generated for the *first* fixed point in the new path.
> 3. **SM_PFG_PATHCYCLE** - If the path curve is cyclic, or a next path has been defined, then this event is generated just as movement transfers onto the new path curve. This is generated *before* the first **SM_PFG_PATHFIXPNT** event of the new path curve.
> 4. **SM_PFG_PATHSTOP** - If the path curve is non-cyclic and a next path has not been defined, then movement will stop when it reaches the end of the path and this event will be generated. Note that in this one case, an **SM_PFG_PATHFIXPNT** event is generated for the final fixed point in the path. It is generated *before* the **SM_PFG_PATHSTOP** event.
>
> If PATHSTART is used to start a path *and "Stop" points are being used*, then events will be generated in the order described above, with the following difference:
>
> - **SM_PFG_PATHSTOP** - If a fixed point is also a "Stop" point, then the **SM_PFG_PATHSTOP** event is generated *immediately after* the **SM_PFG_PATHFIXPNT** event for that fixed point. Note that the last fixed point in a path curve cannot be a "Stop" point.
>
> If all possible events are generated, then the event order will be similar to:
>
> ```
> SM_PFG_PATHSTARTDONE
>  SM_PFG_PATHFIXPNT
> ```

```
                SM_PFG_PATHSTOP
            SM_PFG_PATHFIXPNT
                SM_PFG_PATHSTOP
            SM_PFG_PATHFIXPNT (fixed point prior to final fixed point)
                SM_PFG_PATHSTOP
        SM_PFG_PATHCYCLE
            SM_PFG_PATHFIXPNT (first fixed point in path)
                SM_PFG_PATHSTOP
            SM_PFG_PATHFIXPNT
                SM_PFG_PATHSTOP
            SM_PFG_PATHFIXPNT
                SM_PFG_PATHSTOP
```

Example

An example of how to program a state machine to follow path curves using `SmSystem PFGEVENT`, has been provided as a standard ApossC sample program. Please see Path Curve Sample.

5.5.14.6 Ethernet events

To request the generation of events when telegrams are received on the ethernet interface, use the following command:

`SmSystem(ETHERNET, handle, machine, sig-expr);`

where:	*handle*	This is the handle returned by the EthernetOpenClient or EthernetOpenServer command.
	machine	The state machine which is to receive the event. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then the events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.
	sig-expr	An integer expression that gives the signal number of the event to be sent.

The events are expected to be defined as follows:

`SmEvent signal { handle }`

where:	*handle*	The handle returned by the EthernetOpenClient or EthernetOpenServer command.

It is possible to execute multiple `SmSystem ETHERNET` commands as long as each combination of *handle*, *machine*, and *sig-expr* is unique.

If the events are no longer needed, then they can be disabled using the command:

`SmSystem(DELETE, ETHERNET, handle, machine, sig-expr);`

Applications must execute the EthernetOpenClient or EthernetOpenServer command to initialize the ethernet interface handler *before* events will be generated. `SmSystem` commands may be executed either before or after the EthernetOpenClient or EthernetOpenServer command has been executed.

When a state machine receives the event indicating that a telegram has been received, then the event handler function for that event is expected to execute the EthernetReceiveTelegram command to read the message.

Note that *handle*, *machine*, and *sig-expr* must *exactly* match the original values specified, in order to allow the correct command to be identified. An error is generated if the specified command cannot be found.

> ⚠ Unlike some other interrupts, the ETHERNET interrupt cannot be used simultaneously with the `SmSystem(ETHERNET)` command, even if the SM_RUN_INTERRUPT flag is set (see Specifying the State Machine Configuration). If `SmSystem(ETHERNET)` events are enabled, then ETHERNET interrupts will *not* be generated.

5.5.15 Detecting Changes to Parameters

A mechanism is provided that allows state machines to receive events when arbitrary system parameters or user values change. This can be done using the following command:

`SmParam(param-expr, mask-expr, test-expr, machine, sig-expr);`

where: *param-expr* An integer expression that gives the parameter number to track. Parameter numbers may be specified explicitly using the following hexadecimal format:

```
0x01nnnnss
```

where "`nnnn`" and "`ss`" are the SDO index and subindex of the parameter. For example, 0x01220105 represents User Parameter #5.

mask-expr An integer expression that gives a mask value that is applied to the parameter value before testing.

test-expr An integer expression that specifies the type of test to be applied.

machine The state machine which is to receive the event. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '`*`' character (without quotes). If '`*`' is used, then the events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.

sig-expr An integer expression that gives the signal number of the event to be sent.

The mask value and test type are handled as follows. #defines have been provided for the test type values in the "sysdef.mi" include file.

Test Type	Meaning
0 - `SM_PARAM_CHANGE`	An event is generated whenever the parameter value changes. The mask value is ignored.
1 - `SM_PARAM_EQUAL`	An event is generated whenever the parameter value changes to the mask value.
2 - `SM_PARAM_NOTEQUAL`	An event is generated whenever the parameter value

	changes to a value different from the mask value. No event is generated unless the previous value is the same as the mask value.
3 - `SM_PARAM_MASK`	An event is generated if any bit in the parameter value corresponding to a 1-bit in the mask changes.
4 - `SM_PARAM_RISING`	An event is generated if the bit in the parameter value corresponding to the mask changes from 0 to 1. The mask value is a bit number in the range 0-31.
5 - `SM_PARAM_FALLING`	An event is generated if the bit in the parameter value corresponding to the mask changes from 1 to 0. The mask value is a bit number in the range 0-31.
6 - `SM_PARAM_GT`	An event is generated whenever the parameter value changes to a value greater than the mask value. No event is generated unless the previous value is less than or equal to the mask value.
7 - `SM_PARAM_GE`	An event is generated whenever the parameter value changes to a value greater than or equal to the mask value. No event is generated unless the previous value is less than the mask value.
8 - `SM_PARAM_LT`	An event is generated whenever the parameter value changes to a value less than the mask value. No event is generated unless the previous value is greater than or equal to the mask value.
9 - `SM_PARAM_LE`	An event is generated whenever the parameter value changes to a value less than or equal to the mask value. No event is generated unless the previous value is greater than the mask value.

The events are expected to be defined as follows:

```
SmEvent signal { param, value }
```

where: *param* The ID of the parameter that changed. This is the *param-expr* value from the original `SmParam` command.

value The new value of the parameter.

It is possible to execute multiple `SmParam` commands using different values for each of the command arguments. In this case, events will be received if *any* of the specified parameters change. The command may not be executed more than once for the same set of command arguments.

> ⚠ If multiple parameter values change at the same time, then state machines *cannot* rely on any specific order in which the events will be generated.

When `SmParam` commands are specified, then the commands are saved in a system "Parameter Pool". Note that `SmInput` commands (see Detecting Digital Input Changes) are also saved in the same pool. The total number of

simultaneous SmParam and SmInput commands (for all state machines) that can be saved in the pool at any one time is defined by the user in the SmConfig command (see Specifying the State Machine Configuration). However, if state machines "delete" existing commands, then that space is returned to the pool and may be re-used later by subsequent SmParam commands.

Existing SmParam commands may be deleted using the following command:

```
SmParam(DELETE, param-expr, mask-expr, test-expr, machine, sig-expr);
```

This will find the specified command and remove it from the list of parameters being watched. An error is generated if the specified command cannot be found.

The state machine functionality differs from the PARAM interrupt in the following ways:

1. The total number of parameters that may be watched simultaneously is not limited to 10. PARAM limits the number to 10.
2. Any SDO may be used as a parameter. PARAM cannot be used with many SDO values (e.g. the axis status SDO 0x2500/123).

Example

An example of how to use SmParam has been provided in the following sample program:

File → Sample → State Machine Samples → Operational Events → ISR_Param.mc

5.5.16 Detecting Digital Input Changes

A mechanism is provided that allows state machines to receive events when digital inputs change state. This can be done using the following command:

```
SmInput(input-expr, test-expr, machine, sig-expr);
```

where:	*input-expr*	An integer expression that gives the virtual digital input number to track.
	test-expr	An integer expression that specifies the type of test to be applied.
	machine	The state machine which is to receive the event. Note that this can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then the events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.
	sig-expr	An integer expression that gives the signal number of the event to be sent.

The SmInput command uses the virtual digital input SDO 0x2D0- to watch for input changes. This means the virtual digital inputs must be configured *before* SmInput can be used. However, it also means that any input that can be configured as a virtual input may be tested.

The test type is handled as follows. #defines have been provided for the test type values in the "sysdef.mi" include file.

Test Type	Meaning

0 - `SM_INPUT_CHANGE`	An event is generated whenever the input changes.
1 - `SM_INPUT_RISING`	An event is generated if the input changes from 0 to 1.
2 - `SM_INPUT_FALLING`	An event is generated if the input changes from 1 to 0.

The events are expected to be defined as follows:

```
SmEvent signal { param, value }
```

where: *param* The virtual digital input SDO (formated as `0x01nnnnss`) used to test for the input.

value The value of the SDO.

It is possible to execute multiple `SmInput` commands using different values for each of the command arguments. In this case, events will be received if *any* of the specified inputs change. The command may not be executed more than once for the same set of command arguments.

> ⚠ If multiple inputs change at the same time, then state machines *cannot* rely on any specific order in which the events will be generated.

When `SmInput` commands are specified, then the commands are saved in a system "Parameter Pool". Note that `SmParam` commands (see Detecting Changes to Parameters) are also saved in the same pool. The total number of simultaneous `SmInput` and `SmParam` commands (for all state machines) that can be saved in the pool at any one time is defined by the user in the `SmConfig` command (see Specifying the State Machine Configuration). However, if state machines "delete" existing commands, then that space is returned to the pool and may be re-used later by subsequent `SmParam` commands.

Existing `SmInput` commands may be deleted using the following command:

```
SmInput(DELETE, input-expr, test-expr, machine, sig-expr);
```

This will find the specified command and remove it from the list of inputs being watched. An error is generated if the specified command cannot be found.

The state machine functionality differs from the DIGINPUT interrupt in the following ways:

1. The total number of inputs that may be watched simultaneously is not limited. DIGINPUT limits the number to 32.

> **Example**
>
> An example of how to use `SmInput` has been provided in the following sample program:
>
> **File → Sample → State Machine Samples → Operational Events → ISR_Input.mc**

5.5.17 Detecting Motor Positions

A mechanism is provided that allows state machines to receive events when an axis motor position reaches or passes any desired position. This can be done using the following command:

```
SmPosition(sign postype, axis, position, machine, sig-expr);
SmPosition(sign postype, axis, position, machine, sig-expr, ONCE);
```

where: *sign* This can be either + or − representing passing the specified position in either the forward/positive direction or the reverse/negative direction respectively. If the sign is omitted, then the forward direction is assumed.

postype The position type can be one of APOS, IPOS, MAPOS, MCPOS, or MIPOS.

APOS - When the slave position is passed.
IPOS - When the distance between the last marker position and the actual position is reached.
MAPOS - When the master actual position [mu] is passed.
MCPOS - When the master command position [mu] is passed.
MIPOS - When the distance between two markers is reached.

axis For ApossC applications, axes must be specified using 0-based axis modules numbers.

X(*n*) Optional axis number. The axis will default to 1 if this is omitted.

position Position at which to trigger the event. For APOS and IPOS, this will be in user units (UU). For MAPOS and MIPOS, this will be in master user units (MU). For MCPOS, this will be in curve units (CU).

machine The state machine which is to receive the event. Note that "*machine*" can either be the name of a state machine, an integer expression that gives the ID of a state machine, or the '*' character (without quotes). If '*' is used, then timer events will be published instead of posted. In this case, state machines must subscribe to the events in order to receive them.

sig-expr An integer expression that gives the signal number of the event to be sent. Note that predefined signals (e.g. SIG_ENTRY) cannot be specified; only user-defined signals are allowed.

ONCE The event will only be generated once.

For the forward direction, the event will be generated when the motor first reaches the specified position. For the reverse direction, the event will be generated when the motor leaves the specified position. Regardless of whether or not parameters have been defined in the SmEvent statement for the signal, the generated events will have two event parameters: the axis number and the specified motor position.

The events are expected to be defined by the user as follows:

```
SmEvent signal { axis, position }
```

where: *axis* The axis causing the event. For ApossC applications, axes will be specified using 0-based axis modules numbers.

position The position at the time the event was generated.

If the ONCE parameter or the SmPositionOnce command is specified, then the event will only be generated once.

It is possible to specify multiple SmPosition and SmPositionOnce commands and they will all be active simultaneously. Commands (using the same signal or different signals) may be specified for each different position type and for each direction (forward and reverse). As well, multiple commands can be used to specify different positions. However, all commands corresponding to a single position type and direction must have unique positions; it is not possible to specify multiple signals to be triggered at the same position. For example, it is possible to specify +APOS and +IPOS with the same position or +APOS and -APOS with the same position, but is it not possible to specify +APOS and +APOS with the same position and different signals. Specifying the same position type, direction, and position in a command will cause the previous command to be overwritten.

Each SmPosition and SmPositionOnce command that specifies a unique position type and position (but not direction) will use a record from the "system position pool". The size of this pool is defined in the SmConfig statement (see Specifying the State Machine Configuration). Users must determine an adequate size for this pool based on how many simultaneous SmPosition commands their program will require. Note that if commands are no longer needed and are deleted, then the records are returned to the pool and may be reused for subsequent SmPosition commands.

As with the Position interrupt, the SmPosition command allows "CAM boxes" to be simulated by automatically setting and clearing digital outputs. This is done using either of the following commands:

```
SmPosition(sign postype, axis, position, SETOUT, output);
SmPosition(sign postype, axis, position, SETOUT, output, input);
SmPosition(sign postype, axis, position, SETOUT, output, ONCE);
SmPosition(sign postype, axis, position, SETOUT, output, input, ONCE);
```

For details on how setout is used, please refer to SETOUT. Note that this command does not include the *sign* argument; "opposite" actions are taken depending on the direction in which the position is passed. This command also does *not* result in any event being published; the digital output setting and clearing is done entirely autonomously.

If the ONCE parameter or the SmPositionOnce command is specified, then the digital output is only changed once. It will be set high for forward movement and low for backward movement but once changed, it will not be changed again.

Existing SmPosition and SmPositionOnce commands may be deleted using the following commands:

```
SmPosition(DELETE, axis, sig-expr);
SmPosition(DELETE, axis, SETOUT, output);
```

The first command will delete all existing commands that publish "*sig-expr*" events for all position types and all positions of the specified axis. The second command will delete all existing commands that set digital output "*output*" for all position types and all positions of the specified axis. The delete function currently does not have a finer resolution; it is not possible to delete individual SmPosition or SmPositionOnce commands.

Example

An example of how to use SmPosition and SmPositionOnce has been provided in the following sample program:

File → Sample → State Machine Samples → Operational Events → ISR_Position.mc

5.5.18 Detecting Axis Status Changes

Many application programs rely on changes to axis status bits to trigger some action from the controller. In non-state machine programs, this can be done using the STATBIT interrupt. In state machine programs, it can be done using the SmParam statement to trap changes to the axis state SDO PFG_FLAGS (0x2500/123). For example, the following statements will cause a SIG_REACHED signal to be published when axis 1 reaches its target position.

```
SmEvent SIG_REACHED { }
SmParam(AXE_PROCESS_INDEX(0,PFG_FLAGS), SM_STAT_POSREACHED, SM_PARAM_RISING, *,
SIG_REACHED);
```

Note that `#define` statements (e.g. for "SM_STAT_POSREACHED") are available in the system "sysdef.mi" include file for all axis status bits.

Note the following differences between state machine programs and non-state machine programs:

1. State machines can subscribe to all status bit changes with a single command. Non-state machine programs must register an interrupt handler using STATBIT for each individual bit.
2. State machines can subscribe to both "rising" and "falling" events at the same time. With STATBIT, it is only possible to receive either "rising" or "falling" interrupts but not both.

Example

An example of how to use `SmParam` to detect axis status bit changes has been provided in the following sample program:

File → Sample → State Machine Samples → Operational Events → ISR_Statbit.mc

5.5.19 State Machine Errors

If the controller detects any state machine error (see below) during the execution of state machines, then it will put the controller into an error condition with error #57. This will implicitly execute an "`SmStop -1`" or "`SmStop(-1);`" command; state machine execution will be stopped and the `SmRun` command will return a -1 return code. The error will then be treated in the same way as any other controller error: Either the ERROR interrupt routine is executed or, if there is no ERROR routine, then application program execution is stopped, the motor is stopped, etc.

Unlike other controller errors where it may be possible to clear the error and then **Development → Continue** the application program, this will not be possible for state machine programs. The reason for this is that the event queues are not maintained while execution is stopped. As a result, there would be no continuity of the events in the event queue prior to stopping with the events in the queue after continuing.

Other than for the exceptions described below, the 32-bit data value saved with the error log for error #57 (and that can be viewed using the Error History menu command) will be packed as follows:

0xSSSSMMEE

0xSSSS____	Internal system-defined state ID of the currently active state. If required, this state ID number can be found in the compiler cross reference file (see the **Settings → Compiler** command).
0x____MM__	State machine ID of the state machine that was active when the error was detected.
0x_____EE	State machine error number.

The state machine error numbers (low-order byte of the data value) are as follows:

1 (0x01)	State machines are not running. This error is generated if any state machine command other than the `SmRun` command (e.g. `SmPost`, `SmStop`, etc.) is executed when state machines are *not* running. Note that in this case, the 32-bit data value saved with the error log will

always contain the error number only. The state machine ID and state ID will both be set to 0.

2 (0x02) State machines are already running. This error is generated if an `SmRun` command is executed while state machines are already running. Currently, it is not possible to start new state machines while state machine are already running.

3 (0x03) Invalid signal number. This error is generated if an invalid signal is used with any of the `SmPost`, `SmUrgent`, `SmSubscribe`, `SmPublish`, `SmTimer`, `SmPeriod`, or `SmInput` commands.

4 (0x04) Too many event parameters specified. This error is generated if too many event parameters are specified with any of the `SmPost`, `SmUrgent`, or `SmPublish` commands. Note that it is possible to specify "extra" parameter values for those events that define fewer than the maximum number of parameters in the `SmEvent` statement. The maximum number of parameters is the "global" maximum; the maximum number required by any event defined in the application program.

5 (0x05) No free event in the event pool. This error is generated by the `SmPost`, `SmUrgent`, and `SmPublish` commands if the event pool contains no more available events. It will also be generated by the `SmTimer` and `SmPeriod` timers if there are no available events when needed. Note that this error can also be generated by the controller itself since the controller is publishing system events (e.g. `SIG_ERROR`, `SIG_KEYPRESSED`, etc.). If this error happens, then the size of the event pool defined in the `SmConfig` statement should be increased. *However*, the state machine program should also be analyzed to ensure that there is not a problem with events being generated faster than they are being handled.

6 (0x06) No free record in event queue. This error is generated when an event must be queued for a specific state machine but that state machine's event queue is full. In this case, the state machine ID and state ID in the error log will correspond to the state machine owning the event queue (i.e. the state machine to which the event is being queued) and *not* to the currently active state machine. If this error happens, then the size of the event queue defined in the `SmMachine` statement should be increased. *However*, the state machine program should also be analyzed to ensure that there is not a problem with events being generated faster than they are being handled.

7 (0x07) Transition exits top-most state. This error is generated if a state function attempts to execute a transition that would exit the top-most state of the state machine (e.g. executing "`return(SmTrans(<-<-))`" when in the first substate of the top-most state).

8 (0x08) Could not find transition target state. This error is generated if a state

function attempts to execute a transition where the target state does not exist (e.g. executing "`return(SmTrans(<-StateABC->SubstateXYZ))`" where "`SubstateXYZ`" is not a child state of "`StateABC`").

9 (0x09) Cannot transition "up" in a `SIG_INIT` event handler. This error is generated if the state function for the `SIG_INIT` signal executes a transition that exits that state (e.g. "`SIG_INIT = { return(SmTrans(<-)) }`"). If a transition is done for the `SIG_INIT` signal, then the transition must only be to a substate of that state (or any substate of a substate).

10 (0x0A) State machine not found. This error is generated when any state machine command that references a state machine ID number is executed and the ID number does not correspond to any existing state machine.

11 (0x0B) The specified state machine is not running. This error is generated if any of the `SmPost`, `SmUrgent`, `SmPublish`, `SmTimer`, or `SmPeriod` command is executed and references a state machine that is not currently running.

12 (0x0C) No more timers available. This error is generated if either the `SmTimer` or `SmPeriod` command is executed and all the existing timers are already busy. If this error happens, then the number of timers defined in the `SmConfig` statement should be increased. *However*, the state machine program should also be analyzed to ensure that there is not a problem with too many timers being requested simultaneously.

13 (0x0D) No free record in the subscribe pool. This error is generated by the `SmSubscribe` command if a subscribe record is required but the subscribe pool contains no more available records. If this error happens, then the size of the subscribe pool defined in the `SmConfig` statement should be increased. *However*, the state machine program should also be analyzed to ensure that there is not a problem with multiple `SmSubscribe` commands.

14 (0x0E) State machines are not supported by the firmware currently running on the controller. Please contact zub machine control AG for assistance.

15 (0x0F) A state machine has tried to execute any of the various `Delay` or `WAIT` commands. State machine event functions should never execute commands that require any type of "waiting" since this violates state machine architectural rules. Event functions are expected to "run to completion" in a minimal amount of time. Note that the `SmConfig` statement conatins a flag that will allow these commands to be used. However, this should only be done in exceptional cases since it can lead to state machine misbehaviours.

16 (0x10)	There are no available records in the pool for the `SmParam`, `SmInput`, or `SmSystem` command.
17 (0x11)	An identical `SmParam`, `SmInput`, or `SmSystem` record already exists.
18 (0x12)	The specified `SmParam`, `SmInput`, or `SmSystem` record was not found. This error may be disabled by using the SM_RUN_DELDISABLE flag.
19 (0x13)	If this is an `SmParam` command, then it specifies an unknown parameter index or a parameter that cannot be read. If it is an `SmInput` command, then it specifies a virtual digital input that does not exist.
20 (0x14)	The compiled state machine binary code is corrupt. Please contact zub machine control AG (see Contact and Legal Information).

Unlike other controller errors where it may be possible to clear the error and then **Development** → **Continue** the application program, this will not be possible for state machine programs.

5.5.20 Debugging State Machines

State machine execution history can be recorded for later debugging. If recording is enabled, then a record is saved for each event processed by the executing state machines. The recording consists of the following information:

1. The time from the system clock when the processing of the event started.
2. The state machine ID of the state machine that processed the event. This is the ID number that is specified in the `SmMachine` statement.
3. The ID of the current state. This is a number that can be found in the compiler cross reference (.ad$) file for the compiled application program.
4. The ID of the event signal. This number can also be found in the compiler cross reference file.
5. The first two event parameter values.
6. The ID of the state that handled the event. This will either be the current state or a parent of the current state.
7. A set of flags indicating the disposition of the event. Values can be:

0x01	The event was handled.
0x02	The event caused a transition.
0x04	The event was discarded.
0x08	An event handler function will be called.

The recording may be saved to either a DIM array or to dynamic memory.

> ⚠ State machine history *must not* be saved to dynamic memory at the same time that the Oscilloscope is saving curves to dynamic memory. Dynamic memory cannot be used simultaneously for multiple functions. Doing so will cause data corruption errors.

> To record state machine history, a zub motor controller with firmware version 7.1.29 (or later) is required.

Recording may be configured and started either by the user with the **Controller** → **State Machine** → **Configure History** menu command or by the state machine program itself using SDO values (see Program Configuration of History). Recordings may be viewed using the **Controller** → **State Machine** → **Upload History** menu command. They may be viewed either after the program has stopped or while the program is still running.

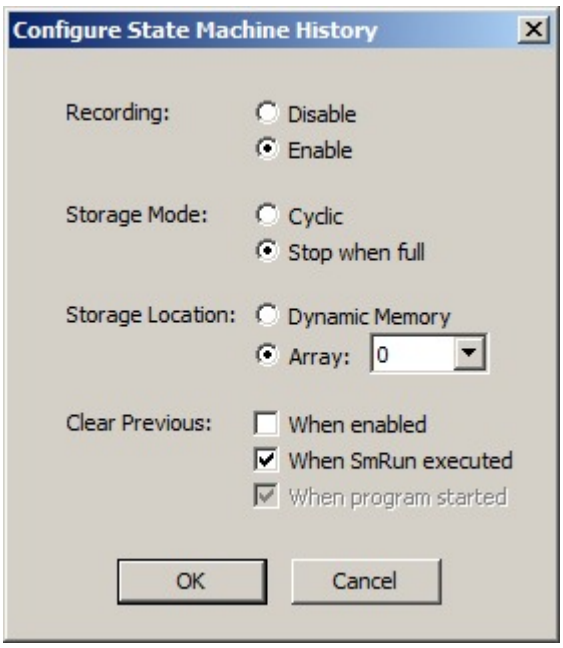
Generally, it is not useful to have a single history recording that spans more than one application program execution. In the case where different application programs are run, this would also cause synchronization problems when the history was viewed. The various ID values in the history records *must* correctly match the application program that recorded them and a single history recording cannot match multiple programs simulateously. Similarly, it may not be useful to have a single history recording span multiple `SmRun` executions or to span multiple recording enable/disable cycles. All of these cases can cause difficulty in interpretting the exact sequence of the history. Hence, when history recording is enabled, an automatic reset is normally done to clear any previously existing history recording. However, should it be necessary, the history recording configuration does provide the user with options to disable the automatic reset.

5.5.20.1 User Configuration of History

The **Controller** → **State Machine** → **Configure History** menu command allows the user to start the recording of state machine history as state machines run. (See Program Configuration of History for information of how the recording can be started from within the state machine program.) This information can then be used later to help debug state machine programs.

History recording may be started or stopped at any time (i.e. before or after an application program has been started or before or after the `SmRun` command is executed). However, since history recordings can become very large, it is anticipated that they be started only after the state machine program is running. The general debugging steps would be:

1. Start the state machine program running.
2. Prepare the test that is to be debugged.
3. Start the history recording.
4. Execute the test.
5. Stop the history recording.
6. View the recording state machine history.



Recording

This value should be interpretted as a *request* to enable or disable history recording. In certain cases (see **Storage Location** below), recording will remain disabled even though this is set to **Enable**. Note that if recording is *already* enabled, then pressing **OK** is equivalent to enabling it again. This may cause previously existing history to be cleared (see **Clear Previous** below).

> If the recording is set to **Disable**, then all other settings are ignored.

Storage Mode

If the storage mode is set to **Cyclic**, then new history records will overwrite old records once the history memory becomes full. With **Stop when full**, history recording will be stopped once the memory becomes full. Note that state machines will continue to run and recording will still be enabled; new history records will simply be discarded.

> If the storage mode is changed, then this will trigger an automatic reset to clear existing history the next time that recording is enabled. The reset will be immediate if **Recording** is set to **Enable**. The reset will be done regardless of the setting of **Clear Previous**.

Storage Location

History recordings can be saved in either dynamic memory or in a DIM array. If dynamic memory is used to record the history, then a maximum of 5000 records will be saved. If a DIM array is used, then the number of records will be determined by the size of the array and the length of each history record.

> If the storage location is changed, then this will trigger an automatic reset to clear existing history the next time that recording is enabled. The reset will be immediate if **Recording** is set to **Enable**. The reset will be done regardless of the setting of **Clear Previous**.

> If the storage mode is set to a DIM array that no longer exists when recording is to be enabled (e.g. the Delete command may have been executed), then the actual recording will remain disabled even if **Recording** is set to **Enable**.

Clear Previous

This setting determines when previously existing history records will be cleared. Previous records should be normally cleared each time that recording is enabled unless there is some specific reason not to do so.

> If **When program started** is selected, then the clear is done only before the application program starts state machines running for the *first* time. The application program itself may be stopped and continued without causing a clear.

5.5.20.2 Program Configuration of History

An application program may configure and start/stop state machine history recording itself by writing to SDO **0x2214/31**. The current configuration settings can be found by reading this SDO. The SDO value has the following format (as a hexadecimal number):

```
0xaaaaffff
```

where: *aaaa* The storage location for the history recording. This should be 0xFFFF for dynamic memory. Otherwise, it should be the DIM array number.

ffff Configuration flags:

0x01	Enable recording.
0x02	Cyclic recording.
0x04	Clear history when program starts.
0x08	Clear history when SmRun executed.
0x10	Clear history when recording enabled.

Example:

```
sysvar[0x0122141F] = 0xFFFF001F;   // Record to dynamic memory, cyclic, always
clear, enable.
sysvar[0x0122141F] = 0x0;          // Disable recording. Flags are ignored so we
can use 0.
```

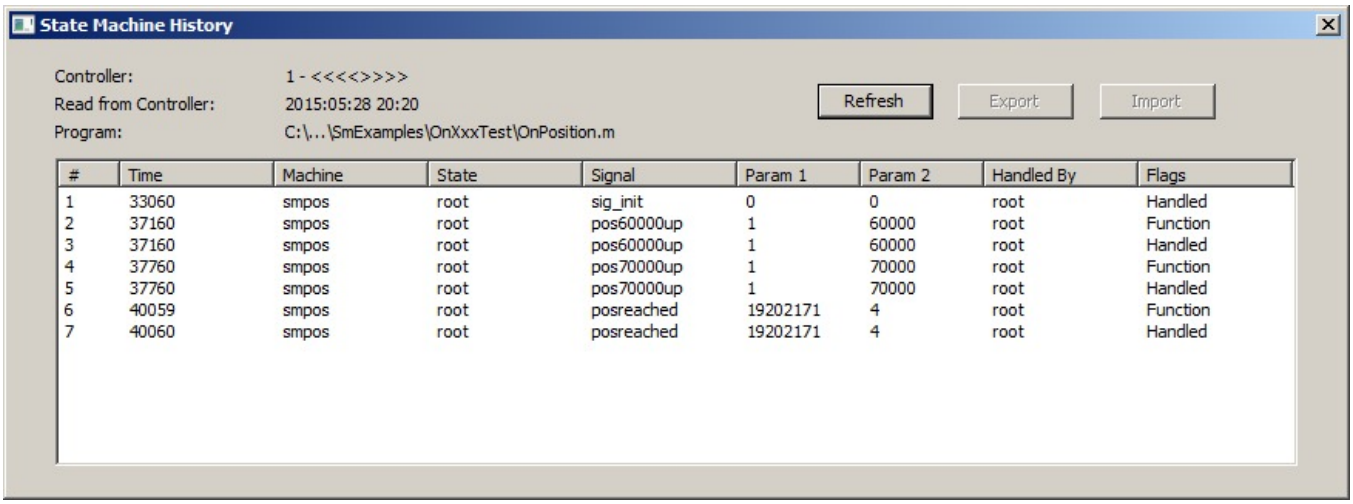
See User Configuration of History for more information on the handling of the storage location and configuration flags.

5.5.20.3 Viewing State Machine History

The **Controller** → **State Machine** → **Upload History** menu command allows the user to upload and view any recorded state hachine history.

> ⚠ The ApossIDE Edit Window *must* contain the application program that generated the history record. If it does not, then the History Window will be unable to correctly decode the history records and the information in the table will be corrupt.

Multiple windows may be opened, even to the same controller. The windows will stay open even when the original Edit Window is closed.



#	Time	Machine	State	Signal	Param 1	Param 2	Handled By	Flags
1	33060	smpos	root	sig_init	0	0	root	Handled
2	37160	smpos	root	pos60000up	1	60000	root	Function
3	37160	smpos	root	pos60000up	1	60000	root	Handled
4	37760	smpos	root	pos70000up	1	70000	root	Function
5	37760	smpos	root	pos70000up	1	70000	root	Handled
6	40059	smpos	root	posreached	19202171	4	root	Function
7	40060	smpos	root	posreached	19202171	4	root	Handled

Each history record represents a single event that was processed by the running state machines and will contain the following information:

#	Sequence number of record.
Time	The controller system clock time at which the event was processed. Note that this is *not* the time at which the event was generated.
Machine	The name of the state machine that processed the event. This name is specified by the `SmMachine` statement (see Defining State Machine Variables).
State	The name of the state that the state machine was in when the event was processed. This name is specified by the `SmState` statement (see Defining the State Hierarchy).
Signal	The name of the event signal. This name is specified by the `SmEvent` statement (see Defining Signals and Event Types).
Param 1, 2	The first two event parameters associated with the event.
Handled By	The name of the state that actually processed the state.
Flags	The flags will give an indication of how the event was handled:
	Handled The event was handled. No additional processing was done.
	Discarded The event was passed up the state heirarchy to the root state and then discarded because it was not handled.
	Transition The event caused a transition to a new state. This record will be followed by a series of zero or more `SIG_EXIT` events and then zero or more `SIG_ENTRY` events. The **Handled By** column will show the path of the transition. The **Handled By** column in the last record in the sequence will show the final target state.
	Function The event caused a handler function to be called. This record will be followed by a sequence of any number of history records, depending on how the function handles the event. For example, the function may end by flagging the event as being "handled" or "not handled" or by doing a transition to another state.

Double-clicking on any of the history records will cause the Edit Window to jump to the state that handled the event. Note that this is not necessarilly the state that the state machine was in when it received the event.

The following functions are also available:

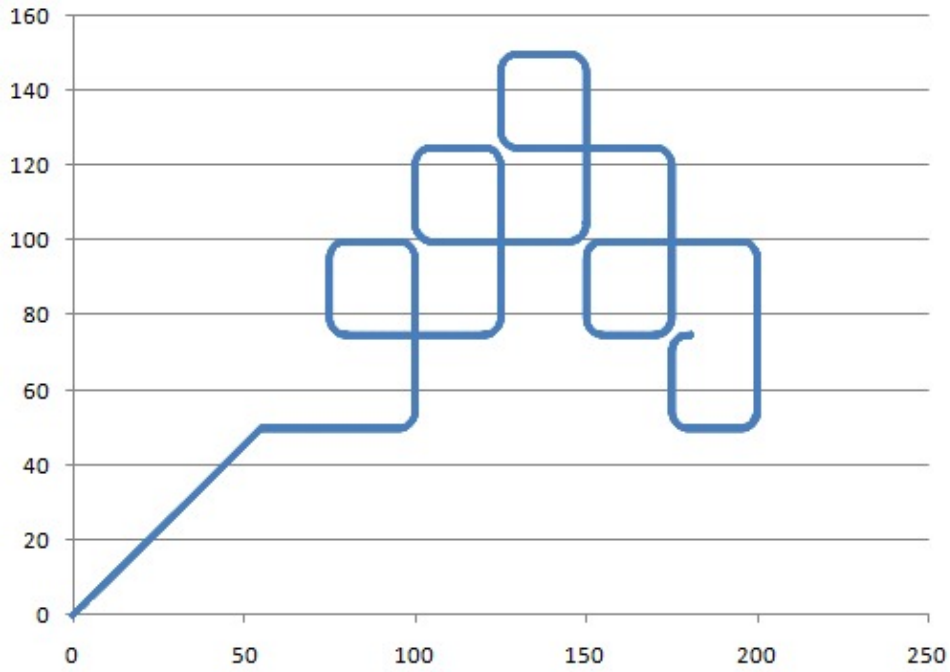
Refresh	Reload the history from the controller.
Export	Not currently implemented.
Import	Not currently implemented.

5.5.21 Sample Programs

5.5.21.1 Path Curve Sample

The Path Curve Sample was created to illustrate how an ApossIDE state machine program may be written to follow path curves. The sample is based on a 2-dimensional pen plotter. The sample will draw a series of 5 "squares" and then stop.

If an actual pen plotter existed, then the drawing should appear as follows:



Please note the following points:

- The sample illustrates *both* state machine concepts in general and path-related state machine events in particular.
- The sample will run either on a real controller or using an Simulator.
- The controller must support at least 3 axes. If a controller is not available, then a simulator may be used.
- The sample is a complete, although simple, application. It will drive a real plotter, if an appropriate plotter is available.
- The plotter is expected to use H-Bot/CoreXY Kinematics. If different kinematics are needed, then the program may be changed quite easily. (Basically, only a different homing procedure would be required.)

To run the sample program:

1. Create a local directory on the hard drive. This will be used to save program files.
2. In the ApossIDE, execute the command:

   File → Sample → State Machine Samples → Path Curves

3. One-by-one, open all files under this heading and save them to the local directory. The compiler cannot embed include files unless they come from files saved on the hard drive.
4. Edit the "PathCurves.m" file in the local directory. This is the main program for the application.
5. Connect to a controller (or Simulator) and execute the program. The program is set up to run with no changes. It will not attempt to run real motors unless it is modified to do so.

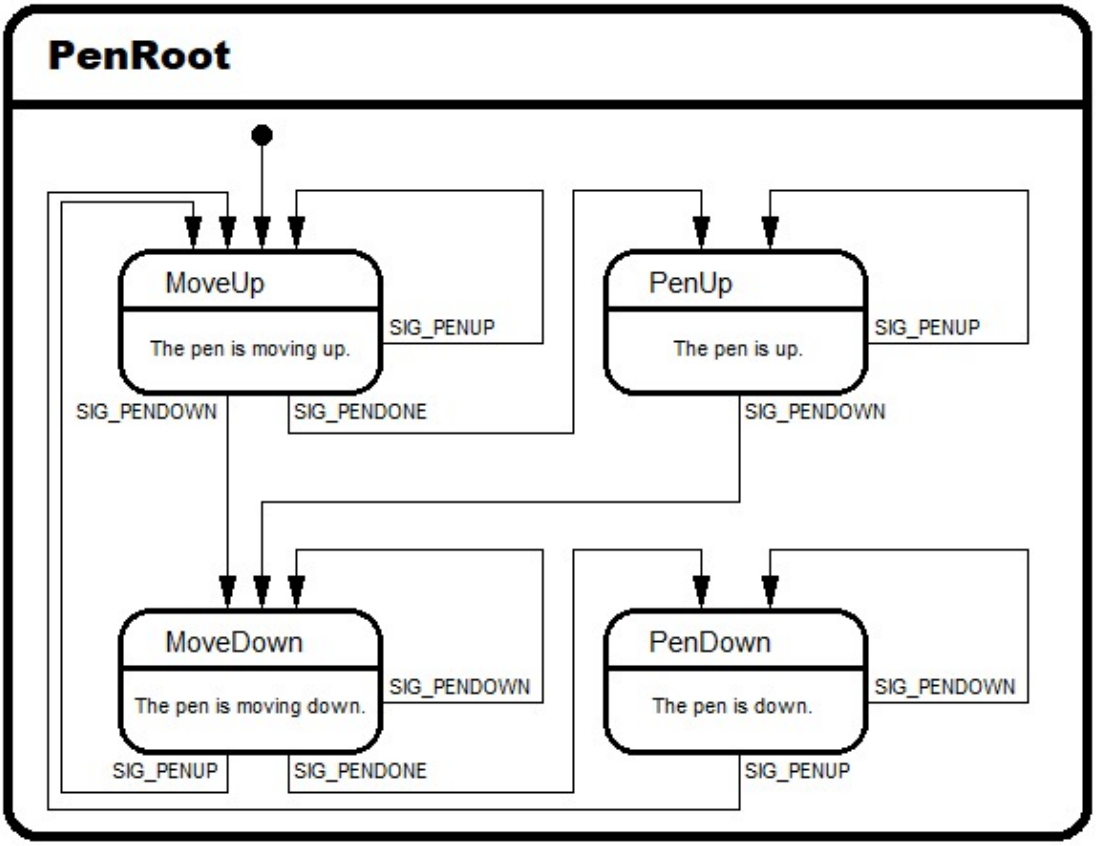
The application is designed to make it possible for users to experiment with different curves, different modes of operation, and different path scenarios. It is important to gain an understanding of the exact behaviour of path-related state machine events, before attempting to create a real application. Please read through the following topics before attempting to modify the Path Curve sample.

State Machine Structure

The application is implemented as two state machines: one to control the pen up/down movements and one to follow

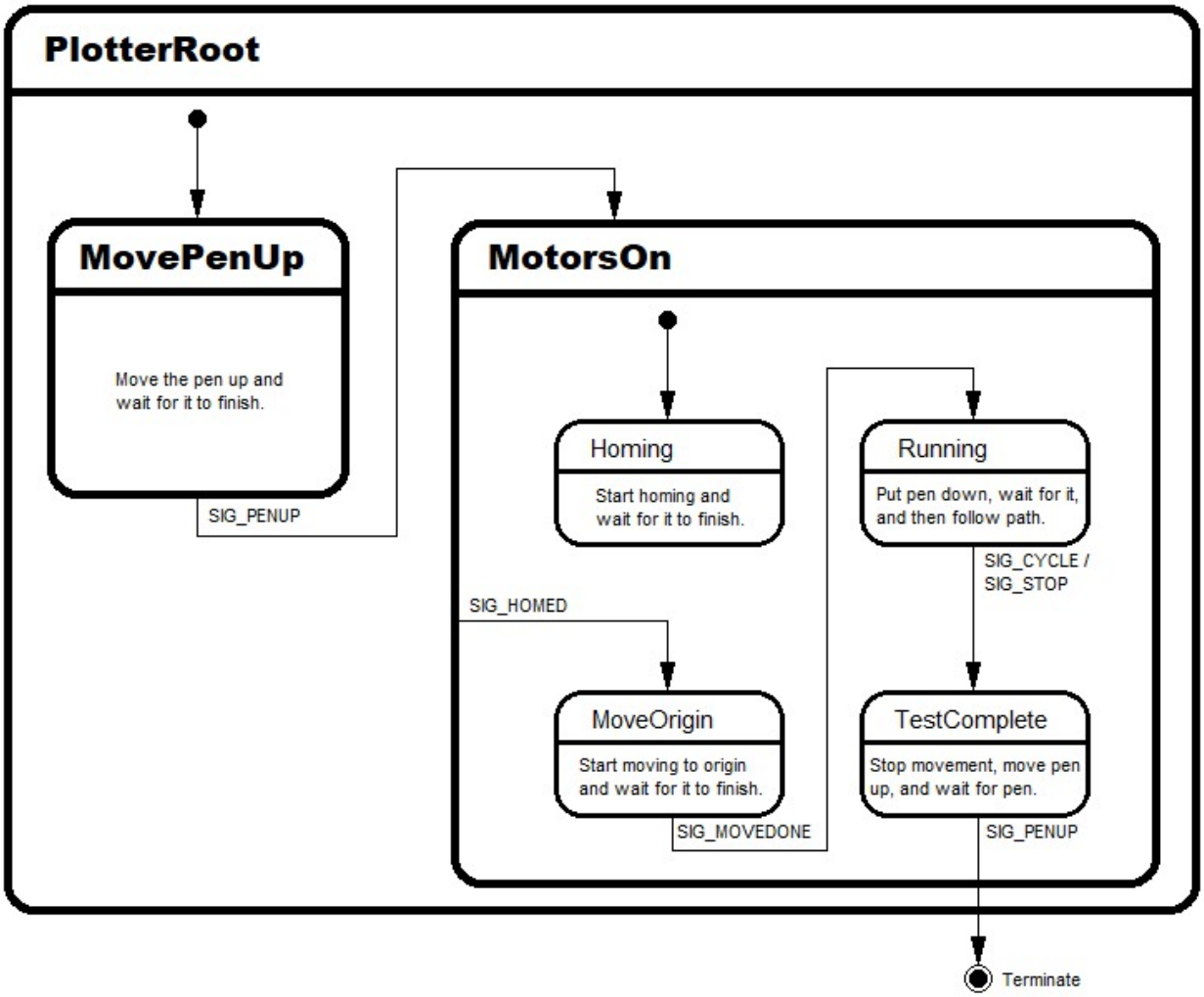
the path curve, along with various "helper" functions. These state machines communicate with each other using application-specific events. The state machines are completely independent; no memory or variables are shared between the two state machines. In addition, the state machines contain no ApossIDE statements that cause delays.

This is the state diagram for the pen up/down state machine:



Its initial action is to move the pen up (in case it is not already up), in order to ensure a known state. Then it waits for pen up/down requests (SIG_PENUP and SIG_PENDOWN) from other state machines. These are the only two external events that the state machine will recognize. Note that the state machine must be able to receive these requests when it is in *any* state. Once the request has been completed, the state machine will post a reply event to the state machine making the request.

This is the state diagram for the path-following state machine:



This state machine's initial action is to request the pen to move up. The purpose of this is *not* to move the pen up (because the Pen state machine already initilizes iteself by moving the pen up) but to ensure that the Plotter state machine waits long enough so that the pen *is* up before it starts moving the plotter mechanism. Once the pen is up, then motors are turned on (by entering the "MotorsOn" state) and homing begins (by entering the "Homing" state). Once homing is complete, then the pen is moved to the paper origin (by entering the "MoveOrigin" state). Once at the paper origin, then the actual drawing will begin (by entering the "Running" state). When the test is complete, the "TestComplete" state is entered, which terminates the test and exits the program. Note that the motors are turned off as the state machine exits the "MotorsOn" state.

The "Running" state contains no substates. It receives and processes all of the various path-related events without actually leaving the state. The "Homing" state contains multiple substates, but these have not been included in the above diagram.

Application File Structure

The sample application consists of the following files:

`PathCurves.mc`	The main application program. This also contains the complete state machine for following the path (i.e. "PlotterRoot").
`Pen.mh`	This include file contains the complete state machine for

handling the pen up/down movements (i.e. "PenRoot").

> Note how include files can be used to implement cpmplete state machines and include them into applications. This same state machine could be used in multiple applications simply by including this file.

`PlotterHome.mh` This include file contains substates that will be used to home the plotter.

> Note how include files can be used to implement portions of state machines and embed them into larger state machines. This method can be used to simplify application development by reducing the need for large monolithic source files. Also note that it provides a compiler-time mechanism for handling such things as machine-specific functions. For example, a different homing procedure could be implemented and easily swapped into the main program with minimal effort.

`zubPlotter.mh` This file contains the helper functions for configuring axis parameters and the #defines for specifying the physical attbributes of the hardware.

`Nanotec_STEP_FOC.mh` This file contains the helper functions for specifically configuring the X and Y axis motors.

`Square.zbc` This configuration file contains the various path curves that are used by the application.

Specifying the Hardware

The sample application contains the following defines that allow some aspects of the test hardware to be specified:

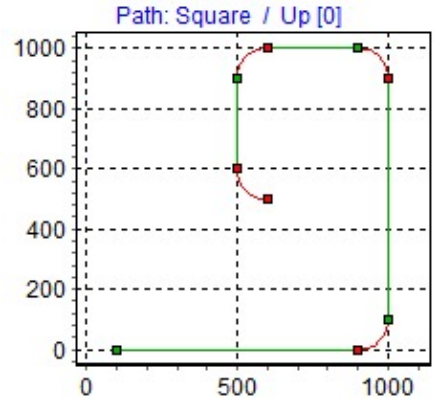
`ZUB_PLOTTER` This specifies that the axes should be configured for the zub demonstration plotter. Although this is currently the only hardware supported by the sample, the define provides a compile-time mechanism to select different hardware. Search for this define within "PathCurves.m" to see how this is done.

`SIMULATION` The application is run in simulation mode (i.e. without real motors) if this is defined. Remove this definition *only* if real plotter hardware is to be used.

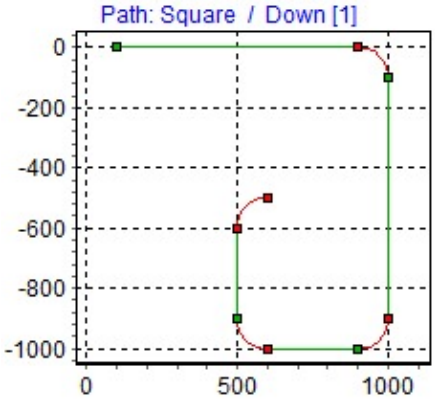
`HOME_ENABLE` If this is defined, then the plotter motors will be homed when the application starts. Otherwise, the startup position is simply defined to be the home position. If simulation mode is being used, then homing need not be enabled (although it can be, if the user would like to simulate homing). If real plotter hardware is being used, then homing should be enabled.

Available Path Curves

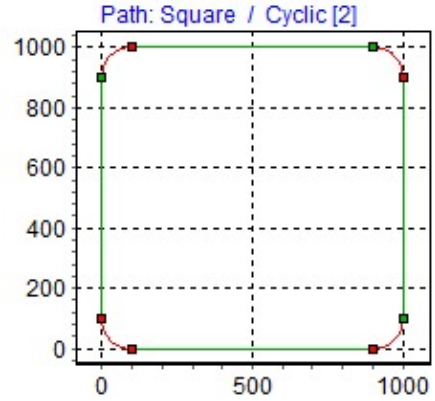
The sample provides several different variations of "square" path curves to choose from. All of these paths are maintained in the "Square.zbc" file. Open this file with the ApossIDE to view or modify the path curves. Note that if the path curves are modified, then it may be necessary to change the DIM statement in "PathCurves.m" if the array size changes. See also CAM Editor Toolbar.



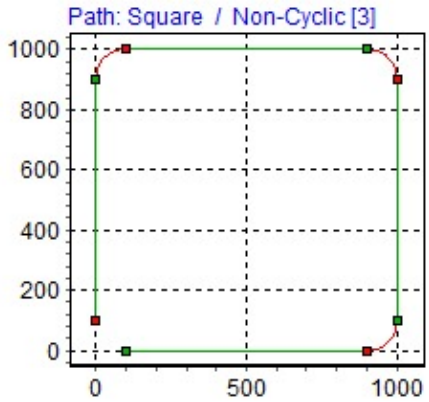
Path 0: This is an "open cyclic" path. This means the path will be repeated if movement continues past the end of the curve. This effectively means that the drawing will "walk up and right" as the curve cycles. The first 3 cycles in the diagram at the beginning of this topic are done with this curve. This curve also defines the fixed point at the beginning of each tangent to be a "Stop" point. This allows users to configure the test with or without stop points.



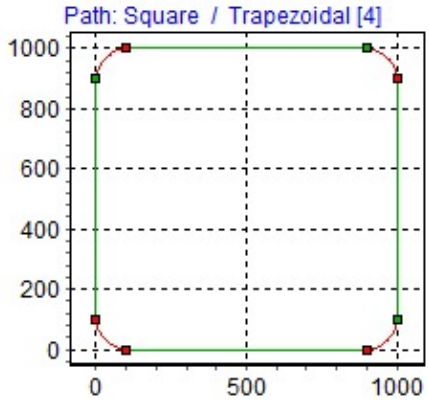
Path 1: This is also an "open cyclic" path. However, this curve will "walk down and right" as the curve cycles. The last 2 cycles in the diagram at the beginning of this topic are done with this curve. There are no "Stop" points defined in this curve.



Path 2: This is a "closed cyclic" path. Since the end point matches the start point, the same path will be redrawn over and over as the curve cycles. There are no "Stop" points defined in this curve.



Path 3: This is a "non-cyclic" path. Note that it is missing the fourth corner segment, so it should not be changed to a cyclic curve. Since this is a non-cyclic curve, movement will automatically stop when it reaches the end of the curve. However, if PathNext has been used to define a pending curve, then movement will automatically switch to the new curve after this curve is complete. There are no "Stop" points defined in this curve.



Path 4: This is a "closed cyclic" path. It is identical to **Path 2** except that the corners have been defined using trapezoidal curves rather than 5'th order curves (see Fix Points Tab).

Configuring the Test Behaviour

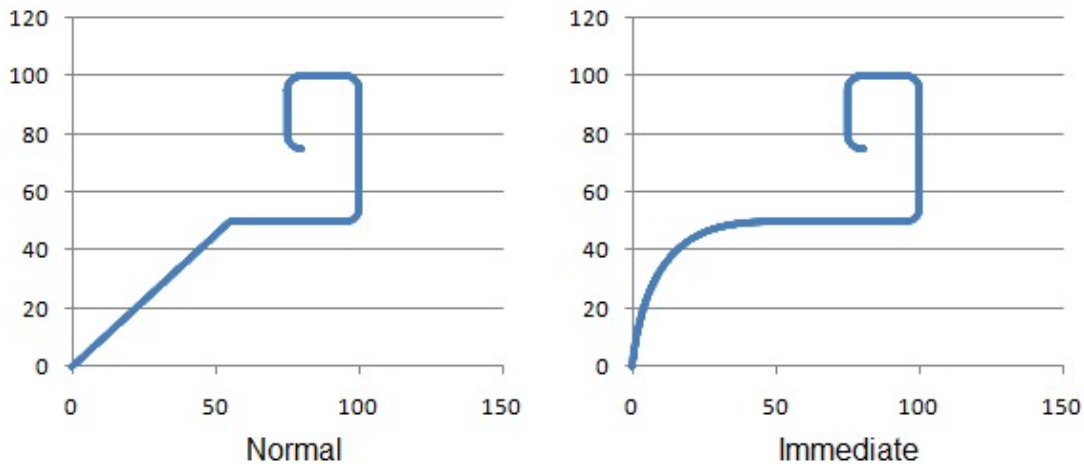
The sample application contains several defines that allow various aspects of the test to be specified. Simply change these defines and rerun the program to see how the behaviour changes.

`MOVETYPE` This define determines the overall behaviour as the curve is being followed (see PathStart). Select from the following options:

- `TYPE_NORMAL` The plotter will move in a straight line from the paper origin to the synchronization point on the curve and then stop. Once stopped, it will immediately start moving again and drawing the square path curve. It will draw continuously without stopping at any "Stop" points.
- `TYPE_STOP` The plotter will move in a straight line from the paper origin to the synchronization point on the curve and then stop. Once stopped, it will immediately start moving again and drawing the square path curve. However, unlike the TYPE_NORMAL option, movement will *also* stop at each "Stop" point. Path 0 (see above) has "Stop" points at the beginning of each tangent section. Once stopped, the plotter will immediately start moving again. Note that an important use of "Stop" points is to allow the plotter to negotiate sharp corners in path curves. This behaviour is difficult to observe when motors are being simulated.
- `TYPE_WAIT` This is identical to the TYPE_STOP option except that the plotter

will also wait at every "Stop" point. The sample application uses this wait time to move the pen up and down. A real application can use this wait time to do whatever is necessary. Once the pen has moved up and down, the path is continued. This behaviour can be observed by watching for "Pen up" and "Pen down" print messages.

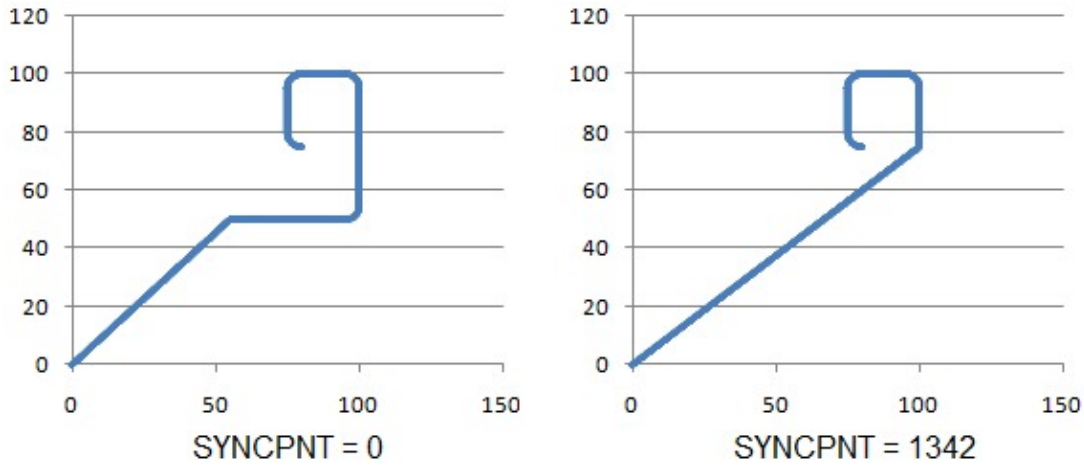
`TYPE_IMMEDIATE` This option is similar to TYPE_NORMAL with one important difference. Movement will *not* stop at the synchronization point (unless this point is also a "Stop" point and "Stop" points are enabled). Instead, the curve will smoothly flow from the start curve into the main curve.



`STARTCURVE` Use this define to select which of the available curves (see above) is to be drawn first. Path 0 was selcted for the diagram at the beginning of this topic.

`NEXTCURVE` Use this define to select which of the available curves (see above) is to be drawn second. Set this to -1 if no second curve is wanted. Path 1 was selcted for the diagram at the beginning of this topic.

`SYNCPNT` This specifies the distance from the beginning of the first curve to the synchronization point (see PathStart). If this is set to 0, then the curve will be drawn from the first fixed point. If this is set higher, then the drawing will start later in the curve. Note that 1342 is half way between fixed point 3 and fixed point 4.



`SWITCHCNT`	This specifies the number of cycles of the first curve (STARTCURVE) to draw before switching to the second curve (NEXTCURVE). In the diagram at the beginning of this topic, SWITCHCNT was set to 3.
`ENDCNT`	The test will be terminated when this cycle just begins. In the diagram at the beginning of this topic, ENDCNT was set to 6 (i.e. 5 full cycles are drawn).
`DEBUGCNT`	Debug recording is started when the first cycle of the curve is drawn. It is stopped when the cycle specified by DEBUGCNT is started. Set this to 0 if no debug recording is wanted.

Homing Behaviour

If homing is enabled (by defining HOME_ENABLE), then the application will try to home the motors during initialization. This will find the mechanical (0,0) position of the plotter. The DefOrigin command is then used to define this as the (0,0) point in the Machine Coordinate System. Note that this point is not the same as the paper origin of the plotter (see below).

If a real plotter is being used, then homing should be enabled. If motors are being simulated (whether using a real controller or the Simulator), then homing may be enabled if the user is interested in simulating the homing process.

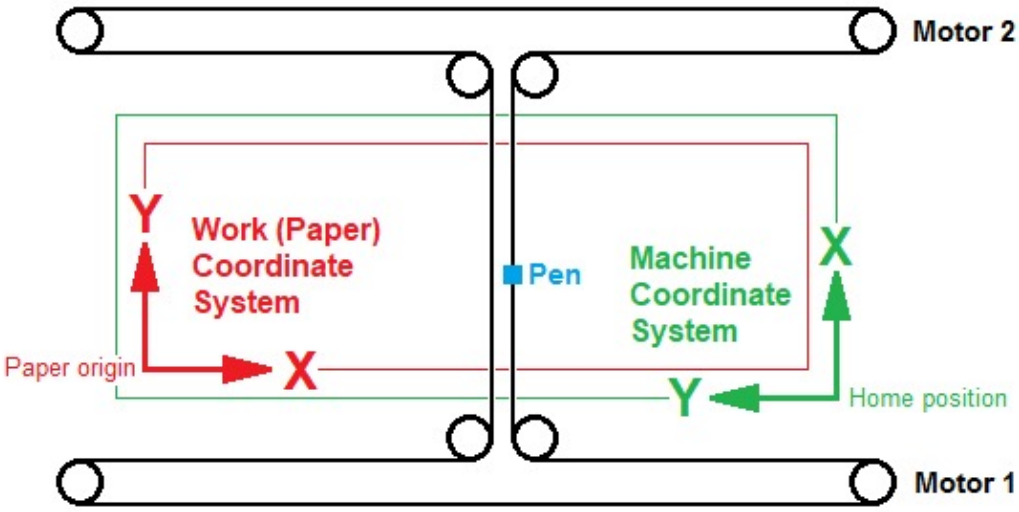
Because the Path Curve Sample is designed to use H-Bot/CoreXY Kinematics, the homing process is a little more complicated than expected. It works as follows:

1. Limit switches, connected to digital inputs, are provided to limit the plotter's horizontal and vertical movement; the limit switches are not associated with the motors themselves. The "horizontal" limit switch will limit how far the plotter can move in the horizontal direction towards the right edge of the paper. The "vertical" limit switch will limit how far the plotter can move towards the bottom edge of the paper. The digital input assignment for the limit switches, is specified in "zubPlotter.mi".
2. The homing will start by driving Motor 1 only, moving the pen diagonally towards Motor 1. This will eventually trigger one or the other limit switch (or perhaps both).
3. If the horizontal limit switch is triggered, then stop. We are now just to the right of the home position and somewhere above it. Start moving vertically down towards the home position and wait for the vertical limit switch to be triggered.
4. If the vertical limit switch is triggered, then stop. We are now just below the home position and somewhere to the left of it. Start moving horizintally right towards the home position and wait for the horizontal limit switch to be triggered.
5. Once both limit switches have been triggered, then stop. We are now just below and just to the right of the home position.
6. Reverse the above procedure, but with reduced velocity. Move diagonally until either limit switch is clear and then move either horizontally or vertically until the other limit switch is clear.
7. Once both motors are stopped, then this point is defined as the home position.

If homing is enabled and an ApossIDE Simulator is being used, then the digital input checkboxes can be used to simulate the limit switches. Once the program is running, wait for the "Waiting for limit switches..." message to be printed. Then click on the checkbox for input 1, followed by input 2, to set them high. Then click on input 2, followed by input 1, set set them low again. This then marks the home position.

If homing is enabled and an actual controller is being used with simulated motors, then inputs 1 and 2 must be physically connected in a way that the user can manually set them high and low. The triggering process is then the same as it is for the Simulator.

Configuring the Work Coordinates



In order to use the path commands, a Work Coordinate System must be defined (see Basic Coordinate Systems). This will correspond to the paper where we want to work. The "natural" coordinate system will have its origin at the bottom left corner of the paper, the X axis will point towards the right, and the Y axis will point upwards. The unit of measurement will be *millimeters*; a relatively "nice" unit from a human perspective.

The Machine Coordinate System for the plotter has its origin at the home position near Motor 1 and the X axis points towards Motor 2 (see H-Bot/CoreXY Kinematics). The unit of measurement is *micrometers*. This unit is determined by the motor gearing factors which are set by the "SetAxisParameters()" routine in "zubPlotter.mi". Note that these units are typically quite small since they need to provide adaquate resolution using only integer values.

The sample application must now define a transformation that will map points from the Work Coordinate System into the Machine Coordinate System. One possible transformation can be built as follows:

1. Imagine a sheet of paper (measured in mms) in the Work Coordinate System. Scale it by 1000. The sheet is now measured in into micrometers; the same units as the Machine Coordinate System.
2. Rotate the sheet of paper *clockwise* by 90 degrees. This is equivalent to rotating it *counterclockwise* by -90 degrees.
3. Translate the sheet by the offset from the machine home point to the paper origin. Note that in the sample, the paper width is 300 mm (300000 micrometers) and we would like a 10 mm (10000 micrometer) margin between the plotter limit switches and the edge of the sheet. So the offset is (10000,310000).
4. This is now the Work-to-World transformation. This complete transformation is then passed to the controller using the KinematicsWorkToMc command.

In the sample program, this transformation is built by the "SetUpWorkCoords()" routine in "PathCurves.m". Also, see KinematicsWorkToMc for a similar derivation with a diagram.

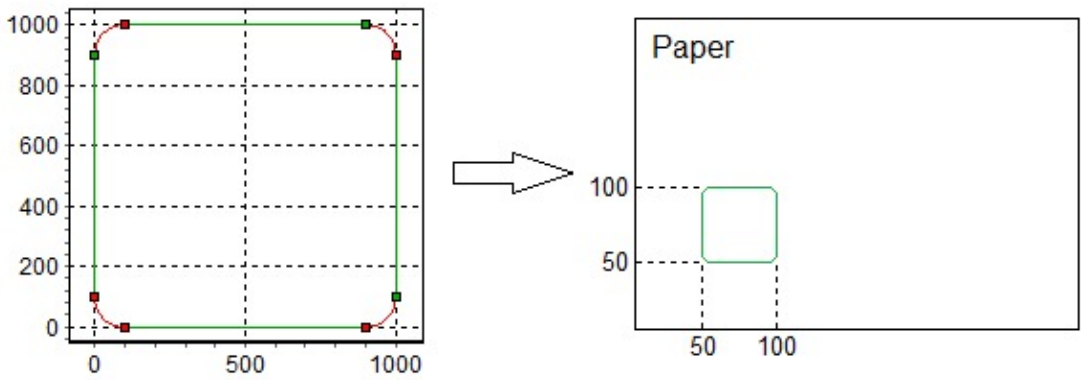
Configuring the Path Coordinates

To use the PathStart command to draw paths, a transformation from the Path Coordinate System (in which the curve itself is defined) to the Work Coordinate System must be defined (see Additional Coordinate Systems). This is eqivalent to defining where to draw the curve on the paper and how big the curve is to be drawn. For the squares drawn by this sample program, the transformation is built as follows:

1. Imagine the square (for example, Path 2 in the above diagrams) as it is in the CAM Editor (see Editing Path Curves). This square is 1000×1000 in size. We would like it to be drawn 50×50 mms in size on the paper. So start by scaling the dimensions by 1/20 (i.e. 0.05). The 1000×1000 square now becomes a 50×50 square.
2. We would like the bottom left corner of the sqaure (i.e. (0,0) in the Path Coordinate System) to be placed at (50,50) mms on the paper. So now translate by (50,50). Note that the first fixed point of the square is actually

placed at (100,0). This point will then end up being drawn at (55,50) on the paper.

3. This is now the Path-to-Work transformation. This complete transformation is then passed to the controller with the PathStart command.



In the sample program, this transformation is built by the "PlotterInit()" routine in "PathCurves.m".

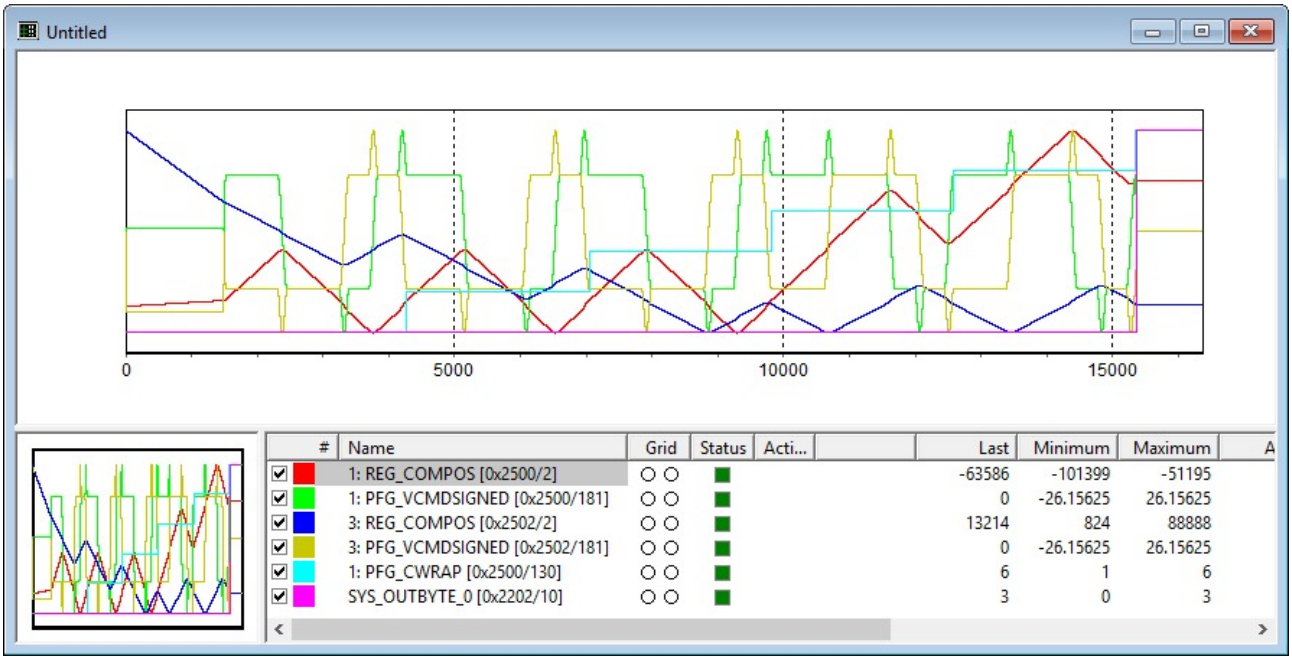
Debug Recording

The debug recording (Oscilloscope (Record) is initialized by the "StartDebugging()" routine in "PathCurves.m". The sample is set up to record the following:

- Motor position of Motor 1.
- Velocity of Motor 1.
- Motor position of Motor 2.
- Velocity of Motor 2.
- Curve cycle count.
- Digital outputs (for pen up/down).

Additional values can easily be recorded simply by adding more parameters to the RecordIndex command.

Once the debug recorded has been stopped, then the Record Oscilloscope can be used to upload and view the results. The result should appear similar to the following:



Setting User Stop Positions

The sample program is set up so that it can be used to experiment with the PathStop command. This can be done as follows:

1. Use the SmTimer command to set a timer to post the SIG_TIMEOUT event when it expires. An example of this can be found in the SIG_PENDOWN event handler in the "Running" state in "PathCurves.m". Note that the placement of this statement at this point in the code, will set the timeout to start when the drawing of the path first starts. The timeout will be triggered 1000 milliseconds after the drawing starts, which will occur while the path's start curve is being drawn. The length of the timeout may be changed to cause the SIG_TIMEOUT event to happen at any time, either earlier or later in the curve. As well, the placement of the SmTimer command may be changed (or additional commands added) if desired.
2. In the SIG_TIMEOUT event handler (which is also in the "Running" event handler), add the desired PathStop command. There are two examples already coded in the sample program.

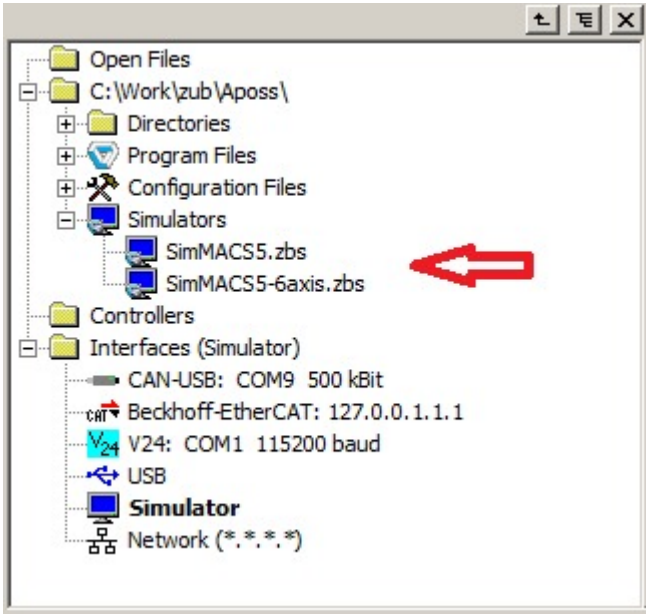
5.6 Simulator

The ApossIDE provides a Simulator that can simulate a generic 6-axis controller in software without the need of a physical controller. In certain types of applications and at certain stages of application development, a Simulator can be a valuable debugging tool. For example, partially developed application programs can be run on a Simulator without the risk of damaging equipment when something goes wrong with the application program. A Simulator can also be used when a physical controller is unavailable.

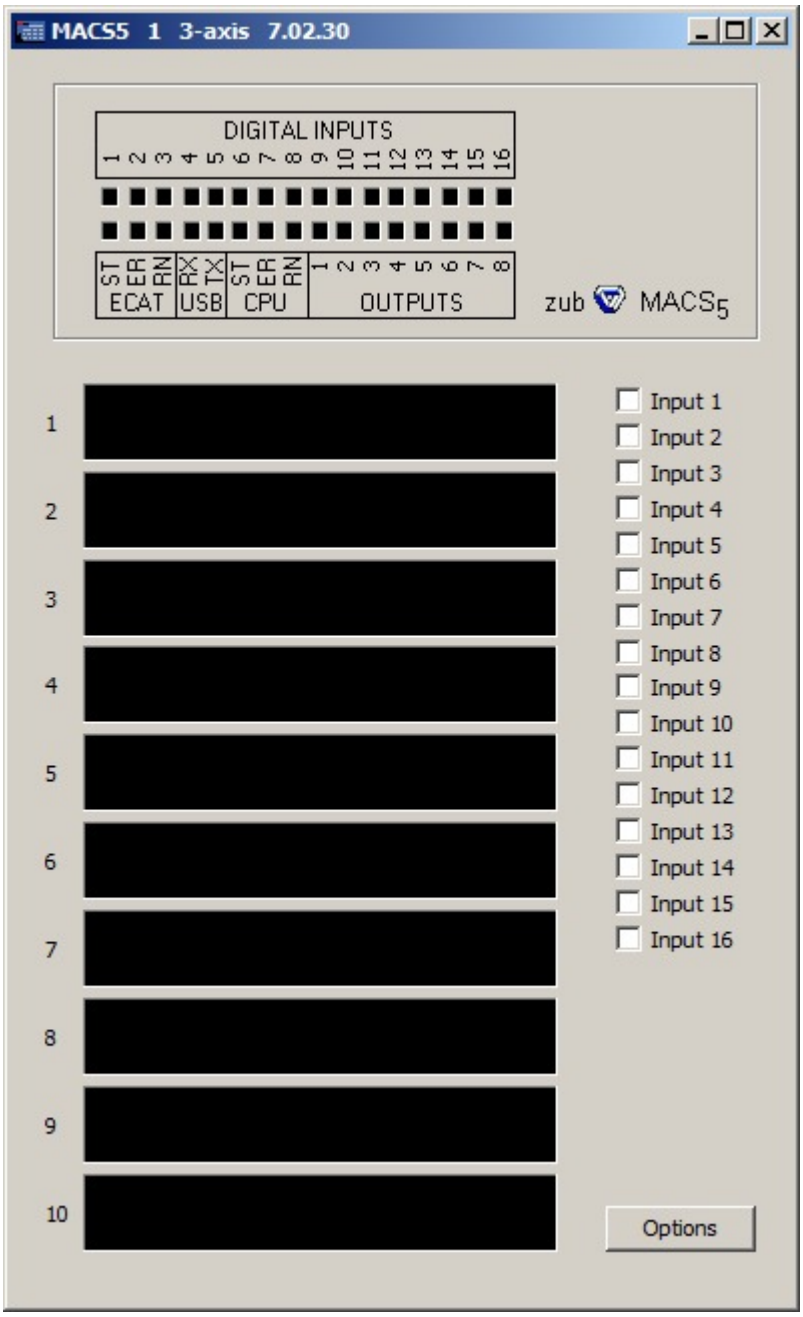
While many parts of application programs may be tested, there are certain aspects of some programs that cannot be simulated. In particular, interactions with other hardware elements such as PLCs or slave CAN devices cannot currently be simulated. The Simulator is purely stand-alone software. It can display all status LEDs (e.g. digital inputs and outputs) and they can accept user-specified digital input transitions, but they have no actual external connections. As a result, certain adaptations must be made to application programs while they are being developed before they can be successfully tested on a Simulator. See Required Program Adaptations for more information.

When a Simulator is to be used, a "Simulator file" must first be created by the user (see Simulator Files). This file will contain all Simulator settings, parameters, and data. These files will have the extension ".zbs" and they will be identified in Windows folders with the icon shown below.

Once created, Simulator files will appear in the ApossIDE Sidebar as shown below. A Simulator can then be started simply by double-clicking on the Simulator file in the sidebar. Any number of Simulators can be created and started and multiple Simulators may be running simultaneously. However, any single Simulator may only have one running instance at a time.



When a Simulator is started, a Simulator window similar to the following will be opened. This is a stand-alone window and it can be moved to another location on the computer monitor so that it does not interfere with the ApossIDE window. This window displays all LEDs and allows the user to set any digital input. It also allows the user to display in real time any internal parameter in the controller. See Simulator User Interface for information on how to use this window.



The ApossIDE connects to Simulators using a "connection interface" in a similar manner to that which it uses to connect to physical controllers with CAN, USB, EtherCAT, or other interfaces. Follow these steps:

1. Open an application program file.
2. Execute the **Settings** → **Interface** command and select the **Simulator** tab.
3. Choose the desired settings for the Simulator. See Simulator Parameters for more information about Simulator settings.

   Enabling the **Search on local computer** option will greatly simplify the process of using Simulators.

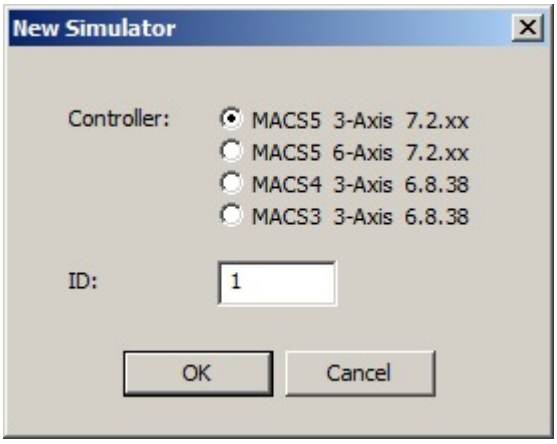
4. Select the **Set as Default** option.
5. Press **OK** to accept the settings.
6. In the program edit window, press the **Esc** key. This will start the Simulator running and make a connection to it.

Once the Simulator is connected, the ApossIDE will be able to use it as if it were a real controller. For example, most functions such as compiling, downloading, and executing application programs, editing configuration parameters, running the Oscilloscope, etc., can be used. The Simulator itself also has a user interface that allows the user to interact with the Simulator (e.g. to trigger digital inputs and to query values). See Simulator User Interface for more information.

5.6.1 Simulator Files

Simulator files contain all settings, parameters, and data associated with a particular Simulator. For example, the type of controller being simulated, the Simulator ID used to identify to the Simulator, all saved DIM arrays and downloaded application program, etc., are maintained in the Simulator file. This allows the Simulator to be restarted at a later time and behave in much the same way as a real controller would behave when powered on at a later time.

As with other types of ApossIDE files, Simulator files can be created using the **File** → **New** command and then selecting **Create Simulator .zbs file**. This will display the following dialog:



Controller The type of controller to be simulated.

ID Specify the Simulator ID that will be used to identify the Simulator. This can be any number in the range 1 to 65535. If multiple Simulators are to be run simultaneously, then each must have a unique ID number. If the Simulator is to be run on a remote computer, then the ID is used as the TCP/IP Port number. See Advanced Simulator Topics for more information on running Simulators on remote computers.

5.6.2 Required Program Adaptations

The Simulator has no physical motor. However, a motor can be simulated by using the following statement in the application program to set the encoder simulation.

```
VIRTCOUNTIN_PARAM(axisNo, VIRTCNTIN_PISRC_COUNTER) =
AXE_PROCESS_SRCINDEX(axisNo, REG_COMPOS);
VIRTCOUNTIN_PARAM(axisNo, VIRTCNTIN_MODE) = VIRTCNTIN_MODE_SIMULATION_SLAVE;
```

This will cause the "actual" position of a system (normaly encoder feedback) to track the "command" position. The command position is the calculated position from the profile generator when performing a positioning command (e.g. AxisPosAbsStart, AxisPosRelStart, etc.). With this setting the actual postion is the same as the last command position and no track error ever occurs. To the controller and the application program (when reading

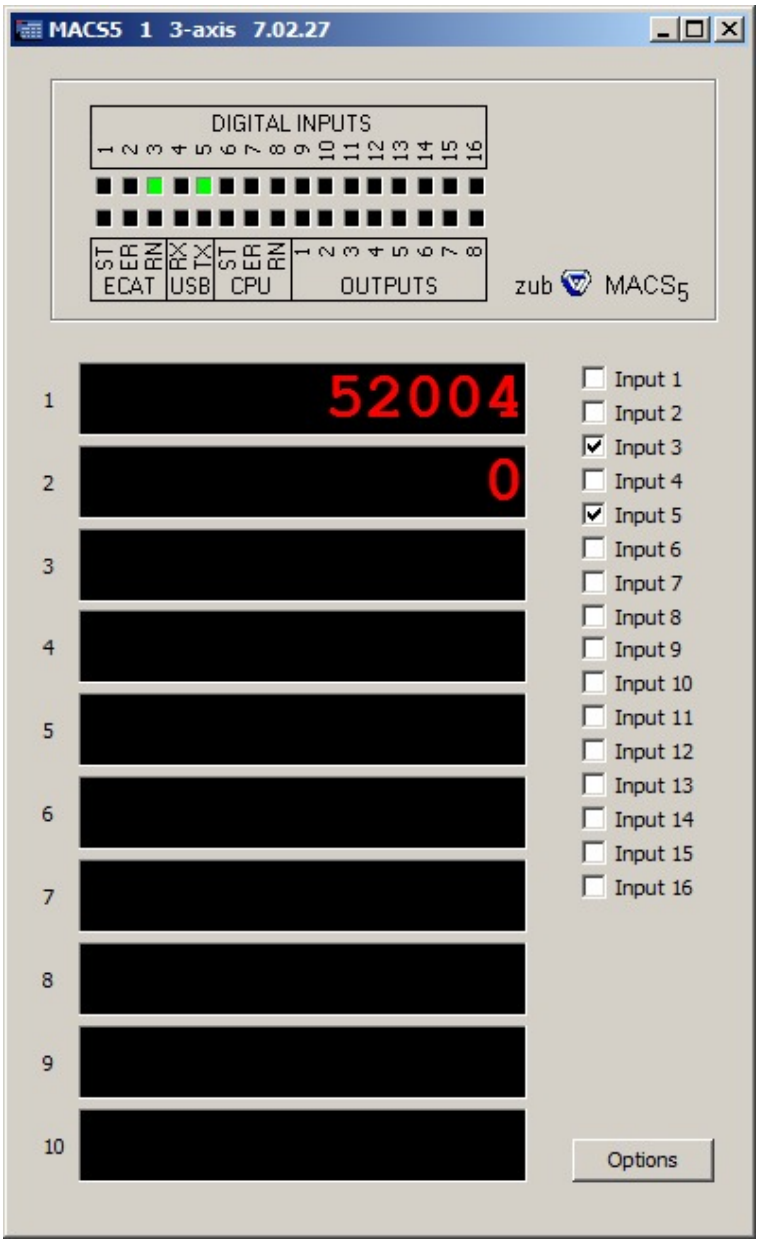
Apos()) it appear that the motor is moving. This command should be placed in the application program somewhere during the initialization phase).

> You may notice that there is a small trackerror when running a simulated axis. This is because the actual position is the command position from the last cycle (1ms).

The upper code section has to be implemented for each axis which should appear simulated. "axisNo" is the axis number in the system.

5.6.3 Simulator User Interface

The Simulator dialog will approximate the appearance of the selected controller. The dialog for a 6-axis MACS5 controller is shown below. The dialog title shows the controller type, an identification number for the Simulator, the axis count, and the firmware version. When running, the Simulator will display the current state of the LEDs in the appropriate boxes.

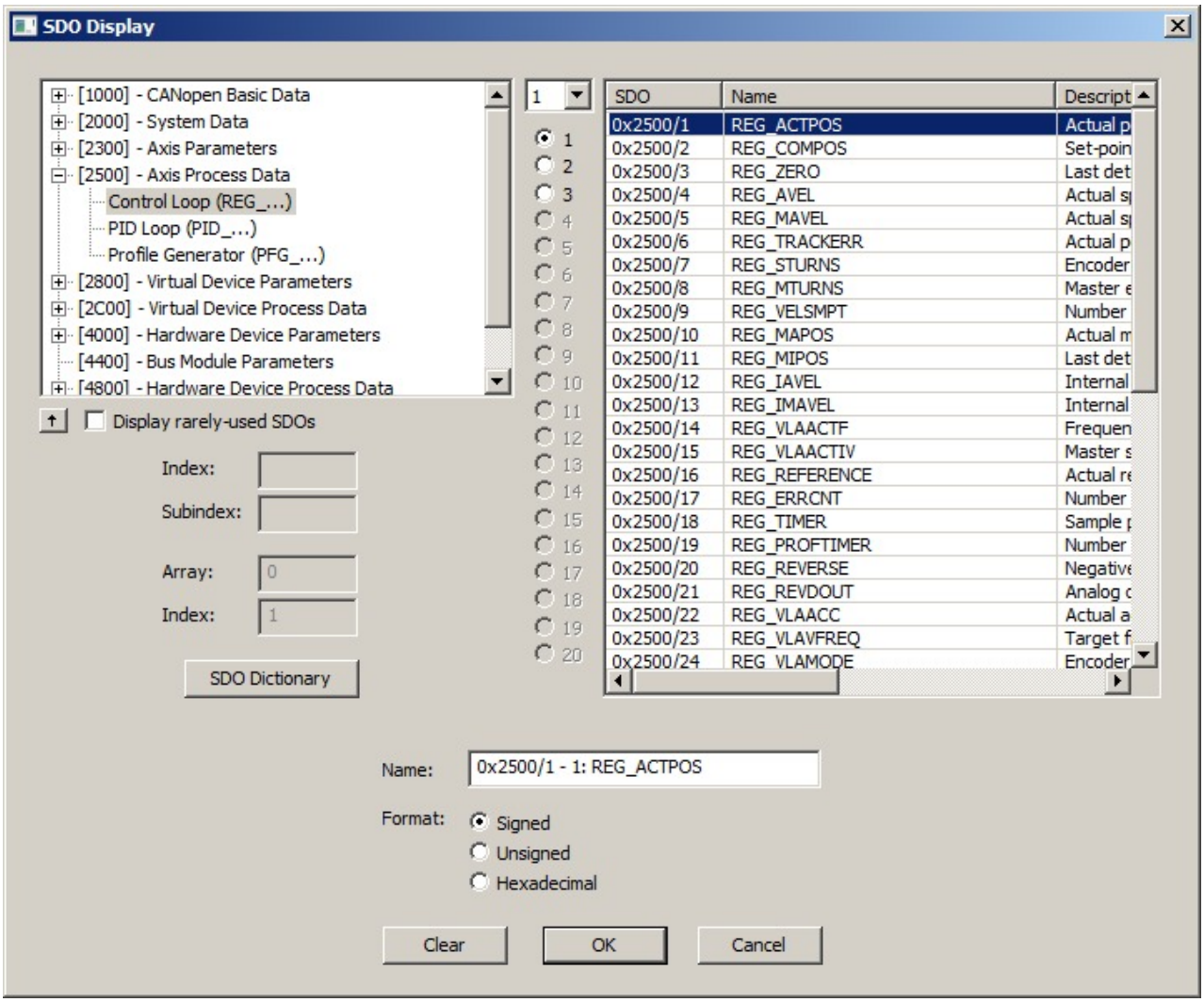


Digital Inputs

The checkboxes labelled "Input 1", etc., correspond to the controller's digital inputs. Clicking on these checkboxes will simulate digital inputs being set or cleared. The digital input states are not saved; they will always start "off" when the Simulator is started.

SDO Display

The black boxes labelled "1", "2", etc., can be used to display current SDO values. To select an SDO value, simply click within one of the black boxes:



Select the SDO to be displayed, enter a user-friendly name if desired, select the display format, and press **OK**. Alternatively, click on the **Index** field and enter an explicit SDO index and subindex or click on the **Array** field and enter a DIM array number and index. The SDO value will then be displayed and continually updated as the Simulator runs. Placing the mouse over one of the displays without clicking, will cause a tooltip to be displayed showing the user-specified name for the SDO. The selected SDO values are saved and will be restored the next time that the Simulator is started.

To clear an SDO value, click on the display and press the **Clear** button.

Options

The **Options** button allows the user to set the available Simulator options. These settings are saved and will

be restored the next time that the Simulator is started.

5.6.4 Advanced Simulator Topics

Manually Starting the Simulator

Simulators can be started manually simply by executing the appropriate Simulator application program in the normal way using the Windows command line or a Windows shortcut. The Simulator file must be specified in quotes on the command line. The following Simulator applications are installed with the ApossIDE in the install directory:

MoconSim-MACS5-6.exe	MACS5, 6-axis, newest available MACS5 firmware.
MoconSim-MACS5-3.exe	MACS5, 3-axis, newest available MACS5 firmware.
MoconSim-MACS4-3-60838.exe	MACS4, 3-axis, firmware 6.8.38. This is a legacy Simulator.
MoconSim-MACS3-3-60838.exe	MACS3, 3-axis, firmware 6.8.38. This is a legacy Simulator.

Example:

```
"C:\Program Files (x86)\Aposs\MoconSim-MACS5-6.exe"
"C:\Work\Aposs\Project\SimMACS5.zbs"
```

Running Multiple Simulators

It is possible to run multiple Simulators concurrently. Simply create a Simulator file for each Simulator (see Simulator Files). Then start each Simulator individually using either the ApossIDE Sidebar or a Windows shortcut. Note that each running instance of a Simulator must have a unique Simulator ID number. This number is shown in the Simulator's dialog title and it will also be shown in the ApossIDE controller selection dialog when connections are made to the Simulators. The ID number is specified when the Simulator file is first created and it may be modified using the **Options** button.

Simulator Debug Files

Each Simulator instance creates a pair of debug files in the Windows Application directory for the ApossIDE. These are generally useful only to zub machine control AG personnel. The files are ("n" is the Simulator ID number):

MoconSim-n.dm	A debug log file. The content of this file is determined by the **Debug level** setting in **Options**.
MoconSimMsg-n.dm	This file contains a log of all messages sent between the ApossIDE and the Simulator.

Running the Simulator on a Different Computer

The Simulator can often be a CPU-intensive application. This can be a problem on older computers with slower CPUs or on computers with only single-core CPUs. To handle this, the ApossIDE allows the Simulator be run on a different computer from the computer running the ApossIDE. Do this as follows:

1. The two computers must be on the same Ethernet network.
2. Ensure that the Internet firewalls on the two computers are configured to allow TCP/IP traffic between the ApossIDE and the Simulator.
3. Install the ApossIDE on both computers.
4. On the computer that will run the Simulator, create a Windows shortcut to manually start the Simulator (see above).
5. Start the Simulator and press the **Options** button. Select **Remote PC** as the location and specify a TCP/IP port number for the Simulator ID (e.g. 56713). Exit the Simulator.

   > Care must be taken when choosing TCP/IP port numbers. Port numbers must *not* conflict with any existing network applications already running on the computer. If a conflict exists, then the ApossIDE will be unable to connect to the Simulator. Generally, the "user" port numbers in the range 1024 to 49151 and the "private" port numbers in the range 49152 to 65535 are safe to use. For information on TCP/IP port numbers, see **http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml**.

6. On the computer that will run the ApossIDE, execute the **Settings** → **Interface** command and select the **Simulator** tab. Enable the **Run on remote computer** option and enter the Simulator computer's IP address and the TCP/IP port number chosen above. Select the **Set as Default** option and press the **OK** button.
7. Use the Windows shortcut to manually start the Simulator.
8. In the ApossIDE, use the **Esc** key to connect in the usual way.

5.7 Monitor

The Monitor feature allows the user to create customized Windows dialog interfaces for operating controllers. These could be designed for such purposes as normal system operation, system configuration, maintenance, diagnostics, etc. They allow the user to create interfaces specific to their own applications. The interfaces can contain most of the common Windows dialog controls such as text strings, buttons, listboxes, editboxes, checkboxes, etc.

Monitors are designed to be run either from within the ApossIDE or as stand-alone applications. When used in the stand-alone application mode, the computer can be configured to automatically start the application whenever the computer boots.

The layout and functionality of a Monitor interface is defined in a Monitor ".zbm" file. This is a normal text file that can be edited by any standard text editor. When a ".zbm" file is opened by the ApossIDE (using any of the standard File Open mechanisms), then the ApossIDE will parse the file and display the dialog.

A Monitor can be started in stand-alone mode by specifying the "/m" argument and the ".zbm" filename on the Windows command line. For example:

```
"C:\Program Files (x86)\Aposs\Aposs32.exe" /m C:\ApossApps\Monitor.zbm
```

When started in stand-alone mode, the ApossIDE will display only the dialog window; the normal ApossIDE window is not displayed.

> Only one instance of the ApossIDE may be running at any one time. Hence, if a stand-alone ApossIDE Monitor is running, then the normal ApossIDE may not be started.

The ".zbm" file consists of:

1. Initial statement. The first statement in the file must always be "[VARIABLES]".
2. Comments and blank lines. Both C-style comments (i.e. "//" and "/* ... */") are supported.

3. Control Statements. These define various parameters that affect the general behaviour of the Monitor when it is being used.
4. Layout Element Statements. These define the various visible and non-visible elements (i.e. text strings, buttons, value fields, etc.) that make up the dialog window.

Each statement must be defined in a single line in the ".zbm" file. Statements may be "continued" on multiple lines only by using the C-style "\" continuation character as the *last* character on a line.

> The same preprocessor statements supported by the ApossIDE compiler, are also supported within ".zbm" files. See Preprocessor for more information.

See Example Monitor Files for examples of Monitor files that have been created to illustrate the various features of Monitors.

5.7.1 Control Statements

Control statements always consist of a keyword, followed by an "=" sign, followed by one or more parameter values separated by commas:

keyword = value1,value2,value3

Space characters can be used to improve readability.

The following control statements are supported:

WindowWidth	Initial width of the dialog window (client rectangle).
WindowHeight	Initial height of the dialog window (client rectangle).
WindowTitle	Title that will appear in the dialog titlebar.
WindowFlags	"Window" flags that control the appearance and usage of the dialog window.
OpFlags	"Operational" flags that control the behaviour of the dialog while it is being used.
PollInterval	Rate at which controllers will be polled for new values.
Fade	Background "fade" colors used to highlight changed values.

5.7.1.1 WindowWidth and WindowHeight

These statements define the default size of the dialog window. The window will be opened with this size if it is not being opened "maximized" (see WindowFlags).

Syntax **WindowWidth** = *width*
WindowHeight = *height*

width	Dialog window width (in pixels).
height	Dialog window height (in pixels).

Example
```
WindowWidth  = 700
WindowHeight = 600
```

5.7.1.2 WindowTitle

The **WindowTitle** statement specifies the title that will appear in the dialog window titlebar.

Syntax **WindowTitle** = *title*

title Dialog title. This string should not be enclosed in quotes.

Example `WindowTitle = Sample Monitor Dialog`

5.7.1.3 WindowFlags

The **WindowFlags** statement specifies various flags that control the behaviour and appearance of the dialog window. The default flag values are used only if the **WindowFlags** statement is omitted altogether. These flags mostly affect the dialog only when the ApossIDE Monitor is running in stand-alone mode.

Syntax **WindowFlags** = *flags*

0x0001	Maximize window: 0 - No (default), 1 - Yes.
0x0002	System menu: 0 - Hide, 1 - Show (default).
0x0004	ESC close: 0 - Disabled (default), 1 - Enabled.
0x0008	Size boxes: 0 - Hide (default), 1 - Show.
0x0010	Resizing: 0 - Disabled (default), 1 - Enabled.
0x0020	Title move: 0 - Disabled (default), 1 - Enabled.
0x0040	Touchscreen mode: 0 - Disabled (default), 1 - Enabled.
0x0080	Enable scrollbars: 0 - Disabled (default), 1 - Enabled.

Example `WindowFlags = 0x0021`

A value of 0x0021 will cause the dialog window to be opened with the specified window size and allow it to be moved. However, the user cannot resize the window or close it. The window can be closed only using a “Button,Close” dialog element.

`WindowFlags = 0x0039`

A value of 0x0039 will cause the dialog window to be opened with the specified window size. The user can also move it and resize it. However, the user cannot close the window except by using a “Button,Close” dialog element.

`WindowFlags = 0x0001`

A value of 0x0001 will cause the dialog window to be opened maximized. The user cannot move the window (e.g. by dragging the titlebar), resize the window, or close the window. This might be used, for example, when an application is to be auto-started and the user should not have easy access to the underlying Windows interface.

5.7.1.4 OpFlags

The **OpFlags** statement specifies various flags that control the operational behaviour of the dialog. The default flag values are used only if the **OpFlags** statement is omitted altogether.

Syntax **OpFlags** = *flags*

0x0001 Start polling on open: 0 - No (default), 1 - Yes.

Example `OpFlags = 0x0001`

5.7.1.5 PollInterval

The **PollInterval** statement specifies the polling interval (in ms) to be used when polling has been started. This will default to 1000 ms if the **PollInterval** statement is omitted. Note that this is the fastest polling rate that will be used. The polling rate of individual values can be further reduced to any multiple of this value.

Syntax **PollInterval** = *time*

time Poll interval in ms.

Example `PollInterval = 2000`

This will cause controllers to be polled once every 2 seconds.

5.7.1.6 Fade

The **Fade** statement defines a list of background colors to be used to "fade" changed values. Fading is typically used for SDO value fields in order to provide positive user feedback when SDO values change. The intent is to catch the user's eye by highlighting important SDO values with a "bright" color when their values change. This color will then fade to paler colors the longer the value remains unchanged.

The list may contain 0 or more time/color pairs. The pairs must be ordered by increasing time.

Syntax **Fade** = *time1,color1,time2,color2,time3,color3,...*

time Time limit (in seconds) after which this color will no longer be used.

color Background color to be used for the corresponding time limit. This may be specified either as a "red/green/blue" color in square brackets (i.e. [red.green.blue]) or by using one of the following predefined keywords: Black, White, Gray, DlgGray, Red, Green, Blue, Yellow, Cyan, Magenta, DarkRed, DarkGreen, DarkBlue, Orange, Olive, Purple.

Example `Fade = 2,Orange,5,[255.192.128],10,[255.224.192]`

Changed values will be displayed with a bright orange background (i.e. "Orange") for the first 2 seconds after the value changes. From 2 to 5 seconds, the background will be a paler orange (i.e. [255.192.128]). From 5 to 10 seconds, the background will be a very pale orange (i.e. [255.244.192]). After 10 seconds, the normal background (i.e. white or gray) will be used.

5.7.2 Layout Element Statements

Layout element statements define the various visible and non-visible elements (i.e. text strings, buttons, value fields, etc.) that make up the dialog window. They always consist of an element "name", followed by an "=" sign, followed by one or more parameter values separated by commas:

name = value1,value2,value3

Space characters can be used to improve readability. Names may consist of letters, digits, "_", "-", and "$". However, they must always begin with the "$" character in order to differentiate layout statements from control statements. Note that layout element names are often used as values *within* element statements to refer to other elements of the dialog. When used this way, the initial "$" character in the name is optional.

The following control statements are available:

Array	Array field.
Bit	Color box to display a bit status.
Bitmap	Bitmap image.
Box	Group box.
Button	Dialog button.
Checkbox	Checkbox.
Constant	Constant integer.
Controller	Connection to a specific controller.
Font	Text font.
List	List box.
Message	Message string.
PDO	PDO.
SDO	SDO.
Slider	Slider.
Table	Table.
Text	Text string.
Value	Value field.

5.7.2.1 Array

The **Array** statement defines an array field. This is displayed as a Windows edit box in which the user can enter a value. Depending on the specified flags, it may or may not include a label and read/write buttons. The value will be displayed in red if it has been changed but not "committed". If the value is associated with a controller, then it is "committed" when the array has been written to the controller. Otherwise, the value is "committed" when the user moves to another field. Currently, only string arrays are supported

Syntax *name* = **Array**,*label,offset,x,y, width, height, flags,ctrl,arrnum,show*

name Element name.

label Text string to appear as field label. This *must* be enclosed in quotes and the first

character *must* be "*". The "*" will not be displayed

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

width Width (in pixels) of the text box that will enclose the text string. The text string will wrap within the box if necessary.

height Height (in pixels) of the text box that will enclose the text string. The text string will wrap within the box if necessary.

flags The following flags are supported:

- 0x0001 Value is read-only.
- 0x0002 Blink the value when Read button is pressed.
- 0x0004 Automatically write if user changes value and leaves field.
- 0x0008 Fade the value when it changes.
- 0x0010 Automatically write whenever the user presses the carriage return on the keyboard. The value will also be reselected so that a new value can be entered immediately. This flag is ignored if the Monitor is running in touchscreen mode.
- 0x0100 Use the "small" child dialog.
- 0x0200 Use the "display" child dialog (no read or write buttons).
- 0x0400 Hide the child dialog.

ctrl Name of controller element. This can also be the name of an element (e.g. value or list element) that contains the controller number of the associated controller element. If this is 0 or -, then the array will not be associated with a controller.

arrnum Array number element name. The DIM array number for this field is retrieved from this element. Note that DIM array numbers start at 0. If *arrnum* is set to 0 or -, then the array will not be associated with a controller.

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

Example `$String = Array,"*Name:",-,20,20,70,15,0x0002,Ctrl,ArrNum,0`

5.7.2.2 Bit

The **Bit** statement defines a color box to display a bit status

Syntax *name* = **Bit**,*offset,x,y,width,height,flags,on,off,undef,element,bitmask,show,compare*

name Element name.

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

width Width (in pixels) of the box that will display the status color.

height Height (in pixels) of the box that will display the status color.

flags The following flags are supported:

- 0x0001 The value is to be compared with *compare* according to the other bits in flags. More than one "compare" bit may be specified.
- 0x0002 Compare: Equals.
- 0x0004 Compare: Greater than.
- 0x0008 Compare: Less than.

on The box will be displayed in this color if the bit is "on". This may be specified either as a "red/green/blue" color in square brackets (i.e. [red.green.blue]) or by using one of the following predefined keywords:
Black, White, Gray, DlgGray, Red, Green, Blue, Yellow, Cyan, Magenta, DarkRed, DarkGreen, DarkBlue, Orange, Olive, Purple.

off The box will be displayed in this color if the bit is "off".

undef The box will be displayed in this color if the bit is undefined (i.e. the value of *element* cannot be determined).

element Element name. The value for the bit is retrieved from this element.

bitmask Bitmask specified either as an integer or as a hexadecimal number (e.g. 0x0002). If *flags* bit 0x0001 is not set, then the bit will be "on" if the value is non-0 when "and"-ed with the bitmask. The bitmask may contain more than one bit. If this bit is set, then see the description for *compare*.

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

compare Compare value specified either as an integer or as a hexadecimal number. This value is used if *flags* bit 0x0001 is set. The bit will be "on" if the value, when "and"-ed with the bitmask, compares successfully with this value.

Example
```
$Bit1 = Bit,-,550,260,10,10,0x0000,Green,Red,DlgGray,Sdo,0x0002,-,0
$Bit7 = Bit,-,590,260,10,10,0x0000,
[177.100.201],Red,Black,Sdo,0x0008,-,0
```

5.7.2.3 Bitmap

The **Bitmap** statement defines a bitmap image that will be displayed in the dialog. Currently, only bitmap files are supported. Other image types (e.g. .jpg, .gif, .png, etc.) are not supported.

Syntax *name* = **Bitmap**,*filename,offset,x,y,flags,show*

name Element name.

filename Filename of the ".bmp" file to be displayed. This *must* be enclosed in quotes. If a relative pathname is specified, then it will be relative to the directory containing the ".zbm" file.

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

flags Currently, no flags are supported. This should be set to 0x0000.

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If show is set to 0 or -, then the control will always be visible.

Example
```
$Picture = Bitmap,"Wizards\EN\Assemble.bmp",-,50,100,0x0000,-
```

5.7.2.4 Box

The **Box** statement defines a "group box" that will be displayed in the dialog.

Syntax *name* = **Box**,*label,offset,x,y,width,height,flags,show*

name Element name.

label Label that will appear in the upper-left of the group box (up to 200 characters). This *must* be enclosed in quotes and the first character *must* be "*". The "*" will not be displayed.

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

width Width (in pixels) of the box.

height Height (in pixels) of the box.

flags Currently, no flags are supported. This should be set to 0x0000.

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

Example `$MiscBox = Box,"*Miscellaneous Values",-,50,250,440,240,0x0000,-`

5.7.2.5 Button

The **Button** statement defines a dialog button. "Clicking" a button will execute an operation that depends on the type of button defined. Buttons can be "clicked" either directly by the user or indirectly by referencing the button in a **Group** button and then "clicking" the **Group** button.

Buttons can be flagged as "hidden". In this case, the button is not displayed in the dialog and the user cannot "click" it. The button can be "clicked" only indirectly by using a **Group** button. This allows a single **Group** button to be used to execute multiple operations.

Syntax *name* = **Button**,*type,label,offset,x,y,width,height,flags,font,textclr,faceclr,show,*{*params*}

name Element name.

type Button type. This can be one of the following:

Close	Close the dialog window. *params* is not required.
PollStart	Start polling. *params* is not required.
PollStop	Stop polling. *params* is not required.
Controller	Controller connection functions.
Get	Retrieve a value (e.g. read an SDO value).
Set	Set a value (e.g. write an SDO value).
Group	Get or set multiple values.
PDO	Send a PDO message (currently not supported).
Download	Download a configuration file to a controller.

label Text string to appear on button. This *must* be enclosed in quotes and the first character *must* be "*". The "*" will not be displayed.

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window

itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

width Width (in pixels) of the button.

height Height (in pixels) of the button.

flags Button flags are type-dependent and are described with the specific **Button** type (see above). However, all button types support the following flags:

0x0001 Hide the button.

0x0002 Use a bold text font for the button label (ignored if a **Font** is used).

font The name of the **Font** element that specifies the font to be used. A positive integer may be specified instead of a **Font** element. In this case, a default font is used and the integer will specify the font height. If 0 is specified, then a default font height is used.

textclr The button text will be displayed in this color. This may be specified either as a "red/green/blue" color in square brackets (i.e. [red.green.blue]) or by using one of the following predefined keywords:
Black, White, Gray, DlgGray, Red, Green, Blue, Yellow, Cyan, Magenta, DarkRed, DarkGreen, DarkBlue, Orange, Olive, Purple.

faceclr The button face will be displayed in this color. It is specified as described above.

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

params Button-specific parameters enclosed in braces. Different button types require different parameters. This can be omitted if the button requires no button-specific parameters (see above).

5.7.2.5.1 Controller

The **Controller** button provides various controller-related functions. The function is determined by the "*flags*" parameter.

Syntax *name* = **Button,Controller**,*label,offset,x,y,width,height,flags,font,textclr,faceclr,show,{ctrl}*

flags The following flags are supported:

0x0001 Hide the button.

0x0002 Use a bold text font for the button label.

0x0100 Select a new controller.

0x0200 Connect (or reconnect) controller.

0x0400 Enable button color for controller status.

ctrl This is the name of the **Controller** element associated with the button. This can also be the name of an element (e.g. value or list element) that contains the controller number of the associated controller element.

Example
```
$Connect = Button,Controller,"*Connect
Controller",-,400,35,130,30,0x0600,-,Black,DlgGray,-,{UsbMACS4}
```

5.7.2.5.2 Download

The **Download** button will download a configuration or Array Editor file to a controller.

Syntax *name* = **Button,Download**,*label,offset,x,y,width,height,flags,font,textclr,faceclr,show*, {*ctrl,cnfflags,filename*}

flags The following flags are supported:

- 0x0001 Hide the button.
- 0x0002 Use a bold text font for the button label.
- 0x0100 Browse for the filename.
- 0x0200 Download an Array Editor file. If this is not set, then the file is assumed to be a configuration file.

ctrl This is the name of the **Controller** element associated with the button. This can also be the name of an element (e.g. value or list element) that contains the controller number of the associated controller element.

cnfflags Configuration download flags (used only if a configuration file is being downloaded):

- 0x00000001 Download controller name.
- 0x00000002 Download global parameters.
- 0x00000004 Download user parameters.
- 0x00000008 Download permanent axis parameters.
- 0x00000010 Download temporary axis parameters.
- 0x00000020 Download DIM arrays.
- 0x00000040 Download advanced parameters.
- 0x40000000 Set parameters regardless of modification state.
- 0x10000000 Do not display user information messages.
- 0x04000000 Retain existing controller DIM arrays.

filename Filename of the file to be downloaded. This must be enclosed in quotes. This should be "*" if the "File Open" dialog is to be used to browse for the file.

Example
```
$Config =
Button,Download,"*Configure",-,560,150,100,30,0x0000,-,Black,DlgGray,-,
{UsbMACS4,0x50000020,"test.zbc"}
```

5.7.2.5.3 Get

The **Get** button will retrieve a value (e.g. read an SDO value).

Syntax *name* = **Button,Get**,*label,offset,x,y,width,height,flags,font,textclr,faceclr,show,{element}*

flags The following flags are supported:

0x0001	Hide the button.
0x0002	Use a bold text font for the button label.

element Element name. A "Get" operation is performed on this element when the button is "clicked". Currently, *element* must be an **SDO** element. "Getting" an SDO will cause its value to be read from the controller.

Example
```
$Status = Button,Get,"*Get
Status",GroupBox,40,190,200,30,0x0000,-,Black,[177.100.201],-,
{StatusSdo}
```

5.7.2.5.4 Group

The **Group** button will "get" or "set" multiple values.

Syntax *name* = **Button,Group**,*label,offset,x,y,width,height,flags,font,textclr,faceclr,show,{elements}*

flags The following flags are supported:

0x0001	Hide the button.
0x0002	Use a bold text font for the button label.
0x0100	Do a "Get" operation on each element.
0x0200	Do a "Set" operation on each element.

elements List of element names separated by commas and enclosed in parentheses.

Example
```
$ReadAll = Button,Group,"*Read
All",-,560,100,100,30,0x0100,-,Black,DlgGray,-,{Sdo1,Sdo2,Sdo3}
```

When a **Group** button is "clicked", either a "Get" or "Set" operation will be performed on each element in the specified list. The operation will be performed on the elements in the order that they are specified.

A "Get" operation will consist of reading each readable element (e.g. reading an SDO value from the controller) and simulating a "click" on each **Get** button. A "Set" operation will consist of setting each settable element (e.g. writing an SDO value to the controller) and simulating a "click" on each **Set** button. Note that **Group** buttons can also be included in the list of elements. *Do not* reference **Group** buttons recursively.

Typically, **Group** buttons are used to execute a sequence of operations. In this case, the buttons referenced by the **Group** button will usually have the "Hide" flag set.

5.7.2.5.5 PDO

The **PDO** button will send a PDO. This button type is not currently supported and should not be used.

Syntax *name* = **Button,PDO**,*label,offset,x,y,width,height,flags,font,textclr,faceclr,show,{ctrl,pdo,(hexbytes)}*

flags The following flags are supported:

0x0001	Hide the button.
0x0002	Use a bold text font for the button label.

ctrl This is the name of the **Controller** element associated with the button. This can also be the name of an element (e.g. value or list element) that contains the controller number of the associated controller element.

pdo PDO number.

hexbytes Bytes to be sent in the PDO. These must be separated by commas and enclosed in parentheses. They may be specified either in integer or hexadecimal notation.

Example
```
$SendPdo = Button,PDO,"*Send PDO",-,560,150,100,30,0x0100,-,Black,DlgGray,-,
{UsbMACS4,1,
(0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88,0x99,0xAA,0xBB,0xCC,0xDD,0xEE,0xFF)}
```

5.7.2.5.6 Set

The **Set** button will set a value (e.g. write an SDO value).

Syntax *name* = **Button,Set**,*label,offset,x,y,width,height,flags,font,textclr,faceclr,show*,{*element,value,addval*}

flags The following flags are supported:

0x0001	Hide the button.
0x0002	Use a bold text font for the button label.

element Element name. A "Set" operation is performed on this element when the button is "clicked". Currently, *element* must be an **SDO** element. "Setting" an SDO will cause its value to be written to the controller.

value Value to be written. This may be specified either in integer or hexadecimal notation.

addval Element name. The value of this element will be added to *value* before the "Set" operation is performed. This should be 0 or - if no value is to be added. Note that referencing an *addval* element will only cause its existing value to be retrieved; it will not cause a "Get" operation to be performed on the element.

Example
```
$Width = Button,Set,"*Set Width",GroupBox,40,190,200,30,0x0000,-,Black,
[177.100.201],-,{UserSdo1,500000,-}
```

5.7.2.6 Checkbox

The **Checkbox** statement defines a checkbox.

Syntax *name* = **Checkbox**,*offset,x,y,flags,element,bitmask,show*

name Element name.

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move

along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

flags The following flags are supported:

0x0001 If *element* is not specified, then the checkbox is initialized to the *bitmask* value. If this bit is not set, then the checkbox is initialized to 0.

element Element name. The value for the checkbox is retrieved from this element. If the value of *element* cannot be determined (e.g. the controller may be disconnected), then the checkbox will be disabled. If this is not specified (i.e. if it is set to 0 or -), then the value of the checkbox will be maintained locally in the dialog.

bitmask Bitmask specified either as an integer or as a hexadecimal number (e.g. 0x0002). The bit will be "on" if the value is non-0 when "and"-ed with the bitmask. Currently, the bitmask may contain only one bit.

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

Example
```
$Check = Checkbox,-,590,260,0x0000,Sdo,0x0008,-
$Show = Checkbox,-,590,260,0x0001,-,0x0001,-
```

5.7.2.7 Child

The **Child** statement defines a complete child dialog. The child dialog will be defined in its own ".zbm" file. Please note the following points:

1. Each child ".zbm" file is completely independent of all other child dialogs and of the main dialog itself. Monitor elements are *not* shared between child dialogs and element names need not be unique among the different child dialogs. For example, each child dialog must have its own controller elements, its own font elements, etc.
2. The WindowWidth, WindowHeight, WindowTitle, and WindowFlags statements in the child ".zbm" file are ignored.

> Microsoft Windows has a maximum limit to the number of controls that may be defined with a dialog window. This limit is a few hundred controls. This limit can easily be exceeded by Aposs Monitor dialogs that contain many checkbox controls or bit display controls. If an Aposs Monitor dialog requires more than perhaps 200 to 300 controls, then the dialog should be divided into two or more child dialogs.

Syntax *name* = **Child**,*offset,x,y,width,height,flags,show,filename*

name Element name.

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -,

then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

width Width (in pixels) of the area in which the child dialog will be placed. The dialog will be clipped if it is wider than this dimension.

height Height (in pixels) of the area in which the child dialog will be placed. The dialog will be clipped if it is higher than this dimension.

flags Currently, no flags are supported. This should be set to 0x0000.

show Show/hide flag. This is the name of the element to be used to determine whether the child dialog is visible or hidden. If the element value is 0, then the dialog is hidden. Otherwise, the dialog is visible. If *show* is set to 0 or -, then the dialog will always be visible.

filename Pathname of the file ".zbm" containing the specification of the child dialog. This may be specified either as an absolute pathname or a relative pathname. If a relative pathname is used, then the path will be relative to the directory containing the current ".zbm" file (i.e. the ".zbm" file that contains the **Child** statement). This *must* be enclosed in quotes and *must* contain at least 1 character.

Example `$Child1 = Child,-,5,30,860,500,0x0000,ShowIO,"IoConfig.zbm"`

5.7.2.8 Constant

The **Constant** statement defines an integer. It is used simply to specify constant integer values. It is not displayed in the dialog in any way.

Syntax *name* = **Constant**,*flags,value*

name Element name.

flags Currently, no flags are supported. This should be set to 0x0000.

value The value of the constant. The values may be specified either as integers or as hexadecimal numbers.

Example `$Five = Constant,0x0000,5`

5.7.2.9 Controller

The **Controller** statement defines a connection to a specific controller. It is not displayed in the dialog in any way.

Syntax *name* = **Controller**,*flags,number*,(*connection*)

name Element name.

flags The following flags are supported:

0x0001 Connect controller when dialog is opened.

number Controller number by which the controller will be known in the Monitor. This can be any integer value, however all controllers defined in the file must have unique numbers. Note that this is *not* the controller ID number (e.g. CAN ID number).

connection Connection definition parameters. This can be either "*" or the actual connection parameters for a specific controller connection. If "*" is used, then a "Select Controller" dialog will be displayed the first time that the Monitor tries to talk to the controller. If actual connection parameters are specified, then they must consist of the Controller ID (e.g. CAN ID) followed by the the interface definition parameters. The interface definition parameters can be copied from the examples shown in the "Specified Controllers" example (see Example Monitor Files) or copied from the APOS.DAT file.

Example
```
$Ctrl = Controller,0x0001,1,(*)
$V24 = Controller,0x0,2,(1,V24,0,115200,0,6,50,0x7,0,25)
$Usb = Controller,0x0,3,(1,USB,6,50,0x0,921600,1)
$Can2 = Controller,0x0,4,(2,COM,5,7,1,32,0x0)
```

5.7.2.10 Font

The **Font** statement defines a text font that can be used by text strings and message strings in the dialog. A font is a "non-visible" element.

Syntax *name* = **Font**,*flags*, (*height,width,escape,orient,weight,italic,underline,strike,charset,outprec,clipprec,quality, pitch,font*)

name Element name.

flags Currently, no flags are supported. This should be set to 0x0000.

height Font height (>0 - Windows "cell" height, <0 - Windows "character" height).

width Font width. If this is 0, then a default aspect ratio is chosen based on the available fonts.

escape Escapement angle of text string (in 10's of a degree).

orient Text angle (in 10's of a degree).

weight Font weight. This can be a multiple of 100 from 100 (thin font) to 900 (black). A value of 400 gives a "normal" font.

italic Specifies an italic font if set to non-0.

underline Specifies an underline font if set to non-0.

strike Specifies a strikeout font if set to non-0.

charset Specifies the Windows character set to be used. This should be set to 0 unless there is a specific reason to change it.

outprec Specifies the Windows output precision to be used. This should be set to 0 unless there is a specific reason to change it.

clipprec Specifies the Windows clipping precision to be used. This should be set to

0 unless there is a specific reason to change it.

quality Specifies the Windows output quality to be used. This should be set to 0 unless there is a specific reason to change it.

pitch Specifies the Windows font pitch and family to be used. This should be set to 0 unless there is a specific reason to change it.

font Specifies the Windows font name. This must be enclosed in quotes. The names of available fonts can be found using "Fonts" in the Windows Control Panel.

Example
```
$Font1 = Font,0x0000,(-11,0,0,0,400,0,0,0,0,0,0,0,0,"MS Shell Dlg 2")
$Font2 = Font,0x0000,(-11,0,0,0,400,0,0,0,0,0,0,0,0,"Times New Roman")
```

5.7.2.11 List

The **List** statement defines a list box that will be displayed in the dialog.

Syntax *name* = **List**,*type,offset,x,y,width,height,flags,show,{params}*

name Element name.

type Button type. This can be one of the following:

Axis List contains axis numbers from a controller.

Table List contains items from a Table element.

Lookup Table list with automatic selection.

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

width Width (in pixels) of the list box.

height Height (in pixels) of the drop-down list box.

flags Flags are type-dependent and are described with the specific **List** type (see above).

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

params List-specific parameters enclosed in braces. Different list types require different parameters. These are described with the specific **List** type (see above).

5.7.2.11.1 Axis

The **Axis** list contains axis numbers from a controller.

Syntax *name* = **List,Axis**,*offset,x,y,width,height,flags,show*,{*ctrl*}

flags Currently, no flags are supported. This should be set to 0x0000.

ctrl Name of **Controller** element. The list will allow the user to select a specific axis if the controller contains multiple axes. Currently, this *cannot* be the name of an element (e.g. value or list element) that contains the controller number of the associated controller element.

Example
```
$AxisList = List,Axis,-,560,413,50,200,0x0000,-,{UsbMACS4}
```

5.7.2.11.2 Lookup

The **Lookup** list is a **Table** list with automatic selection.

Syntax *name* = **List,Lookup**,*offset,x,y,width,height,flags,show*,{*table,element*}

flags The following flags are supported:

0x0001 The list is read-only.

0x0002 Automatically write an SDO (i.e. do a "Set") if the selected list item changes. If the 0x0002 flag is used, then *element* must be an **SDO** element.

table Name of the **Table** element defining the list items and values to be included in the list.

element Element name. If this element value changes, then the currently selected list item is automatically changed to match the new value.

Example
```
$Sdo2400-6 = SDO,0x0000,1,0x2400,6,0,0
$Lst = List,Lookup,-,60,70,90,200,0x0001,-,{Numbers,Sdo2400-6}
```

5.7.2.11.3 Table

The **Table** list contains items from a **Table** element.

Syntax *name* = **List,Table**,*offset,x,y,width,height,flags,show*,{*table*}

flags Currently, no flags are supported. This should be set to 0x0000.

table Name of the **Table** element defining the list items and values to be included in the list.

Example
```
$Numbers = Table,0x0000,("*One",1,"*Two",2,"*Three",3)
$NumList = List,Table,-,560,473,90,200,0x0000,-,{Numbers}
```

5.7.2.12 Message

The **Message** statement defines a message string. This is similar to a **Text** statement except that the text of the

message is determined by the value of another element.

Syntax *name* = **Message**,*type,offset,x,y,width,height,just,font,flags,show,{params}*

name Element name.

type Message type. This can be one of the following:

Lookup	Message is selected automatically.
Value	Message is selected automatically.
Load	Message is the polling load (in percent).
Controller	Message is the controller name.
Print	Message is the last print message from the controller.

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

width Width (in pixels) of the text box that will enclose the text string. The text string will wrap within the box if necessary.

height Height (in pixels) of the text box that will enclose the text string.

just Justification: -1 - Left, 0 - Center, 1 - Right.

font The name of the **Font** element that specifies the font to be used. A positive integer may be specified instead of a **Font** element. In this case, a default font is used and the integer will specify the font height. If 0 is specified, then a default font height is used.

flags Flags are type-dependent and are described with the specific **Message** type (see above).

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

params Message-specific parameters enclosed in braces. Different message types require different parameters. These are described with the specific **Message** type (see above).

5.7.2.12.1 Controller

The **Controller** message will display the name of a controller.

Syntax *name* = **Message,Controller**,*offset,x,y,width,height,just,font,flags,show,{ctrl,textclr}*

flags The following flags are supported:

0x0001	Use bold font (ignored if **Font** is used).

0x0002 Display a sunken border.

ctrl Name of controller element. This can also be the name of an element (e.g. value or list element) that contains the controller number of the associated controller element.

textclr The text will be displayed in this color. This may be specified either as a "red/green/blue" color in square brackets (i.e. [red.green.blue]) or by using one of the following predefined keywords:
Black, White, Gray, DlgGray, Red, Green, Blue, Yellow, Cyan, Magenta, DarkRed, DarkGreen, DarkBlue, Orange, Olive, Purple.

Example `$Name1 = Message,Controller,Title1,0,18,110,20,0,16,-1,Font1,0x0001,-,{Ctrl1,DarkRed}`

5.7.2.12.2 Load

The **Load** message will display the current polling load in percent.

Syntax *name* = **Message,Load**,*offset,x,y,width,height,just,font,flags,show,{format,textclr}*

flags The following flags are supported:

0x0001 Use bold font (ignored if **Font** is used).

0x0002 Display a sunken border.

format This must be a "C" printf() format specification enclosed in quotes. The format must expect one "double" value. Note that improper format specifications may cause the ApossIDE to crash. Users should be familiar with printf() format specifications before attempting to use them.

textclr The text will be displayed in this color. This may be specified either as a "red/green/blue" color in square brackets (i.e. [red.green.blue]) or by using one of the following predefined keywords:
Black, White, Gray, DlgGray, Red, Green, Blue, Yellow, Cyan, Magenta, DarkRed, DarkGreen, DarkBlue, Orange, Olive, Purple.

Example `$Load = Message,Load,Stop,75,2,170,20,-1,16,-1,Font1,0x0,-,{"Current Polling Load: %.1lf%%",DarkGreen}`

5.7.2.12.3 Lookup

The **Lookup** message is a message where the text strings comes from a **Table** element.

Syntax *name* = **Message,Lookup**,*offset,x,y,width,height,just,font,flags,show,{table,element}*

flags The following flags are supported:

0x0001 Use bold font (ignored if **Font** is used).

0x0002 Display a sunken border.

0x0004 Fade background color after a change.

table Name of the **Table** element defining the messages and the corresponding values.

element Element name. If this element value changes, then the displayed message is automatically changed to match the new value.

Example
```
$State = Table,0x0000,("*Start",1,"*Stop",2,"*Run",3)
$Msg = Message,Lookup,-,560,473,70,15,-1,Font1,0x0006,-,{State,Sdo}
```

5.7.2.12.4 Print

The **Print** message will display the last print message from the controller. Note that polling must be enabled for this to work.

Syntax *name* = **Message,Print**,*offset,x,y,width,height,just,font,flags,show,{table,element}*

flags The following flags are supported:

0x0001	Use bold font (ignored if **Font** is used).
0x0002	Display a sunken border.

ctrl Name of controller element. This can also be the name of an element (e.g. value or list element) that contains the controller number of the associated controller element.

textclr The text will be displayed in this color. This may be specified either as a "red/green/blue" color in square brackets (i.e. [red.green.blue]) or by using one of the following predefined keywords:
Black, White, Gray, DlgGray, Red, Green, Blue, Yellow, Cyan, Magenta, DarkRed, DarkGreen, DarkBlue, Orange, Olive, Purple.

Example
```
$Name1 = Message,Print,Title1,0,18,110,20,0,16,-1,Font1,0x0001,-,
{Ctrl1,Black}
```

5.7.2.12.5 Value

The **Value** message will display the value of another element.

Syntax *name* = **Message,Value**,*offset,x,y,width,height,just,font,flags,show,*
{element,format,addval,scale,textclr}

flags The following flags are supported:

0x0001	Use bold font (ignored if **Font** is used).
0x0002	Display a sunken border.
0x0004	Fade background color after a change.

element Element name. If this element value changes, then the displayed message is automatically updated.

format Display format: 0 - Numeric, 1 - Hexadecimal (ignored if scaling is used). This can also be a "C" printf() format specification enclosed in quotes. If a scale factor is used, then this *must* be a "double" specification. If a scale factor is not used, then this *must* be a "long" specification. Note that improper format specifications may cause the ApossIDE

to crash. Users should be familiar with printf() format specifications before attempting to use them.

addval Element name. The value of this element will be added to the value of *element* before the value is displayed. This should be 0 or - if no value is to be added. If this element value changes, then the displayed message is automatically updated.

scale Scale factor for display. This is a floating-point value. The value being displayed (i.e. after *addval* has been included) will be *divided* by this number before being displayed. Set this to 0 if no scaling is to be done.

textclr The text will be displayed in this color. This may be specified either as a "red/green/blue" color in square brackets (i.e. [red.green.blue]) or by using one of the following predefined keywords:
Black, White, Gray, DlgGray, Red, Green, Blue, Yellow, Cyan, Magenta, DarkRed, DarkGreen, DarkBlue, Orange, Olive, Purple.

Example
```
$Dsp = Message,Value,-,560,493,70,15,-1,Font1,0x0000,-,
{Sdo,"0x%08lX",-,0,Red}
```

5.7.2.13 PDO

The **PDO** statement defines a PDO. It is not displayed in the dialog in any way. The value of this element will be set when a PDO message has been received.

Syntax *name* = **PDO**,*flags,ctrl,pdo,start,count,reverse,mask,signed*

name Element name.

flags Currently, no flags are supported. This should be set to 0x0000.

ctrl Name of controller element. This can also be the name of an element (e.g. value or list element) that contains the controller number of the associated controller element.

pdo PDO number.

start Start byte of data in PDO (0-based).

count Number of bytes of data to extract from PDO.

reverse Reverse PDO bytes: 0 - No, 1 - Yes.

mask PDO data extraction mask (in hexadecimal).

signed PDO data is signed: 0 - No, 1 - Yes.

Example
```
$Pdo = PDO,0x0000,UsbMACS4,1,0,4,0,0xFFFF,0
```

5.7.2.14 SDO

The **SDO** statement defines an SDO. It is not displayed in the dialog in any way.

Syntax *name* = **SDO**,*flags,ctrl,index,subindex,addndx,addsub,pollcnt*

name Element name.

flags The following flags are supported:

- 0x0001 SDO should be polled.
- 0x0002 SDO is read-only.
- 0x0004 Poll SDO once after initial connection (or reconnection).
- 0x0008 Poll SDO once if the value of either *addndx* or *addsub* changes.

ctrl Name of controller element. This can also be the name of an element (e.g. value or list element) that contains the controller number of the associated controller element.

index Base SDO index. This may be specified either as an integer or as a hexadecimal number.

subindex Base SDO subindex.

addndx Element name. The value of this element will be added to the value of *index* before the SDO is read or written. This should be 0 or - if no value is to be added.

addsub Element name. The value of this element will be added to the value of *subindex* before the SDO is read or written. This should be 0 or - if no value is to be added.

pollcnt Poll count. When polling is active, a new value will be polled after this many poll intervals. If this is 0, then the value will be polled after each poll interval. This will default to 0 if it is not specified.

Example `$Sdo2202-23 = SDO,0x0001,UsbMACS4,0x2202,23,-,-,1`

The *addndx* and *addsub* parameters are typically used in conjunction with **List,Axis** elements to select a specific axis when a controller contains multiple axes.

5.7.2.15 Slider

The **Slider** statement defines a slider.

Syntax *name* = **Slider**,*offset,x,y,width,height,flags,element,low,high,show*

name Element name.

offset Position offset. This is the name of the element to be used as the “base” position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be “absolute” (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

width Width (in pixels) of the box that will display the slider. The slider will be drawn vertically if the width is less than the height.

height Height (in pixels) of the box that will display the slider.

flags The following flags are supported:

0x0001 Tick marks will be drawn below the slider (or to the right for vertical sliders).

0x0002 Tick marks will be drawn above the slider (or to the left for vertical sliders).

0x0004 Update the slider value continuously. Otherwise, the slider value is only updated when the mouse button comes up at the end of the movement.

0x0008 Reverse the orientation of the slider. Note that the low and high values must not be reversed.

element Element name. The value for the slider is retrieved from this element. If the value of *element* cannot be determined (e.g. the controller may be disconnected), then the slider will be disabled. If this is not specified (i.e. if it is set to 0 or -), then the value of the slider will be maintained locally in the dialog.

low Lowest value of the slider.

high Highest value of the slider.

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

Example `$Value = Slider,-, 50,140,400,40,0x0007,Sdo,-10,10,-`

5.7.2.16 Table

The **Table** statement defines a table. It is used simply to associate strings with values. It is not displayed in the dialog in any way.

Syntax *name* = **Table**,*flags*,(*list*)

name Element name.

flags Currently, no flags are supported. This should be set to 0x0000.

list The list consists of string/value pairs separated by commas. The entire list must be enclosed in parentheses. The strings *must* be enclosed in quotes and the first character *must* be "*". The "*" will not be included in the string. The values may be specified either as integers or as hexadecimal numbers.

Example `$Numbers = Table,0x0000,("*One",1,"*Two",2,"*Three",3,"*Four",4)`

5.7.2.17 Text

The **Text** statement defines a text string that will be displayed in the dialog.

Syntax *name* = **Text**,*string,offset,x,y,width,height,just,font,flags,clr,show*

name Element name.

string Text string (up to 200 characters). This *must* be enclosed in quotes and *must* contain at least 1 character.

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

width Width (in pixels) of the text box that will enclose the text string. The text string will wrap within the box if necessary.

height Height (in pixels) of the text box that will enclose the text string.

just Justification: -1 - Left, 0 - Center, 1 - Right.

font The name of the **Font** element that specifies the font to be used. A positive integer may be specified instead of a **Font** element. In this case, a default font is used and the integer will specify the font height. If 0 is specified, then a default font height is used.

flags The following flags are supported:

0x0001 Use bold font (ignored if a **Font** is used).

clr The text will be displayed in this color. This may be specified either as a "red/green/blue" color in square brackets (i.e. [red.green.blue]) or by using one of the following predefined keywords:
Black, White, Gray, DlgGray, Red, Green, Blue, Yellow, Cyan, Magenta, DarkRed, DarkGreen, DarkBlue, Orange, Olive, Purple.

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

Example `$Label = Text,"Status",-,550,240,40,15,-1,Font1,0x0001,Red,-`

5.7.2.18 Value

The **Value** statement defines a value field. This is displayed as a Windows edit box in which the user can enter a value. Depending on the specified flags, it may or may not include a label and read/write buttons. The value will be displayed in red if it has been changed but not "committed". If the value corresponds to an SDO, then it is "committed" when the SDO has been written to the controller. Otherwise, the value is "committed" when the user moves to another field.

Syntax *name* = **Value**,*label,offset,x,y,flags,element,format,addval,scale,show*

name Element name.

label Text string to appear as field label. This *must* be enclosed in quotes and the first character *must* be "*". The "*" will not be displayed

offset Position offset. This is the name of the element to be used as the "base" position of this element. The *x,y* position will specify an offset relative to this base element. If the base element is moved to a different position in the dialog, then this element will move

along with it. This is often used for elements that are positioned within group boxes. If *offset* is set to 0 or -, then the *x,y* position will be "absolute" (i.e. relative to the window itself). This value must not be omitted.

x,y Position. These specify the horizontal and vertical position of the element (in pixels).

flags The following flags are supported:

- 0x0001 Value is read-only.
- 0x0002 Blink the value when Read button is pressed.
- 0x0004 Automatically write if user changes value and leaves field.
- 0x0008 Fade the value when it changes.
- 0x0010 Automatically write whenever the user presses the carriage return on the keyboard. The value will also be reselected so that a new value can be entered immediately. This flag is ignored if the Monitor is running in touchscreen mode.
- 0x0100 Use the "small" child dialog.
- 0x0200 Use the "display" child dialog (no read or write buttons).
- 0x0400 Hide the child dialog.

element Element name. The value for this field is retrieved from this element. This should be 0 or - if the value is not to be retrieved from any other element. If this is an **SDO** element, then "Read" and "Write" buttons may be shown. These buttons will read and write the SDO. If *element* is not specified, then flag 0x0200 is automatically included so that the buttons are not displayed. If *element* is specified but is *not* an **SDO** element, then flags 0x0001 and 0x0200 are automatically included to make this a read-only value.

format Display format: 0 - Numeric, 1 - Hexadecimal (no scaling).

addval Element name. The value of this element will be added to the value of *element* before the value is displayed. This should be 0 or - if no value is to be added.

scale Scale factor for display. This is a floating-point value. The value being displayed (i.e. after *addval* has been included) will be *divided* by this number before being displayed. The displayed value will be *multiplied* by this number before being used (e.g. before being written to an SDO). Set this to 0 if no scaling is to be done

show Show/hide flag. This is the name of the element to be used to determine whether the control is visible or hidden. If the element value is 0, then the control is hidden. Otherwise, the control is visible. If *show* is set to 0 or -, then the control will always be visible.

Example
```
$VelMax = Value,"*VELMAX",-,20,20,0x0008,Sdo,0,-,0,-
```
This will display a "normal" value field which includes a label, an edit box, and read/write buttons.

```
$Value1 = Value,"*",-,140,520,0x0108,Sdo,0,-,0,-
```
This will display a "small" value field which includes an edit box and small read/write buttons. There will be no label.

```
$Value2 = Value,"*",-,250,520,0x0109,Sdo,0,-,0,-
```

This will display a “small” read-only value field which includes a read-only edit box and a small read button. It includes no write button or label.

```
$Value3 = Value,"*",-,360,520,0x0200,Sdo,0,-,0,-
```

This will display a “small” value field which includes an edit box. It includes no read/write buttons or label.

5.7.3 Example Monitor Files

The following examples are provided to show the user how various capabilities of the Aposs Monitor can be implemented. The examples are also provided so that users can copy/paste when creating their own Monitor files.

All of the examples are complete and it should be possible for users to copy and run them verbatim. Except where stated otherwise, all of the examples require a connection to a single controller but the controller can be of any type and any interface connection can be used. When run, most of these examples will display a “Select Controller” dialog allowing the user to select the controller.

Fonts and Text Strings

Simple Polling

Background Fading

Displaying Status Bits

Polling SDO Values on Demand

Receiving PDO Messages

Reading and Writing SDO Values

Writing Specific SDO Values and Writing Multiple SDO Values

Setting Bits

Listboxes and SDO Index Offsets

Showing and Hiding Dialog Fields

Multiple Controllers, Polling Load, and Dynamically Selecting Controllers

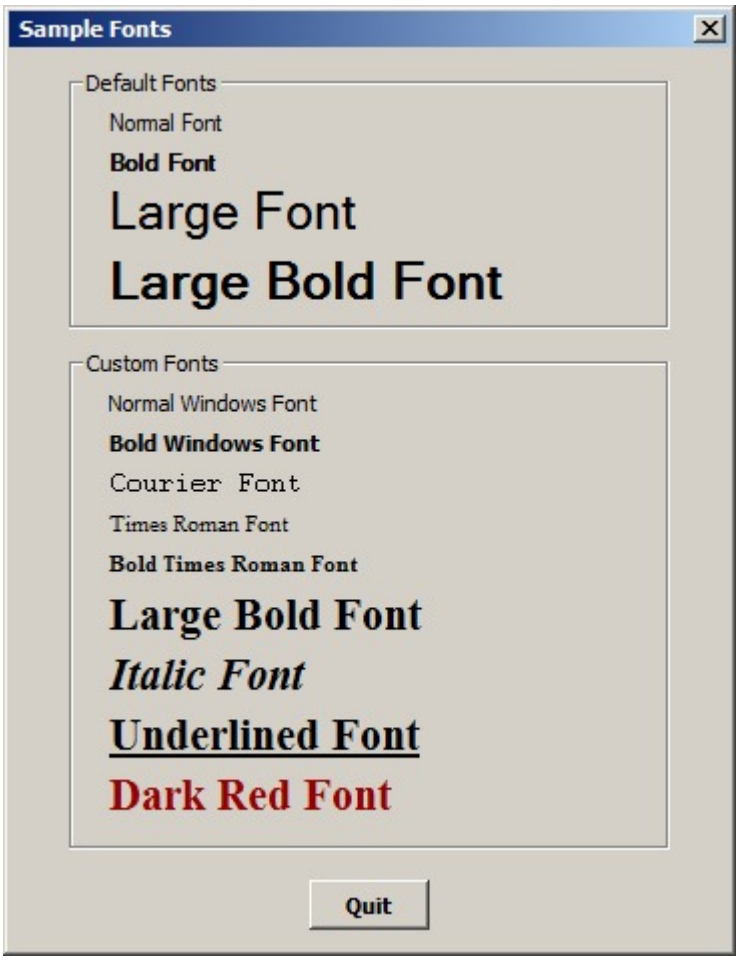
Specified Controllers and Interface Types

Displaying Analog Input

Displaying and Setting Regulation Parameters

Displaying and Setting User Parameters

5.7.3.1 Fonts and Text Strings

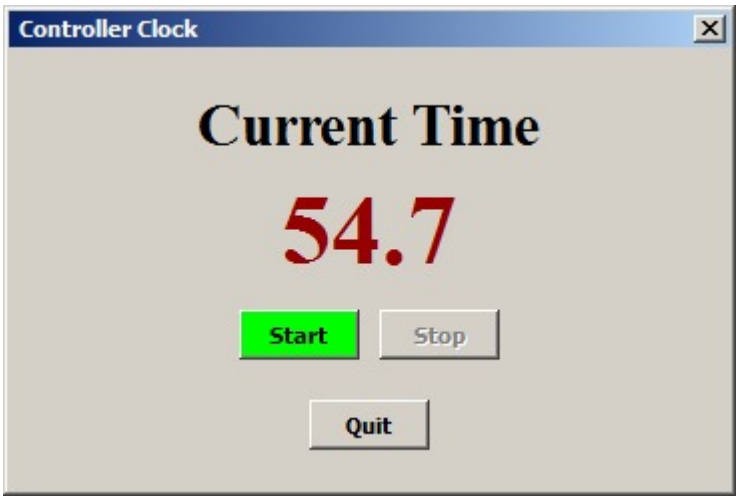



```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 450
WindowTitle  = Sample Fonts
WindowFlags  = 0x0022
OpFlags      = 0x0000

$DefFonts = Box,"*Default Fonts",-,30,10,300,130,0x0000,-
$Text1    = Text,"Normal Font",
DefFonts,20,20,200,15,-1,14,0x0000,Black,-
$Text2    = Text,"Bold Font",
DefFonts,20,40,200,15,-1,14,0x0001,Black,-
$Text3    = Text,"Large Font",
DefFonts,20,55,200,35,-1,30,0x0000,Black,-
$Text4    = Text,"Large Bold
Font",DefFonts,20,90,200,35,-1,30,0x0001,Black,-

$CstFonts = Box,"*Custom Fonts",-,30,150,300,250,0x0000,-
$Windows  = Font,0x0000,(-11,0,0,0,400,0,0,0,0,0,0,0,0,"MS Shell Dlg
2")
```

```
$Bold      = Font,0x0000,(-11,0,0,0,900,0,0,0,0,0,0,0,0,0,"MS Shell Dlg
2")
$Courier   = Font,0x0000,(-11,0,0,0,400,0,0,0,0,0,0,0,0,0,"Courier")
$Times     = Font,0x0000,(-11,0,0,0,400,0,0,0,0,0,0,0,0,0,"Times New
Roman")
$BoldT     = Font,0x0000,(-11,0,0,0,900,0,0,0,0,0,0,0,0,0,"Times New
Roman")
$Large     = Font,0x0000,(-22,0,0,0,900,0,0,0,0,0,0,0,0,0,"Times New
Roman")
$Italic    = Font,0x0000,(-22,0,0,0,900,1,0,0,0,0,0,0,0,0,"Times New
Roman")
$Under     = Font,0x0000,(-22,0,0,0,900,0,1,0,0,0,0,0,0,0,"Times New
Roman")
$Text11    = Text,"Normal Windows Font", CstFonts,20,
20,200,15,-1,Windows,0x0000,Black,-
$Text12    = Text,"Bold Windows Font", CstFonts,20,
40,200,15,-1,Bold, 0x0000,Black,-
$Text13    = Text,"Courier Font", CstFonts,20,
60,200,15,-1,Courier,0x0000,Black,-
$Text13    = Text,"Times Roman Font", CstFonts,20,
80,200,15,-1,Times, 0x0000,Black,-
$Text14    = Text,"Bold Times Roman
Font",CstFonts,20,100,200,15,-1,BoldT, 0x0000,Black,-
$Text15    = Text,"Large Bold Font", CstFonts,20,120,200,25,-1,Large,
0x0000,Black,-
$Text16    = Text,"Italic Font", CstFonts,20,150,200,25,-1,Italic,
0x0000,Black,-
$Text17    = Text,"Underlined Font", CstFonts,20,180,200,25,-1,Under,
0x0000,Black,-
$Text18    = Text,"Dark Red Font", CstFonts,20,210,200,25,-1,Large,
0x0000,DarkRed,-

$Quit      =
Button,Close,"*Quit",-,150,415,60,25,0x0002,0,Black,DlgGray,-
```

This example defines different character fonts and displays a text string using each font. It is specifically intended to allow users to copy/paste useful fonts.

5.7.3.2 Simple Polling




```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 220
WindowTitle  = Controller Clock
WindowFlags  = 0x0022
OpFlags      = 0x0000
PollInterval = 2000

$Ctrl    = Controller,0x0001,1,(*)
$Clock   = SDO,0x0001,Ctrl,0x2202,23,-,-

$Times   = Font,0x0000,(-30,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$BigFont = Font,0x0000,(-50,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$Label   = Text,"Current Time",-,20,20,320,30,0,Times,0x0,Black,-
$Display = Message,Value,-,20,60,320,60,0,BigFont,0x0,-,
{Clock,"%.1lf",-,1000.0,DarkRed}

$Start   =
Button,PollStart,"*Start",-,115,130,60,25,0x0002,0,Black,Green,-
$Stop    = Button,PollStop,
"*Stop",Start,70,0,60,25,0x0002,0,Black,Red,-

$Quit    =
Button,Close,"*Quit",-,150,175,60,25,0x0002,0,Black,DlgGray,-
```

This example shows how an SDO value can be polled and displayed. Please note the following points about the example:

1. “OpFlags” is set to 0x0000 so that polling is *not* started automatically. The “Start” button must be pressed to start polling. Setting “PollInterval” to 2000 will cause the controller to be polled approximately once every 2000 milliseconds.
2. The controller “$Ctrl” has flags set to 0x0001. This will cause the controller connection to be opened as

soon as the Monitor file is opened. Since "(*)" does not explicitly specify a particular controller, the "Select Controller" dialog will be displayed.

3. "$Clock" defines SDO 0x2202/23 (the system clock). Flags 0x0001 will cause the SDO value to be polled (from controller "$Ctrl") when polling is active.
4. "$Display" defines where in the dialog the clock value will be displayed. Note that a scale factor of 1000.0 is used. This will divide the value by 1000 before it is displayed in order to convert the value from milliseconds to seconds.
5. When the Monitor is first opened, "$Display" will attempt to retrieve the value to be displayed from "$Clock". Since "$Clock" does not yet have a value, this will cause "$Clock" to read the SDO value from "$Ctrl". If "$Ctrl" has not already been opened, then this will cause the controller to be opened. The SDO value will then be read and the value displayed.

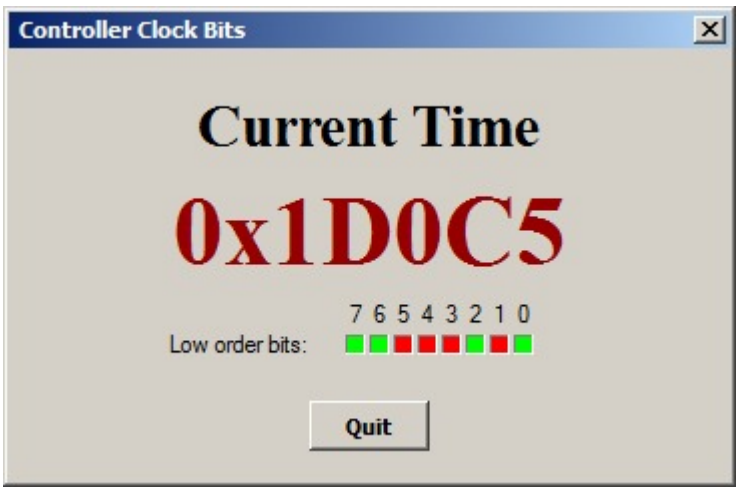
5.7.3.3 Background Fading

```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 180
WindowTitle  = Background Fading
WindowFlags  = 0x0022
OpFlags      = 0x0001
PollInterval = 8000
Fade         = 1,[255.128.0],2,[255.192.128],4,[255.224.192]

$Ctrl    = Controller,0x0001,1,(*)
$Clock   = SDO,0x0001,Ctrl,0x2202,23,-,-

$Times   = Font,0x0000,(-30,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$BigFont = Font,0x0000,(-50,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$Label   = Text,"Current Time",-,20,20,320,30,0,Times,0x0,Black,-
$Display = Message,Value,-,20,60,320,60,0,BigFont,0x0006,-,
{Clock,"0x%lX",-,0.0,Black}

$Quit    =
Button,Close,"*Quit",-,150,140,60,25,0x0002,0,Black,DlgGray,-
```

"Background fading" is intended to be used to "catch the eye" of users when displayed values change. Its use is appropriate for important values that change relatively infrequently. Although any colours can be used, it was designed to display a bright colour as soon as the value changes and then fade to paler and paler colours over time. This will give the user an impression of how long it has been since the value changed. Please note the following points about the example:

1. This example is based on the "Simple Polling" example. See that example for more detail.
2. "OpFlags" is set to 0x0001 so that polling will be started automatically.
3. "$Display" has flags set to 0x0006. This enables background fading and displays the value using a "sunken border". A sunken border gives a more aesthetic appearance when fading is enabled.

5.7.3.4 Displaying Status Bits



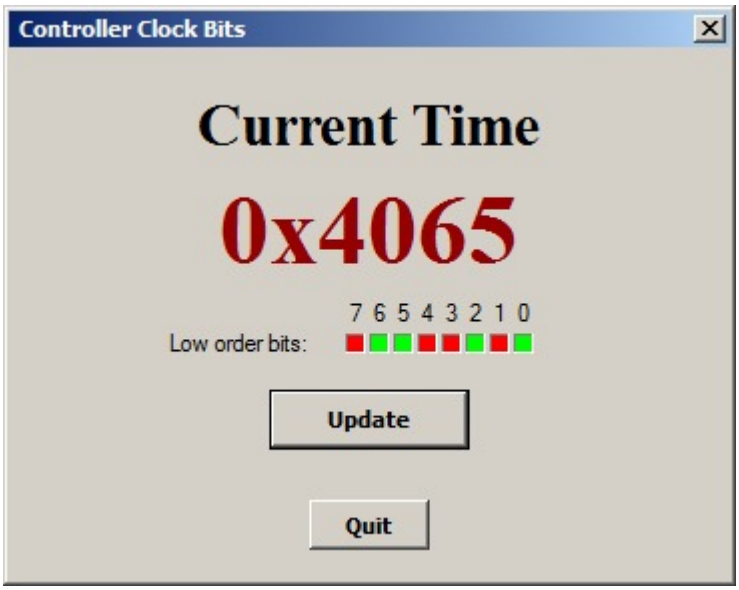

```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 215
WindowTitle  = Controller Clock Bits
WindowFlags  = 0x0022
OpFlags      = 0x0001
PollInterval = 4000

$Ctrl    = Controller,0x0001,1,(*)
$Clock   = SDO,0x0001,Ctrl,0x2202,23,-,-

$Times   = Font,0x0000,(-30,0,0,0,900,0,0,0,0,0,0,0,0,0,"Times New
Roman")
$BigFont = Font,0x0000,(-50,0,0,0,900,0,0,0,0,0,0,0,0,0,"Times New
Roman")
$Label   = Text,"Current Time",-,20,20,320,30,0,Times,0x0,Black,-
$Display = Message,Value,-,20,60,320,60,0,BigFont,0x0,-,
{Clock,"0x%lX",-,0.0,DarkRed}

$First   = Text,"Low order
```

```
bits:",-,80,140,70,15,-1,14,0x0000,Black,-
$Numbers = Text,"7  6  5  4  3  2  1
0",First,90,-15,90,15,-1,14,0x0000,Black,-
$Bit7    =
Bit,First,88,2,10,10,0x0000,Green,Red,DlgGray,Clock,0x0080,-,0
$Bit6    =
Bit,Bit7,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0040,-,0
$Bit5    =
Bit,Bit6,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0020,-,0
$Bit4    =
Bit,Bit5,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0010,-,0
$Bit3    =
Bit,Bit4,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0008,-,0
$Bit2    =
Bit,Bit3,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0004,-,0
$Bit1    =
Bit,Bit2,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0002,-,0
$Bit0    =
Bit,Bit1,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0001,-,0

$Quit    =
Button,Close,"*Quit",-,150,175,60,25,0x0002,0,Black,DlgGray,-
```

Individual bits within values can be displayed as colour patches. This is mostly intended to be used to display the bits of “status” words implemented within the user’s program. Please note the following points about the example:

1. This example is based on the “Simple Polling” example. See that example for more detail.
2. The eight low order bits from the clock are displayed as bit status values. In this example, the bit status will be green when the bit is 1, red when the bit is 0, and gray when the bit cannot be determined (for example, if the controller connection is lost). However, the user can choose any colours that are appropriate to the meaning of the status bit.
3. The status bit colours will change only when the “source” value changes (in this case, “$Clock”). As a result, polling should normally be enabled.
4. The value of any program variable can be read using SDO 0x2000. The details of how to use this SDO are beyond the scope of this example.

5.7.3.5 Polling SDO Values on Demand



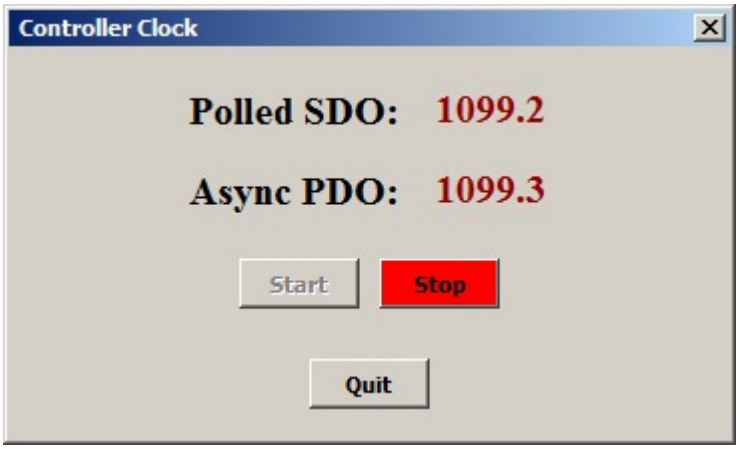

```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 265
WindowTitle  = Controller Clock Bits
WindowFlags  = 0x0022
OpFlags      = 0x0000

$Ctrl    = Controller,0x0001,1,(*)
$Clock   = SDO,0x0000,Ctrl,0x2202,23,-,-

$Times   = Font,0x0000,(-30,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$BigFont = Font,0x0000,(-50,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$Label   = Text,"Current Time",-,20,20,320,30,0,Times,0x0,Black,-
$Display = Message,Value,-,20,60,320,60,0,BigFont,0x0,-,
{Clock,"0x%lX",-,0.0,DarkRed}

$First   = Text,"Low order
bits:",-,80,140,70,15,-1,14,0x0000,Black,-
$Numbers = Text,"7  6  5  4  3  2  1
0",First,90,-15,90,15,-1,14,0x0000,Black,-
$Bit7    =
Bit,First,88,2,10,10,0x0000,Green,Red,DlgGray,Clock,0x0080,-,0
$Bit6    =
Bit,Bit7,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0040,-,0
$Bit5    =
Bit,Bit6,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0020,-,0
$Bit4    =
Bit,Bit5,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0010,-,0
```

```
$Bit3    =
Bit,Bit4,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0008,-,0
$Bit2    =
Bit,Bit3,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0004,-,0
$Bit1    =
Bit,Bit2,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0002,-,0
$Bit0    =
Bit,Bit1,12,0,10,10,0x0000,Green,Red,DlgGray,Clock,0x0001,-,0

$Update  =
Button,Get,"*Update",-,130,170,100,30,0x0002,0,Black,DlgGray,-,
{Clock}

$Quit    =
Button,Close,"*Quit",-,150,225,60,25,0x0002,0,Black,DlgGray,-
```

SDO values can be explicitly polled by the user at any time by creating a “Get” button. This is useful for updating the dialog when polling is not active or when a particular SDO is not being polled. Please note the following points about the example:

1. This example is based on the Displaying Status Bits example.
2. In this example, polling is not active. The SDO is read only once when the dialog is first opened.
3. Pressing the “Update” button will “get” (i.e. read) the “Clock” SDO value. This will cause the dialog to be updated with the new value.
4. Multiple SDO values can be read with a single button by creating a “Group” button. See the Writing Specific SDO Values and Writing Multiple SDO Values example.

5.7.3.6 Receiving PDO Messages




```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 195
WindowTitle  = Controller Clock
WindowFlags  = 0x0022
OpFlags      = 0x0000
```

```
PollInterval = 2000

$Ctrl   = Controller,0x0001,1,(*)
$Clock  = SDO,0x0001,Ctrl,0x2202,23,-,-
$PDO    = PDO,0x0000,Ctrl,1,4,4,0,0xFFFFFFFF,0

$Times  = Font,0x0000,(-20,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$LblSDO = Text,"Polled SDO:",-,90,20,110,35,-1,Times,0x0,Black,-
$DspSDO = Message,Value,LblSDO,120,-7,100,35,-1,Times,0x0,-,
{Clock,"%.1lf",-,1000.0,DarkRed}
$LblPDO = Text,"Async PDO:",LblSDO,0,40,110,35,-1,Times,0x0,Black,-
$DspPDO = Message,Value,LblPDO,120,-7,100,35,-1,Times,0x0,-,
{PDO,"%.1lf",-,1000.0,DarkRed}

$Start  =
Button,PollStart,"*Start",-,115,105,60,25,0x0002,0,Black,Green,-
$Stop   = Button,PollStop,
"*Stop",Start,70,0,60,25,0x0002,0,Black,Red,-

$Quit   =
Button,Close,"*Quit",-,150,155,60,25,0x0002,0,Black,DlgGray,-
```

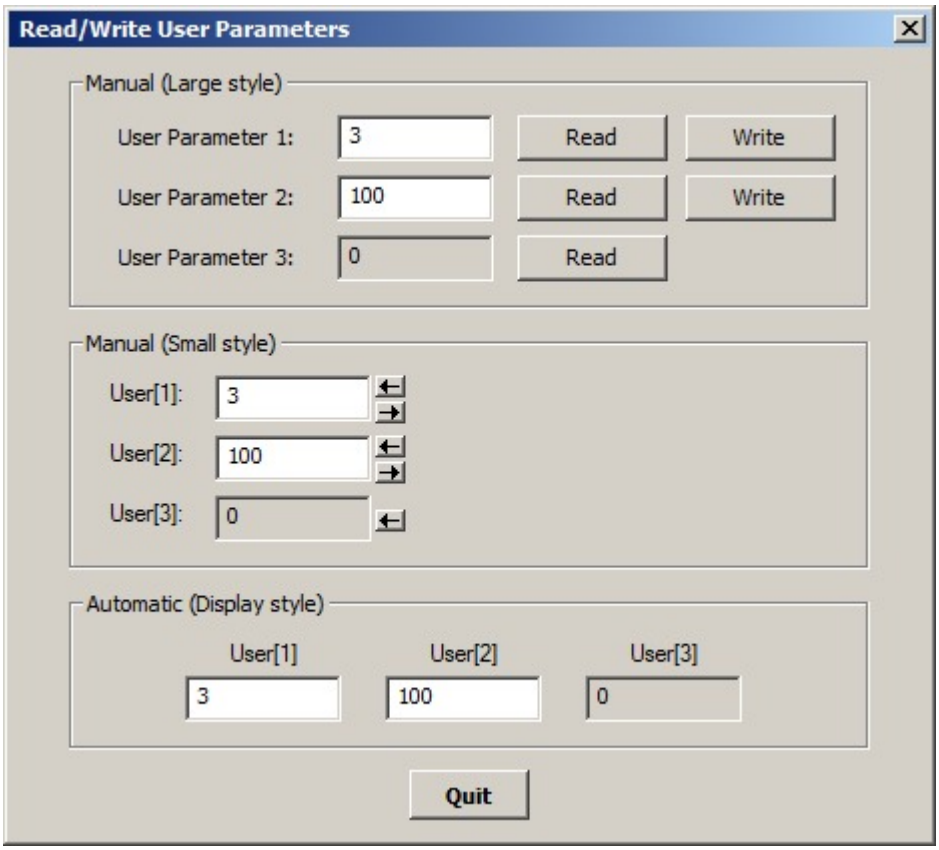
```
// Send PDO test program
// This test program sends PDO1 containing the actual position and
the
// system clock.

// Program Configuration
G_PDOCycle = 1000   // Cycle time for sending PDOs [ms]

// System Variables
C_SYSVAR_APOSQC      = 0x01250001  // Current position [qc]
C_SYSVAR_SYSCLOCK    = 0x01220217  // System clock [ms]
C_SYSVAR_TRANSTYPE   = 0x01180002  // PDO Transmission Type
C_SYSVAR_INHIBITTIME = 0x01180003  // PDO Min. delay between PDO
updates [n * 100us]
C_SYSVAR_EVENTTIME   = 0x01180005  // PDO Max. delay between PDO
send [ms]

// Link the PDOs
linksdo C_SYSVAR_APOSQC   32 1 "APOS-QC" 1  // Position data [qc, 32
Bit]
linksdo C_SYSVAR_SYSCLOCK 32 2 "SYSCLK"  1  // System clock [ms, 32
Bit]

// Set transmission type for PDO sending and send every 1 second
sysvar[C_SYSVAR_TRANSTYPE]   = 0xFF
```

```
sysvar[C_SYSVAR_INHIBITTIME] = G_PDOCycle * 10  // [* 100 us]
sysvar[C_SYSVAR_EVENTTIME]   = G_PDOCycle       // [* ms]

// Send NMT so that sending of PDOs starts
nmt = defcanout 0 2
canout nmt 0x0 ((0x01 << 8) + 0)
delay 100
candel nmt

// Main Program
start:
    delay 2000
    goto start
```

This examples shows how to receive data that is being sent by the controller in asynchronous PDO messages. These PDO messages are normally configured by a program running in the controller. Please note the following points about the example:

1. To use this example, the above program must be compiled and run in the controller. This program will configure the actual position and system clock into a PDO and then start sending the PDO messages.
2. The "PDO" statement specifies that the value is to be extracted from PDO1 messages starting at byte 4 and the value will be 4 bytes long. Since PDO1 is being used, the example must be run using a CAN connection; a serial or USB connection cannot be used. If the example is to be run using a serial or USB connection, then both the program and the example Monitor file must be changed so that they reference the serial PDO type that is supported on serial or USB connections.

5.7.3.7 Reading and Writing SDO Values



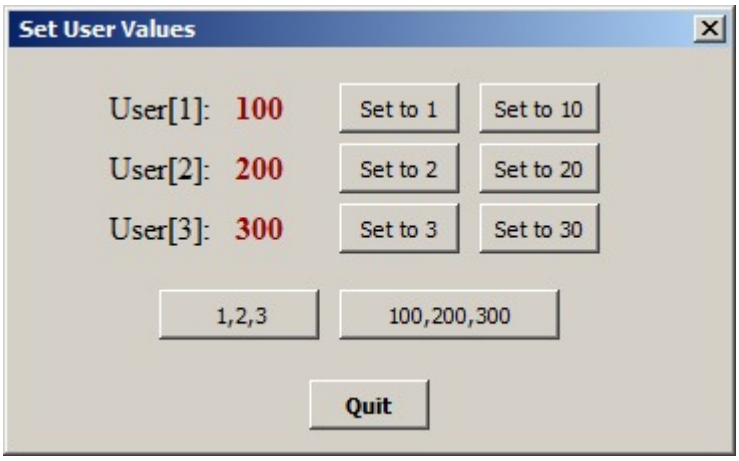

```
[VARIABLES]
WindowWidth  = 460
WindowHeight = 395
WindowTitle  = Read/Write User Parameters
WindowFlags  = 0x0022
OpFlags      = 0x0000

$Ctrl   = Controller,0x0000,1,(*)
$User1  = SDO,0x0000,Ctrl,0x2201,1,-,-
$User2  = SDO,0x0000,Ctrl,0x2201,2,-,-
$User3  = SDO,0x0000,Ctrl,0x2201,3,-,-

$Box1   = Box,"*Manual (Large style)",-,30,10,400,120,0x0000,-
$Large1 = Value,"*User Parameter 1:",Box1,
5,20,0x0000,User1,0,-,0,-
$Large2 = Value,"*User Parameter
2:",Large1,0,30,0x0000,User2,0,-,0,-
$Large3 = Value,"*User Parameter
3:",Large2,0,30,0x0001,User3,0,-,0,-

$Box2   = Box,"*Manual (Small style)",-,30,140,400,120,0x0000,-
$Label1 = Text,"User[1]:",Box2,20,25,40,15,-1,14,0x0000,Black,-
$Label2 = Text,"User[2]:",Label1,0,30,40,15,-1,14,0x0000,Black,-
```

```
$Label3 = Text,"User[3]:",Label2,0,30,40,15,-1,14,0x0000,Black,-
$Small1 = Value,"*",Label1,50,-5,0x0100,User1,0,-,0,-
$Small2 = Value,"*",Label2,50,-5,0x0100,User2,0,-,0,-
$Small3 = Value,"*",Label3,50,-5,0x0101,User3,0,-,0,-

$Box3   = Box,"*Automatic (Display style)",-,30,270,400,80,0x0000,-
$Title1 = Text,"User[1]",Box3,   80,25,40,15,-1,14,0x0000,Black,-
$Title2 = Text,"User[2]",Title1,100, 0,40,15,-1,14,0x0000,Black,-
$Title3 = Text,"User[3]",Title2,100, 0,40,15,-1,14,0x0000,Black,-
$Dsply1 = Value,"*",Title1,-25,15,0x0204,User1,0,-,0,-
$Dsply2 = Value,"*",Title2,-25,15,0x0204,User2,0,-,0,-
$Dsply3 = Value,"*",Title3,-25,15,0x0205,User3,0,-,0,-

$Quit   =
Button,Close,"*Quit",-,200,360,60,25,0x0002,0,Black,DlgGray,-
```

This example shows how specific SDO values can be read and written. Please note the following points about the example:

1. Values can be displayed using three different styles: Large, Small, and Display. The difference between the styles is only in the existence of the label and the size and existence on the “Read” and “Write” buttons. In all other ways, the behaviour of the three styles is identical. This example shows all three styles.
2. No controller is initially connected when this example is started so the values will be blank until a “Read” button is pressed for the first time.
3. The *same* three user parameter values are used in each of the style examples. As a result, when one of the parameters is modified, then the others will be automatically updated.
4. User parameter 3 is flagged as being a “Read only” value. In this case, the value will be displayed with a gray background and there will be no “Write” button.
5. Values can be modified by clicking in a field and typing a new value. Values will be displayed in red until they are actually written to the controller.
6. The “$Dsply1”, “$Dsply2”, and “$Dsply3” values have flag 0x0004 set. This will cause any changed value to be automatically written to the controller when the user leaves that value by clicking on a different value.

5.7.3.8 Writing Specific SDO Values and Writing Multiple SDO Values



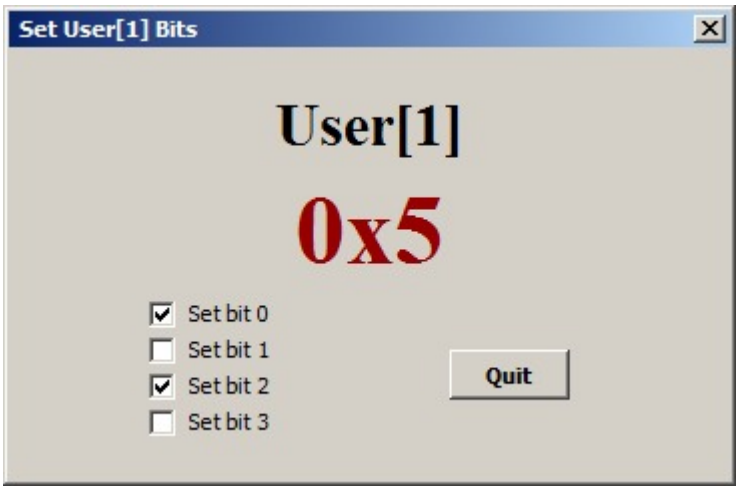
```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 200
WindowTitle  = Set User Values
WindowFlags  = 0x0022
OpFlags      = 0x0000

$Ctrl   = Controller,0x0001,1,(*)
$User1  = SDO,0x0000,Ctrl,0x2201,1,-,-
$User2  = SDO,0x0000,Ctrl,0x2201,2,-,-
$User3  = SDO,0x0000,Ctrl,0x2201,3,-,-

$Times  = Font,0x0000,(-16,0,0,0,400,0,0,0,0,0,0,0,0,0,"Times New Roman")
$Bold   = Font,0x0000,(-16,0,0,0,900,0,0,0,0,0,0,0,0,0,"Times New Roman")

$Label1 = Text,"User[1]:",-,50,20,50,20,0,Times,0x0,Black,-
$Dsp1   = Message,Value,Label1,55,0,40,20,0,Bold,0x0,-,
{User1,0,-,0,DarkRed}
$Set1   = Button,Set,"*Set to
1",Dsp1,60,-3,60,25,0x0000,0,Black,DlgGray,-,{User1,1,-}
$Set10  = Button,Set,"*Set to
10",Set1,70,0,60,25,0x0000,0,Black,DlgGray,-,{User1,10,-}

$Label2 = Text,"User[2]:",Label1,0,30,50,20,0,Times,0x0,Black,-
$Dsp2   = Message,Value,Label2,55,0,40,20,0,Bold,0x0,-,
{User2,0,-,0,DarkRed}
$Set2   = Button,Set,"*Set to
2",Dsp2,60,-3,60,25,0x0000,0,Black,DlgGray,-,{User2,2,-}
$Set20  = Button,Set,"*Set to
20",Set2,70,0,60,25,0x0000,0,Black,DlgGray,-,{User2,20,-}
$Label3 = Text,"User[3]:",Label2,0,30,50,20,0,Times,0x0,Black,-

$Dsp3   = Message,Value,Label3,55,0,40,20,0,Bold,0x0,-,
{User3,0,-,0,DarkRed}
$Set3   = Button,Set,"*Set to
3",Dsp3,60,-3,60,25,0x0000,0,Black,DlgGray,-,{User3,3,-}
$Set30  = Button,Set,"*Set to
30",Set3,70,0,60,25,0x0000,0,Black,DlgGray,-,{User3,30,-}

$All123 = Button,Group,"*
1,2,3",Label3,25,40,80,25,0x0200,0,Black,DlgGray,-,{Set1,Set2,Set3}

$S100   = Button,Set,"*",-,0,0,60,25,0x0001,0,Black,DlgGray,-,
{User1,100,-}
$S200   = Button,Set,"*",-,0,0,60,25,0x0001,0,Black,DlgGray,-,
{User2,200,-}
$S300   = Button,Set,"*",-,0,0,60,25,0x0001,0,Black,DlgGray,-,
```

```
{User3,300,-}
$Set100 =
Button,Group,"*100,200,300",All123,90,0,110,25,0x0200,0,Black,DlgGray,-,
{S100,S200,S300}

$Quit    = Button,Close,"*Quit",-,150,165,60,25,0x0002,0,Black,DlgGray,-
```

This example shows how specific values can be written to SDO values. It also shows how multiple SDO values can be written at the same time. Please note the following points about the example:

1. The example defines three SDO values to access the first three user parameters.
2. For each user parameter, two “Set” buttons are defined which will set the SDO value to the specific value shown on the button.
3. A “Group” button labelled “1,2,3” is defined which will “press” the “Set1”, “Set2”, and “Set3” buttons. This will cause all three SDO values to be set with a single button.
4. A “Group” button labelled “100,200,300” is also defined which will set the SDO values to 100, 200, and 300. This is done using “hidden” buttons. Hidden buttons allow values to be set without requiring buttons to be visible in the dialog.
5. “Group” buttons can also be used to poll multiple SDO values by using the 0x0100 flag.

5.7.3.9 Setting Bits



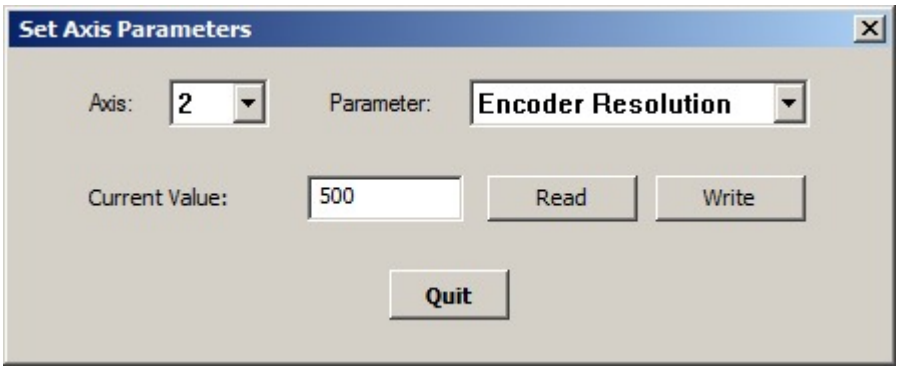

```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 215
WindowTitle  = Set User[1] Bits
WindowFlags  = 0x0022
OpFlags      = 0x0000

$Ctrl    = Controller,0x0001,1,(*)
$User1   = SDO,0x0000,Ctrl,0x2201,1,-,-

$Times   = Font,0x0000,(-30,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
```

```
$BigFont = Font,0x0000,(-50,0,0,0,900,0,0,0,0,0,0,0,0,0,"Times New
Roman")
$Windows = Font,0x0000,(-11,0,0,0,400,0,0,0,0,0,0,0,0,0,"MS Shell Dlg
2")
$Label   = Text,"User[1]",-,20,20,320,35,0,Times,0x0,Black,-
$Display = Message,Value,-,20,60,320,60,0,BigFont,0x0,-,
{User1,1,-,0,DarkRed}

$Check0  = Checkbox,-,70,125,0x0,User1,0x1,-
$Check1  = Checkbox,Check0,0,18,0x0,User1,0x2,-
$Check2  = Checkbox,Check1,0,18,0x0,User1,0x4,-
$Check3  = Checkbox,Check2,0,18,0x0,User1,0x8,-
$SetBit0 = Text,"Set bit
0",Check0,20,0,100,16,-1,Windows,0x0,Black,-
$SetBit1 = Text,"Set bit
1",Check1,20,0,100,16,-1,Windows,0x0,Black,-
$SetBit2 = Text,"Set bit
2",Check2,20,0,100,16,-1,Windows,0x0,Black,-
$SetBit3 = Text,"Set bit
3",Check3,20,0,100,16,-1,Windows,0x0,Black,-

$Quit    =
Button,Close,"*Quit",-,220,150,60,25,0x0002,0,Black,DlgGray,-
```

This example shows how individual bits of a value can be set using checkboxes. Please note the following points about the example:

1. This example allows the user to set the low order 4 bits of user parameter 1.
2. When the user checks or clears a checkbox, then the corresponding bit is changed and the entire value is written to the controller.

5.7.3.10 Listboxes and SDO Index Offsets



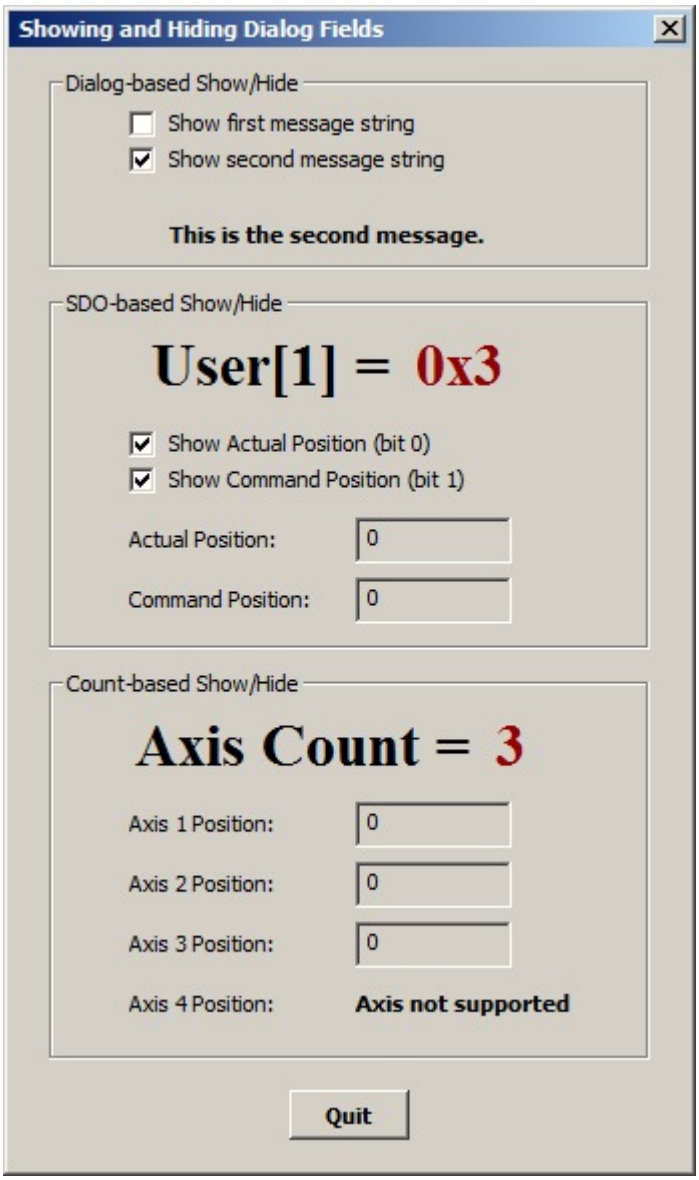

```
[VARIABLES]
WindowWidth  = 440
WindowHeight = 155
WindowTitle  = Set Axis Parameters
```

```
WindowFlags  = 0x0022
OpFlags      = 0x0000

$Ctrl    = Controller,0x0001,1,(*)

$AxisLbl = Text,"Axis:",-,40,20,40,15,-1,14,0x0000,Black,-
$Axis    = List,Axis,AxisLbl,40,-5,50,200,0x0000,-,{Ctrl}

$ParmLbl =
Text,"Parameter:",AxisLbl,120,0,60,15,-1,14,0x0000,Black,-
$Param   = List,Table,ParmLbl,70,-5,170,200,0x0000,-,{ParmTbl}
$ParmTbl = Table,0x0000,("*Maximum Speed [rpm]",2,"*Min. Accel. time
[ms]",32,"*Encoder Resolution",3)

$AxisSdo = SDO,0x000C,Ctrl,0x2400,0,Axis,Param
$Display = Value,"*Current Value:",-,20,60,0x0000,AxisSdo,0,-,0,-

$Quit    =
Button,Close,"*Quit",-,190,110,60,25,0x0002,0,Black,DlgGray,-
```

This example shows various ways that listboxes can be used. It also shows how SDO index offsets can be used to reference different SDO values. Please note the following points about the example:

1. The “$Axis” listbox will contain a list of the axis numbers available on controller “$Ctrl”. The “value” of the listbox will be the axis number of the currently selected axis.
2. The “$Param” listbox will contain the entries from table “$ParmTbl”. This table is set up so that the “value” will be the SDO subindex required to access the corresponding parameter. For example, the minimum acceleration time (RAMP_MIN) uses SDO subindex 32.
3. The SDO “$AxisSdo” has the “addndx” parameter set to “$Axis” and the “addsub” parameter set to “$Param”. When the SDO is read or written, the currently selected axis number (from “$Axis”) will be added to the base SDO index (i.e. 0x2400) and the currently selected parameter subindex (from “$Param”) will be added to the base SDO subindex (i.e. 0). This will give the SDO index and subindex necessary to reference the selected parameter of the selected axis.
4. When a different axis or parameter is selected in the listboxes, then the 0x0008 flag bit in “$AxisSdo” will cause the SDO value to be automatically reread so that the displayed value matches the newly selected axis and parameter.

5.7.3.11 Showing and Hiding Dialog Fields




```
[VARIABLES]
WindowWidth  = 340
WindowHeight = 560
WindowTitle  = Showing and Hiding Dialog Fields
WindowFlags  = 0x0022
OpFlags      = 0x0001
PollInterval = 1000

$Times   = Font,0x0000,(-30,0,0,0,900,0,0,0,0,0,0,0,0,0,"Times New
Roman")
$Windows = Font,0x0000,(-11,0,0,0,400,0,0,0,0,0,0,0,0,0,"MS Shell Dlg
2")
$Bold    = Font,0x0000,(-11,0,0,0,900,0,0,0,0,0,0,0,0,0,"MS Shell Dlg
2")
```

```
$Ctrl    = Controller,0x0001,1,(*)
$AxisCnt = SDO,0x0004,Ctrl,0x2209,5,-,-
$User1   = SDO,0x0004,Ctrl,0x2201,1,-,-
$ActSDO1 = SDO,0x0001,Ctrl,0x2500,1,-,-
$ActSDO2 = SDO,0x0001,Ctrl,0x2501,1,-,-
$ActSDO3 = SDO,0x0001,Ctrl,0x2502,1,-,-
$ActSDO4 = SDO,0x0001,Ctrl,0x2503,1,-,-
$CmdSDO1 = SDO,0x0001,Ctrl,0x2500,2,-,-

$Box1    = Box,"*Dialog-based Show/Hide",-,20,10,300,100,0x0000,-
$DlgShw1 = Checkbox,Box1,40,20,0x0,-,0x1,-
$DlgShw2 = Checkbox,DlgShw1,0,18,0x1,-,0x1,-
$DlgLbl1 = Text,"Show first message string",
DlgShw1,20,0,150,16,-1,Windows,0x0,Black,-
$DlgLbl2 = Text,"Show second message string",
DlgShw2,20,0,150,16,-1,Windows,0x0,Black,-
$DlgTxt1 = Text,"This is the first message.",
DlgLbl2,0,20,200,35,-1,Bold,0x0,Black,DlgShw1
$DlgTxt2 = Text,"This is the second
message.",DlgTxt1,0,18,200,35,-1,Bold,0x0,Black,DlgShw2

$Box2    = Box,"*SDO-based Show/Hide",Box1,0,110,300,180,0x0000,-
$UserLbl = Text,"User[1] =",Box2,20,20,150,35,1,Times,0x0,Black,-
$UserDsp = Message,Value,UserLbl,160,0,110,35,-1,Times,0x0,-,
{User1,1,-,0,DarkRed}
$ActShow = Checkbox,UserLbl,20,50,0x0,User1,0x1,-
$CmdShow = Checkbox,ActShow, 0,18,0x0,User1,0x2,-
$SetBit0 = Text,"Show Actual Position (bit 0)",
ActShow,20,0,150,16,-1,Windows,0x0,Black,-
$SetBit1 = Text,"Show Command Position (bit
1)",CmdShow,20,0,150,16,-1,Windows,0x0,Black,-
$ActLbl  = Text,"Actual Position:",
CmdShow,0,30,100,35,-1,Windows,0x0,Black,ActShow
$CmdLbl  = Text,"Command Position:",ActLbl,
0,30,100,35,-1,Windows,0x0,Black,CmdShow
$ActVal  = Value,"*",ActLbl,110,-7,0x0201,ActSDO1,0,-,0,ActShow
$CmdVal  = Value,"*",CmdLbl,110,-7,0x0201,CmdSDO1,0,-,0,CmdShow

$Hide    = Constant,0x0000,0
$AxsOk1  = Bit,-,0,0,5,5,0x0007,Black,Black,Black,AxisCnt,-1,Hide,1
// True if axis count >= 1.
$AxsOk2  = Bit,-,0,0,5,5,0x0007,Black,Black,Black,AxisCnt,-1,Hide,2
// True if axis count >= 2.
$AxsOk3  = Bit,-,0,0,5,5,0x0007,Black,Black,Black,AxisCnt,-1,Hide,3
// True if axis count >= 3.
$AxsOk4  = Bit,-,0,0,5,5,0x0007,Black,Black,Black,AxisCnt,-1,Hide,4
// True if axis count >= 4.
```

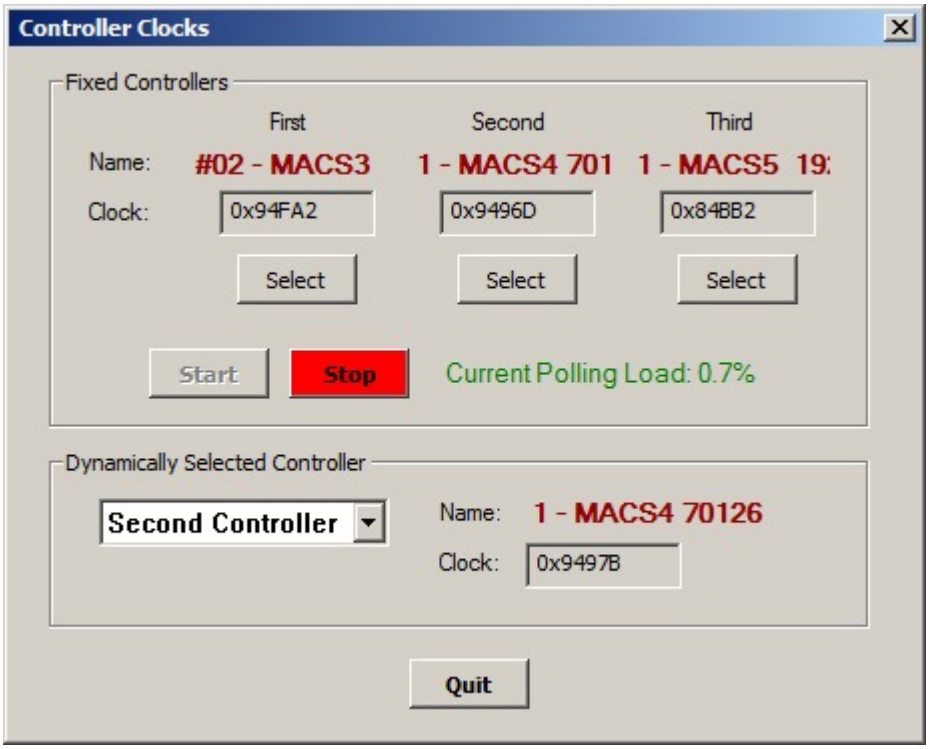
```
$AxsNOk1 = Bit,-,0,0,5,5,0x0009,Black,Black,Black,AxisCnt,-1,Hide,1
// True if axis count < 1.
$AxsNOk2 = Bit,-,0,0,5,5,0x0009,Black,Black,Black,AxisCnt,-1,Hide,2
// True if axis count < 2.
$AxsNOk3 = Bit,-,0,0,5,5,0x0009,Black,Black,Black,AxisCnt,-1,Hide,3
// True if axis count < 3.
$AxsNOk4 = Bit,-,0,0,5,5,0x0009,Black,Black,Black,AxisCnt,-1,Hide,4
// True if axis count < 4.

$Box3    = Box,"*Count-based Show/Hide",Box2,0,190,300,195,0x0000,-
$CntLbl  = Text,"Axis Count =",Box3,20,20,190,35,1,Times,0x0,Black,-
$CntDsp  = Message,Value,CntLbl,200,0,70,35,-1,Times,0x0,-,
{AxisCnt,0,-,0,DarkRed}
$AxsLbl1 = Text,"Axis 1 Position:",CntLbl,
20,50,100,35,-1,Windows,0x0,Black,-
$AxsLbl2 = Text,"Axis 2 Position:",AxsLbl1,
0,30,100,35,-1,Windows,0x0,Black,-
$AxsLbl3 = Text,"Axis 3 Position:",AxsLbl2,
0,30,100,35,-1,Windows,0x0,Black,-
$AxsLbl4 = Text,"Axis 4 Position:",AxsLbl3,
0,30,100,35,-1,Windows,0x0,Black,-
$AxsVal1 = Value,"*",AxsLbl1,110,-7,0x0201,ActSDO1,0,-,0,AxsOk1
$AxsVal2 = Value,"*",AxsLbl2,110,-7,0x0201,ActSDO2,0,-,0,AxsOk2
$AxsVal3 = Value,"*",AxsLbl3,110,-7,0x0201,ActSDO3,0,-,0,AxsOk3
$AxsVal4 = Value,"*",AxsLbl4,110,-7,0x0201,ActSDO4,0,-,0,AxsOk4
$AxsMsg1 = Text,"Axis not
supported",AxsLbl1,113,0,120,35,-1,Bold,0x0,Black,AxsNOk1
$AxsMsg2 = Text,"Axis not
supported",AxsLbl2,113,0,120,35,-1,Bold,0x0,Black,AxsNOk2
$AxsMsg3 = Text,"Axis not
supported",AxsLbl3,113,0,120,35,-1,Bold,0x0,Black,AxsNOk3
$AxsMsg4 = Text,"Axis not
supported",AxsLbl4,113,0,120,35,-1,Bold,0x0,Black,AxsNOk4

$Quit    =
Button,Close,"*Quit",-,140,520,60,25,0x0002,0,Black,DlgGray,-
```

This example shows various ways to show and hide dialog fields. Please note the following points about the example:

1. The “Dialog-based Show/Hide” box shows how checkboxes can be used to show or hide fields. Note that these checkboxes have no “source” element specified. In this case, they will be enabled and initialized when the dialog is first displayed and their values will be maintained locally. The two checkboxes show how checkboxes can be initially unchecked or initially checked.
2. The “SDO-based Show/Hide” box shows how checkboxes can be associated with SDO values from a controller and used to show or hide fields at the same time that the SDO value is updated on the controller. “Bit” fields can be used for the same purpose without allowing the user write access to the SDO values. Note that “Bit” fields can themselves be hidden in which case they can control the visibility of other fields without being visible themselves.

3. The “Count-based Show/Hide” box shows how fields can be shown or hidden based on comparing values retrieved from a controller.

5.7.3.12 Multiple Controllers, Polling Load, and Dynamically Selecting Controllers



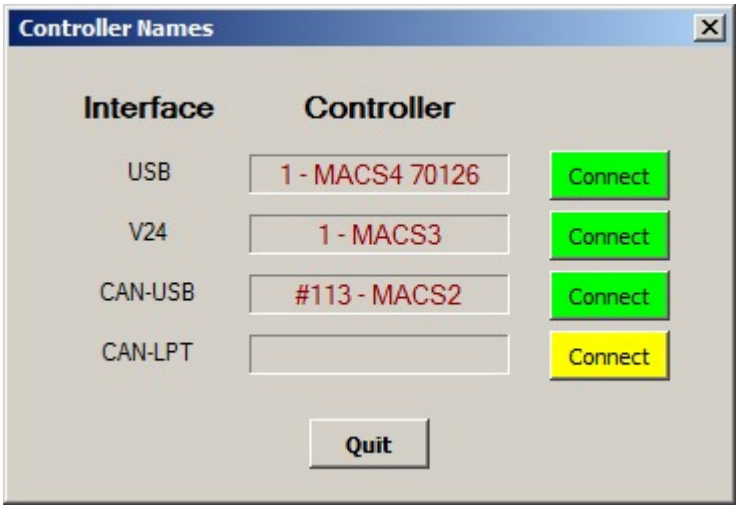

```
[VARIABLES]
WindowWidth  = 455
WindowHeight = 345
WindowTitle  = Controller Clocks
WindowFlags  = 0x0022
OpFlags      = 0x0000
PollInterval = 4000

$Ctrl1   = Controller,0x0001,1,(*)
$Ctrl2   = Controller,0x0001,2,(*)
$Ctrl3   = Controller,0x0001,3,(*)
$Clock1  = SDO,0x0001,Ctrl1,0x2202,23,-,-
$Clock2  = SDO,0x0001,Ctrl2,0x2202,23,-,-
$Clock3  = SDO,0x0001,Ctrl3,0x2202,23,-,-

$Box1    = Box,"*Fixed Controllers",-,20,10,410,180,0x0000,-
$Label1  = Text,"Name:",Box1,20,40,30,15,-1,14,0x0000,Black,-
$Label2  = Text,"Clock:",Label1,0,25,30,15,-1,14,0x0000,Black,-
$Title1  = Text,"First",Label1,45,-20,110,15,0,14,0x0000,Black,-
```

```
$Name1   = Message,Controller,Title1,5,18,100,20,-1,16,0x0001,-,
{Ctrl1,DarkRed}
$Value1  = Value,"*",Name1,12,20,0x0201,Clock1,1,-,0,-
$Select1 =
Button,Controller,"*Select",Value1,12,35,60,25,0x0100,0,Black,DlgGray,-,
{Ctrl1}

$Title2  = Text,"Second",Title1,110,0,110,15,0,14,0x0000,Black,-
$Name2   = Message,Controller,Title2,5,18,100,20,-1,16,0x0001,-,
{Ctrl2,DarkRed}
$Value2  = Value,"*",Name2,12,20,0x0201,Clock2,1,-,0,-
$Select3 =
Button,Controller,"*Select",Value2,12,35,60,25,0x0100,0,Black,DlgGray,-,
{Ctrl2}

$Title3  = Text,"Third", Title2,110,0,110,15,0,14,0x0000,Black,-
$Name3   = Message,Controller,Title3,5,18,100,20,-1,16,0x0001,-,
{Ctrl3,DarkRed}
$Value3  = Value,"*",Name3,12,20,0x0201,Clock3,1,-,0,-
$Select3 =
Button,Controller,"*Select",Value3,12,35,60,25,0x0100,0,Black,DlgGray,-,
{Ctrl3}

$Start   =
Button,PollStart,"*Start",Box1,50,140,60,25,0x0002,0,Black,Green,-
$Stop    = Button,PollStop,
"*Stop",Start,70,0,60,25,0x0002,0,Black,Red,-
$Load    = Message,Load,Stop,75,2,170,20,-1,16,0x0,-,{"Current Polling
Load: %.1lf%%",DarkGreen}

$Box2    = Box,"*Dynamically Selected
Controller",Box1,0,190,410,90,0x0000,-
$CtrlTbl = Table,0x0000,("*First Controller",1,"*Second
Controller",2,"*Third Controller",3)
$Ctrl    = List,Table,Box2,25,25,145,200,0x0000,-,{CtrlTbl}
$Clock   = SDO,0x0005,Ctrl,0x2202,23,-,-
$Label3  = Text,"Name:",Ctrl,170,0,30,15,-1,14,0x0000,Black,-
$Label4  = Text,"Clock:",Label3,0,25,30,15,-1,14,0x0000,Black,-
$Name    = Message,Controller,Label3,43,-3,130,20,-1,16,0x0001,-,
{Ctrl,DarkRed}
$Value   = Value,"*",Label4,40,-6,0x0201,Clock,1,-,0,-

$Quit    = Button,Close,"*Quit",-,200,305,60,25,0x0002,0,Black,DlgGray,-
```

This example shows how multiple controllers can be handled and how different controllers can be dynamically selected by the user. The example is designed to work with three different controllers, however it can be run using only a single controller; simply select the same controller three times. Please note the following points about the example:

1. Three controllers are defined and the 0x0001 flags means that they will be connected when the Monitor starts. The “Select Controller” dialog will be displayed three times, once for each controller. If this dialog is “cancelled”, then a corresponding controller is not opened. At any later time, the “Select” buttons can be used to open controllers or select different controllers.
2. Three system clock SDO values are defined and each is explicitly associated with a different controller. These values will be polled.
3. Polling can be started and stopped but it is not started automatically. Polling consumes resources on the PC; the more values that are polled, the more resources that are consumed. The “$Load” statement shows how to display the amount of resources currently being consumed by the polling. If this value exceeds 100%, then the Monitor is not able to poll all the values within the time specified by the poll interval and it will start to “fall behind”. In this case, the Monitor file should be redesigned so that either fewer values are polled or the poll interval is increased.
4. Any of the three controllers can also be selected dynamically. The “$Ctrl” listbox contains a list of the available controllers. The “value” of the listbox will be the controller number defined for each of the three controllers (i.e. in the “$Ctrl1”, “$Ctrl2”, and “$Ctrl3” statements). The SDO “$Clock” is then defined so that it points to the “$Ctrl” listbox rather than to an explicit controller. When the SDO value is read or written, then it will be done using the controller currently selected in the listbox. If a different controller is selected in the listbox, then the SDO value will automatically be reread from the newly selected controller.
5. Note that when running this example, the clock value shown in the “Dynamically Selected Controller” section will not match the clock value of the same controller in the “Fixed Controllers” section. This happens because the clock value is polled two times: once as “$Clock” and once as “$Clock1”, “$Clock2”, or “$Clock3”.

5.7.3.13 Specified Controllers and Interface Types




```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 225
WindowTitle  = Controller Names
WindowFlags  = 0x0022
OpFlags      = 0x0000

// USB Interface           ID  Interface
//                          |     |
```

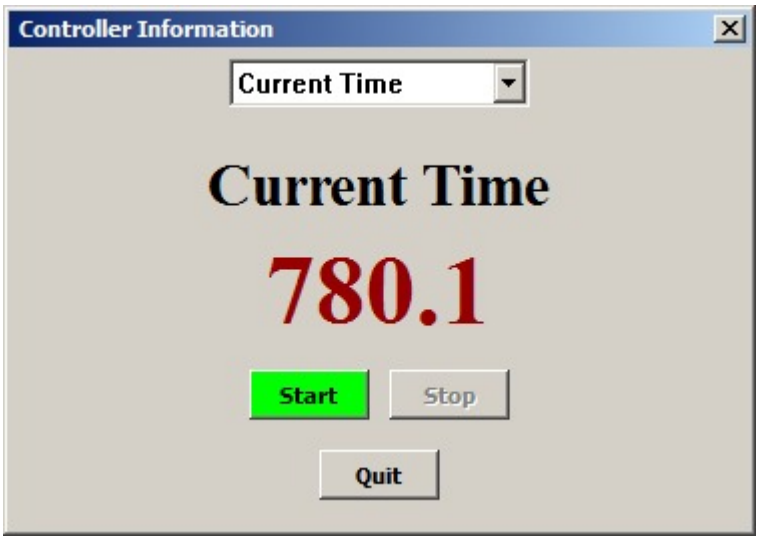
```
$Usb = Controller,0x0000,1,(1,   USB,6,50,0x0,921600,1)

// V24 Interface           ID  Interface  Port  Baudrate
//                          |     |        |       |
$V24 = Controller,0x0000,2,(1,   V24,      0,   115200,0,6,50,0x7,0,25)
// Port: 0 - COM1, 1 - COM2, etc.

// CAN-USB Interface        ID  Interface  Port  Speed  Scanfrom  Scanto
//                           |     |        |      |       |         |
$Can = Controller,0x0000,3,(113,  COM,      9,     7,     113,
113,0x0)
// Port:  1 - COM1, 2 - COM2, etc.
// Speed: 1 - 10kbps, 2 - 20kbps, 3 - 50kbps, 4 - 100kbps, 5 - 125kbps, 6
- 250kbps, 7 - 500kbps, 8 - 1000kbps

// CAN-LPT Interface          ID  Interface  Speed  Scanfrom  Scanto
//                             |     |         |       |         |
$Lpt    = Controller,0x0000,4,(2,   LPT,-1,7,  7,      1,
32,10,10,0x0,3)
// Speed: 1 - 10kbps, 2 - 20kbps, 3 - 50kbps, 4 - 100kbps, 5 - 125kbps, 6
- 250kbps, 7 - 500kbps, 8 - 1000kbps

$Title1 = Text,"Interface",-,30,20,80,15,0,18,0x0001,Black,-
$Title2 = Text,"Controller",Title1,90,0,130,15,0,18,0x0001,Black,-

$Label1 = Text,"USB",Title1,0,35,80,15,0,14,0x0000,Black,-
$Name1  = Message,Controller,Label1,90,-2,130,20,0,16,0x0002,-,
{Usb,DarkRed}
$Btn1   =
Button,Controller,"*Connect",Name1,150,-2,60,25,0x0600,0,Black,DlgGray,-,
{Usb}

$Label2 = Text,"V24",Label1,0,30,80,15,0,14,0x0000,Black,-
$Name2  = Message,Controller,Label2,90,-2,130,20,0,16,0x0002,-,
{V24,DarkRed}
$Btn2   =
Button,Controller,"*Connect",Name2,150,-2,60,25,0x0600,0,Black,DlgGray,-,
{V24}

$Label3 = Text,"CAN-USB",Label2,0,30,80,15,0,14,0x0000,Black,-
$Name3  = Message,Controller,Label3,90,-2,130,20,0,16,0x0002,-,
{Can,DarkRed}
$Btn3   =
Button,Controller,"*Connect",Name3,150,-2,60,25,0x0600,0,Black,DlgGray,-,
{Can}

$Label4 = Text,"CAN-LPT",Label3,0,30,80,15,0,14,0x0000,Black,-
$Name4  = Message,Controller,Label4,90,-2,130,20,0,16,0x0002,-,
```

```
{Lpt,DarkRed}
$Btn4   =
Button,Controller,"*Connect",Name4,150,-2,60,25,0x0600,0,Black,DlgGray,-,
{Lpt}

$Quit   = Button,Close,"*Quit",-,150,185,60,25,0x0002,0,Black,DlgGray,-
```

This example shows how specific controllers and specific interface types can be predefined. The example is intended to allow users to copy/paste controller connections. As it is written, this Monitor example expects to be able to connect to four different controllers on four different interface types. Please note the following points about the example:

1. Four different controllers are defined: Controller 1 on a USB interface, controller 1 on a V24 interface, controller 113 on a CAN-USB interface, and controller 2 on a CAN-LPT interface.
2. None of the controllers will be connected automatically. Each controller has a “Connect” button that must be pressed to connect the controller. This allows the example to be run with fewer than four controllers.
3. Status colour is enabled for the “Connect” buttons. If a controller is not connected, then the button will be yellow. If a controller is connected, then the button will be green. If the controller connection has been lost, then the button will be red. Note that when a controller connection is lost, the button will change to red only the next time that the controller is referenced. This will happen when an SDO value is read or written. If polling is active, then it will also happen at each poll interval time. The button will also change to red if the controller goes into an error state, however this colour change is only supported when polling is active.
4. If a controller connection has been lost, then pressing the “Connect” button will attempt to reconnect the controller.

5.7.3.14 Child Dialogs



ChildDialogs.zbm

```
[VARIABLES]

WindowWidth  = 370
WindowHeight = 240
WindowTitle  = Controller Information
```

```
WindowFlags  = 0x0022

OpFlags      = 0x0000

$FuncTbl = Table,0x0000,("*Current Time",1,"*Digital Inputs",2)
$Func    = List,Table,-,110,5,150,200,0x0000,-,{FuncTbl}
$Hide    = Constant,0x0000,0
$ShowClk = Bit,-,0,0,10,10,0x0003,Green,Red,DlgGray,Func,1,Hide,1
$ShowDig = Bit,-,0,0,10,10,0x0003,Green,Red,DlgGray,Func,2,Hide,2

$Child1 = Child,-,5,30,360,160,0x0000,ShowClk,"ChildTime.zbm"
$Child2 = Child,-,5,30,360,160,0x0000,ShowDig,"ChildDigIn.zbm"

$BTN_QUIT =
Button,Close,"*Quit",-,155,200,60,25,0x0002,0,Black,DlgGray,-
```

ChildTime.zbm

```
[VARIABLES]
OpFlags      = 0x0000
PollInterval = 2000

$Ctrl    = Controller,0x0001,1,(1,TCP,0,[192.168.3.2],23,2,50,0x9,0)
$Clock   = SDO,0x0001,Ctrl,0x2202,23,-,-

$Times   = Font,0x0000,(-30,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$BigFont = Font,0x0000,(-50,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$Label   = Text,"Current Time",-,20,20,320,30,0,Times,0x0,Black,-
$Display = Message,Value,-,20,60,320,60,0,BigFont,0x0,-,
{Clock,"%.1lf",-,1000.0,DarkRed}

$Start   =
Button,PollStart,"*Start",-,115,130,60,25,0x0002,0,Black,Green,-
$Stop    = Button,PollStop,
"*Stop",Start,70,0,60,25,0x0002,0,Black,Red,-
```

ChildDigIn.zbm

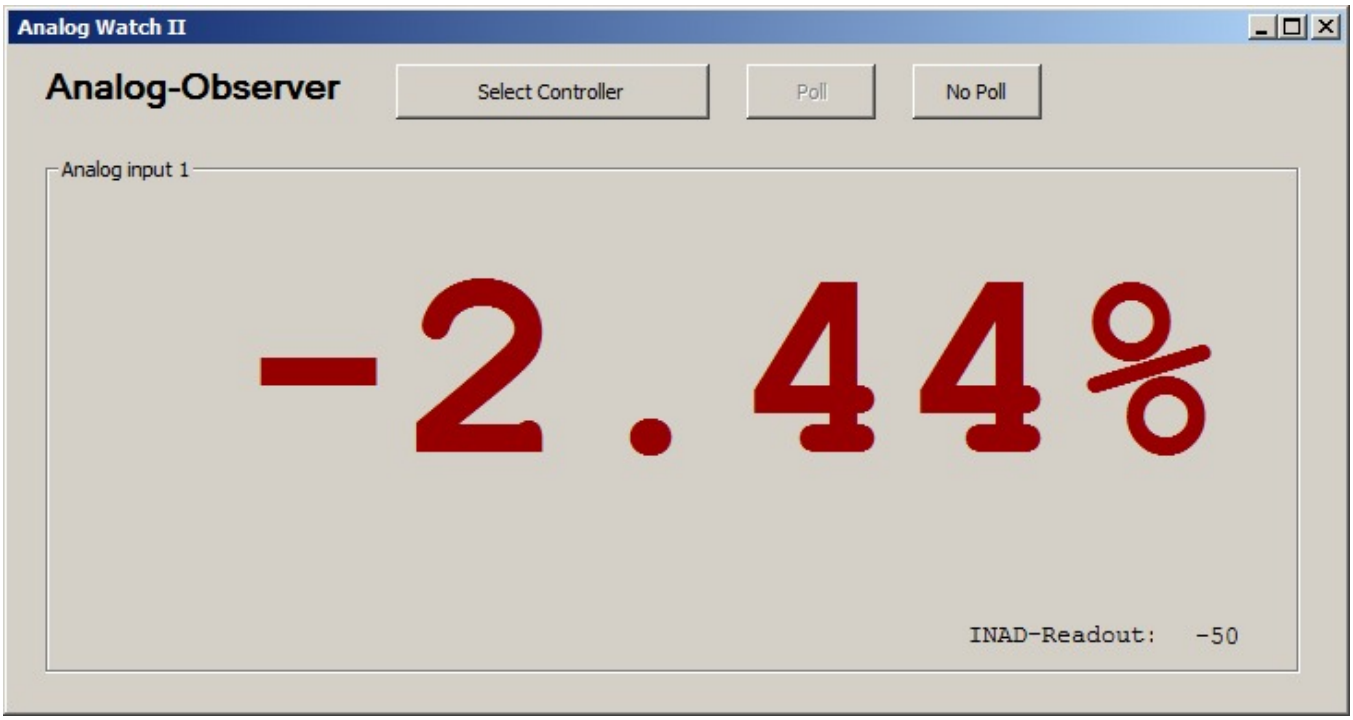
```
[VARIABLES]
OpFlags      = 0x0001
PollInterval = 500

$Ctrl    = Controller,0x0001,1,(1,TCP,0,[192.168.3.2],23,2,50,0x9,0)
$Inputs  = SDO,0x0001,Ctrl,0x2202,2,-,-

$Times   = Font,0x0000,(-30,0,0,0,900,0,0,0,0,0,0,0,0,"Times New
Roman")
$Label   = Text,"Digital Inputs",-,20,20,320,40,0,Times,0x0,Black,-
```

```
$Bit0    =
Bit,-,150,70,10,10,0x0000,Red,DlgGray,DlgGray,Inputs,0x0001,-,0
$Bit1    =
Bit,Bit0,0,20,10,10,0x0000,Red,DlgGray,DlgGray,Inputs,0x0002,-,0
$Bit2    =
Bit,Bit1,0,20,10,10,0x0000,Red,DlgGray,DlgGray,Inputs,0x0004,-,0
$Bit3    =
Bit,Bit2,0,20,10,10,0x0000,Red,DlgGray,DlgGray,Inputs,0x0008,-,0
$Lbl1    = Text,"Input 1",Bit0,20,-2,70,15,-1,14,0x0000,Black,-
$Lbl2    = Text,"Input 2",Bit1,20,-2,70,15,-1,14,0x0000,Black,-
$Lbl3    = Text,"Input 3",Bit2,20,-2,70,15,-1,14,0x0000,Black,-
$Lbl4    = Text,"Input 4",Bit3,20,-2,70,15,-1,14,0x0000,Black,-
```

This example shows how child dialogs may be used. Please note the following points about the example:

1. “ChildDialogs.zbm” is the main Monitor file. This is the file that should be opened by the ApossIDE. “ChildTime.zbm” and “ChildDigIn.zbm” are two child dialogs that are embedded within the main Monitor dialog.
2. The $Func, $Hide, $ShowClk, and $ShowDig elements (in “ChildDialogs.zbm”) illustrate a little trick that can be used to show/hide the two child dialogs, based on a selection from a list box. Note that the $ShowClk and $ShowDig bit flags are permanently hidden in the dialog since these values do not need to be visible to users.
3. Each of the three “.zbm” files is completely independent. Each must define its own controller, fonts, etc. Each will also define its own OpFlags and PollInterval. In this example, polling is started manually in one child dialog and automatically in the other child dialog. Each child dialog also uses a different polling interval.
4. Each of the two child ".zbm" files contains an identical Controller statement, so that they will connect to the same controller. These statements expect one and only one TCP controller to be available and will connect to the controller automatically when the Monitor dialog is opened.

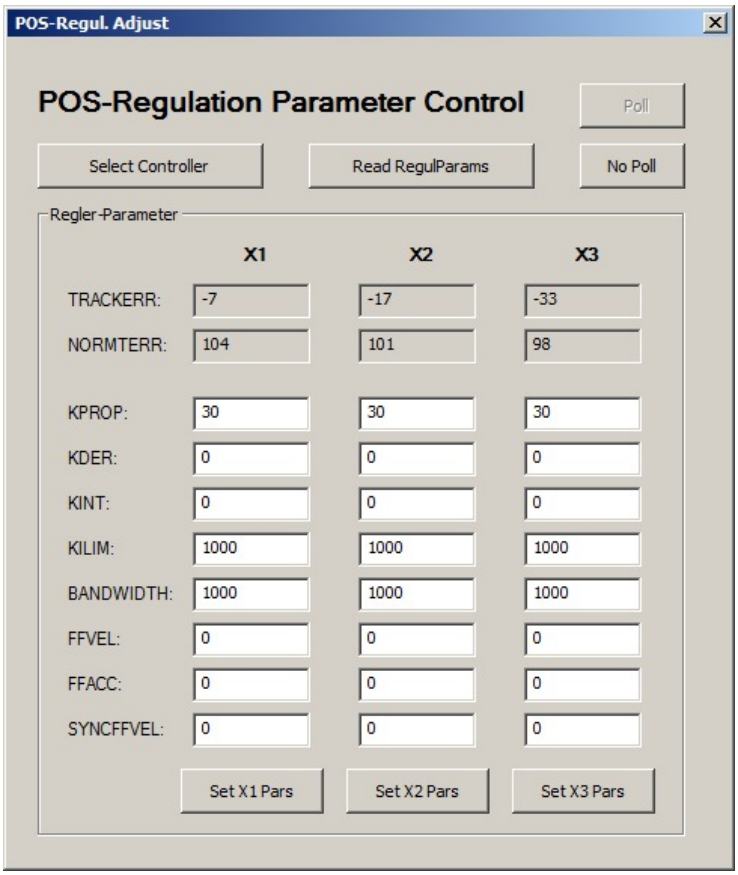
5.7.3.15 Displaying Analog Input




```
[VARIABLES]
WindowWidth  = 724
WindowHeight = 360
WindowTitle  = Analog Watch II
WindowFlags  = 0x002A
OpFlags      = 0x0001
PollInterval = 500
Fade         = 2,[255.128.0],5,[255.192.128],10,[255.224.192]

$Ctrl      = Controller,0x0001,1,(*)
$INAD1     = SDO,0x0001,Ctrl,0x2202,19,0,0,0

$BigFont   = Font,0x0000,(-150,0,0,0,900,0,0,0,0,0,0,0,0,"Courier
New")
$SmallFont = Font,0x0000,( -14,0,0,0,400,0,0,0,0,0,0,0,0,"Courier
New")

$Title     = Text,"Analog-Observer", 0,
20,10,170,40,-1,24,0x0001,Black,-
$Select    = Button,Controller,"*Select Controller",Title, 190,
0,170,30,0x0,0,Black,DlgGray,-,{Ctrl}
$PollOn    = Button,PollStart,"* Poll", Select,190, 0,
70,30,0x0,0,Black,DlgGray,-
$PollOff   = Button,PollStop, "*No Poll", PollOn, 90, 0,
70,30,0x0,0,Black,DlgGray,-

$Box1      = Box,"*Analog input 1",Title,0,50,680,280,0x0000,-
```

```
$PercVal    = Message,Value,Box1, 10, 30,630, 180,-1,BigFont,0x0,-,
{INAD1,"%6.2f%%",-,20.48,DarkRed}
$Info       = Text,"INAD-Readout:
",Box1,500,250,120,20,-1,SmallFont,0x0000,Black,-
$Intern     = Message,Value,Info,100,0,50,20,1,SmallFont,0x0,-,
{INAD1,0,-,1,Black}
```

5.7.3.16 Displaying and Setting Regulation Parameters




```
[VARIABLES]
WindowWidth  = 480
WindowHeight = 550
WindowTitle  = POS-Regul. Adjust
WindowFlags  = 0x0002
OpFlags      = 0x0001
PollInterval = 20
Fade         = 2,[255.128.0],5,[255.192.128],10,[255.224.192]

$Ctrl = Controller,0x0001,1,(*)
$X1Pe = SDO,0x0001,Ctrl,0x2500,  6,0,0   // TRACKERR
$X1Pn = SDO,0x0001,Ctrl,0x2500, 29,0,0   // NORMTRACKERR

$X1P1 = SDO,0x0004,Ctrl,0x2300, 12,0,0
$X1P2 = SDO,0x0004,Ctrl,0x2300, 13,0,0
$X1P3 = SDO,0x0004,Ctrl,0x2300, 14,0,0
$X1P4 = SDO,0x0004,Ctrl,0x2300, 22,0,0
$X1P5 = SDO,0x0004,Ctrl,0x2300, 36,0,0
$X1P6 = SDO,0x0004,Ctrl,0x2300, 37,0,0
```

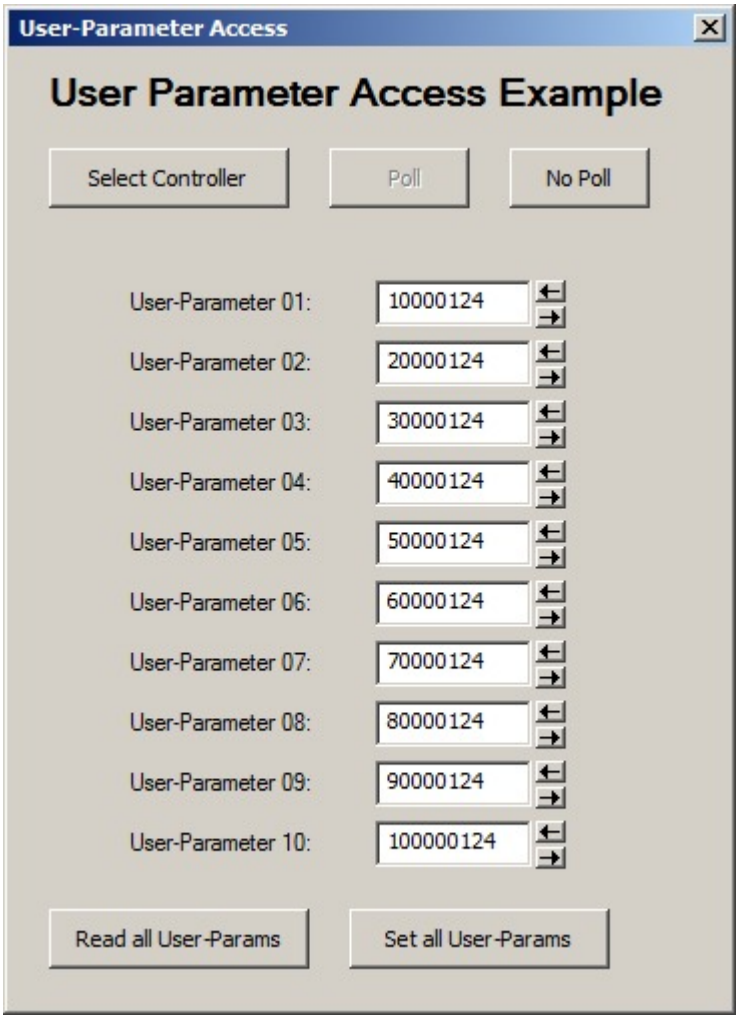
```
$X1P7 = SDO,0x0004,Ctrl,0x2300, 38,0,0
$X1P8 = SDO,0x0004,Ctrl,0x2300,110,0,0

$X2Pe = SDO,0x0001,Ctrl,0x2501,  6,0,0   // TRACKERR
$X2Pn = SDO,0x0001,Ctrl,0x2501, 29,0,0   // NORMTRACKERR

$X2P1 = SDO,0x0004,Ctrl,0x2301, 12,0,0
$X2P2 = SDO,0x0004,Ctrl,0x2301, 13,0,0
$X2P3 = SDO,0x0004,Ctrl,0x2301, 14,0,0
$X2P4 = SDO,0x0004,Ctrl,0x2301, 22,0,0
$X2P5 = SDO,0x0004,Ctrl,0x2301, 36,0,0
$X2P6 = SDO,0x0004,Ctrl,0x2301, 37,0,0
$X2P7 = SDO,0x0004,Ctrl,0x2301, 38,0,0
$X2P8 = SDO,0x0004,Ctrl,0x2301,110,0,0

$X3Pe = SDO,0x0001,Ctrl,0x2502,  6,0,0   // TRACKERR
$X3Pn = SDO,0x0001,Ctrl,0x2502, 29,0,0   // NORMTRACKERR
$X3P1 = SDO,0x0004,Ctrl,0x2302, 12,0,0
$X3P2 = SDO,0x0004,Ctrl,0x2302, 13,0,0
$X3P3 = SDO,0x0004,Ctrl,0x2302, 14,0,0
$X3P4 = SDO,0x0004,Ctrl,0x2302, 22,0,0
$X3P5 = SDO,0x0004,Ctrl,0x2302, 36,0,0
$X3P6 = SDO,0x0004,Ctrl,0x2302, 37,0,0
$X3P7 = SDO,0x0004,Ctrl,0x2302, 38,0,0
$X3P8 = SDO,0x0004,Ctrl,0x2302,110,0,0

$Title   = Text,"POS-Regulation Parameter Control",0, 20, 30,600,40,-1,24,0x0001,-
$Select  = Button,Controller,"*Select Controller" ,0, 20, 70,150, 30,0x0000,0,Black,DlgGray,-,{Ctrl}
$PollOn  = Button,PollStart,"*  Poll"             ,0,380, 30, 70, 30,0x0000,0,Black,DlgGray,-
$PollOff = Button,PollStop, "*No Poll"            ,0,380, 70, 70, 30,0x0000,0,Black,DlgGray,-

$Box  = Box,"*Regler-Parameter", 0,20,110,430,420,0x0000,-
$Lble = Text,"TRACKERR:  ",Box,20, 57,170,14,-1,14,0x0000,Black,-
$Lbln = Text,"NORMTERR:  ",Box,20, 87,170,14,-1,14,0x0000,Black,-
$Lbl1 = Text,"KPROP:     ",Box,20,132,170,14,-1,14,0x0000,Black,-
$Lbl2 = Text,"KDER:      ",Box,20,162,170,14,-1,14,0x0000,Black,-
$Lbl3 = Text,"KINT:      ",Box,20,192,170,14,-1,14,0x0000,Black,-
$Lbl4 = Text,"KILIM:     ",Box,20,222,170,14,-1,14,0x0000,Black,-
$Lbl5 = Text,"BANDWIDTH: ",Box,20,252,170,14,-1,14,0x0000,Black,-
$Lbl6 = Text,"FFVEL:     ",Box,20,282,170,14,-1,14,0x0000,Black,-
$Lbl7 = Text,"FFACC:     ",Box,20,312,170,14,-1,14,0x0000,Black,-
$Lbl8 = Text,"SYNCFFVEL: ",Box,20,342,170,14,-1,14,0x0000,Black,-

$LblX1 = Text,"X1",Box,105,27,80,14,0,14,0x0001,Black,-
$LblX2 = Text,"X2",Box,215,27,80,14,0,14,0x0001,Black,-
$LblX3 = Text,"X3",Box,325,27,80,14,0,14,0x0001,Black,-

$X1Ve = Value,"*", Lble, 80,-7,0x0201,X1Pe,0,0,0,-
$X1Vn = Value,"*", Lbln, 80,-7,0x0201,X1Pn,0,0,0,-
$X1V1 = Value,"*", Lbl1, 80,-7,0x0200,X1P1,0,0,0,-
$X1V2 = Value,"*", Lbl2, 80,-7,0x0200,X1P2,0,0,0,-
$X1V3 = Value,"*", Lbl3, 80,-7,0x0200,X1P3,0,0,0,-
$X1V4 = Value,"*", Lbl4, 80,-7,0x0200,X1P4,0,0,0,-
$X1V5 = Value,"*", Lbl5, 80,-7,0x0200,X1P5,0,0,0,-
$X1V6 = Value,"*", Lbl6, 80,-7,0x0200,X1P6,0,0,0,-
$X1V7 = Value,"*", Lbl7, 80,-7,0x0200,X1P7,0,0,0,-
$X1V8 = Value,"*", Lbl8, 80,-7,0x0200,X1P8,0,0,0,-

$X2Ve = Value,"*", Lble,190,-7,0x0201,X2Pe,0,0,0,-
$X2Vn = Value,"*", Lbln,190,-7,0x0201,X2Pn,0,0,0,-
$X2V1 = Value,"*", Lbl1,190,-7,0x0200,X2P1,0,0,0,-
$X2V2 = Value,"*", Lbl2,190,-7,0x0200,X2P2,0,0,0,-
$X2V3 = Value,"*", Lbl3,190,-7,0x0200,X2P3,0,0,0,-
```

```
$X2V4 = Value,"*", Lbl4,190,-7,0x0200,X2P4,0,0,0,-
$X2V5 = Value,"*", Lbl5,190,-7,0x0200,X2P5,0,0,0,-
$X2V6 = Value,"*", Lbl6,190,-7,0x0200,X2P6,0,0,0,-
$X2V7 = Value,"*", Lbl7,190,-7,0x0200,X2P7,0,0,0,-
$X2V8 = Value,"*", Lbl8,190,-7,0x0200,X2P8,0,0,0,-

$X3Ve = Value,"*", Lble,300,-7,0x0201,X3Pe,0,0,0,-
$X3Vn = Value,"*", Lbln,300,-7,0x0201,X3Pn,0,0,0,-
$X3V1 = Value,"*", Lbl1,300,-7,0x0200,X2P1,0,0,0,-
$X3V2 = Value,"*", Lbl2,300,-7,0x0200,X2P2,0,0,0,-
$X3V3 = Value,"*", Lbl3,300,-7,0x0200,X2P3,0,0,0,-
$X3V4 = Value,"*", Lbl4,300,-7,0x0200,X2P4,0,0,0,-
$X3V5 = Value,"*", Lbl5,300,-7,0x0200,X2P5,0,0,0,-
$X3V6 = Value,"*", Lbl6,300,-7,0x0200,X2P6,0,0,0,-
$X3V7 = Value,"*", Lbl7,300,-7,0x0200,X2P7,0,0,0,-
$X3V8 = Value,"*", Lbl8,300,-7,0x0200,X2P8,0,0,0,-

$Read = Button,Group,"*Read RegulParams",0,200,70,150,30,0x0200,0,Black,DlgGray,-,
{X1V1,X1V2,X1V3,X1V4,X1V5,X1V6,X1V7,X1V8,X2V1,X2V2,X2V3,X2V4,X2V5,X2V6,X2V7,X2V8,X3V1,X3V2,X3V3,X3V4,X3V5,X3V6,X3V7,X3V8}

$SetX1 = Button,Group,"*Set X1 Pars",X1V8,-5,40,95,30,0x0200,0,Black,DlgGray,-,{X1V1,X1V2,X1V3,X1V4,X1V5,X1V6,X1V7,X1V8}
$SetX2 = Button,Group,"*Set X2 Pars",X2V8,-5,40,95,30,0x0200,0,Black,DlgGray,-,{X2V1,X2V2,X2V3,X2V4,X2V5,X2V6,X2V7,X2V8}
$SetX3 = Button,Group,"*Set X3 Pars",X3V8,-5,40,95,30,0x0200,0,Black,DlgGray,-,{X3V1,X3V2,X3V3,X3V4,X3V5,X3V6,X3V7,X3V8}
```

```
set poserr x(1) 100000
set poserr x(2) 100000
set poserr x(3) 100000
deforigin x(1) x(2)

set encodertype x(1) 9
set encodertype x(2) 9
set encodertype x(3) 9

motor on x(1)
motor on x(2)
motor on x(3)

nowait on

target = 80000
acc 10 / 30 / 100
dec 10 / 30 / 100
vel 20 / 50 / 100
posa target / target / target

start:
    axeno = 1
    print " "
    repeat
        print axeno,": ";
        print get kprop x(axeno);
        print " / ",get kder x(axeno);
        print " / ",get kint x(axeno);
        print " / ",get kilim x(axeno);
        print " / ",get bandwidth x(axeno);
        print " / ",get ffvel x(axeno);
        print " / ",get ffacc x(axeno);
        print " / ",get syncffvel x(axeno);
        address = 0x01250006 + ((axeno-1) << 8)
        print " // ",trackerr," / ",sysvar[address]
        axeno++
    until (axeno > 2)
    delay 5000
```

```
if (stat x(1) & 0x040000) then
    if (target == 80000) then
        target = 0
    else
        target = 80000
    endif
    posa target / target / target
    print "posa ",target
endif
goto start
```

5.7.3.17 Displaying and Setting User Parameters




```
[VARIABLES]
WindowWidth  = 360
WindowHeight = 480
WindowTitle  = User-Parameter Access
WindowFlags  = 0x0022
OpFlags      = 0x0001
PollInterval = 1000
Fade         = 2,[255.128.0],5,[255.192.128],10,[255.224.192]
```

```
$Ctrl = Controller,0x0001,1,(*)

$USRP01 = SDO,0x0001,Ctrl,0x2201, 1,0,0
$USRP02 = SDO,0x0001,Ctrl,0x2201, 2,0,0
$USRP03 = SDO,0x0001,Ctrl,0x2201, 3,0,0
$USRP04 = SDO,0x0001,Ctrl,0x2201, 4,0,0
$USRP05 = SDO,0x0001,Ctrl,0x2201, 5,0,0
$USRP06 = SDO,0x0001,Ctrl,0x2201, 6,0,0
$USRP07 = SDO,0x0001,Ctrl,0x2201, 7,0,0
$USRP08 = SDO,0x0001,Ctrl,0x2201, 8,0,0
$USRP09 = SDO,0x0001,Ctrl,0x2201, 9,0,0
$USRP10 = SDO,0x0001,Ctrl,0x2201,10,0,0

$Title  = Text,"User Parameter Access Example",
0,20,10,320,40,-1,24,0x0001,Black,-
$Select = Button,Controller,"*Select
Controller",0,20,50,120,30,0x0000,0,Black,DlgGray,-,{Ctrl}

$UP01TXT = Text,"User-Parameter 01: ",       0, 60,120,
100,14,-1,14,0x0000,Black,-
$UP02TXT = Text,"User-Parameter 02: ", UP01TXT,  0, 30,
100,14,-1,14,0x0000,Black,-
$UP03TXT = Text,"User-Parameter 03: ", UP02TXT,  0, 30,
100,14,-1,14,0x0000,Black,-
$UP04TXT = Text,"User-Parameter 04: ", UP03TXT,  0, 30,
100,14,-1,14,0x0000,Black,-
$UP05TXT = Text,"User-Parameter 05: ", UP04TXT,  0, 30,
100,14,-1,14,0x0000,Black,-
$UP06TXT = Text,"User-Parameter 06: ", UP05TXT,  0, 30,
100,14,-1,14,0x0000,Black,-
$UP07TXT = Text,"User-Parameter 07: ", UP06TXT,  0, 30,
100,14,-1,14,0x0000,Black,-
$UP08TXT = Text,"User-Parameter 08: ", UP07TXT,  0, 30,
100,14,-1,14,0x0000,Black,-
$UP09TXT = Text,"User-Parameter 09: ", UP08TXT,  0, 30,
100,14,-1,14,0x0000,Black,-
$UP10TXT = Text,"User-Parameter 10: ", UP09TXT,  0, 30,
100,14,-1,14,0x0000,Black,-

$UP01VAL = Value,"*", UP01TXT,120,-7,0x0102,USRP01,0,-,0,-
$UP02VAL = Value,"*", UP02TXT,120,-7,0x0102,USRP02,0,-,0,-
$UP03VAL = Value,"*", UP03TXT,120,-7,0x0102,USRP03,0,-,0,-
$UP04VAL = Value,"*", UP04TXT,120,-7,0x0102,USRP04,0,-,0,-
$UP05VAL = Value,"*", UP05TXT,120,-7,0x0102,USRP05,0,-,0,-
$UP06VAL = Value,"*", UP06TXT,120,-7,0x0102,USRP06,0,-,0,-
$UP07VAL = Value,"*", UP07TXT,120,-7,0x0102,USRP07,0,-,0,-
$UP08VAL = Value,"*", UP08TXT,120,-7,0x0102,USRP08,0,-,0,-
```

```
$UP09VAL = Value,"*", UP09TXT,120,-7,0x0102,USRP09,0,-,0,-
$UP10VAL = Value,"*", UP10TXT,120,-7,0x0102,USRP10,0,-,0,-

$PollOn  = Button,PollStart,"*  Poll", Select,140, 0, 70,
30,0x0000,0,Black,DlgGray,-
$PollOff = Button,PollStop, "*No Poll", PollOn, 90, 0, 70,
30,0x0000,0,Black,DlgGray,-
$ReadAll = Button,Group,"*Read all User-Params",UP09TXT,-40, 70,130,
30,0x0100,0,Black,DlgGray,-,{UP01VAL, UP02VAL, UP03VAL, UP04VAL,
UP05VAL, UP06VAL, UP07VAL, UP08VAL, UP09VAL, UP10VAL}
$SetAll  = Button,Group,"*Set all User-Params", ReadAll,150,  0,130,
30,0x0200,0,Black,DlgGray,-,{UP01VAL, UP02VAL, UP03VAL, UP04VAL,
UP05VAL, UP06VAL, UP07VAL, UP08VAL, UP09VAL, UP10VAL}
```

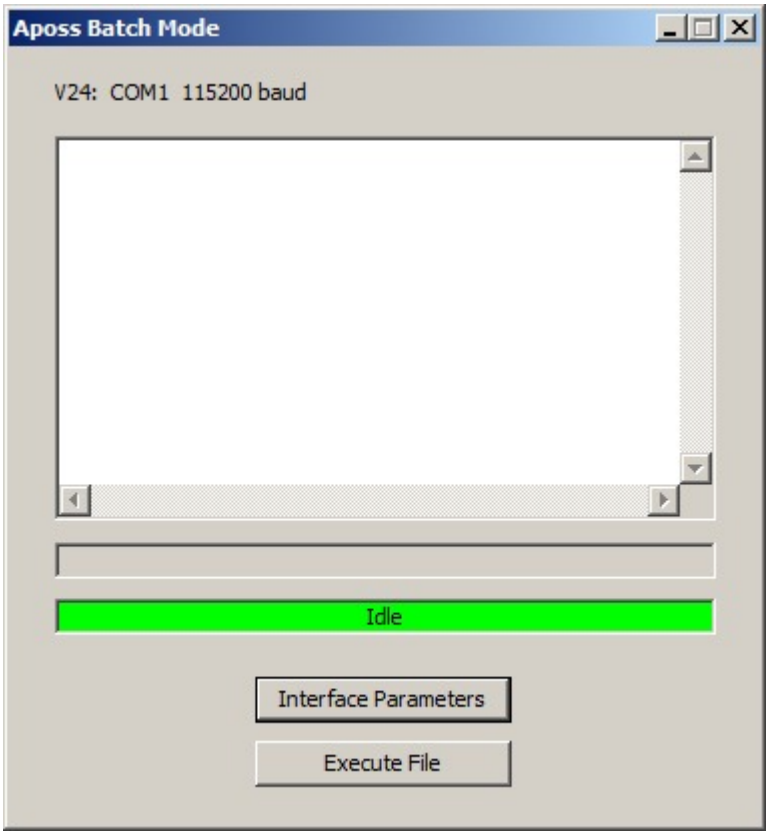
```
UsrParBase = 0x01220100

/*
UsrPar01 = UsrParBase + 1
UsrPar02 = UsrParBase + 2
UsrPar03 = UsrParBase + 3
UsrPar04 = UsrParBase + 4
UsrPar05 = UsrParBase + 5
UsrPar06 = UsrParBase + 6
UsrPar07 = UsrParBase + 7
UsrPar08 = UsrParBase + 8
UsrPar09 = UsrParBase + 9
UsrPar10 = UsrParBase + 10
*/

subidx = 1
repeat
   sysvar[UsrParBase+subidx] = subidx * 10000000
   subidx++
until (subidx > 10)

start:
    subidx = 1
    repeat
        sysvar[UsrParBase+subidx] = sysvar[UsrParBase+subidx] + 1
        print sysvar[UsrParBase+subidx];
        if (subidx != 10) then
            print " / ";
        else
            print " "
        endif
        subidx++
    until (subidx > 10)
```

```
delay 1000

goto start
```

5.8 Batch Mode

Batch Mode provides a simple client/server mechanism that allows communication with controllers without requiring user input. When The ApossIDE is started in Batch Mode, it will wait for a “command file” to be created (by some external means such as another program). When a file appears, then the file is opened and the commands within it are executed. Once execution is complete, an “answer file” is created and the original command file is deleted. The ApossIDE will then wait for a new command file to be created.

When started in Batch Mode, The ApossIDE displays a dialog (shown below) containing a message box, an error box, and a status box. If the status box is green and displays the “Idle” message, then the ApossIDE is waiting for a command file to execute. If the status box is yellow and displays the “Waiting…” message, then there is a command file but it is not possible to open it exclusively. This happens when another program has the file open (e.g. while the file is being written). When the command file can be opened, the status box will turn red and display the “Running…” message. Execution of the file then starts. Commands are displayed in the message box as they are executed and logged to the answer file when execution of the command file completes. The error box will show the completion status of the last command file. It will be empty when no previous command file has been executed and when a command file is in the process of being executed. It will be green and display the “Completed successfully” message if the last command file completed without any errors. If the last command file resulted in errors, then the box will be red and display the message “Errors found during execution”. In this case, if the speakers are turned on, a warning beep will also be sounded.



The dialog also shows the connection interface that will be used to connect to controllers. In the dialog shown above, the V24 interface will be used. The user can change this interface by pressing the “Interface Parameters”

button.

In addition, the Batch Mode dialog allows the user to manually execute command files by pressing the “Execute File” button. These manually executed batch files are not automatically deleted after execution. When the user has activated the manual command file interface (or is setting the interface parameters), then the status box will be yellow and will display the “Waiting...” message. When this happens, The ApossIDE has deactivated the automatic detection and execution of command files so that they cannot interfere with what the user is doing.

Batch Mode can be terminated simply by closing the dialog window. It can also be terminated automatically if the command file contains a special exit command. Note that if the user is manually executing a command file using the “Execute File” button, then exit commands within the command file are ignored. Manually executing a command file will not result in Batch Mode being terminated. See below for an exception.

Batch Mode can also be run in a “simplified” mode. In this mode, Batch Mode does not automatically check for command files. Instead, all command files must be executed manually by pressing the “Execute File” button. In addition, “simplified” mode has the following differences:

- The user is not able to change the interface parameters. It is expected that the interface parameters be entirely specified within the command file and not by the user.
- The selection of controllers is also expected to be specified within the command file. The user is not able to manually specify controllers.
- After a command file has been executed, a clear “success” or “fail” message is displayed. The “fail” message will be displayed if any problem is encountered during the processing of a command file.
- Exit commands within command files will cause Batch Mode to terminate. However, the user must acknowledge the “success” or “fail” message before Batch Mode will terminate.

5.8.1 Configuring and Starting Batch Mode

The default command and answer files will be “SPIC.TXT” and “SPIA.TXT” in the “Aposs” subdirectory under the Windows “Application Data” directory. If necessary, the user can specify different files by starting the ApossIDE in the normal mode and typing Alt-Z. This should be done before Batch Mode is started. If a directory other than the default directory is to be used, then the command and answer files should be specified using absolute path names.

A “Batch Mode debug flag” can also be enabled with Alt-Z in the ApossIDE. If the debug flag is enabled, then additional debug information will be created in both the Batch Mode dialog message box and the answer file.

When a command file is ready to be executed, Batch Mode will try to establish a connection to the controllers using the current default interface specified by the the ApossIDE “Settings/Interface” menu command. This default interface should be set up using the normal ApossIDE before starting the ApossIDE in Batch Mode. The interface can also be changed within Batch Mode itself by pressing the “Interface Parameters” button in the Batch Mode dialog.

Batch Mode is started by executing the ApossIDE with any of the command line arguments “/b”, “/b-”, or “/bs”. If “/b-” is used, then the application will start minimized and the dialog will not be displayed. If “/bs” is used, then the application will start in “simplified” mode. Note that only one instance of the ApossIDE may be running at a time so if the ApossIDE is already running, then a message is displayed and the Batch Mode instance will terminate.

A command file (in quotes) can also be explicitly specified as a command line argument following the “/b”, “/b-”, or “/bs” argument. When this is done, then that command file is executed immediately. The command file is executed only once and will not be deleted after execution completes. This is intended to allow Windows shortcuts to be set up to execute specific maintenance tasks with a minimum of configuration and user intervention.

If a command file is specified in the command line, then a “controller ID range” may be optionally specified after the command file. This is done using the command line argument “/c”, followed by the controller ID range. If

specified, this range will override any range defined within the command file. See Command File Structure for a description of the controller ID range syntax. Example:

```
"C:\Program Files\Aposs\Aposs32.exe" /b "C:\Work\Batch\Download.txt" /c #1-5
```

5.8.2 Using ".zip" Files

Batch Mode also supports the use of ".zip" files as command files. This allows all files that are associated with a particular batch process (e.g. multiple ".m" files) to be packaged into a single file that is delivered to Batch Mode. Note that the ".zip" file is treated in the same way that the normal command file is treated (i.e. if a normal command file would be deleted after execution, then the ".zip" file will also be deleted after execution).

Batch Mode will recognize a ".zip" file as a command file if it has the same filename as the normal command file and a file extension of ".zip". The ".zip" file will be unzipped into a temporary directory. In all cases, this temporary directory and all its files will be deleted when execution completes.

The ".zip" file must contain a single embedded file with a file extension of ".txt"; the name of this file is ignored. This file will be treated as the command file. The normal answer file will be created.

> Currently, the only ".zip" files supported are those created by the FilZip application. This application can be found by searching online.

5.8.3 Controller ID Numbers

Controllers are identified for Batch Mode by means of integer controller ID numbers. Each controller accessible on a connection interface will be assigned its own unique ID number. The ID numbers used will be dependent on the type of interface.

For interfaces such as V24 that support only a single connected controller, the controller will always be assigned ID number 1.

For all interfaces based on the CAN protocol, the controller ID numbers will be identical to the CAN ID numbers of the controllers. The CAN ID numbers are configured into the controllers as part of their initial setup. Similarly, for VLT interfaces, the controller ID numbers will be identical to the VLT ID numbers of the controllers. For EtherCAT, the controller ID numbers will be the EtherCAT port numbers. When multiple controllers are connected simultaneously, this allows specific controllers to be unambiguously identified.

For USB and Network (TCP) interfaces, controller ID numbers are assigned sequentially starting at 1. Controllers are assigned ID numbers based entirely on the order in which Batch Mode locates the controllers. As a result, connecting USB controllers to different USB ports, or assigning different IP addresses to different network controllers, may change the ID numbers assigned to the controllers. Hence, when multiple controllers are connected simultaneously, it is *not* possible to unambiguously identify a specific controller by using the controller ID number.

Batch Mode will generally treat all controllers as being identical; all commands within the command file are executed for each controller. However, to handle the case where all controllers are *not* identical, Batch Mode does provide a mechanism to selectively execute commands on different controllers (see the "&" command).

5.8.4 Command File Structure

The command file is an ASCII file which can be created and edited by any text editor. Commands are specified one per line (i.e. separated by either a line feed or a carriage return/line feed). Leading and trailing blanks are ignored and blank lines are ignored. As well, any line beginning with the asterisk or slash character (i.e. "*" or "/") is treated as a comment and is ignored.

Controller ID Range

The command file may specify an optional controller ID range. If the range is specified, then this must be the first command in the file. The controller ID range is specified using the following command:

```
# range
```

or

```
@ range
```

where "*range*" can be a single integer number (e.g. 2), a list of integers separated by commas (e.g. 2,4,6), a range specified with a "-" (e.g. 4-8), or a combination (e.g. 2,4,7-10,15,20-30). It may also be specified as "*" to indicate "all controllers accessible on the current interface". Controller ID numbers must not be less than 1 or greater than 255. These numbers will define the range of controller ID numbers that will be used. The range will default to "1" if it is not specified. Note that for some types of connections that support only single controllers (e.g. serial connections such as V24), the controller ID range will be ignored if it is specified.

If the controller ID numbers are specified using "#", then a "full" connection is made to the controllers. In this case, all interface types and all commands are supported. If the controller ID numbers are specified using "@", then only a "partial" connection is made. Partial connections are supported only by CAN interface types and offer only SDO read/write commands, the "V" command, as well as the non-controller related commands such as print, exit, etc. Partial connections are useful when non-controller CAN devices must be configured. Note that the special "*" range cannot be used with partial connections.

When executing command files manually, any "#" or "@" command in the command file will be ignored. The controller ID numbers are specified using the Batch Mode interface. An exception to this is made when Batch Mode is running in "simplified" mode. In "simplified" mode, the user is not able to manually specify a controller range. Instead, the controller range will be determined by any "#" or "@" command in the command file.

Each command in the command file will be executed for each controller in the controller ID range before processing goes on to the next command. It is not necessary to have all controllers in the specified ID range physically present; any ID corresponding to a missing controller is skipped.

Commands

The syntax of a command line is as follows:

```
c parameters
```

where "*c*" is a command letter from the following list and "*parameters*" is a set of parameters that depends on the command being specified. For backwards compatibility, a "$" character can precede the command letter.

Command List

Valid commands are as follows. Note that all command letters must be uppercase.

Command	Description
`A nn`	Set autoflag for program "*nn*" (0-254).
`A 255`	Reset autoflag.
`B tttt`	Break (stop a running program or resets an error). Batch Mode will wait up to "*tttt*" milliseconds for a running program to stop. This will default to 60000 milliseconds if it is not specified.
`C`	Factory reset (ram is cleared totally and a reset is executed).

Command	Description
`D ssss`	Download file "*ssss*". The file extension must be included and the filename must not be enclosed in quote characters. If the file extension is ".m" then the file is compiled and downloaded as a temporary program. If the file extension is ".bin", then it is assumed to be a binary file and it will be downloaded without first being compiled. If the file extension is ".zbc" or ".cnf", then the file is assumed to be a configuration file and a "Controller/Restore from file" command will be executed. Either absolute or relative pathnames may be used. If a relative pathname is used, then it will be treated as being relative to the directory in which the command file was found.
`E`	Erase all programs in memory.
`F www sss`	Download firmware file "*sss*" (see restrictions below). Do not enclose the filename in quote characters. Either absolute or relative pathnames may be used. If a relative pathname is used, then it will be treated as being relative to the directory in which the command file was found. If "*www*" is omitted or set to "wait" (without quotes), then the command will wait for the controller to be restarted after the firmware has been downloaded. If "*www*" is set to "nowait" (without quotes), then the controller will be restarted after the firmware is downloaded, but the command will not wait for the restart to complete. This may be useful if firmware is being downloaded to multiple controllers. Note that if the command is not waiting for controllers to restart, then the "J" command should be executed immediately after the "F" command. This effectively results in a single wait being done for all controllers rather than waits being done for each controller individually.
`G ndx sub`	Gets (reads) SDO index "*ndx*", subindex "*sub*" and prints its value.
`I interface`	Set the controller connection interface to "*interface*" (see below). Wait for the controller to restart and then reconnect to it. If the controller is already connected, then this command does nothing.
`J`	Wait for the controller to restart and then reconnect to it. If the controller is already connected, then this command does nothing.
`L`	Erase flash eprom.
`M`	Save ram to flash eprom.
`P string`	Prints "*string*" to the log file.
`Q ssss`	Set project name to "*ssss*" (name of controller).
`R ssss nn`	Save temporary program (using number "*nn*") and set name

Command	Description
	to "*ssss*".
`S ssss`	Save temporary program (using the next free number) and set name to "*ssss*".
`T tttt`	Sets the SDO Read/Write timeout to "*tttt*" milliseconds (-1 for default timeout).
`V nn ss mm ff tt`	Wait until the value of SDO index "*nn*", subindex "*ss*" matches the mask specified by "*mm*". If "*ff*" is 1, then the wait is over when the value "anded" with the mask is non-0. If "*ff*" is 0, then the wait is over when the value "anded" with the mask is 0. If the timeout "*tt*" in milliseconds expires, then the command will end with an error.
`W ndx sub val`	Write value "*val*" to SDO index "*ndx*", subindex "*sub*".
`X nn`	Execute program number "*nn*" (0-254).
`X 255`	Execute temporary program.
`Y options`	Set compiler options to "*options*" (see below).
`Z`	Exit Batch Mode and close window. This command is ignored if a command file is executed manually using the "Execute File" button. In all other cases, it is executed.
`= name string`	Substitute "*name*" with "*string*" in the parameter string.
`?`	Query and print the current controller error status (the Batch Mode debug flag must be enabled).
`& condition`	All commands following this command (up to the next "&" command) will only be executed on controllers that match the specified condition (see below).

Interface Specification

The interface to be used to connect to controllers can be selected using the I command. If used, then this command must be placed before any command that opens a connection to a controller. Otherwise, the previous interface type will be used. Valid interface types are:

V24	RS232 serial connection.
LPT	CAN connection using the LPT port.
COM	CAN connection using a CAN-USB stick.
USB	USB connection.
CAT	EtherCAT connection.
VLT	RS485 connection to Danfoss VLT5000 or FC300 drive.
MCO	RS485 connection to MCO305 onboard RS485 port.
TCP	Ethernet connection.

ZBC	zub EtherCAT connection.

Full interface specifications are also supported. This allows individual interface parameters to be specified. For example:

```
USB,2,50,0x0,921600,1,0
TCP,0,[192.168.4.2],23,2,50,0x9,0
```

If full interface specifications are needed, then please contact zub machine control AG directly for more information.

Note that changing the interface will also change the interface for future command files. If an invalid interface is specified, then this will cause the command file execution to be aborted.

Compiler Options

If compiler options must be set (Y command), then this should be done before any download (D command). The compiler options consist of the following parameters (separated by commas and containing no spaces):

1. Maximum executable size.
2. Maximum number of variables.
3. Maximum number of labels. This should be set to 60000.
4. Stack size.
5. Enable preprocessor (0 – No, 1 – Yes).
6. Generate cross reference (0 – No, 1 – Yes).

Please use **Settings** → **Compiler** to find appropriate values for these parameters. Note that the Y command will cause the include file search path to be cleared.

Example:

```
Y 65000,500,60000,200,1,0
```

String Substitution

The "=" command allows strings to be substituted within the parameter string. This is particularly useful when reading and writing SDO values since it allows SDO indices and subindices to be replaced with more user-friendly strings.

Controller Selection

The "&" command allows subsets of the available controllers to be selected. For example, only MACS5 controllers or only controllers with firmware older than a specified firmware version. The syntax of the command is:

```
& condition
```

where "condition" can be as follows:

`Controller op type`	"*op*" may be "=" to select only controllers of type "*type*" or "!=" to select all controllers except those of type "*type*". "*type*" may be any one of the following: SSI-EMS, MiniMACS, MiniMACS-AMP, MasterMACS, DDE-V85, Maxsine, ZCNT, MACS5, MACS5v1, MACS5v2, MACS4, MACS3, MACS2, MACS27010, Mocon, VLT5000, FC302, CANopen, Simulator, Encoder, Drive, IOModule.
`Firmware op version`	"*op*" may be "=", "!=", "<", "<=", ">", or ">=". "*version*" should be a firmware version specified in a format similar

	to “7.1.20”.
`0xnnnn/ss op list`	“0x*nnnn/ss*” may be any SDO (e.g. 0x2209/3). “*op*” may be “=” or “!=” (meaning “is in list” or “is not in list”). “*list*” should be a list of one or more numbers or number pairs (giving a low and high bound of a range) separated by commas. All numbers will be treated as signed 32-bit numbers. For example: 1-0,15,0x1000-0x2000
`*`	Clear all previous controller selection conditions.

If more than one condition is specified, then all conditions will be “anded”. However, each type of condition may only be specified once. For example, if a specific controller type has been selected, then a second “Controller” condition will simply replace the previous “Controller” condition.

The selection conditions will stay in effect for all following commands, up until they are changed by subsequent “&” commands or cleared using the “& *” command.

If controller selection commands are being used, then the safest way to avoid problems caused by selection conditions staying in effect longer than is appropriate, is to explicitly clear all conditions after the block of commands to which they apply. See below for an example.

The following commands are not related to any specific controller and are therefore always executed, regardless of the current controller selection: P, I, Y, Z, T, &.

5.8.5 Answer File Structure

When a command file is executed, the result of each command is written to the answer file in the following format:

```
#C nn id ssss
```

where “*nn*” gives the error code for the command that was executed (see Error Codes), “*id*” gives the ID of the controller on which the command was executed, and “*ssss*” is a string specifying the command that was executed.

If the BATCH-DEBUG flag was set in the APOS.DAT file, then debug messages of the following form will be written to the answer file:

```
#D id ssss
```

where “*id*” gives the ID of the controller on which the command is being executed and “*ssss*” is a string specifying the debug information.

If errors occur, then error messages of the following form will be written to the answer file:

```
#E id ssss
```

where “*nn*” gives the error code (see Error Codes) and “*ssss*” is a string specifying error information. If appropriate, the error will be followed by a “#C” message giving the ID of the controller which had the error. Note that some errors are fatal and will cause execution of the command file to be terminate. Other errors, particularly errors related to a specific controller, are non-fatal. If a non-fatal error is encountered, then the offending controller is removed from the list active of controllers and execution will continue with the other controllers.

The last line in the answer file will specify the result of the complete execution as follows:

```
#R nn from to exit
```

where “*nn*” gives the error code for the complete execution (see Error Codes), “*from*” and “*to*” give the controller

ID range, and "*exit*" gives the program exit code. The error code will be 0 if the entire command file executed without errors. Otherwise, the error code will be the last error encountered. The exit code will be 0 if Batch Mode is still running and 1 if Batch Mode has been terminated.

5.8.6 Error Codes

The possible error codes written into the answer file are as follows:

- 0 No error.
- 1 Not used.
- -1 Not used.
- -2 Invalid controller ID range specified (i.e. # or @ command).
- -3 Not used.
- -4 Not used.
- -5 $V command timed out.
- -6 Not used.
- -7 Errors generated during compile (i.e. $D command).
- -8 Invalid command specified.
- -9 Command not available when "@" is being used.
- -10 Could not open connection to controllers.
- -11 Command execution failed.
- -12 Could not (re)connect to controller.
- -13 Warning message.
- -14 One or more controllers specified in the controller ID range could not be found.
- -15 Invalid connection interface specified.
- -16 Binary file not found or contains invalid binary code (i.e. $D command).
- -17 Configuration file not found or is invalid (i.e. $D command).
- -18 Invalid controller selection condition.

5.8.7 Batch Mode Examples

Executing a Test Program

The following example will connect to controller with ID 1 and compile, load, and execute a program. The program is not saved so, after execution, it is not necessary to erase it. When the power is turned off, the controller will be in same condition as before.

```
D test.mc
X 255
```

The answer file should be as follows:

```
#C 0 1 D test.mc
#C 0 1 X 255
#R 0 1 1 0
```

Loading an Application

The following example demonstrates how to download and save an application program that we want to deliver and that should be started on power up. To be sure that we have only one program in the controller, all existing programs are erased before the download.

```
E
D application.mc
S APP01
A 0
```

The last command is not necessary for VLT or MACS controllers because they store automatically in flash eprom.

The answer file should be as follows:

```
#C 0 1 E
#C 0 1 D test.mc
#C 0 1 S APP01
#C 0 1 A 0
#R 0 1 1 0
```

Loading an Application into a Range of Controllers

Same as above, but for controllers 1 to 2.

```
# 1 - 2
E
D application.mc
S APP01
A 0
```

The answer file should be as follows:

```
#C 0 1 E
#C 0 2 E
#C 0 1 D test.mc
#C 0 2 D test.mc
#C 0 1 S APP01
#C 0 2 S APP01
#C 0 1 A 0
#C 0 2 A 0
#R 0 1 2 0
```

Loading Different Applications into MACS4 and MACS5 Controllers

Loading “MACS5Appl.m” into all MACS5 controllers and “MACS4Appl.m” into all MACS4 controllers.

```
// Connect to all controllers.
# *
// Erase all programs.
E

// Download "Master.mc" to all MACS5 controllers.
& Controller = MACS5
D Master.mc
S MASTER
// Clear all controller selection conditions.
```

```
& *

// Download "Slave.mc" to all MACS4 controllers.
& Controller = MACS4
D Slave.mc
S SLAVE
// Clear all controller selection conditions.
& *
```

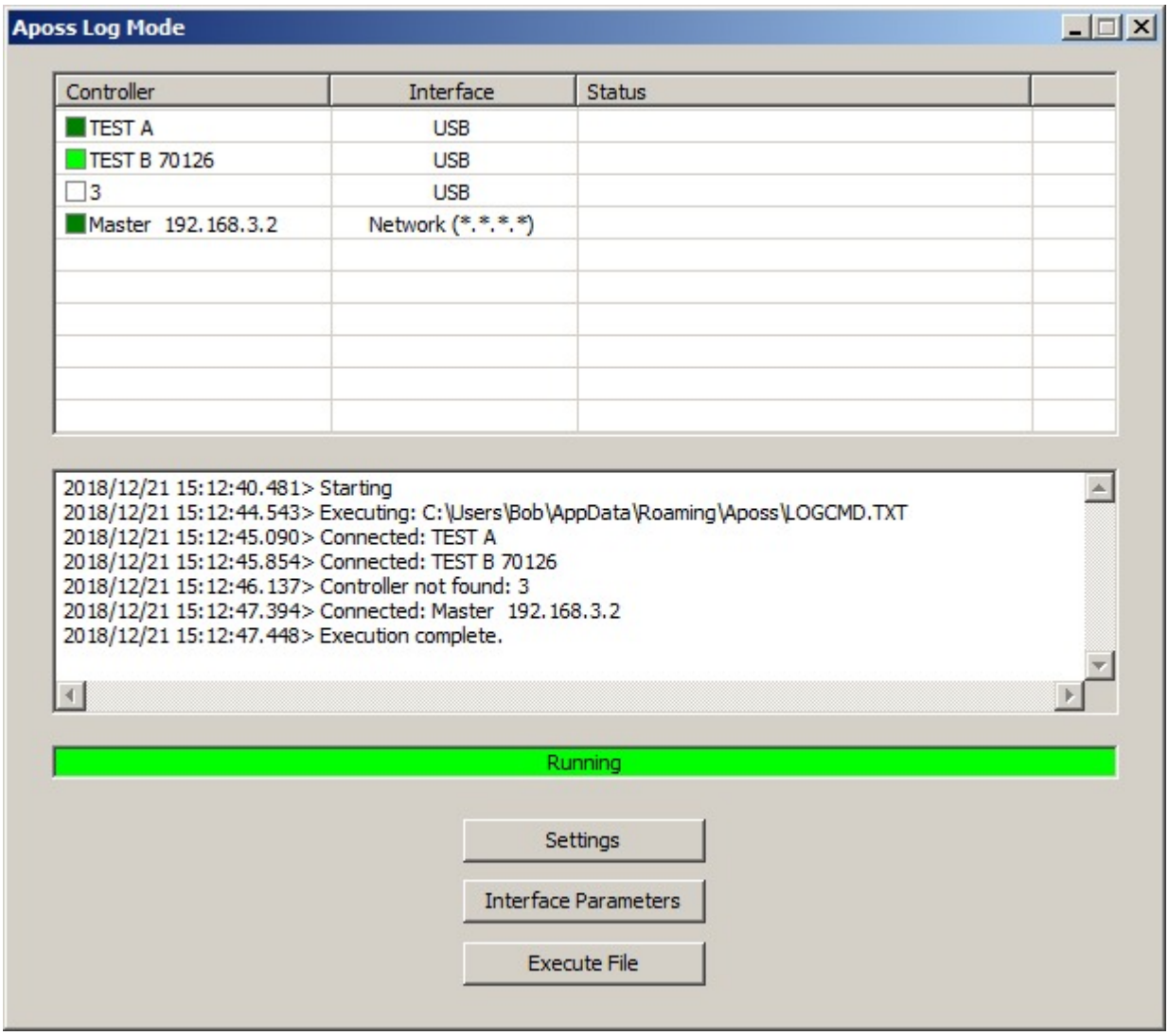
The answer file should be as follows (assuming that controller 1 is a MACS5 and controller 2 is a MACS4):

```
#C 0 1 E
#C 0 2 E
#C 0 1 D Master.mc
#C 0 1 S MASTER
#C 0 2 D Slave.mc
#C 0 2 S SLAVE
#R 0 1 2 0
```

5.9 Log Mode

Log Mode provides a simple client/server mechanism that allows active controllers to be monitored. The monitored values for each controller will be written to a separate log file. This can include both print messages and data values. The log files can then be subsequently retrieved and analyzed. For example, the data values can be imported into the Oscilloscope. When the ApossIDE is started in Log Mode, it will wait for a “command file” to be created (by some external means such as another program). When a file appears, then the file is opened, the commands within it are executed, and logging begins. Once execution is complete and logging has begun, an “answer file” is created and the original command file is deleted. the ApossIDE will then wait for a new command file to be created. Note that a new command file will not necessarily stop the logging begun by a previous command file. For example, new command files can add additional controllers to the list of controllers being monitored.

When started in Log Mode, the ApossIDE displays a dialog (shown below) containing a controller list, a message box, and a status box. If the status box is yellow and displays the “Idle” message, then the ApossIDE is not monitoring any controllers and is waiting for a command file to execute. If the status box is red and displays the “Please wait” message, then the ApossIDE is busy and is not checking for command files. This will happen when the user is modifying parameters or manually executing a command file. When one or more controllers are actively being monitored, the status box will turn green and display the “Running” message. All controllers that are currently being monitored will be shown in the controller list. The message box is used to display various messages to the user.



The controller list contains 3 columns:

Controller This column will display the ID number and name of the controllers being monitored. In addition, there will be a colour box indicating the status of the controller. The colours will be as follows:

White	Controller was not found.
Black	Controller connection has been lost or Log Mode encountered an error.
Red	Controller has entered the error state. The user program may or may not be running.
Green	Controller is connected but there is no user program running. The Application Error bit may or may not be set.
Bright green	Controller is connected and there is a user program running.
Yellow	A user program is running and the Application Error bit is set.

Interface This column displays the interface connection currently being used to monitor the

controller.

Status — If Log Mode encounters an error that prevents proper logging, then the controller colour box will be displayed as black and an explanatory message will be displayed in this column. This will usually only happen with controllers that cannot be found when the command file is executed. For these controllers, logging is configured and started at a time when no answer file is being created.

The user can change various Log Mode settings by pressing the “Settings” button. The interface parameters can be changed by pressing the “Interface Parameters” button.

In addition to automatically detecting command files, the Log Mode dialog allows the user to manually execute command files by pressing the “Execute File” button. These manually executed log files are not automatically deleted when executed. When the user has activated the manual command file interface (or is changing settings or interface parameters), then the status box will be red and will display the “Please wait” message. When this happens, the ApossIDE has deactivated the automatic detection and execution of command files so that they cannot interfere with what the user is doing.

Logging can be stopped in any of the following ways:

1. If a new command file is found, then it may contain commands to stop logging for any controller.
2. If Log Mode is terminated (either by closing the dialog or by executing a command file that causes Log Mode to terminate), then any current logging will be stopped.

A detailed report of the controller status is added to the log file for any of the following reasons:

1. If controller execution terminates for any reason. In particular, this includes the controller terminating as a result of an unhandled fault state.
2. If an Application Error bit is defined and that bit is set high. For more details, see the ‘A’ command.

5.9.1 Configuring and Starting Log Mode

Log Mode is designed to be run as a stand-alone tool that may be started using either a Windows command line or a Windows shortcut. However, it is also available from within the ApossIDE by using the **Tools → Log Mode** command.

Log Mode is started by executing the ApossIDE with either the command line argument “/l” or “/l-”. If “/l-” is used, then the application will start minimized and the dialog will not be displayed. Note that only one instance of the ApossIDE may be running at a time so if the ApossIDE is already running, then a message is displayed and the Log Mode instance will terminate. In this case, Log Mode should be run from within the ApossIDE.

A command file (in quotes) can also be explicitly specified as a command line argument following the “/l” or “/l-” argument. When this is done, then that command file is executed immediately. The command file is executed only once and will not be deleted when executed. This is intended to allow Windows shortcuts to be set up to execute specific logging tasks with a minimum of configuration and user intervention. When a command file is specified in this way, Log Mode will not check for other command files to be created; only that single command file will be executed.

The default command and answer files will be “LOGCMD.TXT” and “LOGANS.TXT” in the “Aposs” subdirectory under the Windows “Application Data” directory. If necessary, the user can specify different files by pressing the “Settings” button.

Similarly, Log files will be placed by default into the “Aposs” subdirectory under the Windows “Application Data” directory. If necessary, the user can specify different files by pressing the “Settings” button. If no directory is specified with the filename, then the directory will default as above.

When a command file is being executed, Log Mode will try to establish connections to controllers using the current default interface specified by the **Settings** → **Interface** command. This default interface should be set up using the ApossIDE before starting Log Mode. The interface can also be changed within Log Mode itself by pressing the "Interface Parameters" button. Command files also contain a mechanism that allows interface parameters to be changed temporarily. This is useful if multiple controllers must be connected using different interfaces.

5.9.2 Command File Structure

The command file is an ASCII file which can be created and edited by any text editor. Commands are specified one per line (i.e. separated by either a line feed or a carriage return/line feed). Leading and trailing blanks are ignored and blank lines are ignored. The command file is preprocessed using the ApossIDE Preprocessor before it is executed. This allows preprocessor statements such as #include, #define, and #if to be used as well as any C-style comments.

Commands

The syntax of a command line is as follows:

```
c parameters
```

where "*c*" is a command letter from the following list and "*parameters*" is a set of parameters that depends on the command being specified.

Command List

Valid commands are as follows. Note that all command letters must be uppercase.

Command	Description
`D ssss`	Write new log files into directory "*ssss*". The directory name must not be enclosed in quote characters. If the directory does not exist, then it will be created. This includes parent directories if parent directories also do not exist. Note that this does not affect any log files that have already been opened. This command should be placed before the '#' command. ⚠ The directory "*ssss*" must not contain a trailing '\' character (e.g. "D C:\Log\"). Since command files are preprocessed using the ApossIDE compiler preprocessor, a trailing '\' is interpretted as a line continuation character.
`I interface`	Set the controller connection interface to "*interface*" (see below). This command should be placed before the '#' command.
`T tt ss`	Set limits on the log file. A new log file will be automatically opened every "*tt*" minutes. The maximum value is 35791 minutes or approximately 24 days. If this is 0, then there is no time limit on log files. The default is 0. A new log file will also be opened automatically if the existing log file size becomes larger than "*ss*" megabytes. The maximum value is 1000 megabytes. If this is 0, then there is no size limit on log files. The default is 100. This command should be placed before the '#' command. These values will be used for every '#' command up until the next "T" command.
`# id`	Connect to controller "*id*" using the current connection interface (i.e. either the default interface or the interface specified by the last 'I' command). Controller ID numbers must not be less than 1. The ID will default to "1" if it is not specified. Note that for some types of connections that support only single controllers (e.g. serial connections such as V24), the controller ID will be ignored if it is specified. All controller-specific commands up until the next '#' command will apply to this controller. If a new controller is referenced and the connection is successful, then the following actions are taken: 1. The log file is opened. 2. Logging is configured according to the following commands in the command file. 3. When the next 'L' command is executed, the configuration is downloaded to the controller and logging is started.

Command	Description
	If a new controller is referenced but the controller cannot be found, then the following actions are taken: 1. Logging is configured according to the following commands in the command file and the 'L' command, if any, is saved. 2. Connection attempts are repeated once per minute until successful. 3. The log file is opened. 4. If an 'L' command was saved, then the configuration is downloaded to the controller and logging is started. If a previously opened controller (i.e. either previously in the same command file or in a previous command file and whether the controller is currectly connected or not) is referenced, then the following actions are taken: 1. If logging is currently active, then it is stopped, the log file is closed, and the connection to the controller is closed. 2. Any existing configuration is cleared. 3. An attempt is made to connect to the controller. Depending on whether or not the connection is successful, one or the other of the above sets of actions are taken.
`A ndx sub bit flags`	Set the Application Error bit to SDO index "*ndx*", subindex "*sub*", bit "*bit*" where the bit number is 0-based. If this bit goes "high", then it will indicate an application error and a detailed data dump will be added to the log file. Note that this bit is tested only once per second so the bit must be low for at least 1 second and then high for at least 1 second in order to ensure that Log Mode recognizes the change. Only one bit may be specified per controller; a second 'A' command will overwrite the previous 'A' command. This command will apply only to the preceding '#' command. The command file will be aborted if there is no preceding controller or if the SDO cannot be read. The "*flags*" value is optional and can be set using the following flags: **0x0001** Add a detailed data dump to the log file. Otherwise, it is not added. **0x0002** Add "Record" data to the log file. Otherwise, it is not added. **0x0004** Stop logging and close the log file. **0x0008** Reopen the log file. "*flags*" will default to 0x0007 if it is omitted.
`M`	Log print messages. This command will apply only to the preceding '#' command. The command file will be aborted if there is no preceding controller or if the command follows the 'L' command.
`S ndx sub`	Log SDO index "*ndx*", subindex "*sub*". This command will apply only to the preceding '#' command. The command file will be aborted if there is no preceding controller or if the command follows the 'L' command. If the SDO cannot be read from the controller, then it will be ignored and an error code will be reported in the answer file for the command. However, this will not abort the command file.
`P nn ee ii`	This command is used with CAN connections (**COM** or **LPT**) to configure PDOnn using event time "*ee*" and inhibit time "*ii*". It is important that this command is used after the sequence of 'S' commands (see **Example2**). In this ways all SDO indices defined in previous 'S' commands will be configured into PDOnn. The command file will be aborted if there is no preceding controller.
`L PRINT`	Start logging. Only print messages are logged. In this case, the "M" command is unnecessary. This command will apply only to the preceding '#' command. The command file will be aborted if there is no preceding controller or if the command follows a previous 'L' command.
`L FREERUN nn ee ii`	Start logging. Logging will be done using SDOs configured into a PDO. SDO indices will be configured into PDO*nn* using event time "*ee*" and inhibit time "*ii*". Note that the event and inhibit

Command	Description
	times may be automatically increased if the specified values are smaller than the minimum supported by the connection interface. The actual values used will be recorded in the answer file when the 'L' command is executed. This command will apply only to the preceding '#' command. The command file will be aborted if there is no preceding controller or if the command follows a previous 'L' command.
`L TESTSETP aa tt cc nn`	Start logging. Logging will be done using SDOs configured into a "Record" (TESTSETP) array. DIM array "*aa*" will be used (-1 for dynamic memory), the sample time will be "*tt*", cyclic recording will be done if "*cc*" is 1, and the sample count will be set to "*nn*" (0 for "fill available space"). This command will apply only to the preceding '#' command. The command file will be aborted if there is no preceding controller or if the command follows a previous 'L' command.
`R`	Reopen new versions of any existing log files (with an updated instance number). This will update the existing logs with all current information before closing them. The files can then be copied by other applications.
`W ndx sub val`	Write the value "*val*" to the specified SDO. This command will apply only to the preceding '#' command. The command file will be aborted if there is no preceding controller or if the command fails.
`X id`	Stop monitoring controller "*id*". The ID will default to "1" if it is not specified. The controller is expected to be using the current default interface (or the interface specified by the last 'I' command). If the controller cannot be found, then an error code will be reported in the answer file for the command. However, this will not abort the command file. Note that this value is *not* ignored even for the types of connections that support only single controllers (e.g. serial connections such as V24).
`Z`	Stop monitoring all controllers, exit Log Mode, and close the window. This command is ignored if it is in a command file that is being executed manually using the "Execute File" button or if it is executed in a command file specified on the Windows command line. In all other cases, it is executed. Note that the exit is immediate and all following commands will be ignored.
`?`	Query the current list of controllers and append the controller information to the answer file.
`! string`	Print "*string*" to the message window.

The interface to be used to connect to controllers can be selected using the 'I' command. If used, then all subsequent '#' commands (i.e. that open controllers) up until the next 'I' command, will use that interface. Note that all changes made by the 'I' command are temporary and are local to that one command file. Changing the interface will *not* change the interface for future command files. The 'I' command does not affect the original default ApossIDE interface or any of the saved interface parameters. If an invalid interface is specified, then this will cause the command file execution to be aborted.

Valid interface types are:

V24	RS232 serial connection.
LPT	CAN connection using the LPT port.
COM	CAN connection using a CAN-USB stick.
USB	USB connection.
CAT	Beckhoff EtherCAT connection.
VLT	RS485 connection to Danfoss VLT5000 or FC300 drive.
MCO	RS485 connection to MCO305 onboard RS485 port.
TCP	Ethernet connection.
ZBC	zub EtherCAT connection.

Note that full interface specifications are also supported. This allows individual interface parameters to be specified. For example:

```
USB,6,50,0x0,921600,1
TCP,0,[172.16.1.1],23,6,50,0x1
```

If full interface specifications are needed, then please contact zub machine control AG directly for more information.

Example1:

The following example will connect to controller ID 2 on the CAN-USB interface (i.e. “COM”). The directory “\Work\Log” will be created (one level at a time) if it does not already exist and the log files will be placed in that directory. The Application Error bit is set to bit 24 (the low-order bit in the high-order byte) of SDO 0x2206, subindex 0. Print messages will be logged. PDO1 will be used to transmit the system clock (SDO 0x2202 subindex 23) and user parameter 1 (SDO 0x2201 subindex 1) every 500 milliseconds. New log files will be started every hour. Logging will be done using the “freerun” method.

```
I COM                // CAN-USB interface.
D C:\Work\Log        // Create " C:\Work\Log" if it does not exist.
T 60 100             // Start a new log every hour (max 100MB).
# 2                  // Controller ID 2.
A 0x2206 0 24        // Application Error bit.
M                    // Log print messages.
S 0x2202 23          // System clock.
S 0x2201 1           // User parameter 1.
L FREERUN 1 500 500  // Start logging. Generate a PDO every 500 ms.
```

Example2:

The following example will connect to controller ID 2 on the CAN-USB interface (i.e. “COM”). The directory “\Work\Log” will be created (one level at a time) if it does not already exist and the log files will be placed in that directory. The Application Error bit is set to bit 24 (the low-order bit in the high-order byte) of SDO 0x2206, subindex 0. Print messages will be logged. PDO1 will be used to transmit the system clock (SDO 0x2202 subindex 23) and user parameter 1 (SDO 0x2201 subindex 1) every 500 milliseconds. PDO2 will be used to transmit user parameter 2 (SDO 0x2201 subindex 2) and user parameter 3 (SDO 0x2201 subindex 3) every 500 milliseconds. PDO3 will be used to transmit user parameter 4 (SDO 0x2201 subindex 4) and user parameter 5 (SDO 0x2201 subindex 5) every 500 milliseconds. New log files will be started every hour. Logging will be done using the “freerun” method.

```
I COM                // CAN-USB interface.
D C:\Work\Log        // Create " C:\Work\Log" if it does not exist.
T 60 100             // Start a new log every hour (max 100MB).
# 2                  // Controller ID 2.
A 0x2206 0 24        // Application Error bit.
M                    // Log print messages.
S 0x2202 23          // System clock.
S 0x2201 1           // User parameter 1.
P 1 500 500          // Configure PDO1 with mappings(SDO 0x2202 subindex 23, SDO 0x2201 subindex
1).
S 0x2201 2           // User parameter 2.
S 0x2201 3           // User parameter 3.
P 2 500 500          // Configure PDO2 with mappings(SDO 0x2201 subindex 2, SDO 0x2201 subindex
3).
S 0x2201 4           // User parameter 4.
S 0x2201 5           // User parameter 5.
L FREERUN 2 500 500  // Configure PDO with mappings for PDO3(SDO 0x2201 subindex 4, SDO 0x2201
subindex 5) and start logging. Generate a PDO every 500 ms.
```

5.9.3 Answer File Structure

When a command file is executed, the result of each command is written to the answer file in the following format:

```
#C nn cmd
```

where “*nn*” gives a ZbMoc error code for the command that was executed and “*cmd*” is a string specifying the command that was executed. Any non-0 error code is not fatal; the command is ignored and execution will continue. Note that if the ‘#’ command returns an error, then all controller-specific commands (e.g. ‘M’, etc.) up until the next ‘#’ command will be ignored.

Commands may be following by any number of information messages of the following form:

```
#I nn ssss
```

where "*nn*" gives an information code (see below for a list of information codes) and "*ssss*" is a string specifying the information. These messages will be used to return useful information to the client.

If a command file error occurs, then error messages of the following form will be written to the answer file:

```
#E nn ssss
```

where "*nn*" gives the error code (see below for a list of error codes) and "*ssss*" is a string specifying error information. Note that all of these errors are fatal and will cause execution of the command file to be terminate. These errors indicate a problem in the client that occurred during the creation of the command file.

The last line in the answer file will specify the result of the complete execution as follows:

```
#R nn mm
```

where "*nn*" gives the error code for the complete execution (see below for a list of error codes) and "*mm*" gives the first non-0 command error code. The command error code will be 0 if the entire command file executed without errors.

Example:

The following example is a possible answer file to the command file in Command File Structure.

```
#C 0 I COM
#C 0 D C:\Work\Log
#C 0 # 2
#I 1 2 - MACS4 70126
#I 2 USB
#I 3 C:\Work\Log\2 - MACS4 70126_2013-06-21-1.zbl
#I 4 0 100
#I 5 0x20 0
#C 0 A 0x2206 0 24
#C 0 M
#C 0 S 0x2202 23
#C 0 S 0x2201 1
#C 0 T 60
#C 0 L FREERUN 1 500 500
#I 6 1 500 500
#I 8 2 0x2202/23 0x2201/1
#R 0 0
```

Error Codes

The possible error codes written into the answer file are as follows:

0	No error.
-1	Could not open, read, and/or preprocess command file.
-2	Command file is empty.
-3	Invalid command.
-4	Controller is not open.
-5	Invalid directory (D command).
-6	Invalid interface specification (I command).
-7	Invalid controller ID.
-8	Could not open log file.

-9 Invalid command arguments.

-10 Application Error SDO could not be read.

-11 Commands are in the wrong sequence.

-12 SDO write failed.

-13 "Record" (TESTSETP) logging could not be started.

-14 Too many SDOs have been configured for PDO.

-15 Logging is already active (L command).

-16 PDO number is not supported by controller.

Information Codes

The possible information codes written into the answer file are as follows:

0 Number of controllers being monitored.

1 Controller name.

2 Interface type.

3 Log file name.

4 Log file limits (time in minutes followed by size in MB).

5 Controller status code (see below) and error code (see above).

6 FREERUN logging: PDO number, PDO event time, PDO inhibit time.

7 "Record" (TESTSETP) logging: Array number, sample time, cyclic flag, sample count.

8 SDO count, list of SDO indices and subindices.

9 Logging mode: PRINT, FREERUN, or TESTSETP.

Information status codes are as follows:

0x01 Log Mode encountered an error while configuring and starting logging.

0x02 Controller is not open.

0x04 Controller connection has been lost.

0x08 Controller has an error.

0x10 Controller has no error and is *not* executing a user program.

0x20 Controller has no error and *is* executing a user program.

Note that status 0x01 will usually only happen with controllers that cannot be found when the command file is executed. For these controllers, logging is configured and started at a time when no answer file is being created.

5.9.4 Log File Structure

When logging is active, all data recorded from the controller will be written to a log file. Each controller will have its own log file. The name of the log file will be the same as the controller name displayed in the Log Mode controller list.

However, all characters that are invalid in filenames (e.g. <, >, *, etc.) will be removed. Log files will always have the file extension ".zbl".

The log file name will include the date on which the log was started and an instance number (for the case when multiple logs are started on the same day). The number of days to keep the log files can be configured using the Settings button. Log files older than this will be automatically deleted.

Log files will be ascii text files and may be viewed using any text editor.

Each record in the log file will be structured as follows:

Type	Description
`I nn ssss`	Information record. These records are similar to the information records in the Answer File Structure.
`T e clk time`	Time synchronization record. "*e*" specifies the elapsed time in seconds since Log Mode was started. "*clk*" specifies the approximate controller system clock value that corresponds to this time. This should be synchronized to within a few milliseconds. "*time*" is the actual time that corresponds to "*e*".
`S nn ssss`	SDO record. "*nn*" specifies the number of SDO values that are being monitored. "*ssss*" specifies the list of SDO indices and subindices. The order of this list will specify the order of the values in the log file.
`L type`	Logging has been started. "*type*" specifies the type of logging and may be any type specified with the 'L' command (e.g. "PRINT", "FREERUN", etc.).
`M e sss`	Print message record. This will be written each time that a print message is received (if the 'M' command was executed). "*e*" specifies the elapsed time in seconds since Log Mode was started, when the message was received. "*sss*" specifies the message text.
`V e clk sss`	Value record. If data are being collected using PDO messages, then this will be written each time that a PDO is received. "*e*" specifies the elapsed time in seconds since Log Mode was started, when the PDO was received. If the controller system clock was specified as an SDO, then "*clk*" specifies the system clock value when the PDO was sent. If the system clock was not specified as an SDO, then this value will be -1. "*sss*" specifies the SDO values that were configured into the PDO (by the 'S' commands). If data is being collected using a "Record" array, then "*e*" will be the elapsed time when the the recording was stopped and "*clk*" will be the system time when each data record was recorded. In this case, "clk" will always be valid and will not be -1.
`D e`	Detailed report record. This will be written each time that a detailed status report is written to the log file. "*e*" specifies the elapsed time in seconds since Log Mode was started, when the detailed report was started. This will be followed by the detailed report. A second 'D' record will terminate the report. Note that 'M' and 'P' records may be interspersed within the report if they are received while the report is being generated.
`E e nn ...`	Event record. This will be written when various errors or other events are detected. "*e*" specifies the elapsed time in seconds since Log Mode was started, when the error or event was detected. "nn" specifies the event code as follows: 1 Controller status has changed. This is followed by two parameters: the old status and the new status. See Information Codes for a description of the possible status codes.

Example

The following example is a possible log file for the command file in Command File Structure.

```
I 1 2 - MACS4 70126
I 2 2,USB,6,50,0x0,921600,1
I 3 C:\Work\Log\2 - MACS4 70126_2013-06-25-1.zbl
I 4 0 100
I 5 0x20 0
T 7.254 8031956 2013/06/25 10:10:45.708
I 6 1 500 500
S 2 0x2202/23 0x2206/0
L FREERUN
V 7.632 8031988 8031988 16512
M 7.629 This is a print message.
V 8.132 8032488 8032488 16512
M 8.662 This is a print message.
V 8.632 8032988 8032988 16512
V 9.132 8033488 8033488 16512
M 9.669 This is a print message.
V 9.632 8033988 8033988 16512
V 10.132 8034488 8034488 16512
M 10.673 This is a print message.
V 10.632 8034988 8034988 16512
V 11.132 8035488 8035488 16512
M 11.693 This is a print message.
V 11.632 8035988 8035988 16512
V 12.132 8036488 8036488 16512
M 12.701 This is a print message.
V 12.632 8036988 8036988 16512
V 13.132 8037488 8037488 16512
M 13.729 This is a print message.
```

5.9.5 Oscilloscope

The ".zbl" log files created by Log Mode can be opened and displayed by the Oscilloscope. This can be done using any of the standard mechanisms to open files (i.e. using the **File** → **Open** command, double-clicking on the ".zbl" file, etc.).

When a ".zbl" file is opened, the ApossIDE will automatically create a separate ".txt" file containing only the print messages from the ".zbl" file. This will allow print messages to be more easily viewed with a simple text editor. The file name of the ".txt" file will be the same as the original ".zbl" filename with "_msg" appended to it. If the ".zbl" file contains no print messages, then this file will not be created.

If the ".zbl" file contains diagnostic report data, then this data will also be separated out into ".txt" files. These files will have "_diag" appended to the original filename. Since there can be more than one diagnostic report in a ".zbl" file, each ".txt" file will also have a sequence number appended to it.

5.10 Recorder

The Recorder is a stand-alone PC program that has been created to demonstrate a data recorder application using one or more zub MACS5 (or newer) controllers connected using TCP. It is intended to provide a function similar to a black-box recorder. It will record user-specified values at a user-specified interval and save the values to both a PC and to microSD memory cards on the controllers. Any value accessible to the controller may be recorded. This data can then be analyzed at any future point in time, for example after a crash.

> Source code for this application is available upon request. Please contact zub machine control AG.

5.10.1 Setting Up the ApossIDE Recorder

The Recorder application is installed along with the ApossIDE. However before running it, the application must be set up as follows:

1. Ensure that the Windows FTP Client is allowed through the Windows Firewall. For instructions on how to do this, see External Storage Device (microSD Card/ USB Memory Stick).
2. Create a directory into which the Recorder program will save the output data files containing the recorded data.
3. Start the Recorder program by selecting "Recorder" in the "zub machine control AG" group in the Windows start menu. Press the "Settings" button and look for the "Output Directory" field in the "Setting" dialog. Either drag-and-drop the directory created in step 2 to this field or press "Browse" to select the directory. Press OK to accept the directory and then close the Recorder program.

The Recorder program uses an initialization file called "Recorder.dat" that is placed in the Aposs Application Data directory. On Windows 7, this directory is usually "\Users\...\AppData\Roaming\Aposs". Note that it may be necessary to set Windows to show hidden directories before this directory can be seen. If a Windows shortcut is created to start the Recorder.exe program, then the initialization file may optionally be specified on the shortcut command line. Edit the shortcut properties and append the full pathname of the "Recorder.dat" file to "Target". "Target" should be something similar to this:

```
"C:\Program Files (x86)\Aposs\Recorder.exe" "C:\Users\Bob\AppData\Roaming\Aposs\Recorder.dat"
```

Recording may be auto-started by appending the flag /s to the shortcut target after the file, like this:

```
"C:\Program Files (x86)\Aposs\Recorder.exe" "C:\Users\Bob\AppData\Roaming\Aposs\Recorder.dat" /s
```

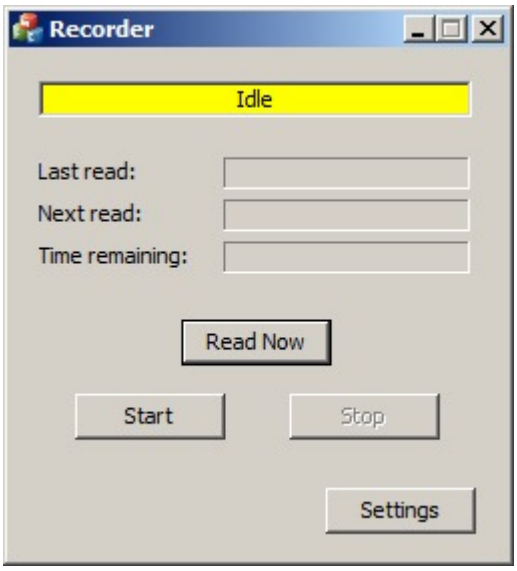
5.10.2 Preparing the Controller

The Recorder application only runs on a MACS5 (or later) controller connected using TCP. Ensure that the controller can be connected and that it contains the appropriate firmware. Do this as follows

1. Start the ApossIDE and open the "SamplingMemDump.m" program. This can be found in the Aposs Application Data directory. A backup copy of this program is also saved in the Help subdirectory under the ApossIDE install directory.
2. Make sure that the "Network" interface is the default interface and that the "Search" option has been enabled. This is done using the "Settings/Interface" menu command.
3. Press ESC to connect to the controller. This will ensure that the controller can be accessed using TCP.
4. Look at the connection banner reported by the controller (in the ApossIDE communications window) and make sure that the firmware is at least version 7.2.19. If not, use the **Controller** → **Download Firmware** command to download the correct firmware to the controller. Contact zub machine control AG to obtain the appropriate firmware.
5. Power the controller off. Note that this will cause the connection to the controller to be lost and a warning message will be displayed. Insert a blank microSD card into the card slot on the controller. Then power on the controller again. Do not insert a microSD card into the controller while it is powered on.
6. Press ESC to re-connect to the controller.
7. Use the Array Editor toolbar icon to start the Array Editor. Upload "User, Global, Axis Parameters". The other parameters need not be uploaded. Ensure that User Parameter 1 (index 130) is set to 0. If not, then set it to 0 and write the parameters back to the controller. Close the Array Editor. The application uses this value to determine file numbers to be used when saving data files on the microSD card.
8. Press F5 to download and start the Data Recorder application running on the controller. If multiple controllers are being used, then start "SamplingMemDump.m" running on each controller. The "SamplingMemDump.m" application will start recording data as soon as it is started.
9. Use the "Close interface" toolbar icon to disconnect from the controller. The ApossIDE and the Recorder.exe program cannot both be connected to the controller at the same time.
10. Close the ApossIDE.

5.10.3 Starting the Recorder on the PC

Start the Recorder using the Windows Start menu. The Recorder will be located under the "zub machine control AG" menu heading. If a shortcut was created (see Setting Up the ApossIDE Recorder), then double-clicking the shortcut will also start the Recorder program running. The Recorder dialog will be as follows:



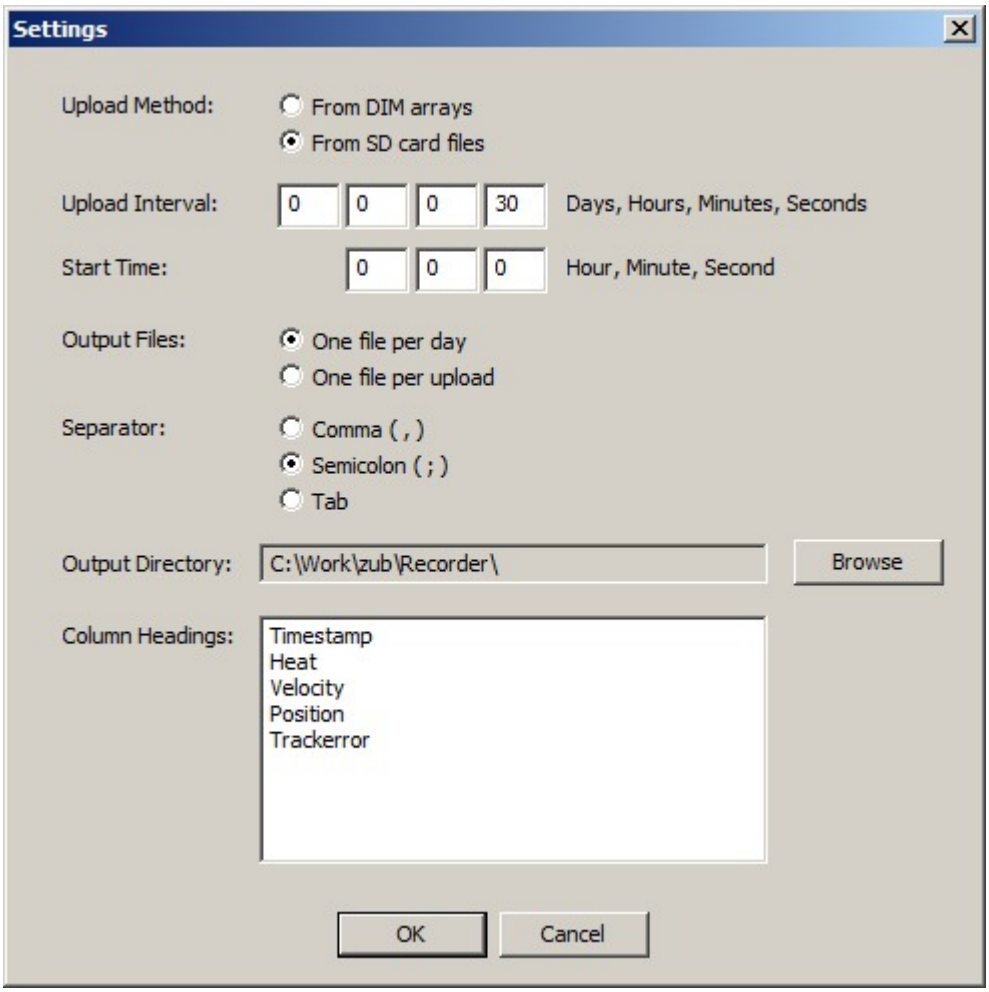
The buttons are:

Read Now	This will cause the currently recorded data to be uploaded immediately. Note that this button can be used even when uploads are already being done automatically.
Start	This will start the automatic uploading of recorded data. The time interval between uploads can be set using the **Settings** button. Note that the first upload is done immediately. Thereafter, uploads are done on the given schedule.
Stop	This will immediately do one final upload and then stop the automatic uploading of recorded data.
Settings	This allows the user to adjust various settings.

Ensure that the settings are correct before starting to upload recorded data from the controller.

5.10.4 Recorder Settings

The Settings dialog is shown below:



Upload Method	Either recorded data may be uploaded directly from the DIM arrays being used to save the data or the microSD card files themselves may be uploaded.
Upload Interval	This is the interval at which the automatic uploading of recorded data will be done.
Start Time	This is the time of day at which the first automatic upload will be done. Note that if the upload interval does not divide evenly into a day, then the daily start time will “drift”.
Output Files	The type of output files to be created.
Separator	The separator character to be used in the output files.
Output Directory	The directory where the output files are to be written.
Column Headings	The column headings to be written to the output files.

5.10.5 microSD Card Files

Recorded data is saved onto the microSD card in files named “DatFile##.txt” where “##” is a sequential number starting at the value saved in User Parameter 1 (normally a value of 0). A new file is saved in the following

situations:

1. Whenever the user press the Read Now button.
2. Whenever the Recorder program does an automatic upload.
3. Whenever the "SamplingMemDump.m" application fills the internal buffer it is using to save recorded data.

User Parameter 1 is used as the starting number of file numbers so that the controller may be powered off/on without causing previously saved files to be overwritten. The "SamplingMemDump.m" application updates this value each time a file is saved.

5.10.6 PC Output Files

The Recorder program will automatically create a subdirectory under the specified output directory for each connected controller. The controller's IP address will be used as the subdirectory name. All data files saved from the controllers will be placed into the appropriate subdirectory. The following files are saved:

1. If the From SD card files Recorder setting has been selected, then all "DatFile##.txt" files will be uploaded and saved. Otherwise, these files are not uploaded to the PC. The user can retrieve them manually from the microSD card.
2. If the One file per day Recorded setting has been selected, then one output file is created per day. All recorded data from that day will be appened to the file. The file will have the name "yyyymmdd.csv" where "yyyymmdd" is the year, month, and day. This file is a "comma-separated value" file that may be opened with a spreadsheet application.
3. If the One file per upload Recorder setting has been selected, then one output file will be created each time data is upload (either from microSD card files or from DIM arrays). These files will have the name "yyyymmdd_hhmmss.csv" where "yyyymmdd_hhmmss" is the date and time when the upload was completed. These files may have recorded data from more than one day in them.

5.10.7 Customizing the Application

The "SamplingMemDump.m" application is currently configured to save 5 values once per second. DIM arrays of 30000 are used to save the data. This means that data will be recorded for approximately 1½ hours before being saved to the microSD card. Follow these steps to change this:

1. Edit the "SamplingMemDump.m" program.
2. Search for the "SAMPLING_RATE" definition. Set this to the required sampling interval.
3. Search for the "SAMPLING_ARRAYSIZE" definition. This is the length of the DIM arrays that will be used to save the data. Calculate how long it will take to fill an array of this size when saving the recorded data at the specified rate. Set the array length appropriately.

Note that data is saved to the microSD card each time the "Recoder.exe" program uploads the data. This will also affect how long the DIM must be.

The "SamplingMemDump.m" application is currently configured to save the following 5 values:

1. System time.
2. Dummy "heat" value.
3. Commanded motor position.
4. Actual motor position.
5. Tracking error.

To change this, follow these steps:

1. Edit the “SamplingMemDump.m” application.
2. Search for the “SampleRecord” function.
3. Modify the lines at the beginning of the function to change the values being saved. You are not limited to 5 values. Note that the system clock value must always be saved and it must be saved first.
4. Search for the “SAMPLING_COLUMNCNT” definition. Change it to match the number of values being saved.
5. Calculate the required length of the DIM arrays. Change their lengths if necessary using the “SAMPLING_ARRAYSIZE” definition.
6. Execute the “Recorder.exe” program on the PC. Press the “Settings” button.
7. Change the “Column headings” so that they match the new values being saved by the “SamplingMemDump.m” application.

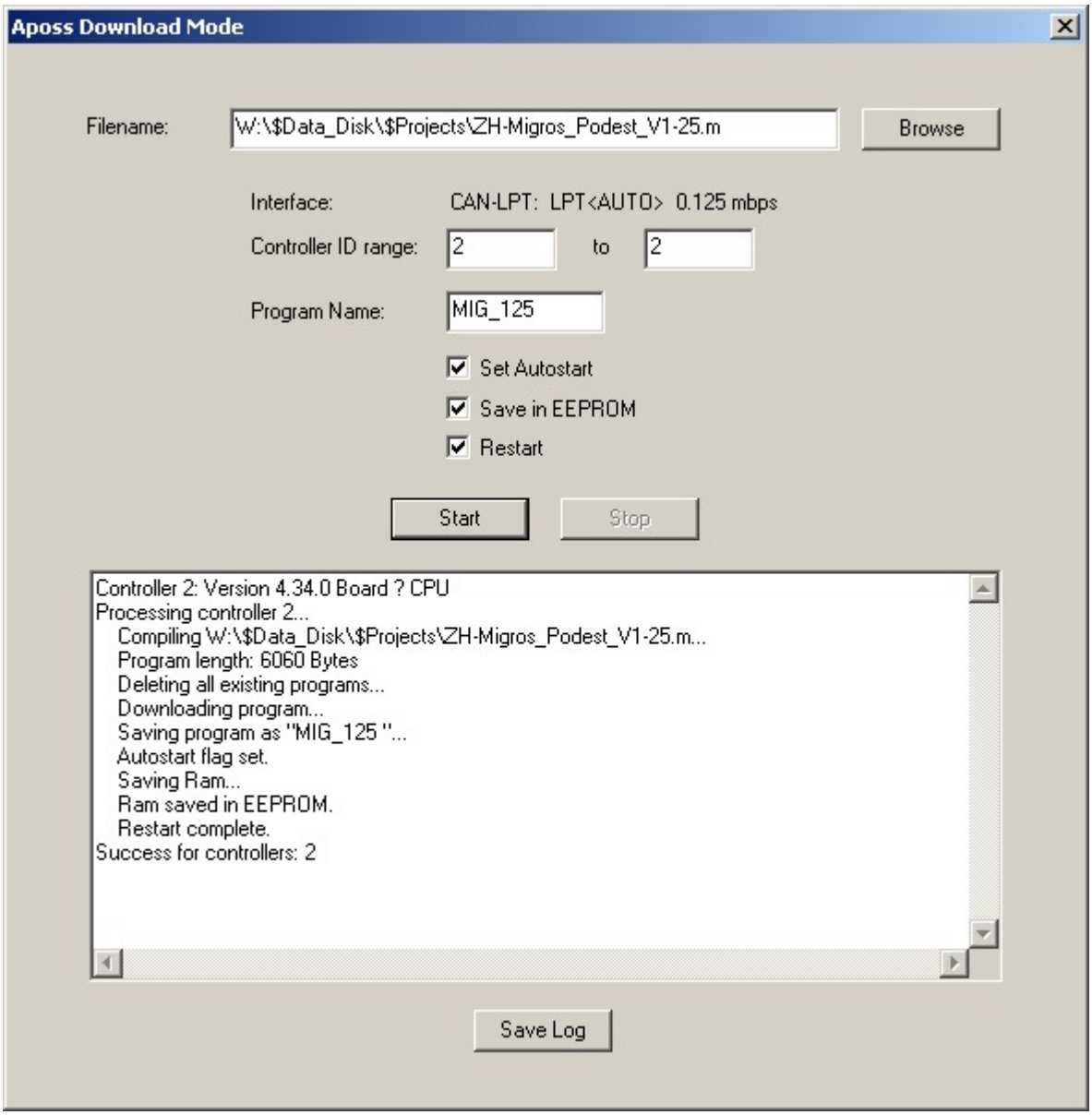
5.10.8 Recorder Limitations

1. If multiple controllers are connected, then all controllers are expected to record the same set of values. Values cannot be specified on a per-controller basis.
2. If a controller is powered off/on, then all unsaved data will be lost. The amount of lost data can be reduced by increasing the frequency of uploads.
3. If a controller is powered off/on more than once between uploads, then all time sychronization will be lost for the saved data.
4. If From DIM arrays is selected in the Recorder settings, then recorded data will not be uploaded from “DatFile##.txt” files that are saved at a faster rate than Recorder.exe is uploading.
5. Recorder.exe must be set to do automatic uploads at a rate faster than the interval at which the controller’s system clock wraps. Currently, the automatic upload should be done at least every 24 days. If not, then the time synchronization in the recorded data will be lost.

5.11 Download Mode

5.11.1 Download Mode

Download Mode provides a simple interface that allows compiling and downloading of programs to multiple controllers. It will also allow downloading of configuration files to multiple controllers.



Starting Download Mode

Download Mode is started by executing the ApossIDE program with the command line argument "/d". Note that only one instance may be running at a time so if the ApossIDE is already running, then a message is displayed and the Download Mode instance will terminate.

When downloading is ready to be begin, the ApossIDE will try to establish a connection to the controllers using the default interface specified by the **Settings** → **Interface** command. This default interface should be set up using the normal ApossIDE before starting the ApossIDE in Download Mode.

Using Download Mode

The Download Mode dialog is shown above. To use it, follow these steps:

1. Enter the name of the application program, binary program file, or configuration file to be downloaded. If the Browse button is clicked, then the normal "File Open" dialog will be displayed allowing the file to be selected.
2. If the controller interface supports multiple controllers, then enter the range of controller ID's to which the program should be downloaded.

3. Enter a program name (up to 8 characters) to be used to identify the program on the controller.
4. Select the desired flags/operations using the check boxes.
5. Click the Start button. The download will then start.

The download will then be done as follows:

1. A test is made to ensure that the specified file exists.
2. A connection is established to the controllers using the default interface.
3. A Break command is sent to all accessible controllers. This will stop any controller which is currently executing.
4. The following steps are performed for each accessible controller. All steps are performed for one controller before going on to the next controller.
    a. (".m" files only) The program is compiled using compiler settings appropriate to the controller.
    b. (".m" and ".bin" files only) All existing programs on the controller are deleted.
    c. The program or configuration is downloaded.
    d. (".m" and ".bin" files only) The program is saved as program number 0 with the specified program name. If no program name was specified, then the first 8 characters of the filename are used.
    e. (".m" and ".bin" files only) If the Verify flag is set, then the downloaded program is uploaded again and the downloaded and uploaded versions are compared. If they differ, then processing is halted for this controller. Note that not all older controllers contain firmware that will support uploading the program. This flag is ignored if the controller does not support uploading.
    f. The Autostart flag is set for the program if requested.
    g. All data is saved in EEPROM if requested. Note that this will be done automatically for newer controllers.
    h. The controller is restarted if requested.
5. The connection to the controllers is closed.

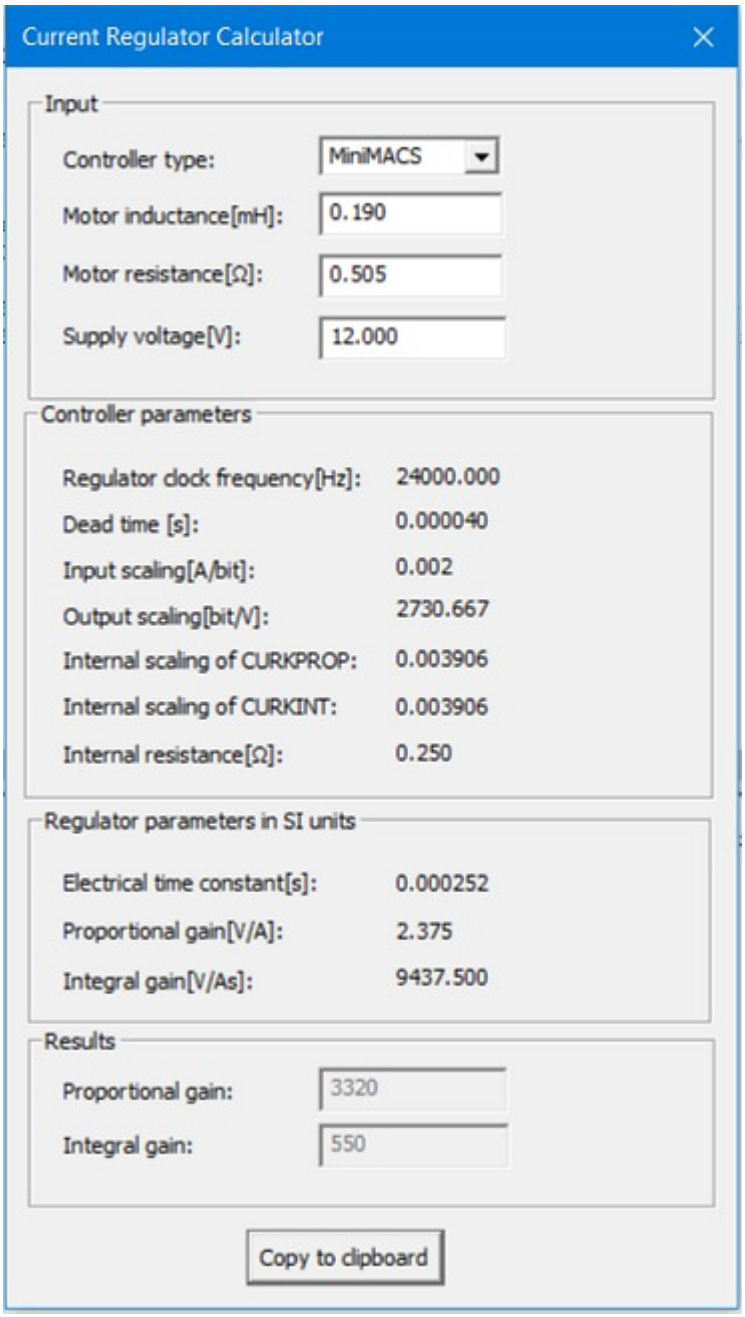
The progress of the download is displayed in the large text box. If desired, this information can be saved to a text file by clicking the Save Log button.

If any error occurs during the processing of a controller, then processing is halted for that controller. However, processing will continue for subsequent controllers in the ID range. When the entire download operation is complete, a summary is displayed in the text box listing the ID's of those controllers that encountered no errors and those controllers that encountered errors. Use the scroll bars to scroll up in the text box in order to determine the nature of any problems encountered.

5.12 Current Regulator Calculator

5.12.1 Current Regulator Calculator

Current regulator calculator provides a simple user interface that allows calculation of Proportional and Integral gain for the Current regulator.



The user interface contains 4 groups of elements:

Input	This group contains input parameters that can be changed by user. They are as follows:	
	Controller Type	Drop down box of controller types supported.
	Motor inductance	Inductance of the motor in millihenry.
	Motor resistance	Resistance of the motor in ohm.
	Supply voltage	Supply voltage in volt.
Controller parameters	This group displays several parameters that cannot be changed as they are dependent on the selected controller type.	

Regulator parameters in SI-units	This group displays several regulator parameters that are calculated based on either input parameters or controller parameters. They cannot be changed by the user.
Results	This group contains the results of the calculation: **Proportional gain, HWAMP_CURKPROP** **Integral gain, HWAMP_CURKINT**

The calculation will be performed every time the user changes one of the parameters in the **Input** group.

The button **Copy to clipboard** enables the user to copy the results to the clipboard and use them in source code. They will be copied in the form:

HWAMP_PARAM(0,HWAMP_CURKPROP) = XXX; where XXX is the value of calculated proportional gain.

HWAMP_PARAM(0,HWAMP_CURKINT) = YYY; where YYY is the value of calculated integral gain.

6 Motion Control Topics

6.1 Scaling

The units for the drive or the slave and the master, respectively, can be defined by the user in any way desired so that the user can work with meaningful measurements.

The factors SYNCFACTM / SYNCFACTS, POSFACT_Z / POSFACT_N are no longer limited to small values starting with firmware version 6.4.43.

Internally, we act as follows: Whenever a value must be multiplied by the gear factor (i.e. master increments per ms), we first look if a multiplication will result in an overflow. If so, we use a factor (64 bit) which consists of

SYNCFACTS/SYNCFACTM

to multiply the delta_master.

If no overflow occurs, we first multiply by SYNCFACTS and then divide by SYNCFACTM.

Concerning the error we are dealing with, this means:

Normal case

Multiplying by SYNCFACTS has no error, but dividing by SYNCFACTM means that the result may be wrong by $1/2^{32}$. That means that (worst case) such an error occurs every ms, i.e. that after 1193 hours (49.71 days) an error of 1 qc (Slave) is made.

Large factors

In that case, the used factor (SYNCFACTS/SYNCFACTM) itself could be wrong by $1/2^{32}$. This means that in the worst case an error of delta_master * $1/2^{32}$ occurs every ms. Assume that we have an encoder with 1000 counts (4000 qc) per revolution. Assume further, that we drive with 2000 RPM, i.e. we have a velocity of 133 qc/ms. This means we make an error of 133 * $1/2^{32}$ per ms. From this follows that in worst case (maximum error every ms always in same direction) we could have an error of 1 qc after 9 hours.

This should not be relevant in most applications.

6.1.1 Master Units [MU]

The curve length or the master cycle length and other information (e.g. the marker distance) for the CAM control are indicated in master units MU.

A factor (fraction) is used for the conversion into [mu] = SYNCFACTM / SYNCFACTS

6.1.2 How to Set Up Resolution and Gear Parameters

In general, motion control applications are designed to work in "user units (uu)" that are in some way appropriate for their specific application. For example, user units might be chosen to be millimeters, inches, degrees of arc, or some other measurement unit. On the other hand, the original source of the measurements is provided by encoder hardware connected to the motor that is driving the machine. Encoders normally provide measurements by generating discrete digital pulses at uniform angular increments as the motor turns. These pulses are resolved into encoder increments called "quadcounts (qc)". The quadcount is the basic resolution of motor positions; motor positions cannot be determined or specified more accurately than to an integral quadcount. For example, the motor position can be 1 qc or 2 qc, but it cannot be 1.5 qc.

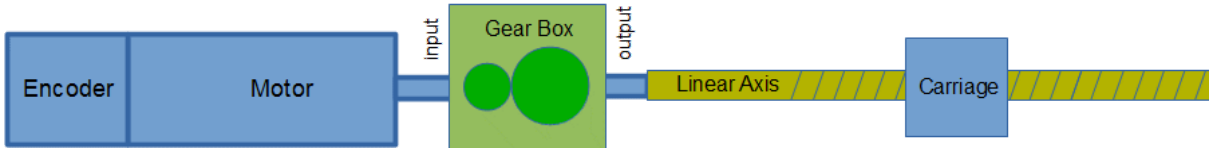
Typically, incremental encoders provide two square wave output signals (Channel A and Channel B) where the second

signal is phase shifted by 90° from the first signal. By tracking the rising and falling edges of these signals, it is possible to identify four positions within each cycle of the outputs. Hence, the name "quadcount". On the other hand, absolute encoders usually have a fixed zero point and simply count increments as the motor moves away from the zero point. While technically these are not "quadcounts", for the purposes of application programs they will be treated as if they are quadcounts and refered to as "quadcounts".

The resolution and gear parameters allow the user to specify the conversion from quadcounts to user units. The controller then uses this information to calculate distances, velocities, accelerations, etc., that are in are units that are natural for the application.

Drive schematic

The following diagram shows an example drive schematic. There will always be a motor with an encoder either mounted directly on the motor shaft or somehow connected indirectly to the motor. Usually, there will also be a gear box and perhaps additional drive mechanics. The gear box typically "reduces" the revolutions of the motor. For example, 3 revolutions of the input shaft may result in 1 revolution of the output shaft. This is called a 3:1 reduction gear.



Encoder parameters

Parameters POSENCREV and POSENCQC taken together define the encoder resolution in quadcounts per motor revolution (qc/rev).

POSENCREV	Number of revolutions of the motor.
POSENCQC	Number of quadcounts in POSENCREV revolutions.

The actual resolution value is calculated as the fraction POSENCQC/POSENCREV. This allows integral POSENCREV and POSENCQC values to be used to specify fractional resolutions to as high a precision as is required. For example, if the exact resolution is 500.5 qc/rev, then POSENCREV could be specified as 2 and POSENCREV would be 1001.

If the encoder is mounted directly on the motor shaft, then typically POSENCREV will be 1 (for 1 revolution) and POSENCQC will be set to the hardware resolution of the encoder in quadcounts/revolution.

Gear box parameters

Parameters POSFACT_Z and POSFACT_N taken together define the ratio of the gear box.

POSFACT_Z	Number of revolutions of the input shaft.
POSFACT_N	Number of revolutions of the output shaft in POSFACT_Z revolutions of the input shaft.

For example, with a 3:1 reduction gear, POSFACT_Z should be set to 3 and POSFACT_N should be set to 1.

The actual gear ratio is calculated as the fraction POSFACT_Z/POSFACT_N. This allows integral POSFACT_Z and POSFACT_N values to be used to specify fractional ratios to as high a precision as is required. For example, if the exact ratio is 3⅓, then POSFACT_Z could be specified as 10 and POSFACT_N would be 3.

If there is no gear box, then both parameters can be set to 1 (default).

Feed parameters

Parameters FEEDREV and FEEDDIST taken together define the feed distance of the axis mechanics per revolution of the gear box output shaft (uu/rev).

FEEDREV	Number of revolutions of the gear box output shaft.
FEEDDIST	Distance travelled (in user units) in FEEDREV revolutions of the gear box output shaft.

This is usually where the conversion to user units (uu) is defined. For example, if the axis moves 1000 millimeters when the gear box output shaft rotates once, then FEEDDIST would be specified as 1000 and FEEDREV would be 1.

The actual conversion factor is calculated as the fraction FEEDDIST/FEEDREV. This allows integral FEEDDIST and FEEDREV values to be used to specify fractional conversions to as high a precision as is required.

Conversion Between User Units and Quadcounts

The following formulas can be used to convert between user units and quadcounts:

$$qc = UU \times \left(\frac{FEEDREV}{FEEDDIST}\right) \times \left(\frac{POSFACT\_Z}{POSFACT\_N}\right) \times \left(\frac{POSENCQC}{POSENCREV}\right)$$

$$UU = qc \times \left(\frac{POSENCREV}{POSENCQC}\right) \times \left(\frac{POSFACT\_N}{POSFACT\_Z}\right) \times \left(\frac{FEEDDIST}{FEEDREV}\right)$$

User Unit Resolution

The basic resolution of the entire machine is 1 encoder quadcount. Using the above formula, this can be converted to the resolution of the machine in user units. This is the highest precision to which an application may specify positions. For example, if 1 uu equals 10 qc, then the command "AxisPosRelStart(0,1);" will move the motor foward by 10 qc; it will not be possible for the application program to move the motor by only 5 qc. Conversely, if the motor moves forward by only 1 qc, then the Apos command is likely to return the same actual position value even though the motor has moved to a new position; the application program will be unable to differentiate between these two motor positions.

As a result of this, the magnitude of user units must be carefully chosen during system design so that the final application program can achieve adaquate resolution.

Example with user unit conversion to [mm]

Using the above diagram, consider the following:

- We would like to convert quadcounts to user units (mm) so that we can work with target positions, etc., in millimeters.
- Encoder resolution is **2000 qc/rev**. The encoder is directly mounted on the motor shaft.
- There is a gear box with **3:2** reduction.
- The linear axis has a pitch of **2 mm/rev**.

POSENCQC	= 2000 [qc]
POSENCREV	= 1
POSFACT_Z	= 3
POSFACT_N	= 2
FEEDDIST	= 2 [mm]
FEEDREV	= 1
Resolution:	1 qc = 1/1500 mm or 1 mm = 1500 qc

With this resolution, the command "AxisPosAbsStart(0,10);" will move the actual position to 10±1/1500 mm.

Example with user unit conversion to [degrees]

In this example, we would like to convert quadcounts to degrees.

- Encoder resolution is **600 qc/rev**. The encoder is directly mounted on the motor shaft.
- There is no gear box.
- There are 360 degrees per revolution.

POSENCQC	= 600 [qc]
POSENCREV	= 1
POSFACT_Z	= 1
POSFACT_N	= 1
FEEDDIST	= 360
FEEDREV	= 1
Resolution:	1 qc = 6/10 deg or 1 deg = 1.66 qc

With this resolution, the command "AxisPosAbsStart(0,10);" will move the actual position to 10±6/10 degrees. Note that this is quite a large error; likely larger than is acceptable to the application. One possible solution is to add a reduction gear box into the system. For example, a 10-to-1 reduction gear would reduce the positioning error to 6/100 degrees. A second possible solution is to replace the encoder with one that has a much higher resolution. For example, a 6000 qc/rev encoder would also reduce the error to 6/100 degrees. Doing both would reduce the error to 6/1000 degrees.

6.1.3 Encoder Position Values

Encoder Position Values

Many commands work with "encoder position values" that define the position of an axis. Basically, encoder position values are 32-bit **long** values. However, their range is limited to -(2^{30}-1) ... +(2^{30}-1) (i.e. -1073741823 ... +1073741823). The MLONG keyword is often used in the Help pages to refer to this range.

Encoder position values are designed so that they properly support overflows and underflows of motor positions as motor movements pass either forward or backward across the ends of the range. This allows the controller to smoothly handle positioning and synchronization without the loss of any encoder pulses. A result of this is that position -1073741823 is *equivalent* to position +1073741823. As well, the incrementing of positions differs slightly depending on whether the movement is forward (overflow) or backward (underflow). The sequence of positions is as follows (note carefully that the center number represents the same position despite the different signs):

Forward: ... ⇒ +1073741821 ⇒ +1073741822 ⇒ **+1073741823** ⇒ -1073741822 ⇒ -1073741821 ⇒ ...

Backward: ... ⇐ +1073741821 ⇐ +1073741822 ⇐ **-1073741823** ⇐ -1073741822 ⇐ -1073741821 ⇐ ...

6.2 Synchronization

6.2.1 Setting Synchronization Parameters

For synchronization applications, when setting up a new controller or connecting a new motor to an existing controller, the following synchronization parameters must be checked to ensure that they are set correctly. It will normally be necessary to adjust other parameters as well, depending on the requirements of the application. However for these other parameters, the factory settings can usually be used at first and then adjusted later as the application is developed. For a detailed description of all global parameters and axis parameters, please refer to

Parameter Reference.

If the controller is connected to more than one motor, then the parameters must be set for *each* axis.

Encoder Data

The following parameters must be set to specify the correct behaviour of the master encoder:

MENCODER	Resolution of the master encoder (counts/revolution).

Synchronization

The following synchronization parameters must be set:

SYNCFACTM	Synchronization factor for master.
SYNCFACTS	Synchronization factor for slave.

Sync Marker

The following parameters are only relevant when using synchronization with marker correction (SYNCM):

SYNCMARKM SYNCMARKS	Master and slave marker count.
SYNCMPULSM SYNCMPULSS	Master and slave marker distance.

Setting Parameters

Axis parameters must be explicitly set at the beginning of every application program. This can be done by modifying the following code and incorporating it into the program. Set "axes" according to the controlled axis number. See Initializing Temporary Parameters for more information on setting parameter values.

```
AXE_PARAM(axis, MENCODER)      = 500;
AXE_PARAM(axis, SYNCFACTM)     = 1;
AXE_PARAM(axis, SYNCFACTS)     = 1;
AXE_PARAM(axis, SYNCMARKM)     = 1;
AXE_PARAM(axis, SYNCMARKS)     = 1;
AXE_PARAM(axis, SYNCMPULSM)    = 500;
AXE_PARAM(axis, SYNCMPULSS)    = 500;
```

⚠ If parameters are modified by an application program (for example, with Sysvar), then it is necessary to re-execute the Vel, Acc, and Dec commands in order to activate the new settings.

6.2.2 Synchronization Commands

Commands to synchronize a slave with a master:

SyncCam	Enter CAM mode synchronization and wait in an "idle" state.

SyncCamMarkerMaster	Synchronization in CAM mode with master marker correction.
SyncCamMarkerSlave	Synchronization in CAM mode with slave marker correction.
SyncCamStart	Start slave for synchronization in CAM mode.
SyncCamStop	Stop slave after the CAM synchronization.
SyncDefOrigin	Defines master-slave relation for the next SyncPos or SyncMarker command.
SyncError	Queries actual synchronization error of the slave.
SyncSetCurve	Set a CAM curve.
SyncShiftOrigin	Relative shifting of the origin of synchronization.
SyncMarker	Synchronization of angle/position with marker correction.
SyncMarkerRestart	Resets a marker or resets marker handling.
SyncPos	Synchronization of angle/position.
SyncStatus	Queries flag for synchronization status.
SyncStatusClear	Resetting of the flags MERR and MHIT.
SyncVel	Synchronization of velocity.

Commands to access elements within a CAM array:

CamArrayIndex	Determine the array index of various data blocks within a CAM curve array.
CamArrayLen	Calculate the required length of a CAM array.

6.3 Optimizing the PIDController

6.3.1 How the Control Process Works

The position controller has two parts:

1. The Set point Generator interprets the various positioning commands and generates a series of set point positions that eventually ends with the desired position.
   Normally, all positioning processes have a trapezoidal shaped velocity curve. That means that after a phase of constant acceleration there is a phase with constant velocity and finally a phase with constant deceleration, which ends at the desired target position.
2. The PID controller receives the set point positions from the Set point generator and in turn calculates the speed reference needed for the motor to follow the current set point position.
   By setting the PID control parameters you can directly influence to what degree and how quickly a deviation from a theoretical set path (as specified by the set point series) should be counteracted.

The following behavior indicates that the control parameters are not optimally adjusted:

- Drive vibrates
- Drive is loud
- Frequent occurrence of position errors
- Poor control accuracy

> The control parameters are load-dependent. Thus the drive must be optimized under the actual conditions of use.

In exceptional cases it may be necessary to determine various sets of control parameters while working with heavily changing load conditions and to re-program them in subsequent application programs depending on the motion process.

6.3.2 Position Control Loop Design

The PID control unit of the positioning system transfers the necessary output frequency via an internal speed reference to the amplifier. This set value is periodically re-calculated with an interval of one millisecond (interval is programmable by the TIMER parameter). Controllers are by default set with soft 'fit for all' controller parameters.

The PID filter works according to the following formula:

1 = (FFVEL standardized) * (Setpoint velocity)

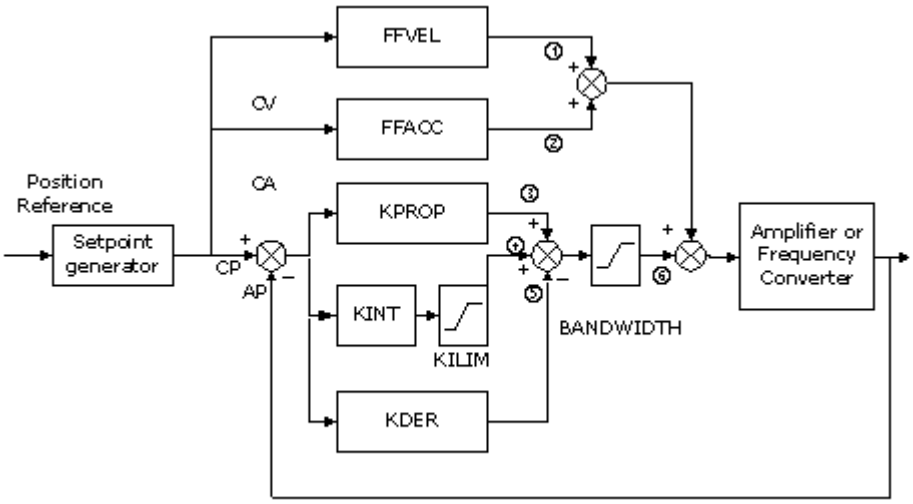
2 = (KFFACC and KFFDEC standardized) * (Setpoint acceleration)

3 = KPROP * (Position deviation CP - AP)

4 = KINT * (Sum of all previous position deviations) (limited by KILIM)

5 = KDER * (Difference of position deviation)

6 = 3 + 4 + 5 (limited by BANDWIDTH)



AV actual velocity, calculated as difference of the actual velocity minus the last velocity

AP actual position (calculated from encoder feedback) in qc.

CP current position set point in qc.

CV velocity set point in qc/ms; it is calculated as difference of the actual set point position minus the last set point position.

CA acceleration set point

Position deviation	=	is calculated by CP-AP.
Diff. of the position deviation	=	position deviation - old position deviation
KILIM standardized	=	a KILIM value of 1000 corresponds with a command value of 100 %.
FFVEL standardized	=	a FFVEL value of 1000 and a command velocity = = VELMAX corresponds with a command value of 100 %
KFFACC standardized	=	a KFFACC value of 1000 and an acceleration of 100% (related to RAMPMIN), results in a amplifier reference current of (VIRTAMP_REFACC100PERC * 2mA).
KFFDEC standardized	=	a KFFDEC value of 1000 and a deceleration of 100% (related to RAMPMIN), results in a amplifier reference current of -(VIRTAMP_REFACC100PERC * 2mA).

In SyncVel mode the PID controller is working with speed deviation instead of position deviation. Speed deviation is calculated by CV-AV.

The controller utilizes two control strategies at the same time:

1. An open-loop feed-forward control. Since an asynchronous motor inherently has a good open loop performance, the feed-forward control is a very important part of the controller in most applications. Benefits from using feed-forward control include a very fast and accurate response to changes in the Set point reference.
2. A closed-loop PID control. The PID controller monitors the difference between the actual position and the Set point position. Based on this information, it calculates a control signal to minimize the position deviance. Thus the controller is able to compensate for changes in load or friction. The PID controller is also necessary to compensate for any position deviance caused by inaccurate setting of the open-loop feed-forward control.

In short: The feed-forward control is used to handle changes in the Set point reference (especially important in synchronization applications), while the PID control is used to handle changes in load conditions or inaccuracies of the feed-forward control.

Please see the section on how to optimize your controller settings.

6.3.3 PID Factors

Proportional Factor: KPROP

The ***Proportional Factor*** is multiplied with the position deviance and the result is added to the control signal. Since the calculated control signal is proportional to the position deviance (or error) this kind of control is called proportional control. The behavior of the proportional control is similar to that of a spring - the further the spring is extended the stronger the counter-force it produces.

Influence of the ***Proportional Factor***:

KPROP too small	large position deviation since the controller cannot adequately compensate for load and friction changes
KPROP larger	quicker reaction, smaller steady-state deviation, larger overshoot, less damping
KPROP too great	heavy vibrations, instability

Derivative Factor: KDER

The ***Derivative Factor*** is multiplied with the derivative of the position deviance (the 'velocity' of the position deviance) and the result is added to the control signal. The behavior of the derivative control is similar to that of an absorber - the faster the absorber is extended the stronger the counter-force it produces. Thus using the ***Derivative Factor*** increases damping in your system.

Influence of the ***Derivative Factor***:

KDER small	no effect
KDER larger	better dampening, less overshoot; if KPROP is increased simultaneously: faster reaction to control deviation at the same level of vibration.
KDER too large	heavy vibrations, instability

If you are working with a normal PID algorithm, then FFVEL must always be just as high as the KDER share in order to achieve this typical KDER-damping.

However, if you are controlling large, slow-acting motors which could vibrate, then the control will be limited via the bandwidth. For this set the BANDWIDTH very low, e.g. 10% and work with FFVEL, KFFACC and KFFDEC, which are both set relatively high.

Integral Factor: KINT

The sum of all error is calculated every time the control signal is updated. The ***Integral Factor*** is then multiplied with the sum of all positioning errors and added to the overall control signal. Thus in case that steady-state position errors occurs in your application, make sure you use the integral part of the controller. Steady-state errors will be leveled out as the summed error increases over time until the control signal eventually matches the load.

It is possible to set a limit for the control signal generated by the integral part of the controller (anti-windup).

Influence of the ***Integral Factor***:

KINT very small	steady-state position deviance is very slowly regulated to zero
KINT larger	faster regulation towards zero steady-state position deviance, larger overshoot
KINT too large	heavy vibrations, instability

Integration Limit: KILIM

The ***Integration limit*** sets a limit for the control signal generated by the integral part of the controller. This helps to prevent the so called “wind-up” problem which typically occurs in applications where the overall control signal (the internal speed-reference) is at a maximum for long periods of time.

Velocity Feed-forward: FFVEL

The ***Velocity Feed-forward*** factor is a scaling factor that is multiplied with the derivative of the setpoint position (the velocity of the setpoint). The result of this operation is added to the overall control signal. This feature is especially useful in applications where there is a good correlation between the control signal and the speed of the motor. This is indeed the case with most applications.

With a setting of 1000 (recommended in most applications) an output of 100% is generated when the setpoint velocity is equal to its maximum setting (specified by the parameters VELMAX and encoder settings.

The scaling of the FFVEL parameter is dependent on the correct setting of the maximum reference as well as VELMAX and encoder settings.

Acceleration/Deceleration Feed-forward: KFFACC / KFFDEC

The ***Acceleration Feed-forward*** factor is multiplied with the 2nd derivative of the setpoint position (the acceleration of the setpoint) and the result is added to the control signal. This feature should be used to compensate for the torque used to accelerate/decelerate the system inertia.

> Scaling of the KFFACC and KFFDEC factors are dependent on the setting of the ***Shortest Ramp Time*** RAMPMIN. You should therefore increase the KFFACC and KFFDEC accordingly when decreasing ***Shortest ramp time*** RAMPMIN and vice versa.

Sampling time for PID Control: TIMER

For particularly slow systems you can slow down the entire control system by entering multiples of 1 ms as the sampling time. However, it is important to remember that such a change influences all the controller parameters!

Therefore, normally you should not deviate from the value of 1 ms.

BANDWIDTH

A ***Bandwidth*** of 1000 means that the set value is being executed 100%, thus Derivative, Proportional and Integral Factors are effective as defined.

But if you are operating a system which could be jeopardized by vibrations, for example, a crane with a heavy load then you can limit the bandwidth in which the PID controller should function. BANDWIDTH of 300 makes a limitation of 30% possible. The build-up of a vibration is thus prevented since the control is only moved to with 30% of the calculated set value.

However, then it is necessary also to use the feed-forward part of the controller in order to achieve the corresponding control.

6.3.4 Optimizing your Controller Settings Step-by-step

6.3.4.1 Setting Controller Behavior

Before you adjust the controller parameters it is important to determine which controller behavior should be achieved.

> The drive elements must never be operated outside of the technical specifications. Thus the maximum acceleration is determined by the 'weakest' drive element.

- 'Stiff' axis: the fastest reaction possible is mainly influenced by the Proportional Factor. You can judge the results on the basis of the velocity graph.
- Damping of vibrations is mainly influenced by the Derivative Factor. The results can best be assessed in the velocity graph.
- Temporary (static) deviations in position are mainly reduced by the Integral factor and can best be judged on the basis of the position graph.

6.3.4.2 The quickest route to optimal control

The following simplified procedure usually leads to acceptable results in most cases:

1. Set the ***Integration Factor*** KINT to zero.
2. Increase the ***Proportional Factor*** KPROP in small increments until a slight vibration is visible in the velocity graph.
3. Now increase the ***Derivative Factor*** KDER thus decreasing the vibrations.
4. Repeat steps 2 and 3 until there the machine no longer tends to vibrate and the velocity curve has been adjusted to the desired controller behavior.

> We recommend only changing one value at a time and subsequently determining the improvement using tests from within the oscilloscope.

6.3.4.3 Ten steps for optimum control for MACS controllers

The following procedure will optimize the position controller settings for most applications:

1. Make sure that you have specified correct values for VELMAX, RAMPMIN and encoder settings. If you change these settings again at a later point you may need to optimize the controller again.
2. Set the parameter POSERR to a very high value (e.g. 1000000) to avoid getting position errors (Error 8) during the following tests.

   > To avoid damaging the system, make sure that you set the POSERR within the limits of the system since position error monitoring is not active with extremely high values.

3. Optimize the ***Velocity Feed-forward*** control:
   Execute a **Testrun** with KPROP=0, KDER30=0, KINT=0, KFFACC=0, KFFDEC=0, and FFVEL =100.
   View the velocity profiles: If the Actual Velocity profile is *lower* than the Commanded Velocity profile, increase FFVEL. Of course if the Actual Velocity profile is *higher* than the specified Commanded Velocity you should decrease FFVEL.
   Run successive test runs until the two velocity profiles shown in the **Testrun** graph have the same maximum value.
   FFVEL is now optimized, save the current value.
4. In systems with large inertia and/or rapid changes in the reference velocity it is a good idea to use and optimize the ***Acceleration Feed-forward*** control (make sure the inertial load is connected when optimizing this parameter):
   Execute a **Testrun** with KPROP=0, KDER=0, KINT=0, KFFACC=0, KFFDEC=0, and FFVEL at the optimized value found above. Use the highest possible acceleration setting. If RAMPMIN is adjusted properly an acceleration value of 100 and a deceleration value of 100 should be sufficient. Start out with a low setting of KFFACC/KFFDEC approx. 2.
   View the velocity profiles: If during acceleration the actual velocity is constantly lower than the reference velocity profile, then set a higher value of KFFACC/KFFDEC and **Start** the **Testrun** again.
   Run successive test runs until the two velocity profiles shown in the **Testrun** graph have similar ramp-up

and ramp-down curves.
KFFACC/KFFDEC is now optimized, save the current value.

5. Next step is finding the maximum stable value of the ***Proportional Factor*** in the PID controller:
   Execute a **Testrun** with KPROP=0, KDER=0, KINT=0. Set FFVEL, KFFACC and KFFDEC at the optimized values found above.
   View the velocity profile. If the velocity profile is not oscillating then increase KPROP and repeat the test run.
   Run successive test runs until the actual velocity profile is oscillating mildly.
   Decrease this "mildly" unstable KPROP value to about 70%. Save this new value.
6. In order to dampen the oscillations created by the KPROP-part of the controller, the ***Derivative Factor*** should now be optimized.
   Start a **Testrun** with KINT=0 and KDER=200. Set FFVEL, KFFACC, KFFDEC and KPROP at the optimized values found above.
   Run successive test runs with increasing values of the KDER factor. At first the oscillations will gradually reduce. Stop increasing KDER when the oscillations begin to increase.
   Save the last value of KDER.
7. In any system that requires a zero steady-state error, the ***Integral*** part of the controller must be used. Setting this parameter though is a trade-off between achieving zero steady-state error fast (which is good) and increasing overshoot and oscillations in the system (which is bad).
8. If you are using the ***Integral*** part of the PID controller, remember to reduce the KILIM as much as possible (without losing the ***Integral Factor*** effect of course) in order to reduce oscillations and overshoot as much as possible.
9. Reduce the BANDWIDTH as much as possible. With a properly optimized open-loop control BANDWIDTH could be reduced to as little as 6% or 12% (60 - 120).
10. Set the POSERR parameter back to normal e.g. 20000.

Setting Parameters from application program

The axis parameters must be explicitly set at the beginning of every application program. This can be done by modifying the following code and incorporating it into the application program. The vlaue axis 1. Other axes can be set in a similar way. See Initializing Temporary Parameters for more information on setting parameter values.

```
AXE_PARAM(axis, POSERR)    = 0;
AXE_PARAM(axis, KPROP)     = 0;
AXE_PARAM(axis, KDER)      = 0;
AXE_PARAM(axis, KINT)      = 0;
AXE_PARAM(axis, ILIM)      = 0;
AXE_PARAM(axis, FFVEL)     = 0;
AXE_PARAM(axis, KFFACC)    = 0;
AXE_PARAM(axis, KFFDEC)    = 0;
AXE_PARAM(axis, BANDWIDTH) = 0;
```

Setting Parameters from monitor file

As all parameter are available over SDO (see in group 0x2300) it is possible to use a monitor file for tuning the position controller. Helpful for analysing the result is the Oscilloscope.

6.3.4.4 What to do, if ...?

... there is a tendency towards instability?

In the event of a strong tendency towards instability reduce the parameters ***Proportional*** KPROP and

Derivative Factor KDER again,
or reset the parameter ***Integral Factor*** KINT.

... stationary precision is required?

If stationary precision is required then the parameter ***Integral Factor*** KINT must be increased.

... the tolerated position error is exceeded?

If the test run is constantly interrupted with the message “position error” set the parameter for the ***Maximum Tolerated Position Error*** POSERR - within the tolerable limits - as large as possible.

If the position error occurs during the acceleration phase, then that suggests that the set acceleration can not be achieved under the existing load conditions. Increase the ***Max. Tolerated Position Error*** or determine a maximum acceleration suitable for the entire system.

If position errors do not occur until after the acceleration phase and they can be delayed but not eliminated by increasing the ***Max. Tolerated Position Error***, this suggests that the maximum velocity (RPM) chosen is too high. Determine a maximum velocity suitable for the entire system.

... the maximum acceleration is not achieved?

In general, the technical data for a drive are only valid for a freely rotating axle end. If the drive is carrying a load the maximum acceleration is reduced.

The theoretical maximum acceleration will also not be achieved if, for example, the PID controller output is too small or the amplifier / motor is not sized correctly and therefore does not provide enough energy for peak consumption during acceleration.

6.4 Limited Jerk

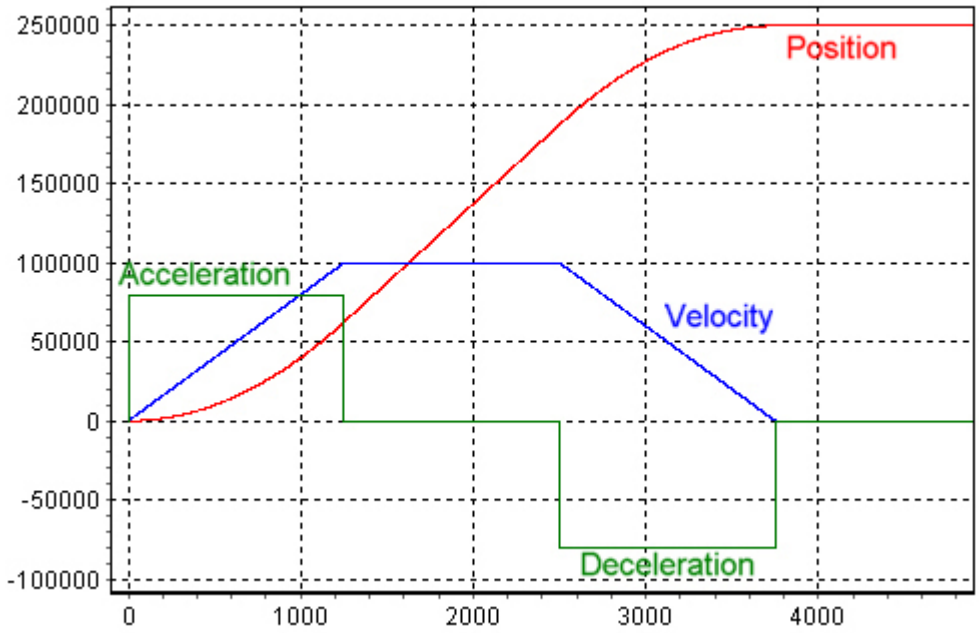
Understanding Limited-Jerk Movements

Limited-jerk movements are similar to normal Trapezoidal movements except that the user may control the "gentleness" of the acceleration and deceleration ramps. This allows the user to limit the "jerk" caused by the "instantaneous" acceleration of a Trapezoidal movement.

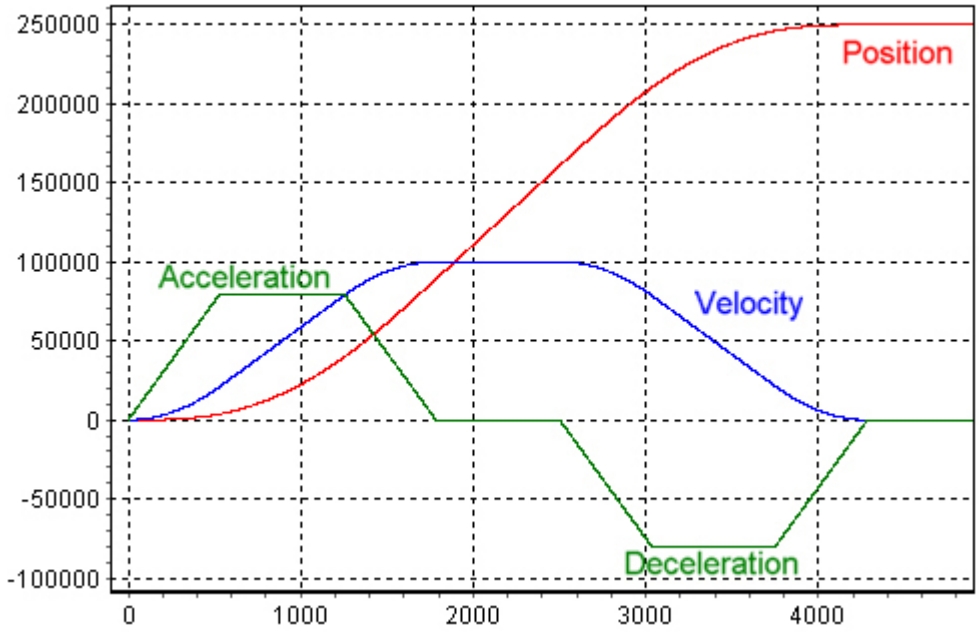
Typical applications where limited jerk is required:

- lift
- movement of heavy loads

As an example, the following chart shows the acceleration, velocity, and position curves for a Trapezoidal movement from one position to another position. The sharp changes in acceleration cause the motor to experience a "jerk" at the beginning and end of each velocity ramp.



The following chart shows the same movement done using a Limited-jerk movement. Note that the acceleration changes are no longer "instantaneous" and that the "corners" of the velocity curve have become rounded. This will result in a smoother motor movement. It will also require slightly longer to get to the desired target position because the motor takes longer to accelerate to maximum acceleration.



The user can control the "gentleness" of the acceleration ramp using 4 parameters:

JERKMIN Acceleration ramp-up constant. This specifies the number of milliseconds required to ramp the acceleration *up* from 0 to maximum acceleration.

JERKMIN2 Acceleration ramp-down constant. This specifies the number of milliseconds required to ramp the acceleration *down* from maximum acceleration to 0 (i.e. normally to constant maximum velocity). If set to 0, this

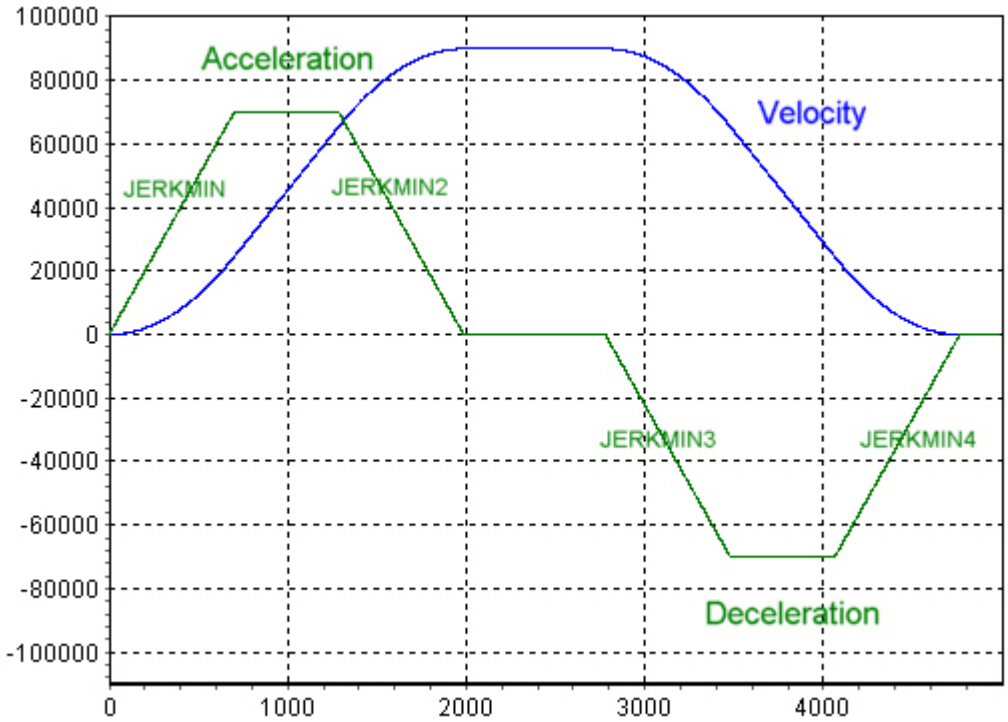
will default to the same value as JERKMIN.

JERKMIN3 Deceleration ramp-up constant. This specifies the number of milliseconds required to ramp the deceleration *up* from 0 to maximum deceleration. If set to 0, this will default to the same value as JERKMIN.

JERKMIN4 Deceleration ramp-down constant. This specifies the number of milliseconds required to ramp the deceleration *down* from maximum deceleration to 0 (i.e. normally to 0 velocity). If set to 0, this will default to the same value as JERMIN.

These constants correspond to the "slopes" of the different portions of the acceleration curve. This is shown in the chart below. When set to larger and larger numbers, the acceleration and/or deceleration will become gentler and gentler as the ramps become longer and longer.

> The acceleration slope determined by the JERKMIN acceleration ramp-up constant will be used *whenever the acceleration is being ramped-up*. It is not used only to ramp-up from 0 to maximum acceleration. Similarly, JERKMIN2 is used whenever the acceleration is being ramped-down, JERKMIN3 is used whenever the deceleration is being ramped-up, and JERKMIN4 is used whenever the deceleration is being ramped-down.



Limited-jerk movements will not normally exceed the velocity and acceleration limits set for the controller (e.g. limits set by the Vel, Acc, Dec, etc., commands). In the above chart, these limits can be recognized by the "plateaus" in the acceleration curve. If the current velocity and/or acceleration are outside these limits when the Limited-jerk movement is started, then the movement is accelerated or decelerated as necessary to bring the movement within the set limits.

It is important to understand that for Limited-jerk movements, "acceleration" is defined as "speeding up" in *either* direction (i.e. either forwards or backwards) and "deceleration" is defined as "slowing down" in either direction. A result of this is that maximum velocity, maximum acceleration, maximum deceleration, and the four JERKMIN values are all *independent of the direction of movement*. This can have important consequences when a Limited-jerk movement must "reverse" the direction of the motor, particularly when maximum deceleration differs from maximum acceleration. In this case, the Limited-jerk movement will ensure that the *deceleration* ramp flows smoothly into an *acceleration* ramp at exactly 0 velocity when the direction changes and without exceeding either

the deceleration or acceleration limits.

A Limited-jerk movement can be used in three different situations:

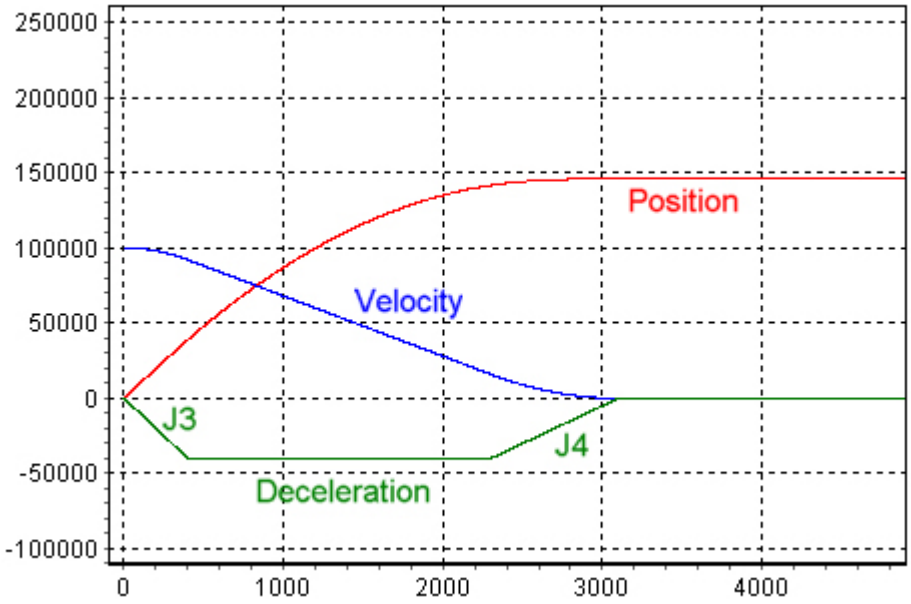
1. Stopping from the current velocity and acceleration (where the final position is not important).
2. Changing from the current velocity and acceleration to a specified constant velocity (where the positions are not important).
3. Moving from the current position (and the current velocity and acceleration) and stopping at a specified position.

6.4.1 Examples of Limited-Jerk Movements

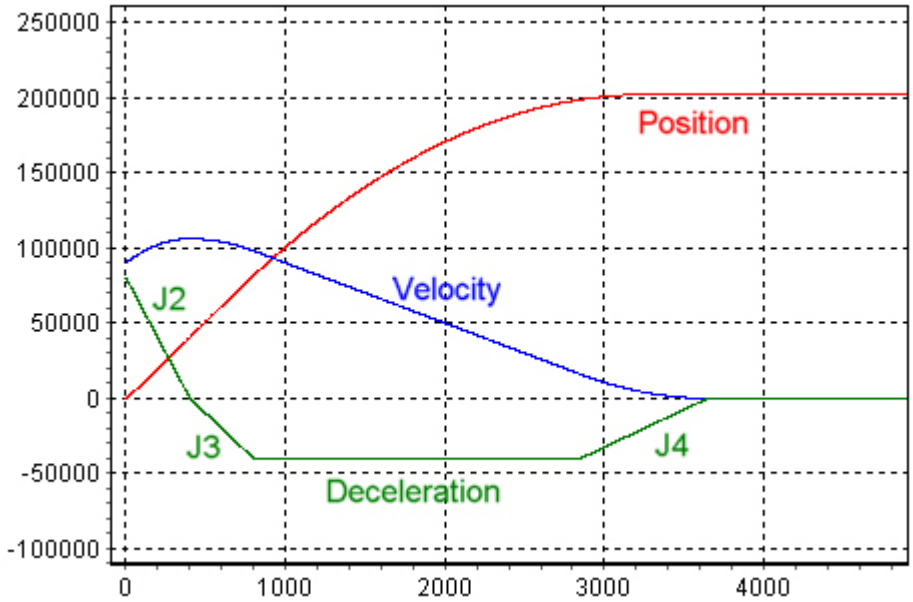
In the following examples, maximum acceleration has been set to a value higher than maximum deceleration so the motor can speed up faster than it can slow down. As well, JERKMIN has been set smaller than JERKMIN2, JERKMIN2 smaller than JERKMIN3, and JERKMIN3 smaller than JERKMIN4. This was done so that the various segments of the curves can be visually distinguished in the charts. The four JERKMIN values have been labeled simply J1, J2, J3, and J4.

Stopping

The chart below shows a stopping movement that begins from a positive constant velocity. The curve consists of a deceleration ramp-up segment (using JERKMIN3), followed by a constant deceleration segment (at maximum deceleration), and finally a deceleration ramp-down segment to 0 velocity (using JERKMIN4).

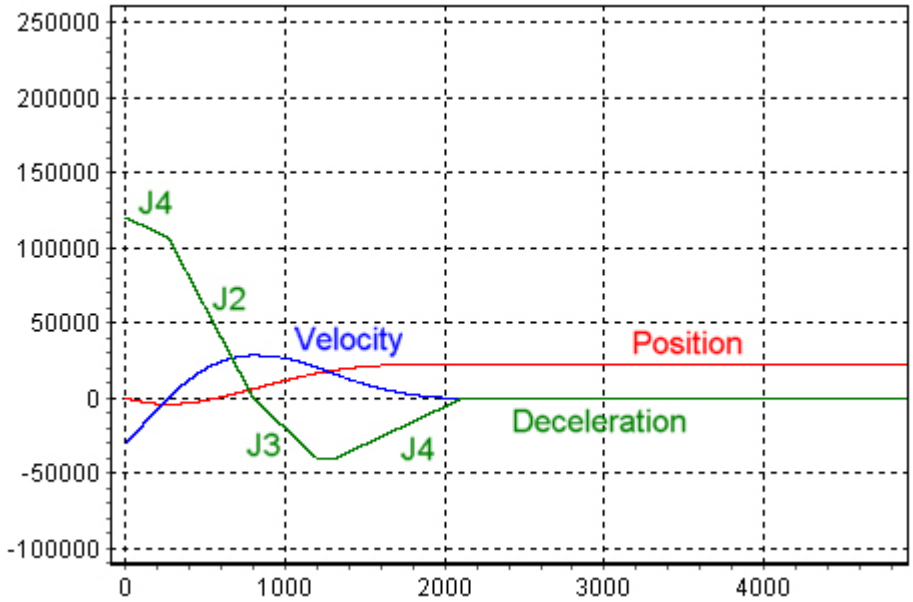


The chart below shows a stopping movement that begins from a positive velocity and a positive acceleration. Since the initial acceleration is positive, the curve must start with an acceleration ramp-down segment to 0 acceleration (using JERKMIN2). This will then be followed by a deceleration ramp-up segment (using JERKMIN3), a constant deceleration segment, and a deceleration ramp-down segment to 0 velocity (using JERKMIN4).



The chart below shows a stopping movement that begins from a *negative* velocity and a very high deceleration. (It is a deceleration because the speed is slowing down.) However, because the initial deceleration is so high, the motor is unable to stop without overshooting 0 velocity and 'coming back'.

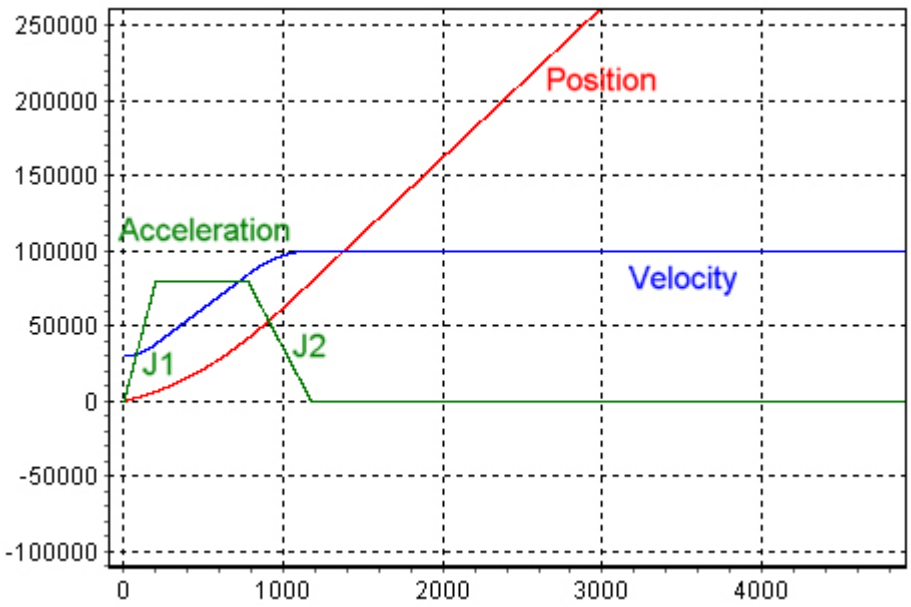
So the curve starts with a deceleration ramp-down segment (using JERKMIN4) to slow the deceleration as much as possible before reaching 0 velocity. At 0 velocity, the 'deceleration' becomes an 'acceleration' because the direction has changed. Hence, the curve continues by ramping-down the *acceleration* (using JERKMIN2) until it gets to 0 acceleration. The motor is now at a constant positive velocity and so the curve will finish in the normal way with a deceleration ramp-up segment (using JERKMIN3), a constant deceleration segment (very short in this example), and a deceleration ramp-down segment to 0 velocity (using JERKMIN4).



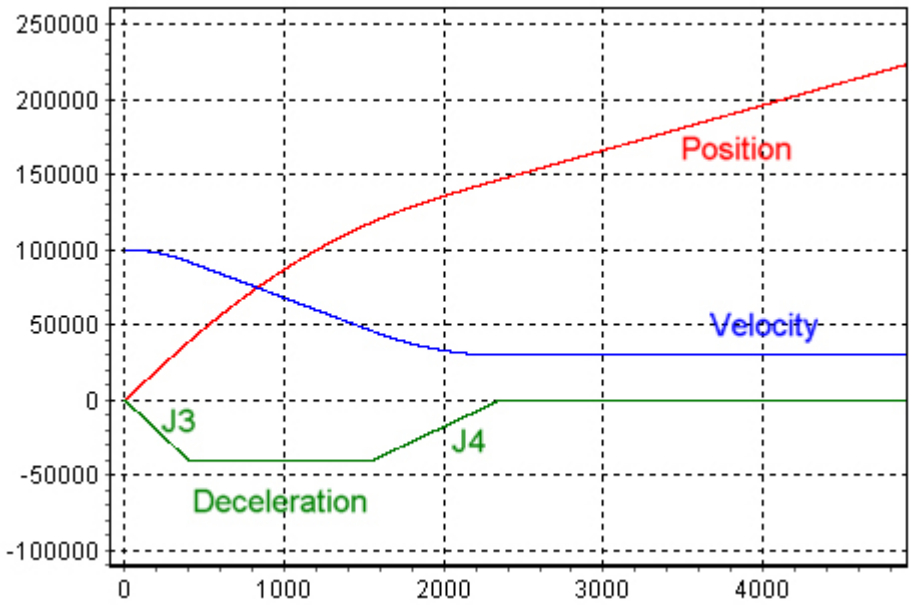
Changing to a Constant Velocity

The chart below shows a movement that begins from a positive constant velocity and increases its speed to a

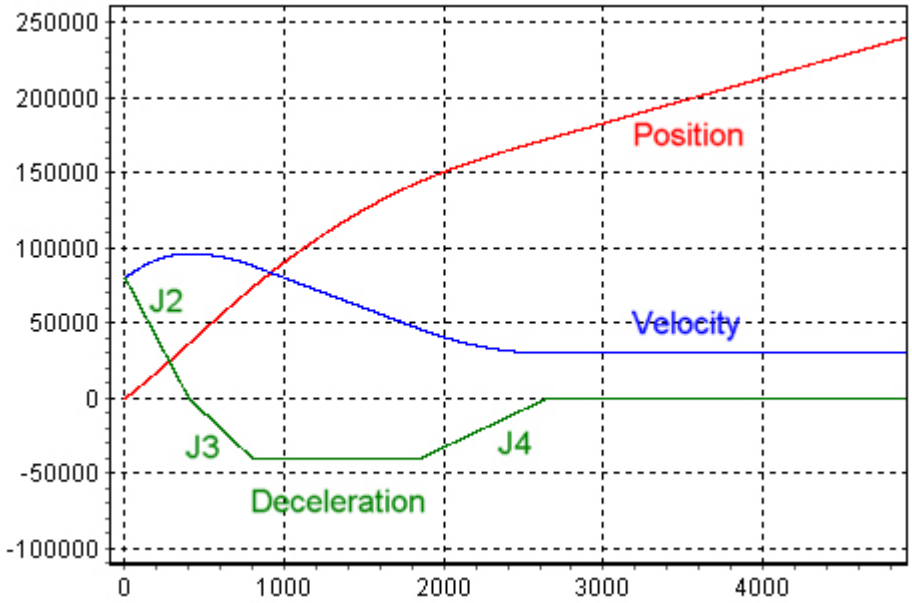
higher positive constant velocity. This curve consists of an acceleration ramp-up segment (using JERKMIN), followed by a constant acceleration segment (at maximum acceleration), and finally an acceleration ramp-down segment to constant velocity (using JERKMIN2). Note that the deceleration JERKMIN3 and JERKMIN4 values are not used because there is never any deceleration.



The chart below shows a movement that begins from a high positive constant velocity and decreases its speed to a lower positive constant velocity. This curve consists of a deceleration ramp-up segment (using JERKMIN3), followed by a constant deceleration segment (at maximum deceleration), and finally a deceleration ramp-down segment to constant velocity (using JERKMIN4). Note that the acceleration JERKMIN and JERKMIN2 values are not used because there is never any acceleration.

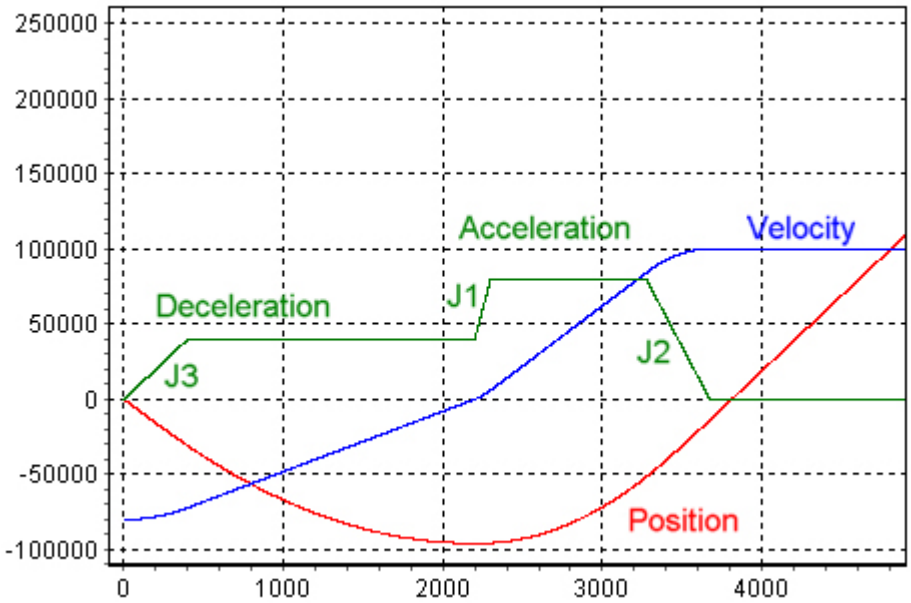


The chart below is similar to the previous example except that it begins with a positive acceleration. In this case, the curve must start with an acceleration ramp-down segment (using JERKMIN2). Once the acceleration reaches 0, then the curve can continue as before.



The chart below shows a movement that begins with a *negative* constant velocity and changes direction to a *positive* constant velocity. The curve must start by slowing down the speed so that it can 'turn around'. Hence the curve begins with a deceleration ramp-up segment (using JERKMIN3) until it reaches maximum deceleration.

The deceleration continues at maximum deceleration until it reaches 0 velocity. Note that there is no deceleration ramp-down segment because the movement is not stopping. At exactly 0 velocity, the direction reverses and the movement is now *accelerating* in the other direction. But since maximum acceleration is higher than maximum deceleration for this example, the curve is now able to include an acceleration ramp-up segment (using JERKMIN). The curve finishes in the normal way with a constant acceleration segment and an acceleration ramp-down segment to constant velocity (using JERKMIN2).

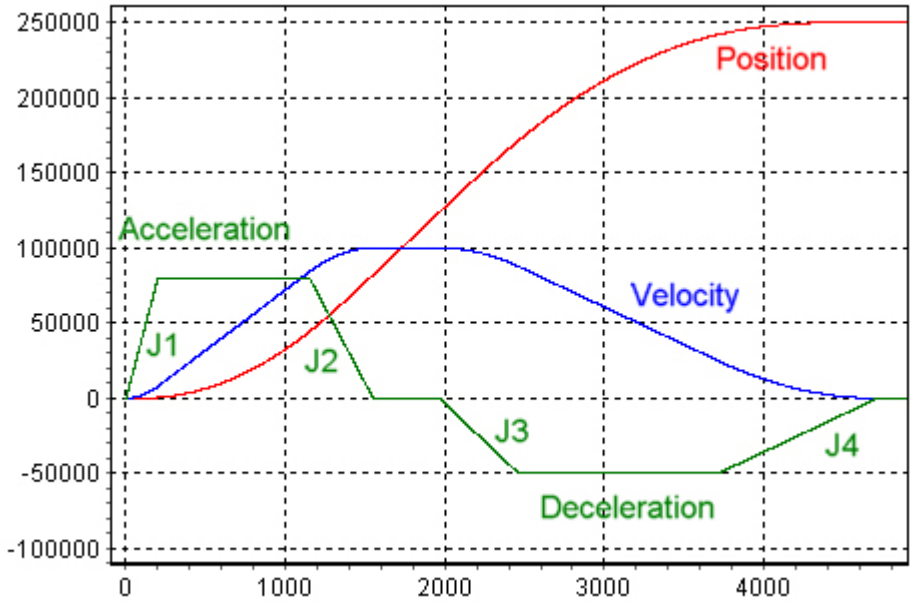


Moving to a Specified Position

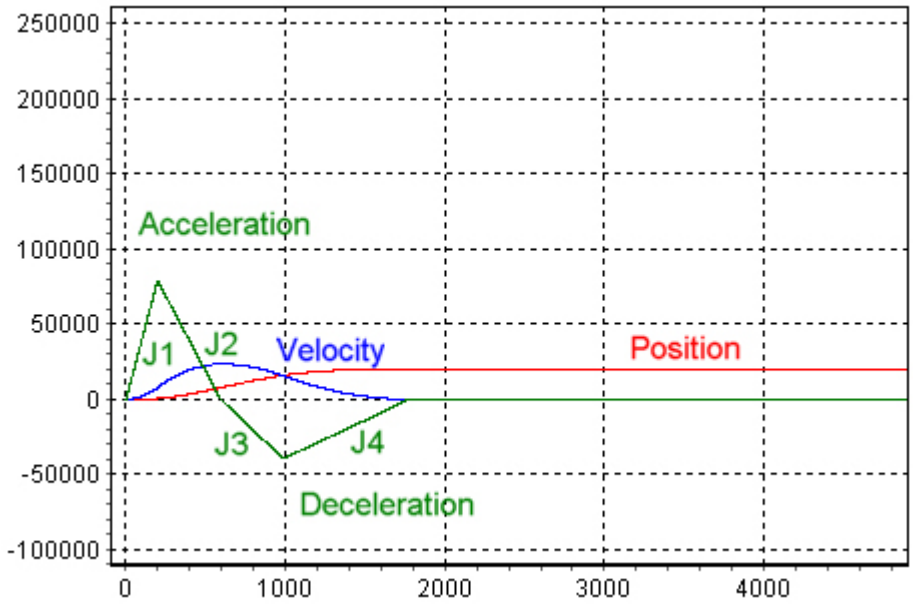
The chart below shows a 'normal' movement from being stopped at one position and then moving forward to

stop at another position. The curve starts with an acceleration ramp to maximum velocity. This portion of the curve will be similar to the first of the examples shown in "*Changing to a Constant Velocity*". The curve is simply changing to a constant velocity where the constant velocity is the maximum velocity.

Hence, the curve will consist of an acceleration ramp-up segment (using JERKMIN), a constant acceleration segment at maximum acceleration, and then an acceleration ramp-down segment to maximum velocity (using JERKMIN2). The movement then proceeds at maximum velocity until it needs to start the deceleration ramp that will stop the movement at the desired position. The deceleration ramp is identical to the first of the examples shown in "Stopping". The curve will consist of a deceleration ramp-up segment (JERKMIN3), followed by a constant deceleration segment (at maximum deceleration), and finally a deceleration ramp-down segment to 0 velocity (using JERKMIN4) stopping at the desired position.

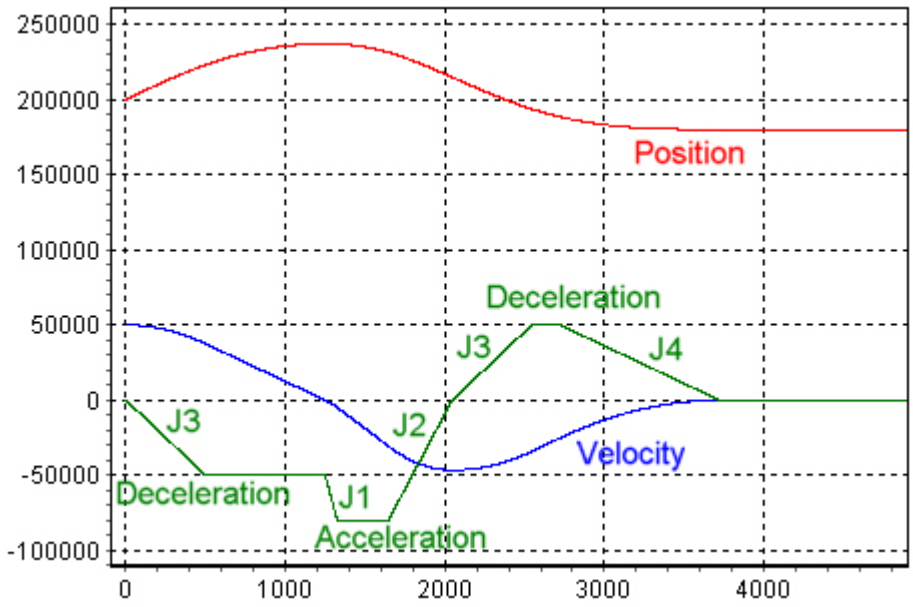


The chart below shows a typical 'short' movement where maximum velocity cannot be reached. In this case, the curve ramps-up the acceleration (using JERKMIN) for as long as possible. This may or may not reach maximum acceleration, depending on how far away the target position is. There will then be an acceleration ramp-down (using JERKMIN2), followed immediately by a deceleration ramp-up (using JERKMIN3). Again, depending on the target position, there may or may not be a constant deceleration segment. The curve ends with a deceleration ramp-down to 0 velocity at the target position.



The chart below shows an example where the motor is initially moving in the 'wrong' direction and must turn around and 'go back' to the target position. Since it must "turn around", the curve starts with a deceleration ramp-up segment (using JERKMIN3) to maximum deceleration.

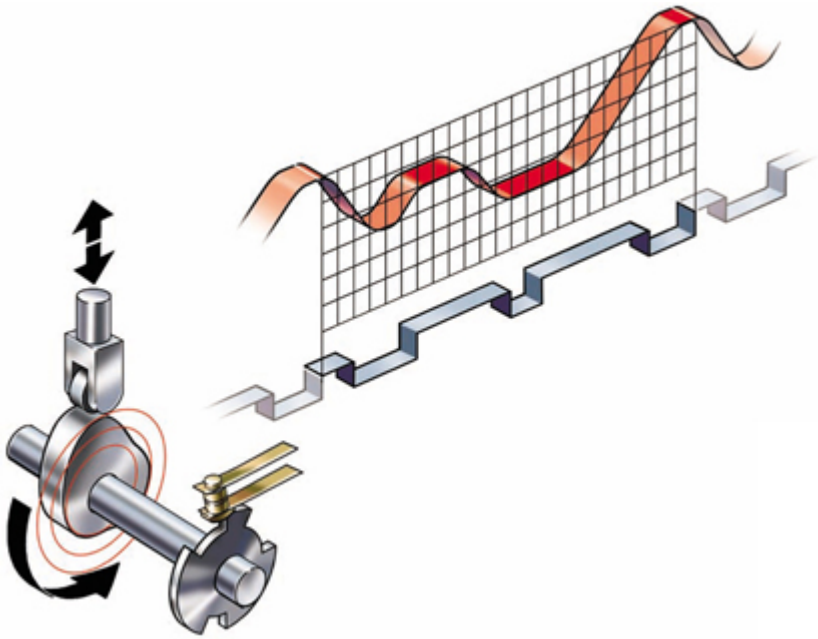
This will slow the speed down until it turns around. Deceleration continues at maximum deceleration until 0 velocity is reached and the direction reverses. At exactly this point, the motor is now 'speeding up' but in the *other* direction. From this point, the curve is similar to a normal movement to a target position, except that the whole curve is inverted because the direction has changed. The curve will have an acceleration ramp-up segment (going backwards), it may or may not have a constant acceleration segment, it will have an acceleration ramp-down segment, it may or may not have a constant velocity segment, it will have a deceleration ramp-up segment, it may or may not have a constant deceleration segment, and finally it will have a deceleration ramp-down to the target position.



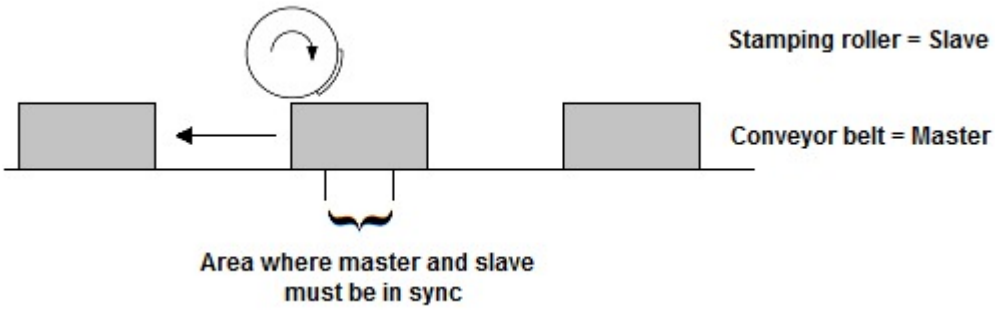
6.5 CAM Curves

Many motion control applications require one motion to be synchronized with a second "master" motion. The "slave" will follow the "master" but often in a complex manner. The motions are also often cyclic. In a mechanical system, this can be implemented using camshafts as is shown in the diagram below on the left. The curve of the cam determines the motion of the synchronized slave. In the controller, the same synchronized motion is reproduced electronically using a "CAM curve" similar to that shown in red on the right. As the master moves along the X axis, the slave follows the red curve on the Y axis.

A related requirement is for a synchronized state rather than a synchronized motion. The mechanical version of this is the cam disc; the electronic version is the "CAM Box". This is also shown in the diagram below.

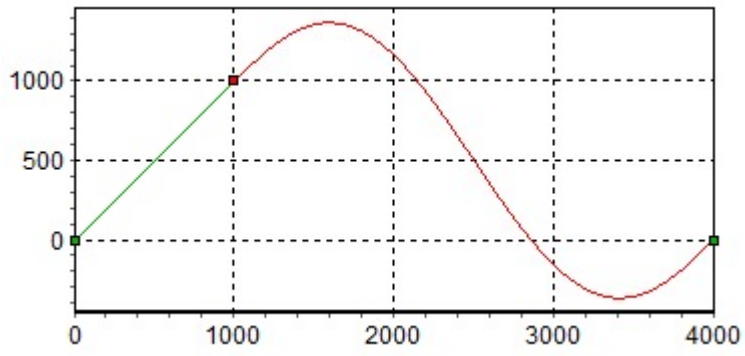


A simplified "real-world" example of where an application might encounter a CAM curve, is shown below. In this example, a stamping roller is to stamp an inscription on cardboard boxes. The stamping roller is at a fixed position and the cardboard boxes are transported on a conveyor belt past the roller. During the printing process, the stamping roller and the cardboard box *must* be synchronized to produce a clear inscription. At other times, the stamping roller does not need to be synchronized with the boxes or belt. Here, the conveyor belt is the "master" and the stamping roller is the "slave". The movement of the roller is linked to the movement of the belt; if the belt speeds up or slows down, the roller must follow so that its position is always correct with respect to the boxes. This synchronization of the roller to the belt is done using a CAM curve.



The CAM curve for the above example, might will look similar to illustration show below. The green portion of the curve corresponds to the printing process where the slave is synchronized and moves at the same speed as the

master. The red portion of the curve corresponds to the time interval between printing where the slave "rewinds" and gets ready for the next box.



The ApossIDE allows both CAM Curves and CAM Boxes to be implemented very easily. The ApossIDE also allows considerable flexibility in how the CAM curve is to be incorporated into an actual real-world application. For example, synchonization can be performed if products are transported irregularly on a conveyor belt and the timing changes. Or if slippage occurs or compensation must be made for the accumulation of systematic errors. The ApossIDE allows behaviours to be defined where a non-synchronized slave must become synchronized by "engaging" a CAM curve and the opposite where a slave "disengages". The examples in the following sections contain simple applications that illustrate how to use CAM curves in these various situations.

The CAM curve itself is implemented within The ApossIDE using a set of user-defined "fixed points" (i.e. points through which the curve *must* pass). Spline interpolation is then used between the fixed points to produce a smooth curve that is optimized to minimize torque and that consumes a minimum of memory on the controller. Since the curves are usually cyclic, the splines are calculated in such a way that the beginning of the curve matches the end of the curve. Also included within the CAM curve data are the various parameters that control the other behaviours such as re-synchronization, enganging and disengaging, etc. The entire set of CAM curve data is then stored in a normal DIM array.

The ApossIDE CAM Editor tool gives the user complete control over creating and modifying CAM curves.

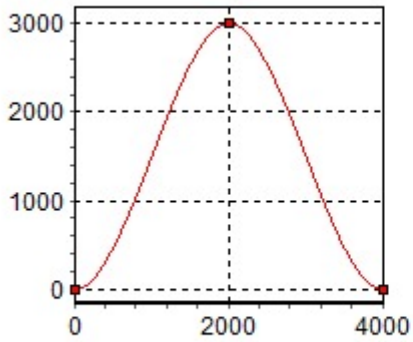
6.5.1 CAM Curve Behaviour

CAM curves created by the CAM Editor are static curves; they are created by the editor, downloaded to the controller in DIM arrays, and then used by the running application program without changing them. There are two general classes of CAM curves:

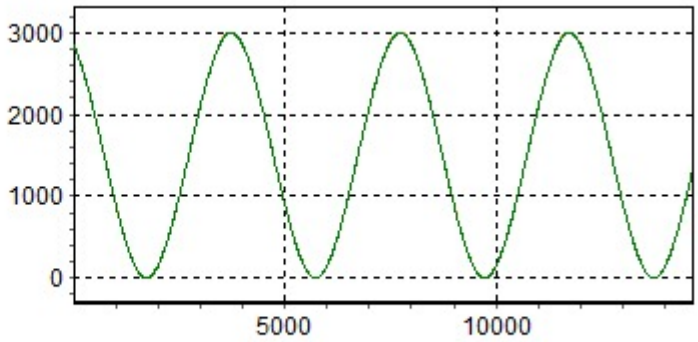
Cyclic CAM Curves

This class includes all curve types available in the CAM Editor (see CAM Curve Tab) except for the two "Specify" curve types.

When a cyclic curve is active and the master runs off the end of the curve, then the master is automatically repositioned to the start of the curve again. This results in the slave repeating the same curve over and over again. Note that in order for the slave to move smoothly as the master position is reset from the end of the curve back to the start of the curve, it is critical that the velocity and acceleration at the start of the curve match those at the end of the curve. For cyclic curves, the CAM Editor will ensure that this is true. The left illustration below shows a simple CAM curve with three fixed points. The right illustration below shows a chart with the corresponding slave positions as the master cycles several times.

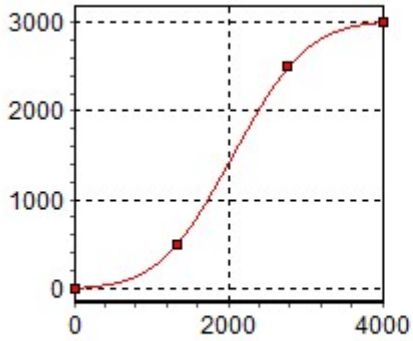


CAM Curve

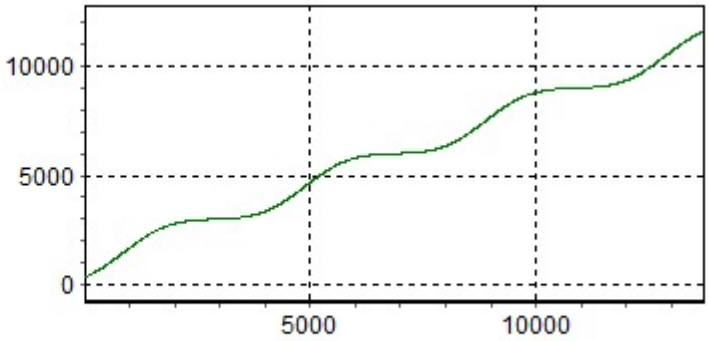


Slave Position

If the slave end position of a cyclic CAM curve matches the slave start position (as above), then the slave will cycle through exactly the same set of positions. However, this is not necessary. If the end position is offset from the start position, then the slave will repeat the same basic shape of movement except that the position at the start of each cycle will be offset from the position at the start of the previous cycle. The illustrations below show a simple CAM curve with an offset and the corresponding slave positions through several cycles.



CAM Curve

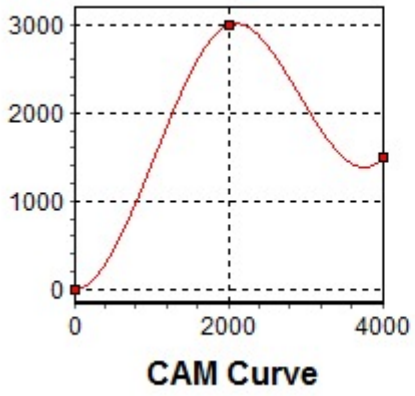


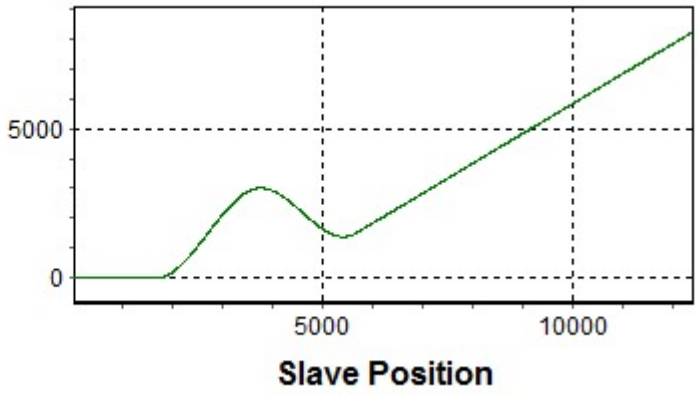
Slave Position

Non-cyclic CAM Curves

This class includes the two "Specify" curve types (see CAM Curve Tab).

When a non-cyclic curve is active and the master runs off the end of the curve, then the slave will continue indefinitely at 0 acceleration and at the velocity it had at the end of the curve. The left illustration below shows a simple non-cyclic CAM curve with a starting velocity of 0 and an ending velocity of 1 uu/mu. The right illustration below shows a chart with the corresponding slave positions. Note that in this example, the slave synchronizes at approximately 2000, runs through the CAM curve until approximately 6000, and then continues along a straight line at the ending velocity beyond 6000.





Any application that uses non-cyclic curves is expected to recognize when the curve is complete and initiate a new movement rather than simply allow the slave to "run away". Generally, this is done using Dynamic CAM Curves.

⚠ When the master runs off the end of a non-cyclic curve, the controller will *overwrite the actual fixed point values in the DIM array containing the CAM curve*. This means the DIM array cannot be used again for a subsequent CAM movement. If the application requires the orginal CAM curve again, then there are two possibilities:

1. Save the original CAM curve in one DIM array and copy it to a second "temporary" DIM array before using it. For example:

   ```
   dim long camcurve[55], tempcurve[55];
   ...
   arraycpy(tempcurve, camcurve);
   SyncSetCurve(0, tempcurve);
   ```

2. Explicitly reset the fixed point count and the fixed point values in the DIM array before executing the SyncSetCurve command again. This may or may not be feasible, depending on the application and the complexity of the curve. Please see Modifying the CAM Curve Array.

6.5.2 CAM Curve Scaling and Positioning

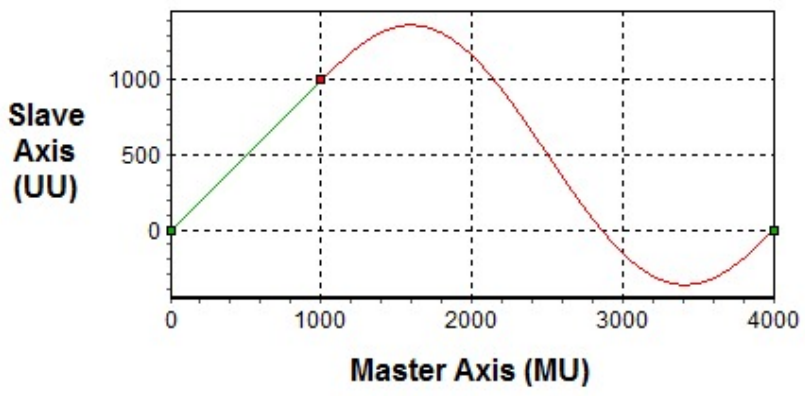
In order to use CAM curves, two important issues must be addressed: Scaling of the CAM curve and positioning of the CAM curve relative to the actual master and slave positions.

Scaling

CAM curve diagrams displayed by the ApossIDE always show curve master coordinates on the horizontal axis and curve slave coordinates on the vertical axis. Master curve coordinates are always measured in "master units (MU)" with coordinates increasing towards the right. Slave curve coordinates are always measured in "user units (UU)" with coordinates increasing upwards. These user units are the same user units that are described in How to Set Up Resolution and Gear Parameters. CAM curves have a restriction: the master cycle length of the curve must be an integer and the intermediate fixed points along the curve must fall on an integer grid. Reducing the size of the master and user units will increase the resolution with which the shape of the curve may be defined.

In general, CAM curves are designed to be followed in the forward direction (i.e. with curve master coordinates increasing). While curves may also be followed in the backward direction, this is not recommended since support for Dynamic CAM Curves is not available. If the "natural" master axis movement is backwards, then the CAM curve should still be designed as if this were the forward direction.

The orientation of the master axis should then be reversed. This is described below.



Once the size of master and user units have been determined, then scaling factors must be provided to convert between these units and actual axis units (i.e. quadcounts). For user units, this is done using the parameters described in How to Set Up Resolution and Gear Parameters. For master units, this is done using the following parameters:

SYNCFACTS	Conversion factor numerator.
SYNCFACTM	Conversion factor denominator.

Parameters SYNCFACTS and SYNCFACTM taken together define the conversion factor from master encoder quadcounts to the master units used by the CAM curve. The actual conversion factor is calculated as the fraction SYNCFACTS/SYNCFACTM (i.e. 1 mu = (SYNCFACTS/SYNFACTM) × 1 master qc). This allows integral SYNCFACTS and SYNFACTM values to be used to specify fractional conversion factors to as high a precision as is required. For example, if the exact conversion is 500.5 mu/qc, then SYNCFACTS could be specified as 1001 and SYNFACTM would be 2.

> Although the term "master encoder quadcounts" is used here, the actual master input signal may have no relationship to encoders or quadcounts. For example, the master signal could come form some completely external source that simply provides "increments" or "ticks" for synchronization. The conversion factor will then be used to convert from "master ticks" to the master units defined for the CAM curve.

Example:

- We would like to convert slave quadcounts to user units (mm) so that we can work with target positions, etc., in millimeters.
- Slave encoder resolution is **2000 qc/rev**. The encoder is directly mounted on the motor shaft.
- The slave has a gear box with **3:2** reduction.
- The linear axis of the slave has a pitch of **2 mm/rev**.
- The CAM curve representing one complete "processing cycle" will be chosen to be 4000 master units long (as in the above diagram). This length is somewhat arbitrary, but must be at least long enough to provide adaquate resolution to the shape of the curve. However, it is often easier when designing the shape of the curve, to have the master units correspond to some appropriate physical measurement (e.g. mms).
- Exactly 3 processing cycles will be completed each time the master moves 10000 quadcounts.

POSENCQC = 2000 [qc]

POSENCREV = 1

POSFACT_Z = 3

POSFACT_N = 2

FEEDDIST = 2 [mm]

FEEDREV = 1

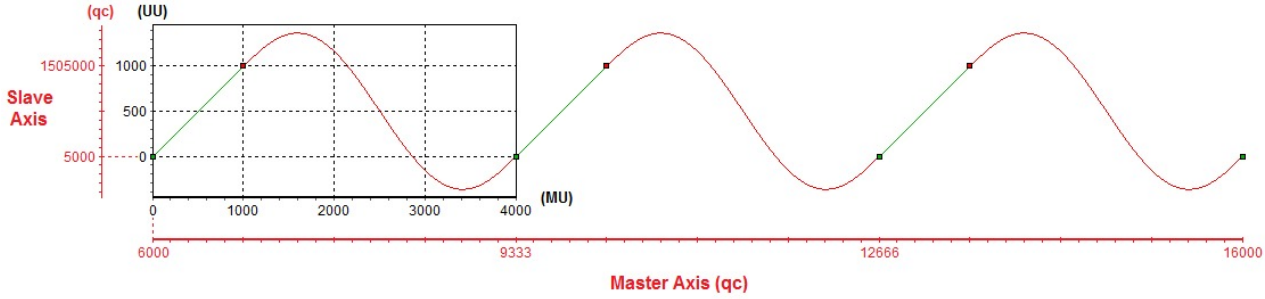
SYNCFACTM = 10000 [qc]

SYNCFACTS = 3 [cycles] × 4000 [mu/cycle] = 12000 [mu]

The slave will complete 3 cycles of 4000 master units each time the master moves 10000 quadcounts. Note that since the final factor is SYNCFACTS/SYNCFACTM, the parameters could also be specified as SYNCFACTS = 6 and SYNCFACTM = 5. This would also give the correct conversion factor. Note that the slave conversion factor in this example is 1 uu = 1500 qc (see How to Set Up Resolution and Gear Parameters).

Positioning

CAM curve coordinates themselves are relative to the (0,0) origin at the bottom left corner of the curve diagram. The scaling factors convert the master and slave units into master and slave quadcounts. The next step in using CAM curves is to position the curve somehow in the "real world". This means determining the actual master and slave quadcounts that correspond to the (0,0) origin of the CAM curve. After this, the controller knows exactly where the slave should be at each position of the master and it is able to maintain synchronization of the slave with the master as the master moves. For example, if a master position of 6000 quadcounts and a slave position of 5000 quadcounts matches the start of the CAM curve cycle shown above, then the synchonization is as follows:



The position of a CAM curve is initialized when the curve is first "activated". Typically, this is done when the SyncSetCurve command is executed to set the CAM curve or the SyncCam, SyncCamMarkerMaster, or SyncCamMarkerSlave command is executed to enter curve synchronization mode. The position is initialized using one of the following methods:

1. **User Specified Position:** The slave axis is manually moved to a position that corresponds to a known position on the curve (often the start of the curve). The current master and slave curve positions are then adjusted to match this position on the curve. This is usually the simplest method of initializing the curve position.
2. **Current Axis Position:** The curve position is adjusted to match the current actual position of the master and slave axes. Usage of this method is not recommended since the algorithm for determining the initial position may not always produce the position expected by the user.
3. **Marker Position:** The slave axis is manually moved to a known position on the curve (often the start of the curve) where it stops and waits while the master axis continues to move. At some point in time as the master moves, an external event (such as an object arriving on a conveyor belt) triggers a signal to the controller. The controller then aligns the curve with that point.

User Specified Position

With this method of initializing the curve position, the master and slave axes are either manually, or by some other means, put into a known initial position on the curve. Typically,

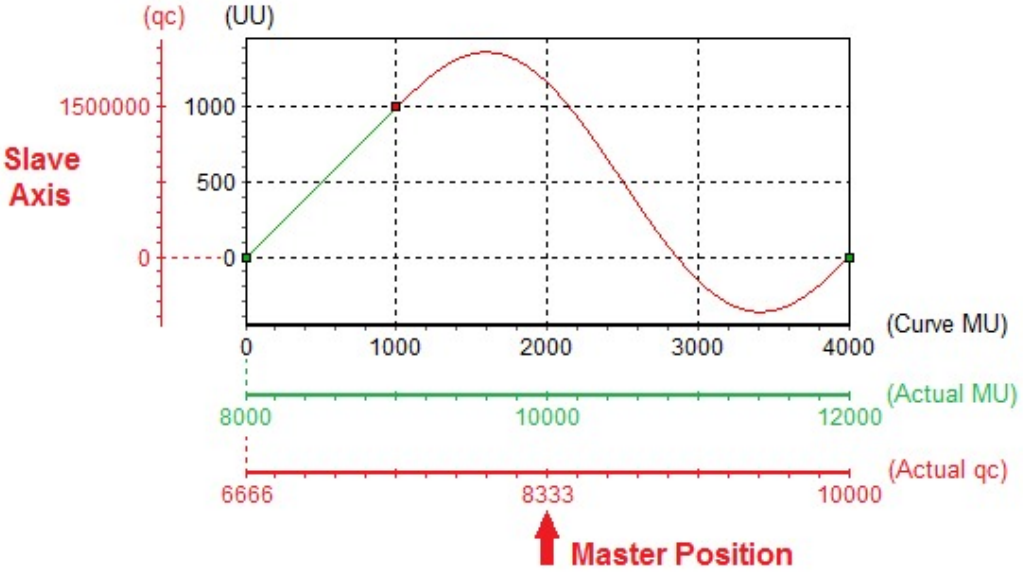
this is some type of default or home position of the system and it will often match the start position of the curve. However, the actual master and slave axes will likely be at some non-0 position. The correspondence between the master and slave axis positions and the curve position is then set using the SyncDefOrigin command. See SyncDefOrigin for a detailed description of exactly how the definition is done. After this, the SyncSetCurve command should be executed. The new curve will then be positioned as specified by the SyncDefOrigin command.

Current Axis Position

The initial position is based only on the current axes positions. The initialization is done assuming that an actual master position of 0 corresponds to the start of a curve and then removing complete master cycle lengths of the curve. For example, if the actual master position (Mapos) corresponds to 10000 mu (i.e. 8333 qc) and the master cycle length of the curve is 4000 mu, then the start of the curve is assumed to be to at 8000 mu and the current master curve position (PFG_CMASTERCPOS) is initialized to 2000 mu (i.e. half way through the curve).

The initialization of the starting slave position, depends on whether the CAM curve has a 0 slave cycle length or a non-0 slave cycle length. See "Cyclic CAM Curves" in CAM Curve Behaviour for more information about slave cycle lengths.

If the CAM curve has a 0 slave cycle length, then the start position of the curve (PFG_CSSTART) is set to 0 qc, regardless of the current actual slave position. For example, using the above numbers, the synchronization is as follows:



> ⚠ Since the start position of curves with 0 slave cycle length is set to 0, if the actual slave position is *not* 0 when the curve starts to be followed, then then slave will try to correct the position as quickly as possible. In certain cases, this will be done *using the maximum possible torque*. As a result of this behaviour, it is recommended that the "User Specified Position" method (above) be used for these curves.

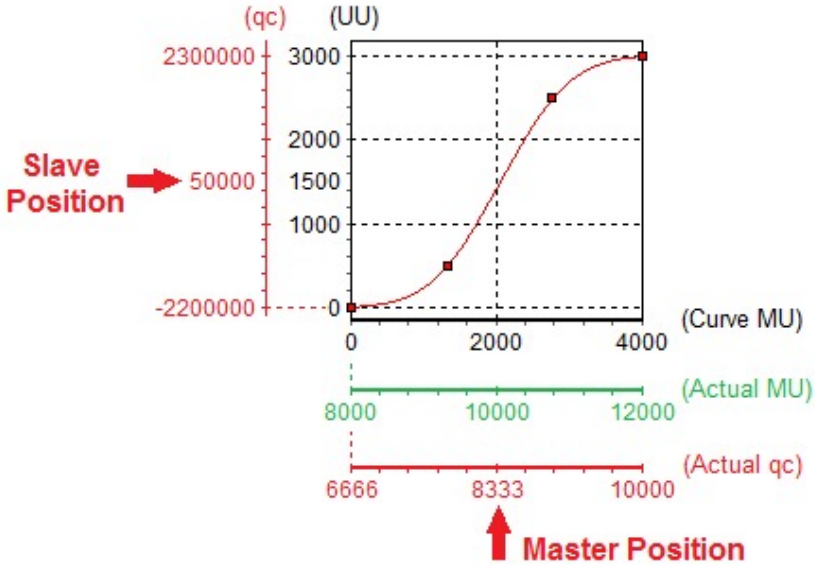
If the CAM curve has a non-0 slave cycle length, then the controller assumes that the master and slave are currently synchronized. It attempts to determine where the slave start position must be in order to have the current actual slave position exactly synchronized with the current master position. This is done as follows:

- Using the current master curve position determined above, the corresponding slave

curve position is calculated. This tells us how far along we are in the cycle.

- This distance is converted to slave quadcounts and subtracted from the current actual slave position. This tells us what the actual slave start quadcount must have been in order to place us at this slave position with this master position.
- The start position of the curve is set to this value.

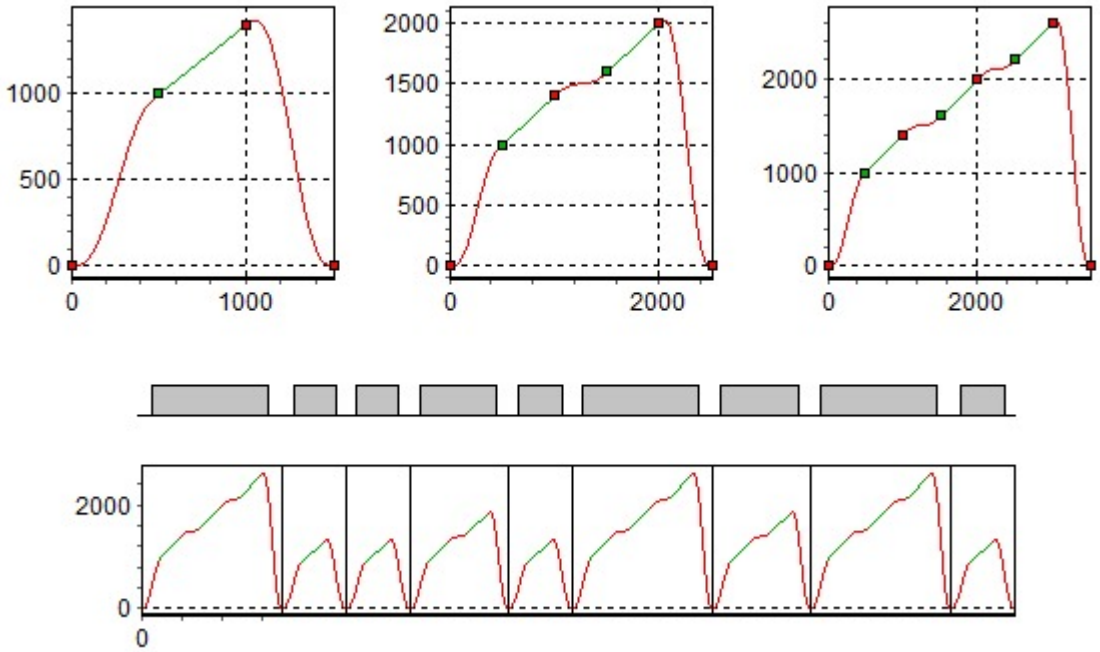
Using the same scaling factors as above, assume that the slave cycle length is 3000 uu (i.e. 4500000 qc) and the current slave position is 50000 qc. Since the master curve position is 2000 mu, the corresponding slave curve position will be 1500 uu. This means 50000 qc corresponds to 1500 uu. The synchronization is as follows:



6.5.3 Dynamic CAM Curves

In addition to static CAM curves that are used over and over without change, the ApossIDE also supports dynamic CAM curves. These are CAM curves that are selected and/or created by the running application program itself in response to events triggered by the target system being controlled. For example, the controller could be following one CAM curve and then, when finished, automatically start a different CAM curve that has been preselected and preloaded from several available CAM curves, based on some prior event that happened. This allows the movement to smoothly flow from one CAM curve into a different CAM curve.

This could look something like the illustration below. Here, boxes of three different lengths are moving along a conveyor belt (middle of illustration) and a CAM curve has been prepared for each length (top). The application then chooses the correct CAM curve for the upcoming box (bottom).



Dynamic CAM curves are only supported by controllers with firmware version 6.4.44 or later. For MCO305 controllers, the firmware version must be 6.6.0 or later.

The SyncSetCurve command is used to select the CAM curve to be followed. When the selected curve is actually started (e.g. with the SyncCamStart command), then the controller will process the CAM curve and create an internal representation of the curve. The controller then follows this internal representation and no longer requires the original CAM curve. This procedure allows the SyncSetCurve command to be executed again, before the active curve ends, to preselect the *next* CAM curve. This new curve is then started as soon as the active curve ends. Note that either cyclic or non-cyclic CAM curves may be used; the important point to remember is that the next curve must be selected before the preceding curve ends.

In order to ensure the smooth transition from one CAM curve to the next, it is important for the CAM curves to be designed so that the starting velocity and acceleration of the following curve matches the ending velocity and acceleration of the preceding curve.

Each time a CAM curve is actually started (including when a cyclic curve is restarted), the PFG_CWRAP parameter is incremented. This can then be used by the application program to determine when it is possible to execute the next SyncSetCurve command. The normal logic is as follows:

1. Use SyncSetCurve to select the first curve.
2. Start the curve (e.g. using SyncCamStart).
3. Wait until PFG_CWRAP changes from 0 to 1. This indicates that the first curve has become active.
4. Use SyncSetCurve to select the next curve. This must be done before the first curves ends.
5. Wait until PFG_CWRAP changes from 1 to 2. This indicates that the first curve has ended and the next curve has become active.
6. Use SyncSetCurve to select the next curve.
7. Wait until PFG_CWRAP changes from 2 to 3.
8. Continue as above.

As an alternative to using a set of predefined CAM curves, the use of dynamic CAM curves also provides the

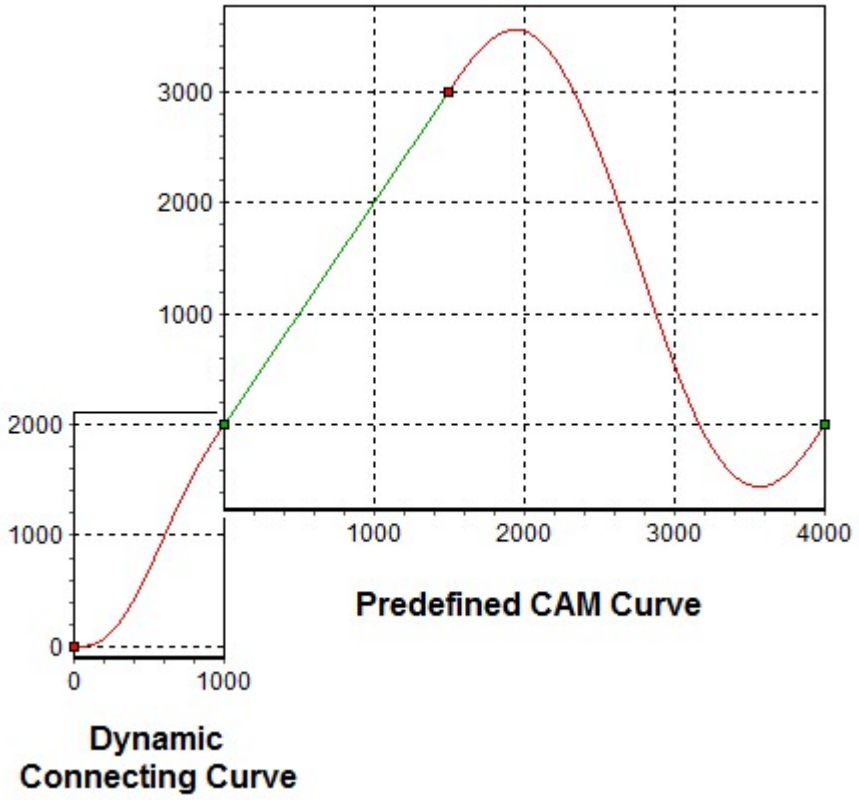
opportunity for application programs to adjust existing CAM curves to suit some property determined during the operation of the target system. For example, boxes moving on a conveyor belt may have arbitrary lengths rather than fixed lengths and that may affect how the boxes are processed. The CAM curve can then be adjusted to account for the box length before the curve is selected by SyncSetCurve. See Modifying the CAM Curve Array for a description of how an application program may modify CAM curves.

6.5.3.1 Dynamic Connecting Curves

Many drive applications have the requirement to smoothly transition from some current known velocity and acceleration state into the start of a predefined CAM curve. The ApossIDE supports this by using special 2-point non-cyclic "dynamic connecting curves". Since the connecting curve is non-cyclic, both the velocity and acceleration at each end of the curve may be specified (see CAM Curve Behaviour). This allows the start of the curve to smoothly match the current state and the end of the curve to smoothly match the start of the predefined curve. The drive application can then follow this connecting curve before automatically switching to the predefined curve. See Dynamic CAM Curves for a description of automatic switching between dynamic curves. Unlike most normal CAM curves that are intended to be repeated and therefore have matching start and end velocities, connecting curves are non-cyclic, are intended to be executed only once, and are normally followed by another CAM curve.

Note that these curves may also be used to transition *out* of predefined curves. In fact, they may be used by a drive application any time that it is necessary to transition from an existing velocity and acceleration to some target velocity and acceleration. There is no requirement that they be used only to transition into or out of a CAM curve.

The illustration below shows how a connecting curve might look. The predefined CAM curve on the right starts with a velocity of 3000/1500 uu/mu and 0 acceleration. In the current state, the master is 1000 mu and the slave is 2000 uu away from the position they need to be in to be synchronized. The slave's current velocity and acceleration are both 0. The curve on the left shows the 2-point non-cyclic connecting curve. Note that it starts with a 0 velocity to match the current state and ends with a velocity of 2/1 uu/mu to match the start of the predefined CAM curve.



Dynamic connecting curves are only supported by controllers with firmware version 6.5.05 or later.

Dynamic connecting curves can be implemented using the following steps:

1. Use the CAM Editor to create a "skeleton" 2-point CAM curve. The position of the second fixed point is not important since the application program will be modifying it. Set the curve type to **Specify start and end velocities** (i.e. curve type 3). Note the array length required for this CAM array. This length can be seen in the CAM Editor toolbar.
2. Download the CAM array to the controller.
3. In the application program, define a DIM array for this CAM array. The array must be at least as long as the length determined in step 1.
4. Before the connecting curve is required, the application program should modify the skeleton DIM array as necessary. This means setting the position of the second fixed point and setting the both the starting and ending velocities and accelerations. See Modifying the CAM Curve Array.
5. Use the SyncSetCurve command in the normal way to start the connecting curve.
6. Use PFG_CWRAP to determine when SyncSetCurve can be used to select the predefined CAM curve (see Dynamic CAM Curves).

Any existing CAM array can be used to create a connecting curve (i.e. if it is not possible to use a predefined "skeleton" array) by overwriting the previous content of the array. However, this will require much more setup in order to configure the array as a connecting curve. It is strongly recommended that a skeleton curve intended for the purpose be used since this ensures that the curve array has been properly initialized.

Modifying a Predefined Skeleton CAM Array

```
#include <SysDef.mh>
```

```
dim long Connect[49];
long mstlen, slvlen;
long v0num, v0den, a0num, a0den, v1num, v1den, a1num, a1den;
...
// Calculate the connecting curve start and end values.
mstlen = ...   // Master length of connecting curve.
slvlen = ...   // Slave length of connecting curve.
v0num = ...    // Velocity numerator at start of curve.
v0den = ...    // Velocity denominator at start of curve.
a0num = ...    // Acceleration numerator at start of curve.
a0den = ...    // Acceleration denominator at start of curve.
v1num = ...    // Velocity numerator at end of curve.
v1den = ...    // Velocity denominator at end of curve.
a1num = ...    // Acceleration numerator at end of curve.
a1den = ...    // Acceleration denominator at end of curve.
// Modify the skeleton array.
CAM_FixpointMaster(Connect,2)       = mstlen;  // Master end coordinate.
CAM_FixpointSlave(Connect,2)        = slvlen;  // Slave end coordinate.
CAM_StartVelocityNum(Connect)       = v0num;   // Start velocity numerator.
CAM_StartVelocityDenom(Connect)     = v0den;   // Start velocity denominator.
CAM_StartAccelerationNum(Connect)   = a0num;   // Start acceleration numerator.
CAM_StartAccelerationDenom(Connect) = a0den;   // Start acceleration
denominator.
CAM_EndVelocityNum(Connect)         = v1num;   // End velocity numerator.
CAM_EndVelocityDenom(Connect)       = v1den;   // End velocity denominator.
CAM_EndAccelerationNum(Connect)     = a1num;   // End acceleration numerator.
CAM_EndAccelerationDenom(Connect)   = a1den;   // End acceleration denominator.
```

6.5.3.2 Immediate Curves

Type 4 is available starting with version 6.5.05. This type is nearly identical to type 3 (CU_GRAD).

The big difference with curves of type CU_GRAD_INTRPT is that they are started immediately when the SyncSetCurve is executed. When this is done, the actual values for velocity and acceleration are used for the calculation. The actual values of the MCPOS and CURVEPOS are subtracted from the end coordinates of the curve before it is calculated (curves must always internally start at 0,0). This guarantees that the original end coordinates are absolute to the start of the interrupted curve.

For example, assume that a curve is running which starts at 0,0 and ends at 2000,2000 (master,slave). Now we define a curve of type CU_GRAD_INTRPT, which starts anywhere and ends at 4000,4000. If this curve is now set by SyncSetCurve at the moment when the original curve passes 1500,1800, for example, then the new curve is calculated in such a manner that it starts at this point (1500,1800) and ends at 4000,4000. To realise this, it uses the velocity and acceleration in the actual point, sets MCPOS and CURVEPOS to 0 and reduces the end coordinates to (4000-1500, 4000-1800) = (2500,2200). It will have the defined velocity (v1) and acceleration (a1) defined in the curve array.

These types of curve are used for processes where the standard curve looks more or less like a straight line (SyncPos / SyncMarker behaviour) and where the poly5 curves are used to align start or stop or restart processes to defined points.

> The responsibility for the correctness of poly5 curve lies with the user / application. The firmware does not do any plausibility test.

To allow readability by CAM Editor, the CurveVersion (index 2) should be 102. Otherwise, the CAM editor will not accept those new curves.

6.5.3.3 Special Cases

Curve type 3 CU_GRAD with SYNCCSTART

Such curves (Poly 5) can now also be started with SyncCamStart. This means the curve starts immediately and does not wait for next marker. (In previous versions, it was only possible to start such a curve with SYNCMSTART 2001.)

If such a curve is started now with SyncCamStart, the user is responsible for the correct setting of the endpoints. Startpoint is 0 which will be internally set to the actual command position. Hence, the curve must be defined from 0 .. endpoint.

Example:

```
G_CFPIdx = StartCurve[9]-1;      // Array index, where fixed point
definition starts
// Fixed Point 1 must be 0/0 because curve must always start at this
position
StartCurve[G_CFPIdx+0]  = 0;    // Master start coordinate
StartCurve[G_CFPIdx+1]  = 0;    // Slave start coordinate
StartCurve[G_CFPIdx+2]  = 1;    // Fixed Point Type = curvepoint
StartCurve[G_CFPIdx+3]  = G_ShingleDistance * 4;      // Master end
coordinate
StartCurve[G_CFPIdx+4]  = G_ShingleDistance * 2;      // Slave end
coordinate
StartCurve[G_CFPIdx+5]  = 1;    // Fixed Point Type = curvepoint
StartCurve[G_CFPIdx+6]  = 0;    // Velocity v0
StartCurve[G_CFPIdx+7]  = 1;    // Divisor v0
StartCurve[G_CFPIdx+8]  = 0;    // Acceleration a0
StartCurve[G_CFPIdx+9]  = 1;    // Divisor a0
StartCurve[G_CFPIdx+10] = 0;    // Jerk j0
StartCurve[G_CFPIdx+11] = 1;    // Divisor j0
StartCurve[G_CFPIdx+12] = 1;    // Velocity v1
StartCurve[G_CFPIdx+13] = 1;    // Divisor v1
StartCurve[G_CFPIdx+14] = 0;    // Acceleration a1
StartCurve[G_CFPIdx+15] = 1;    // Divisor a1
StartCurve[G_CFPIdx+16] = 0;    // Jerk j1
StartCurve[G_CFPIdx+17] = 1;    // Divisor j1

SyncSetCurve(axis, StartCurve);
...
SyncCam(axis, 0);
SyncCamStart(axis, 0);
```

In such a case, when SyncCamStart is executed, the 0 point of the curve is mapped onto the actual command position. Also, the actual velocity is calculated (and start acceleration is set to zero).

Curve type 3 CU_GRAD with SyncCamStart and SyncDefOrigin

In addition to the above mentioned possibility, you have the option to define the endpoints of the curve with absolute values. So the start of such a curve could look like:

```
mendpos = Mipos(axis) + mdistqc;
// apos must be in qc
sendpos = (sdistqc % G_SlaveQcProProdukt) * G_SlaveQcProProdukt +
REG_ACTPOS(axis, REG_ZERO);
SyncDefOrigin(mendpos, axis, sendpos);
// define target position for master and slave (in qc)
SyncCam(axis, 0);
SyncCamStart(axis, 0);
```

In this case, the curve is again started immediately and the actual command position will be the Curve zero position. However, the end position of master and slave are calculated in such a way that the curve will end at the given absolute positions.

Final velocity is given by the curve (1 in our example above) and start velocity is taken from actual values.

Curve types CU_GRAD with stability check

Whenever the new curve types are calculated, a check for extremes within a poly5 is also done if possible. (This only works if start acceleration is 0.) If an extreme within the interval is found, the curve error flag PG_FLAG_CURVE_ERR is set and the error number is stored in the PFG_LastError which allows the user to detect this situation. At the same time, the REG_ACTPOS(0, PFG_CPOLYMAXVEL) and REG_ACTPOS(0, PFG_CPOLYMINVEL) (qc/st) are stored. Writing to REG_ACTPOS(0, PFG_LASTERROR) clears the flag PG_FLAG_CURVE_ERR (Bit 64 <<24) in the AxisStatus.

So a sample could look like:

```
if ((AxisStatus(0))&(64<<24)) { // error bit set
    switch(REG_ACTPOS(0, PFG_LASTERROR)) {
        case 5:     print(" Minimum in interval ",(REG_ACTPOS(0,
PFG_CPOLYMINVEL)*100/128));
                    break;
        case 6:     print(" Maximum in interval ",REG_ACTPOS(0,
PFG_CPOLYMAXVEL)*100/128) );
                    break;
        case 7:     print(" Minimum in interval ",REG_ACTPOS(0,
PFG_CPOLYMINVEL)*100/128) );
                    print(" Maximum in interval ",REG_ACTPOS(0,
PFG_CPOLYMAXVEL)*100/128) );
                    break;
        default:    print(" Other curve error ",REG_ACTPOS(0,
PFG_LASTERROR));
    }
}
```

6.5.4 Modifying the CAM Curve Array

For various reasons, applications may need to modify existing CAM curves in response to some event that occurs while the system is running. This can be done by directly modifying the DIM array containing the CAM curve prior to executing the SyncSetCurve command. Before doing this, please see CAM Array Structure for a detailed description of the structure of CAM curve arrays.

> ⚠ The curve array is used by the internal controller procedures for as long as the curve is running. *Hence, the curve array of a running curve must never be modified.* One solution to this problem, is to have two DIM arrays which are used alternately. So while one curve is active, the next one can be prepared and started. As soon as the new one becomes active, the first curve can be modified again.

The ApossIDE provides the CamArrayIndex command and the "SysDef.mh" include file contains #define statements for several associated offset values. Together, these can make access to the CAM array much easier. See SysDef Defines for an up-to-date list of available macros. The **Help** → **SYSDEF Defines** command can also be used to access this page directly.

> When modifying CAM arrays, please note the following:
> 1. The "index" values in the CAM array header block *must* be correct before the macros may be used.
> 2. If a gradient block exists, then it must immediately follow the last fixed point. There can be no "unused" space between the last fixed point and the gradient block.

Example:

```
#include <sysdef.mi>
DIM camcurve[55]
...
CAM_NumberOfFixpoints(camcurve) = 3    // Point count.
CAM_FixpointMaster(camcurve,2)  = 2000 // Fixed point 2.
CAM_FixpointSlave(camcurve,2)   = 3000 // Fixed point 2.
CAM_FixpointType(camcurve,2)    = 1    // 1 = Curve point.
CAM_FixpointMaster(camcurve,3)  = 4000 // Fixed point 3.
CAM_FixpointSlave(camcurve,3)   = 0    // Fixed point 3.
CAM_FixpointType(camcurve,3)    = 1    // 1 = Curve point.
...
SETCURVE camcurve   // Select the adjusted CAM curve.
```

6.5.5 Quick Start Tutorial for Users

1. Create a new configuration file by either using **File** → **New** or pressing the **CAM Editor** toolbar button. Then press press the **Read from controller** toolbar button to select a controller and the load controller parameters. Alternatively, opening an existing configuration zbc file using **File** → **Open** will automatically start the CAM Editor and load the parameters.

2. Use the **Add CAM array** toolbar button to add a new CAM curve array. Note that **Interpolation** is not necessary except for controllers with very old firmware versions (older than 6.4.44). If **interpolation** is necessary, then define the **Number of Intervals** for a master cycle length. The **Interval Time** in the CAM Curve Tab should not be less than 20 ms.

3. Add at least three **Fixed Points** for master and slave.

4. Add **Start Stop Points** for the engaging and disengaging.

5. Define the **Slave Stop Position** in the CAM Curve Tab.

6. Enter the number of **Cycles/min Master** in the CAM Curve Tab.

7. Enter gearing factors or set user units MU and uu in the Axis Data Tab.

8. Check the velocity and acceleration of the slave with the help of the diagram. Use the Display Tab.

9. Save the parameters in the controller by using the **Write to controller** toolbar button. Alternatively, **File** → **Save** can be used to save the parameters in a configuration zbc file which can later be downloaded to the controller.

6.5.6 Example 1: Basic Synchronization

The following example gives a step-by-step explanation for how to edit the curve for a CAM control application and then how to incorporate it into a control program.

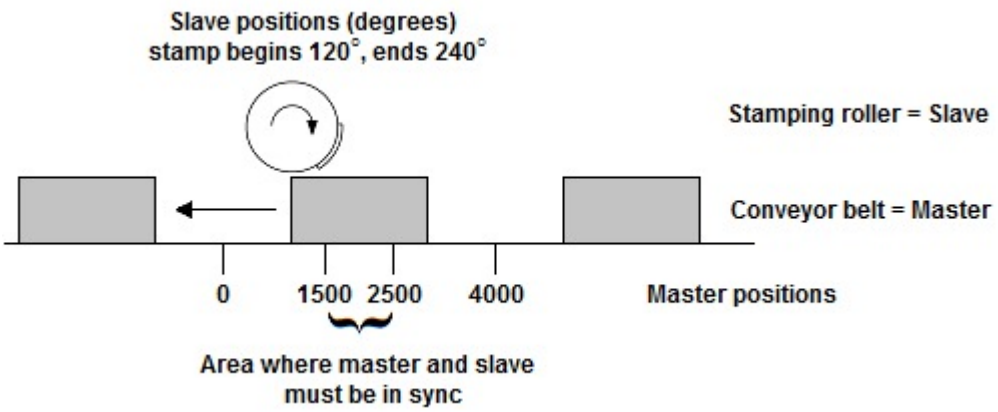
A stamping roller is to stamp a "use-by date" inscription on cardboard boxes. The cardboard boxes are always transported on a conveyor belt at exactly the same distance from each other (e.g. by means of a mechanical set pattern). During the printing process, the stamping roller and the cardboard box *must* run in sync to produce a clear inscription. At other times, the stamping roller does not need to be in sync with the boxes or belt.

Conveyor Belt specification:

- 60 cardboard boxes are transported on the conveyor belt per minute.
- Boxes are 400 mm apart.
- The expected resolution of positioning of the conveyor belt is 1/10 mm.
- The mechanical connection from the main motor drive to the conveyer belt drive pulley has a gearing ratio of 25:11 (i.e. the motor makes 25 revolutions while the belt drive pulley makes 11 revolutions).
- The belt drive pulley has 20 teeth and each tooth corresponds to 5 mm of belt feed.
- An incremental encoder is mounted directly on the main motor and has an encoder resolution of 4096 counts per revolution.
- A digital signal is available indicating the state of the main motor. This signal will be off (low) when the motor is turned off and on (high) when the motor is turned on.
- When the system starts, the conveyor belt will be stopped and it will be positioned so that the front edge of the first box is exactly aligned with the stamping roller (see diagram below).

Stamping Roller specification:

- The stamped inscription is 100 mm in length and corresponds to the roller section from 120 degrees to 240 degrees.
- The expected resolution of the roller is 1/10 degree.
- The inscription is to begin 5 mm from the edge of the box.
- The mechanical connection from the roller motor drive to the roller has a gearing ratio of 5:1.
- An incremental encoder is mounted on the roller motor and has an encoder resolution of 500 counts per revolution.
- When the system starts, the stamping roller must have been manually aligned so that it is positioned at 0 degrees.



For this example, the roller will be synchronized to the conveyor belt so its motion will be determined by the motion of the belt. When a box is in position, then the stamping roller must also be in position. This means that for the CAM curve, the conveyor belt will be the "master" and the roller will be the "slave".

Before Beginning

This example assumes the following:

- The controller will drive the stamping roller motor using axis 1 and encoder 1.
- The controller will watch the conveyor belt main motor but this motor will be driven by some external system.
- The encoder of the conveyor belt motor will be connected to the default master encoder for the first axis. For a 3-axis controller, the default master encoder for axis module 0 is encoder module 3. Axis parameter PISRC_MVIRTCOUNT can be set for axis 1 if a different encoder is to be used as the master encoder.

- The digital signal indicating the state of the main motor will be connected to digital input 1.
- The controller axis parameters have been previously tuned for controlling the stamping roller motor and are currently set properly in the controller. See Optimizing the PID Controller for a description of how to tune the controller. It is also possible that the controller parameters have been previously saved to a Configuration zbc file (see CAM Editor).

How to Edit the Curve

1. Determine the **Master Length**. This is the cycle length of the "master" and is measured in "master units" (MU). Boxes are 400 mm apart so the stamping cycle will repeat every 400 mm. Since CAM curves are based on integral numbers and a resolution of 1/10 mm is required, this means we can choose a **Master Length** of 4000 MU and 1 mm = 10 MU. The entire stamping process will cycle once each time the belt moves 4000 MU.

2. Determine the **Slave Length**. This is the cycle length of the "slave" and is measured in "user units" (UU). We will measure the roller position in degrees so one cycle will be 360 degrees; the roller will rotate once for each box. Since a resolution of 1/10 degree is required, the **Slave Length** will become 3600 UU.

3. Start the CAM Editor using one of the following methods:

   - If the axis parameters for the stamping roller are currently set properly in the controller, then create a new configuration file by either using **File** → **New** or pressing the **CAM Editor** toolbar button. Then press press the **Read from controller** toolbar button to select the controller and load the controller parameters.
   - If the axis parameters have been previously saved in a Configuration zbc file, then open the file using **File** → **Open**.
   - For the purposes of this example, a sample Configuration zbc file called "Stamping.zbc" has been included in the ApossIDE. To open this file, use:

     File → **Sample** → **Synchronization and Markers** → **Stamping Sample** → **Stamping.zbc**

4. Use the **Add array** toolbar button to add a new CAM curve array. Give the curve a name (e.g. "Stamping") and enter the **Master Length** of 4000.

5. Select the CAM Curve Tab. Set the **Cycles/min Master** to 60. This is the (maximum) number of cardboard boxes that are processed per minute. Also, we would like this curve to be a true cyclic curve so set the **Curve Type** to "Cyclic". This will give the smoothest motion to the roller.

   Cycles/min Master = 60
   Curve Type = Cyclic

   > Older controllers do not support true cyclic curves. For these controllers, set the **Curve Type** to "Average start and end velocities".

6. Select the Fix Points Tab . There should be two fixed points at (0,0) and (4000,0). Set the **Slave Length** by double clicking on the slave coordinate of the second point and entering 3600.

Point	Master	Slave	Type	Λ
1	0	0	C	
2	4000	3600	-	

7. When the belt is at position 1500 MU, the roller must be at 120 degrees (1200 UU) in order to start

stamping the inscription. When the belt is at position 2500 MU, the roller must be at 240 degrees (2400 UU) to end the inscription. Press the **Add** button in the Fix Points Tab and add two fixed points as shown below.

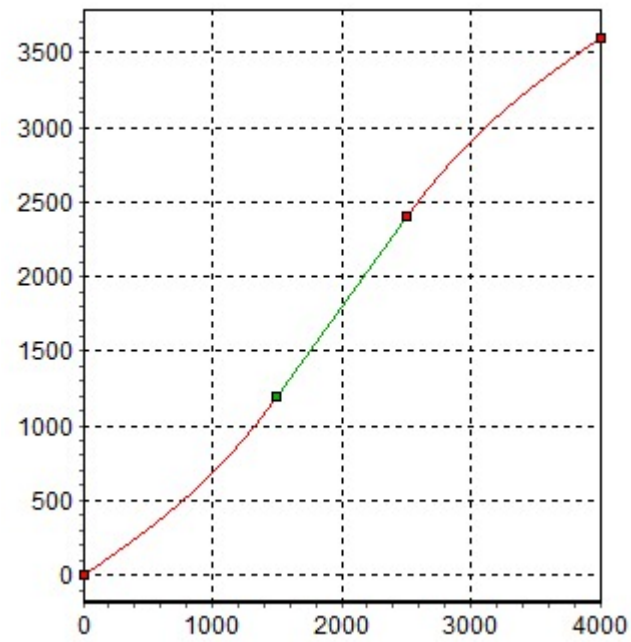
Point	Master	Slave	Type	Λ
1	0	0	C	
2	1500	1200	C	
3	2500	2400	C	
4	4000	3600	-	

For very old controllers that require interpolation, the **Snap on Grid** function should be enabled.

8. Master and slave must run synchronously with the same velocity between the position 1500 MU and 2500 MU. This requires a straight line connecting these two points.

   Double-click in the **Type** column for the fixed point on the left at position 1500 MU and select **Tangent**. Alternatively, you can move the mouse to the fixed point at 1500 MU, click the right mouse button, and select **Tangent** in the subsequent pop-up menu.

   A straight line (colored green) will now connect this point and the following point. Note that the curve is cyclic and the gradient of the curve at the beginning matches the gradient at the end.



9. Determine the conversion factor for master encoder quadcounts (qc) to master units (MU):

1 pully revolution	=	gearing factor * 1 main motor revolution
20 teeth * 5 mm/tooth	=	gearing factor * encoder resolution
100 mm	=	gearing factor * (encoder resolution * 4) qc
100 mm * 10 MU/mm	=	(gearing factor * encoder resolution * 4) qc
1000 MU	=	(gearing factor * encoder resolution * 4) qc
1 MU	=	(gearing factor * encoder resolution * 4) / 1000 qc
1 MU	=	(25/11 * 4096 * 4) / 1000 qc
1 MU	=	(25 * 4096 * 4) / (11 * 1000) qc
1 MU	=	4096/110 qc

1 MU	=	2048/55 qc	= SYNCFACTM/SYNCFACTS
55 MU	=	2048 qc	

In other words, when the main motor moves 2048 qc, then the belt will move 55 MU.

Enter these values in the Axis Data Tab (the units must always be whole numbers):

Syncfactor Master:	SYNCFACTM	=	2048
Syncfactor Slave:	SYNCFACTS	=	55

10. Determine the conversion factor for slave encoder quadcounts (qc) to user units (uu):

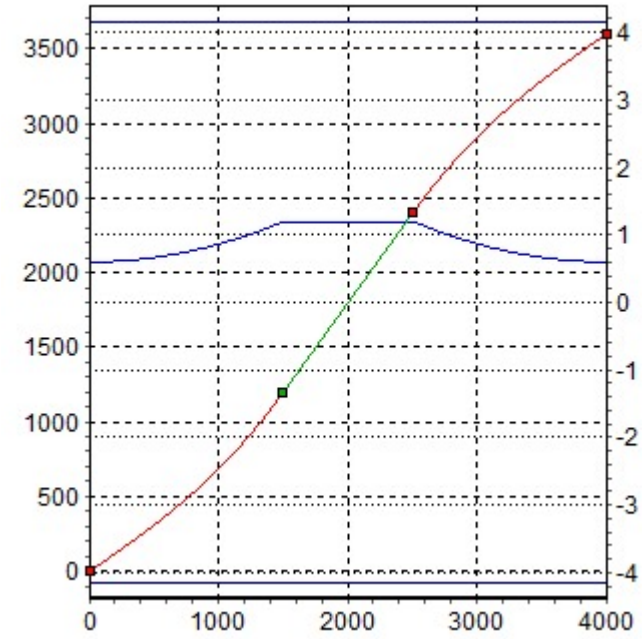
1 roller revolution	=	gearing factor * 1 roller motor revolution
360 degrees	=	gearing factor * encoder resolution
360 degrees * 10 UU/degree	=	gearing factor * (encoder resolution * 4) qc
3600 UU	=	(gearing factor * encoder resolution * 4) qc
1 UU	=	(gearing factor * encoder resolution * 4) / 3600 qc
1 UU	=	(5/1 * 500 * 4) / 3600 qc
1 UU	=	10000/3600 qc
1 UU	=	25/9 qc = POSFACT_Z/POSFACT_N
9 UU	=	25 qc

When the rollor motor moves 25 qc, then the rollor will move 9 UU.

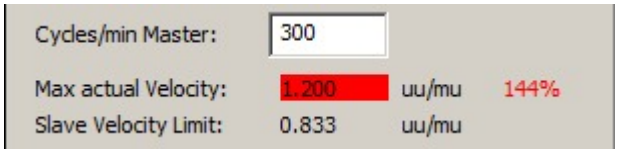
Enter these values in the Axis Data Tab:

User factor numerator:	POSFACT_Z	=	25
User factor denominator:	POSFACT_N	=	9

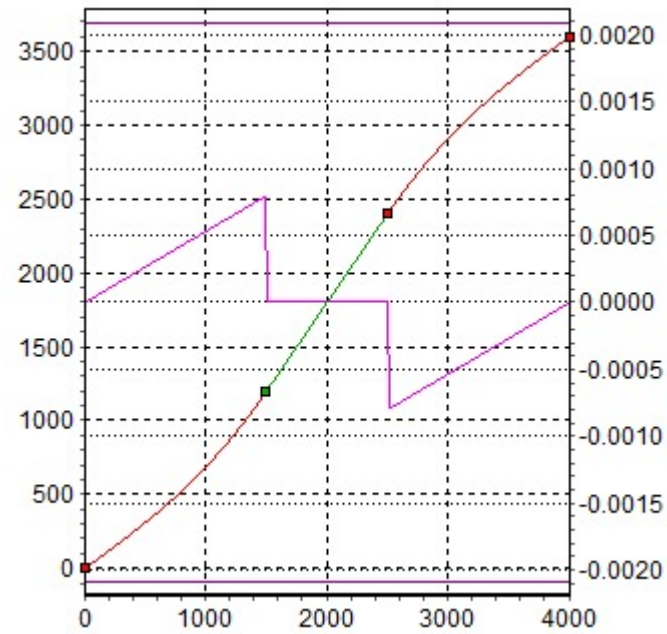
11. Enable ☑ **Velocity** and ☑ **Vel. Limit** in the Display Tab to display the corresponding velocity curve and the velocity limits. The velocity will be shown in bright blue and the velocity limits will be shown as horizontal dark blue lines. The velocity scale is shown on the right side of the diagram and is measured in UU/MU. You can see that the velocity of the roller is constant between 1500 MU and 2500 MU. The rollor velocity is also well within the velocity limits.



You can experiment by specifying a different **Cycles/min Master** or **Curve Type** in the CAM Curve Tab. Look to see how the shape and magnitude of the velocity curve changes. When the velocity or acceleration exceeds the maximum limit, then the actual velocity and/or acceleration will be displayed with a red background in the tab. In the example below, the **Cycles/min Master** has been set to 300; the system is now running too fast.



12. Disable the velocity display and enable ☑ **Acceleration** and ☑ **Acc. Limit**. The acceleration is shown in bright magenta and the acceleration limit is shown as horizontal dark magenta lines. Verify that the acceleration of the roller is within the limits.



13. You will see the number of array elements required for this CAM curve in the title bar. Remember this number as you will need it later during programming.

14. The CAM curve is now complete. Save the CAM curve and settings as a zbc file using **File** → **Save**. You will need the name of this file later during programming.

15. If the ApossIDE is connected to a controller, then the CAM curve can be downloaded to the controller using one of the following methods. Note that only the DIM Arrays need to be downloaded.

    - Use the **Write to Controller** toolbar button in the CAM Editor.
    - Use **Controller** → **Parameters** → **Restore from file** to download the zbc file saved in the previous step.

Program Sample

A sample application program that implements the stamping application is shown below and can be opened in the ApossIDE using:

File → **Sample** → **Synchronization and Markers** → **Stamping Sample** → **Stamping.mc**

> This application program has been designed so that it is compatible with both the Simulator and the Oscilloscope. If you would like to experiment with this example using the Simulator, then please see Example 1: Using the Simulator.

```
/*******************************************************************
**
**    This program illustrates synchronized positioning of one axis
**     acting as a slave, relative to a virtual master with using a
CAM Curve.
**    The axis can be considered as a stamping roller and the master
```

```
as a
**    conveyor belt.
**
** Notes:
**
**    1. This example is terminated after a low of the digital input
1.
**
**    2. Setting four of the Simulator's SDO Display outputs to:
**           "0x2500/1 - REG_ACTPOS" (Actual position of axis 0)
**           "0x2500/4 - REG_AVEL"   (Actual velocity of axis 0)
**       will allow the progress of the movement to be displayed by
**       the Simulator.  See the topic "ApossIDE Tools/Simulator/
**       Simulator User Interface" in the ApossIDE Help.
**
**       3. The CAM curve stamping.zbc must be written to the
simulated
**          controller with a write command.
**
**       4. The file stamping.zbo is prepared as osiloscope. This must
be
**          started first using the play button.
*/



#include <SysDef.mh>

dim long stamping[43];

/*
** Main Program
*/

//
#define AXIS 0      // Use axis module 0.
#define SYS_INT_NO_OF_AXIS      Sysvar[0x01220905]
//#define I_MARKER_SENSOR        4     // Marker signal

// function declaration
void SetupAxisSimulation(long axisNo);

long main(void)
{

    long startpos;
    /*
```

```
    ** Set the basic motor and encoder parameters. These are normally
    ** required before any actual drive can be moved.  Please see the
    ** topic "Getting Started and Safety Tips/Setting Basic
Parameters"
    ** in the ApossIDE Help. How to set the parameters is described in
    ** the ApossIDE Help "How to Set Up Resolution and Gear
Parameters".
    **
    ** Note:
    **         If this sample program is to be used to drive an actual
    **       motor, then these parameters MUST be set correctly for
that
    **       motor.
    **
    */

    AXE_PARAM(AXIS, POSENCREV) = 1;        // Number of revolutions of
the motor
    AXE_PARAM(AXIS, POSENCQC)  = 4096*4;// Number of quadcounts in
POSENCREV revolutions
    AXE_PARAM(AXIS, POSFACT_Z) = 25;    // Number of revolutions of
the input shaft
    AXE_PARAM(AXIS, POSFACT_N) = 11;    // Number of revolutions of
the output shaft in POSFACT_Z revolutions of the input shaft
    AXE_PARAM(AXIS, FEEDREV)   = 1;        // Number of revolutions of
the gear box output shaft
    AXE_PARAM(AXIS, FEEDDIST)  = 4096*4;// Distance travelled (in user
units) in FEEDREV revolutions of the gear box output shaft

    AXE_PARAM(AXIS, VELMAX)      = 500;    // Rated speed of drive
    AXE_PARAM(AXIS, RAMPMIN)     = 500;    // Maximum acceleration

    /*
    **     The parameterization of the two factors SYNCFACTM and
SYNCFACTS is
    **    described in the ApossIDE Help under the chapter
    **    SDO Dictionary -> Axis Parameters -> SYNCFACTM.
    */

    AXE_PARAM(AXIS, SYNCFACTM)  = 2048;    // Synchronization factor
master (M:S) or in CAM mode conversion qc in MU units
    AXE_PARAM(AXIS, SYNCFACTS)  = 55;    // Synchronization factor
slave (M:S) or in CAM mode conversion qc in MU units
    AXE_PARAM(AXIS, SYNCMPULSM) = 0;    // Master marker distance
    AXE_PARAM(AXIS, SYNCMPULSS) = 0;    // Slave marker distance

    /*
    ** This function will simulate a motor moving when
```

```
    ** there is no motor.
    **
    ** Note: Comment out the "SetupAxisSimulation(AXIS)" function if
this sample
    **       is to be executed using an actual drive.  Please see the
warnings
    **       described in the topic "ApossIDE Programming/Program
Samples"
    **       in the ApossIDE Help.
    */

    SetupAxisSimulation(AXIS);

    /*
    ** VELRES determines the resolution of measurement used by the
VEL,
    ** ACC, and DEC commands.  Setting this to 100 means that values
can
    ** be in the range 0-100 where 100 corresponds to the maximum
rating
    ** of the motor.  This is equivalent to percent.
    */

    AXE_PARAM(AXIS, VELRES) = 100;

    /*
    ** For this sample, acceleration and deceleration will be limited
to
    ** 100% of the maximum rating of the motor.
    */

    Vel(AXIS, 100);
    Acc(AXIS, 100);
    Dec(AXIS, 100);

    /*
    **     The conveyor MUST NOT be running at this time (so digital
input 1
    **     must be low).  Otherwise, we can't synchronize the stamping
roller.
    */

    if (DigInput(1))
        {
        print("ERROR: The conveyor belt is already running.");
        print("       Please stop the conveyor belt (Input 1) and then
restart this program.");
```

```
        Exit(0);
        }

    /*
    **     Definition of a virtual master as master of axis 0 In a
real
    **  environment an encoder is used as master.
    */

     VIRTMAST_PARAM(0,VIRTMAST_MODE)= VIRTMAST_MODE_PROFILE;
       VIRTCOUNTIN_PARAM(AXIS + SYS_INT_NO_OF_AXIS,
VIRTCNTIN_PISRC_COUNTER) = VIRTMAST_PROCESS_SRCINDEX(0,
PO_VIRTMAST_POS);


       /*
    ** Turn the stamping roller motor on.  It will not move unless we
do this.
    */

    AxisControl(AXIS, ON);

    /*
    **     The stamping roller uses an incremental encoder so it has
no physical
    **    "home" position.  However, before the system is first
started, the
    **    stamping roller should have been manually placed into the
position
    **    that corresponds to 0 degrees.  Define this as the "origin"
of the
    **     roller's encoder (i.e. the position where the encoder reads
0).
    */

    DefOrigin(AXIS);

    /*
    ** Set the stamping curve.
    */

    SyncSetCurve(AXIS, stamping);    // Set the stamping curve.

    /*
    **     The conveyor belt should already have been aligned so that
the front
    **     edge of the first box is aligned with the stamping roller.
This will
```

```
    **     correspond to position 1000 MU in the CAM Curve.  Set the
initial
    **     position of the master to this value. As a result of the
gear ratio 25/11,
    **    the value entered in the curve is multiplied by the factor
25/11.
    **    0x2500/1 - 0: REG_ACTPOS => 1563.
    */

    DefMasterCpos(1000, AXIS);

    /*
    **     Move the stamping roller to the CAM Curve position that
corresponds to
    **     the master position.  Everything is now positioned
correctly for when
    **     the conveyor starts moving.  Note that this command will
not return
    **    until the stamping roller arrives at the target position.
    */

    AxisPosAbsStart(AXIS, CurvePos(AXIS));
    AxisWaitReached(AXIS);

    /*
    *     Change into CAM Mode.  The 0 means that we will stay in CAM
Mode until
    *     we explicitly leave CAM Mode.
    */
    SyncCam(AXIS, 0);

    /*
    **    Start synchronization.  The 0 means "engage roller
immediately with the
    **    set maximum velocity".  This does not cause any movement
since the master
    **     is idle and in the correct position.  The stamping roller
will start to
    **     move when the conveyor starts to move.
    */

    SyncCamStart(AXIS, 0);

    /*
    **     Now the master (conveyor) can be started.  Wait for digital
input 1 to
    **    go high.  This will indicate that the conveyor belt has been
```

```
turned on.
    */

    print("Ready.  Please start the conveyor belt (Input 1).");

    while (!DigInput(1));

    print("Running...");

    /*
    ** SIMULATOR ONLY: Enable the virtual master.  These commands will
make it
    **                 appear as if the conveyor belt is moving.
    */

    VIRTMAST_PARAM(0,VIRTMAST_ACC)= 100;
    VIRTMAST_PARAM(0,VIRTMAST_VEL)= 100;

    /*
    **     Wait for digital input 1 to go low.  This will indicate
that the conveyor
    **     belt has been turned off.  Once this happens, we can turn
off the stamping
    **     roller motor.
    **
    **     Note:     We are testing for the digital input to go low
very quickly after it
    **           has gone high.  If the switch driving the digital
input is "rough" or
    **           "bounces", then it may appear as if the input goes
on-off-on.  So wait
    **           a short while (100 ms) for the input to stabilize
before starting to
    **           test it again.
    */

    Delay(100);

    while (DigInput(1));

    /*
    **    The conveyor belt has been turned off.  Turn off the
stamping roller motor.
    **    SIMULATOR ONLY: Disable the virtual master.
    */

    AxisControl(AXIS,OFF);
    VIRTMAST_PARAM(0,VIRTMAST_ACC)= 0;
```

```
    VIRTMAST_PARAM(0,VIRTMAST_VEL)= 0;

    print("Finished.  Stamping roller has been stopped.");

    return(0);
}

/*********************************************************************
** Functions
*********************************************************************/

/*
** This function will simulate a motor moving when
** there is no motor.  Then turn the motor on.
*/
void SetupAxisSimulation(long axisNo)
{
    // Parameterization Encoder  ==> Simulation
    VIRTCOUNTIN_PARAM(axisNo, VIRTCNTIN_PISRC_COUNTER) =
AXE_PROCESS_SRCINDEX(axisNo, REG_COMPOS);
    VIRTCOUNTIN_PARAM(axisNo, VIRTCNTIN_MODE) =
VIRTCNTIN_MODE_SIMULATION_SLAVE;
    VIRTLATCH_PARAM(axisNo, VIRTLATCH_MODE) =
VIRTLATCH_MODE_SIMULATOR;
    VIRTLATCH_PARAM(axisNo, VIRTLATCH_PISRC_COUNTER) =
AXE_PROCESS_SRCINDEX(axisNo, REG_ACTPOS);
}
```

6.5.6.1 Example 1: Using the Simulator

The application program for this example has been designed so that it is compatible with the Simulator. To run the program on the Simulator follow these steps:

1. Open the sample program using:

   File → Sample → Synchronization and Markers → Stamping Sample → Stamping.m
2. Modify the parameter initialization statements near the beginning of the program so that they match the zbc file that you previously saved in Example 1: Basic Synchronization (step 14).
3. Connect to a Simulator (see Simulator Connections).
4. In the Simulator window, set the first SDO display to PFG_CMASTERCPOS (0x2500/135) and the second to PFG_CCURVEPOS (0x2500/133) (see Simulator User Interface). This will let you see the master and slave positions respectively as the Simulator is running.
5. Use **Controller → Parameters → Restore from file** to download the zbc file that you previously saved in Example 1: Basic Synchronization (step 14) to the Simulator. Note that only the DIM Arrays need to be downloaded. This will put the CAM curve into permanent memory in the Simulator. For your convenience, a completed zbc file is also available. Use:

   File → Sample → Synchronization and Markers → Stamping Sample → Stamping.zbc

to open it. Download the file using **Write to Controller**.
6. Start the program by pressing **F5** or using **Development** → **Execute**.
7. When prompted by the "Ready" message, set digital input 1 in the Simulator window to simulate the conveyor belt being turned on.
8. When you are finished, clear digital input 1 in the Simulator window to simulate the conveyor belt being turned off.

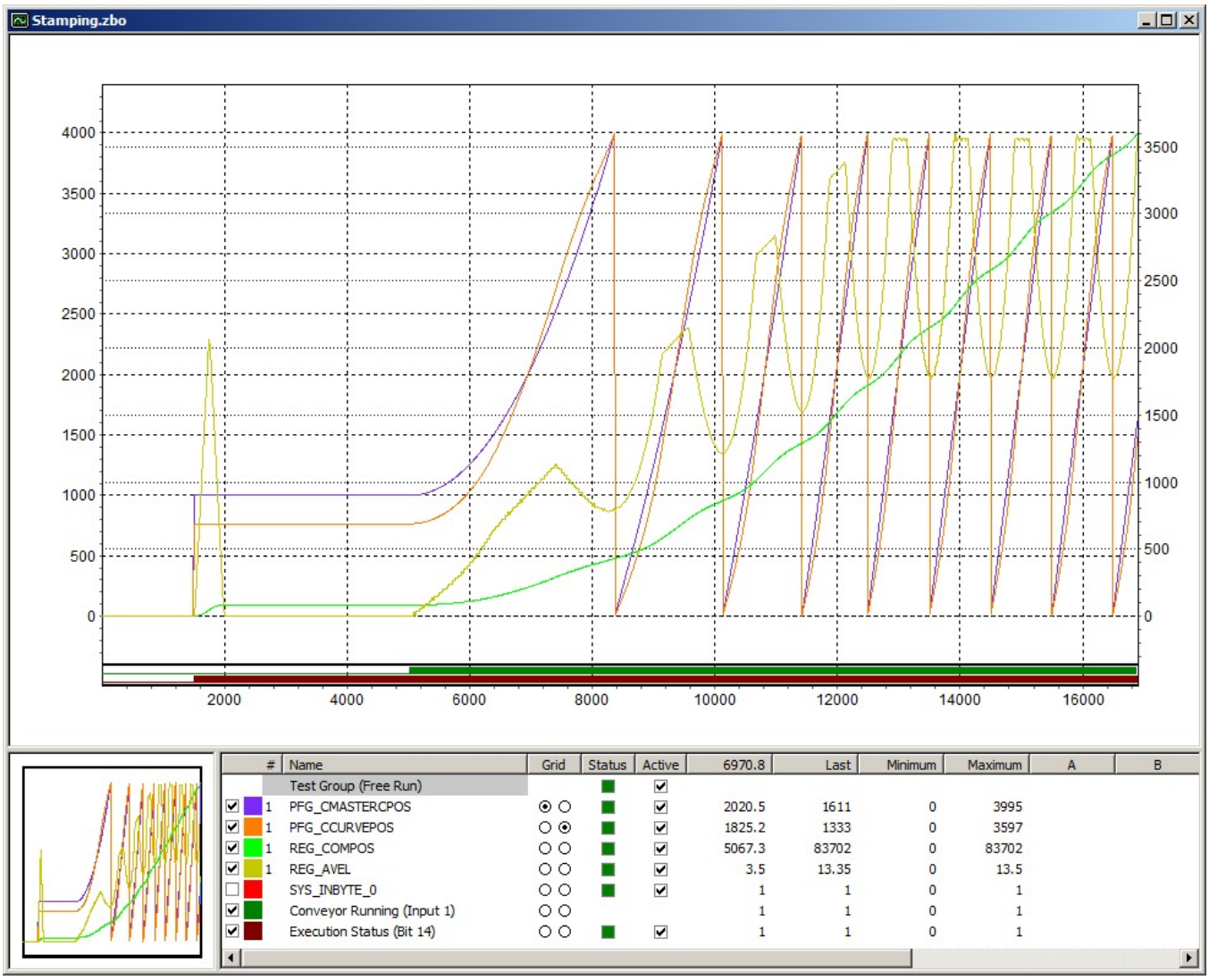
It is also possible to use the Oscilloscope to "watch" the Simulator running. Follow these steps:

1. The Simulator should already be running and the STAMPING.M program open from the previous steps.
2. Open the predefined Oscilloscope file using:

   File → **Sample** → **Synchronization and Markers** → **Stamping Sample** → **Stamping.zbo**
3. Right-click in the **Status** column (below the column title) and select **Reconnect** from the popup menu. The **Status** boxes should turn green indicating that the Oscilloscope is now connected to the Simulator.
4. Press **Start** to start the Oscilloscope running. Then switch to the program window and press **F5** to start the application program running on the Simulator. Wait for the "Ready" message to be displayed. Then switch to the Simulator window and set digital input 1. This will simulate the conveyor belt starting.
5. Switch back to the Oscilloscope window to watch the Simulator running.
6. Let the simulation run for 5-10 seconds. When you want to stop, press **Stop** to stop the Oscilloscope. Then press **Zoom Reset** to redisplay the entire run sequence. Finally, switch to the Simulator window and clear digital input 1 to simulate the conveyor belt being turned off.

The Oscilloscope window should appear similar to that shown below. The curves are as follows:

PFG_CMASTERCPOS - The current master position on the CAM curve in MU.
PFG_CCURVEPOS - The current slave position on the CAM curve in UU.
REG_COMPOS - The encoder position of the stamping roller in qc.
REG_AVEL - The velocity of the stamping roller in qc/ms.
SYS_INBYTE_0 - The status of the first 8 digital inputs. This value does not need to be displayed.
Running (Input 1) - The status of digital input 1. This is displayed as a "signal" at the bottom of the diagram. It is derived from the value of **SYS_INBYTE_0**.
Execution Status (Bit 14) - This bit indicates whether or not an application program is running. This is displayed as a "signal" at the bottom of the diagram. It is derived from the value of SDO 0x2206.



Time in ms is displayed along the X-axis of the chart. MU are displayed up the Y-Axis on the left side of the chart. The range from 0 to 4000 represents one cycle of the CAM curve. UU are displayed on the right side of the chart. The range from 0 to 3600 also represents one CAM curve cycle.

Note the following points of interest regarding the curves in the chart:

Time (ms)	Description
0-1500	The curves are all flat. This is the time during which the Oscilloscope was running but the Simulator was idle before the application program was started. You can see the application program is not running by looking at the **Execution Status (Bit 14)** signal.
1500	The application program was started and the following happens: • **Execution Status (Bit 14)** goes high showing that the application program is now executing. • **PFG_CMASTERCPOS** jumps to 1000 MU. This corresponds to the "DEFMCPOS 1000" command that resets the master position on the CAM curve so that it matches the front edge of

the first box. Note that the current value of the master encoder is not important. It is only important that this encoder value corresponds to 1000 MU.

- **PFG_CCURVEPOS** jumps to 688 UU. This happens automatically when the CAM curve master position is set. This is the slave position that corresponds to the current master position.
- The application program then excutes the "POSA CURVEPOS" command to reposition the stamping roller.

1500-2000 The stamping roller is being repositioned from its starting orientation (0 degrees) to the position that matches the current master position on the CAM curve (1000 MU and 688 UU). You can see **REG_COMPOS** move up as the stamping roller turns into the correct position. You can also see **REG_AVEL** spike up and down as the roller speeds up and then slows down to stop.

2000 The stamping roller stops and the "POSA CURVEPOS" command completes. At this point, stamping roller is synchronized with the conveyor belt and the application program enables synchronization. Then it starts waiting for digital input 1 to signal that the conveyor belt has started. Note that this is only necessary for the Simulator (so that it can start the virtual master encoder); it would not be necessary for a "real" application.

2000-5000 The application program is waiting for digital input 1 to signal the start of the conveyor belt movement.

5000 Digital input 1 is set and the following happens:

- **Running (Input 1)** goes high indicating that input 1 is set.
- The conveyor belt starts moving.
- Both **PFG_CMASTERCPOS** and **PFG_CCURVEPOS** to start moving up (and along the CAM curve) as they track the moving conveyor belt.
- Since **PFG_CCURVEPOS** is driving the stamping roller, both **REG_COMPOS** and **REG_AVEL** also start moving up as the stamping roller rotates forward and gains speed.

>5000 **PFG_CMASTERCPOS** shows successive cycles of the CAM curve as it moves from 0 to 4000 and then resets to 0 for the next cycle. Similarly, **PFG_CCURVEPOS** cycles from 0 to 3600. Note that the first cycle takes about 3500 ms, the second about 1800 ms, and after about 4 cycles, they are running at 1000 ms per cycle. This happens because the "PULSACC 5000" command simulates the conveyor belt slowly accelerating from stopped to full speed. The final 1000 ms per cycle corresponds to the required stamping rate of 60 boxes per minute.

6400-7400	**REG_AVEL** is a straight line sloping up. This means that the stamping roller is accelerating (i.e. the slope) but it is accelerating at a rate that exactly matches the acceleration of the converyor belt (i.e. the straight line). This is the area of the CAM curve where the stamping roller is actually printing the inscription on the box and it shows that the stamping roller is truely synchronized with the conveyor belt.
7400	**REG_AVEL** starts dropping. The printing is complete and the stamping roller is slowing down in order to wait for the next box to arrive. You can verify this by looking at **PFG_CMASTERCPOS** which is at 2500 MU. This is exactly the point in the CAM curve where the printing is expected to be complete.
12800-13200	**REG_AVEL** is horizontal. Again, this is the area of the CAM curve where the stamping roller is actually printing the inscription. The horizontal line indicates that the stamping roller is moving at constant velocity.

6.5.7 Example 2: Synchronization with Marker Correction

The following example is based on the previous example (see Example 1: Basic Synchronization). However, it assumes that the boxes are not always spaced at exactly the same distance apart. In this case, markers will be required that can signal the start of a box and correct the synchronization.

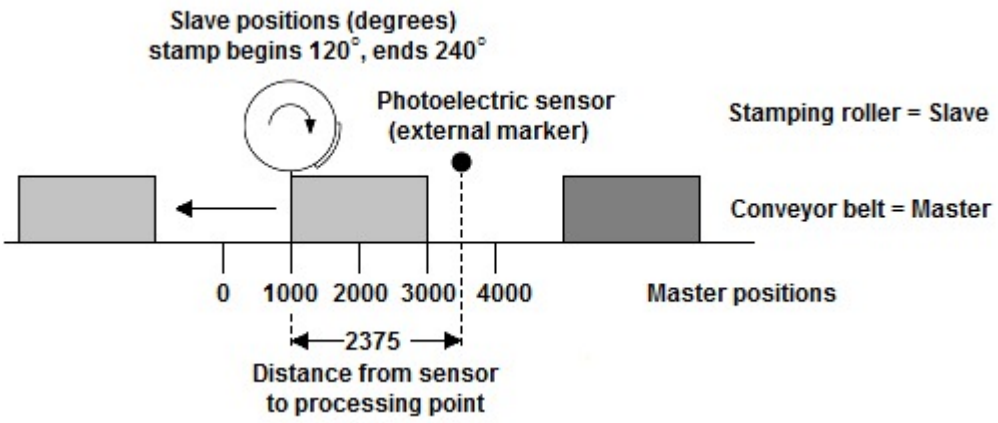
> The same technique of synchronization with marker correction can be used to handle such things as slippage and the accumulation of systematic positioning errors.

The following step-by-step explanation describes how you can adapt the curve of the previous example to this application.

The basic setup of the system is identical to the previous example except that a photoelectric sensor has been added to signal the start of a box as it passes on the conveyor belt. The photoelectric sensor will be connected to the controller by means of one of the controller's digital inputs. The ApossIDE refers to this type of sensor as an "external marker".

Specification:

- The distance from the photoelectric sensor to the processing point (the axis of the stamping roller) is 237.5 mm. Note that the processing point is at position 1000 MU.
- The sensor will set digital input 2 high when it is triggered.
- The nominal box-to-box spacing will be 400 mm, however the actual spacing may vary by +/-20 mm.
- When the system starts, the conveyor belt will be stopped and it will be positioned so that the front edge of the first box (dark gray) is slightly further away than the photoelectric sensor (see diagram below). The stamping roller must have been manually aligned so that it is positioned at 0 degrees.



Basically, the controller will operate as follows: Initially, both the conveyor belt and stamping roller will be running but will essentially be waiting until the first box arrives. As soon as possible after the front edge of the first box triggers the photoelectric sensor, the controller will start to synchronize the velocity and position of the stamping roller with the arriving box. This must be complete by the time that the incription location arrives at the processing point. During the inscription stamping, the conveyor and stamping roller *must* stay synchronized. However, after the stamping is complete, then the stamping roller is free to start synchronizing with the *next* box. It will do this based on the timing of the trigger event for the next box (which may or may have already been received).

> ⚠ For this example to work correctly, the distance from the sensor to the processing point must be *less* than the **Master Length** of the CAM curve (i.e. the sensor must be *within* the range of the CAM curve). See Example 2: Large Sensor Distances and Example 2: Problematic Situations for how to adapt this example to other cases.

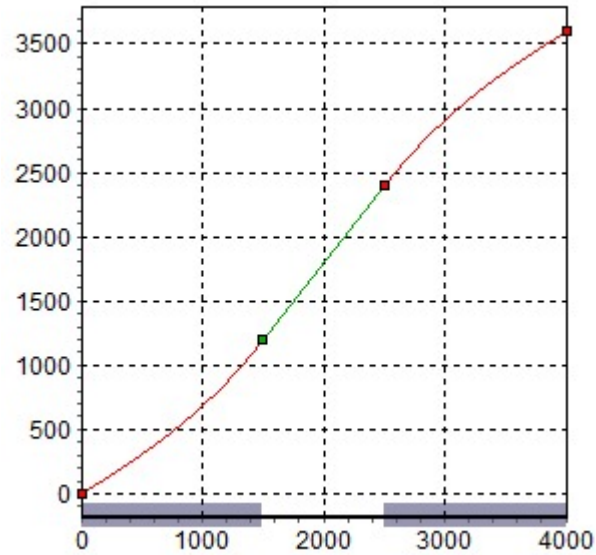
How to Edit the Curve

1. Steps 1 to 12 (verifying velocity and acceleration) are the same as in the previous example (see Example 1: Basic Synchronization).

2. While the conveyor belt is between 1500 MU and 2500 MU and the current box is being printed, the stamping roller *must* stay synchronized. However, outside of this range, the controller is free to re-synchronize the stamping roller with the next box to arrive. The re-synchronization may start at any time after the conveyor belt passes 2500 MU (i.e. the end of the current printing) and it must be complete by the time the conveyor belt reaches 1500 MU on the next cycle (i.e. the start of the next printing). This is the "Correction" range. Enter the values in the CAM Curve Tab.

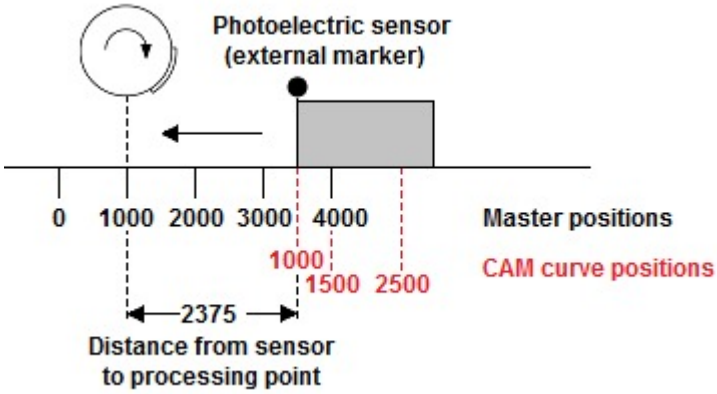
   Correction Start = 3000
   Correction End = 1000

   Enable ☑ **Correction** in the Display Tab. The area of correction will be shaded at the bottom of the diagram.



3. When the photoelectric sensor (external marker) detects the front edge of a box, then this position will correspond to master position 1000 MU *in the CAM curve* (i.e. where the front edge of the box is positioned in the CAM curve). This is shown below in red. Remember that the inscription will start at 1500 MU. In the CAM Curve Tab, set the **Master Marker Position** to 1000 MU.

   Master Marker Position = 1000



The distance from the processing point (1000 MU) to the photoelectric sensor is 237.5 mm or 2375 MU. This is the distance offset that is required to match the original master positions (shown black above) to the corresponding CAM curve positions (shown red above). Enter this value as the **Master Marker Distance** in the Axis Data Tab.

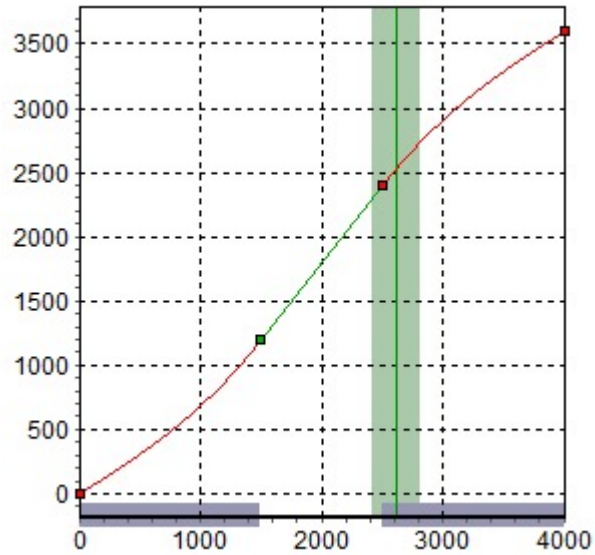
Master Marker Distance:	SYNCMPULSM	=	2375

Since the spacing of the boxes can vary by +/-20 mm, set the **Master Marker Tolerance** to 200 MU in the Axis Data Tab. The photoelectric sensor is an external sensor and the signal goes high when triggered, so set the **Master Marker Type** to 2.

Master Marker Tolerance:	SYNCMWINM	=	200
Master Marker Type:	SYNCMTYPM	=	2

Enable ☑ **Master Marker** in the Display Tab. The master marker is displayed as a vertical green line. The

green background area shows the tolerance window for the appearance of the master marker. The display shows that the trigger for the *next* box may be received at any time that the conveyor belt is within the green background area for the *current* box. You can see in this example that a small portion of the green background occurs *before* 2500 MU while the current box is still being printed. However, this is outside of the correction range so if the next trigger is received in this area, then the controller will *not* start re-synchronization. Instead, the controller will wait until 2500 MU and *then* start re-synchronizing. The rest of the green background overlaps the correction range. If the next trigger is received in this area, then the re-synchronization may begin immediately. Note that the stamping roller will continue to follow the original CAM curve until the next trigger is received.



4. You will see the number of array elements required for this CAM curve in the title bar. Remember this number as you will need it later during programming.

5. The CAM curve is now complete. Save the CAM curve and settings as a zbc file using **File** → **Save**. You will need the name of this file later during programming.

6. If the ApossIDE is connected to a controller, then the CAM curve can be downloaded to the controller using one of the following methods. Note that only the DIM Arrays need to be downloaded.

   - Use the **Write to Controller** toolbar button in the CAM Editor.
   - Use **Controller** → **Parameters** → **Restore from file** to download the zbc file saved in the previous step.

Program Sample

A sample application program that implements the marker application is shown below and can be opened in the ApossIDE using:

File → **Sample** → **Synchronization and Markers** → **Marker Sample** → **Marker.m**

> This application program has been designed so that it is compatible with both the Simulator and the Oscilloscope. If you would like to experiment with this example using the Simulator, then please see Example 2: Using the Simulator.

```
// The CAM Curve will be stored in a DIM array and DIM arrays must
be declared
// first in application programs.  The number of elements in the
array (43)
// comes from the title bar of the CAM Editor.

DIM marker[43]

// The axis parameter settings should be set before the rest of the
application
// program starts.  These values are taken from the Configuration
ZBC file
// that was saved by the CAM Editor.
//
// Note: After axis parameters are set, the VEL, ACC, and DEC
commands must
//       be executed in order to allow the controller to start using
the
//       new axis parameters.  100 will allow the controller to
drive the
//       motor at up to 100% of its rated values.

sysvar[0x01230002] = 500         // VELMAX
sysvar[0x01230018] = 25          // POSFACT_Z
sysvar[0x0123001B] = 9           // POSFACT_N
sysvar[0x01230020] = 500         // RAMPMIN
sysvar[0x01230032] = 2048        // SYNCFACTM
sysvar[0x01230033] = 55          // SYNCFACTS
sysvar[0x0123003B] = 2375        // SYNCMPULSM
sysvar[0x0123003C] = 0           // SYNCMPULSS
sysvar[0x01230045] = 200         // SYNCMWINM

VEL 100
ACC 100
DEC 100

// This example is slightly different from the Stamping.m example.
The
// Stamping.m example required the conveyor belt to be stopped
before we
// could synchronize the stamping roller.  This example only
requires
// that no boxes arrive and trigger the sensor before we have
initialized
// the stamping roller.  However, to keep things simple, we will
still
```

```
// treat digital input 1 as indicating whether or not the conveyor
belt
// is moving.

if (IN 1) then
    PRINT "ERROR: The conveyor belt is already running."
    PRINT "       Please stop the conveyor belt (Input 1) and then
restart this program."
    EXIT
endif

// SIMULATOR ONLY: Set the encoder type for the stamping roller to
9.
//                 This is a special encoder type that simply copies
//                 the requested encoder position to the actual
encoder
//                 position.  This makes it look as if the stamping
//                 roller is moving even though a motor does not
exist.

SET ENCODERTYPE 9

// Set the master encoder to be an incremental encoder (type 0)
connected
// to encoder input 4.

SET MENCODERTYPE 40

// Set the master marker type.  This will set the master marker to
use
// digital input 3 and to trigger on a rising edge.

SET SYNCMTYPM 32

// SIMULATOR ONLY: This sets marker type for the virtual master
encoder
//                 so that it also uses digital input 2 and triggers
on
//                 a rising signal.

SET VMENCMTYP 22

// Turn the stamping roller motor on.  It will not move unless we do
this.

MOTOR ON

// The stamping roller uses an incremental encoder so it has no
```

```
physical
// "home" position.  However, before the system is first started,
the
// stamping roller should have been manually placed into the
position
// that corresponds to 0 degrees.  Define this as the "origin" of
the
// roller's encoder (i.e. the position where the encoder reads 0).

DEFORIGIN

// Set the stamping curve.

SETCURVE marker

// When a box triggers the sensor, this corresponds to the position
1000 MU
// on the CAM curve.  When this happens, the processing point is
237.5 mm
// (i.e. SYNCMPULSM) before this point.  Calculate this point and
use it to
// define the initial position of the conveyor belt (i.e. the
master).  This
// position may be negative.

dist = GET SYNCMPULSM
DEFMCPOS (1000-dist)

// Set the starting behaviour for synchronization with marker
correction.
// The 2000 means the master encoder will be ignored until the next
sensor
// trigger is received.  We can do this since the DEFMCPOS command
(above)
// will already have set the master position correctly to coincide
with
// the first trigger event.

SET SYNCMSTART 2000
set synctype 3

// Change into CAM Mode with master marker correction.  The 0 means
that
// we will stay in CAM Mode until we explicitly leave CAM Mode.

SYNCCMM 0
```

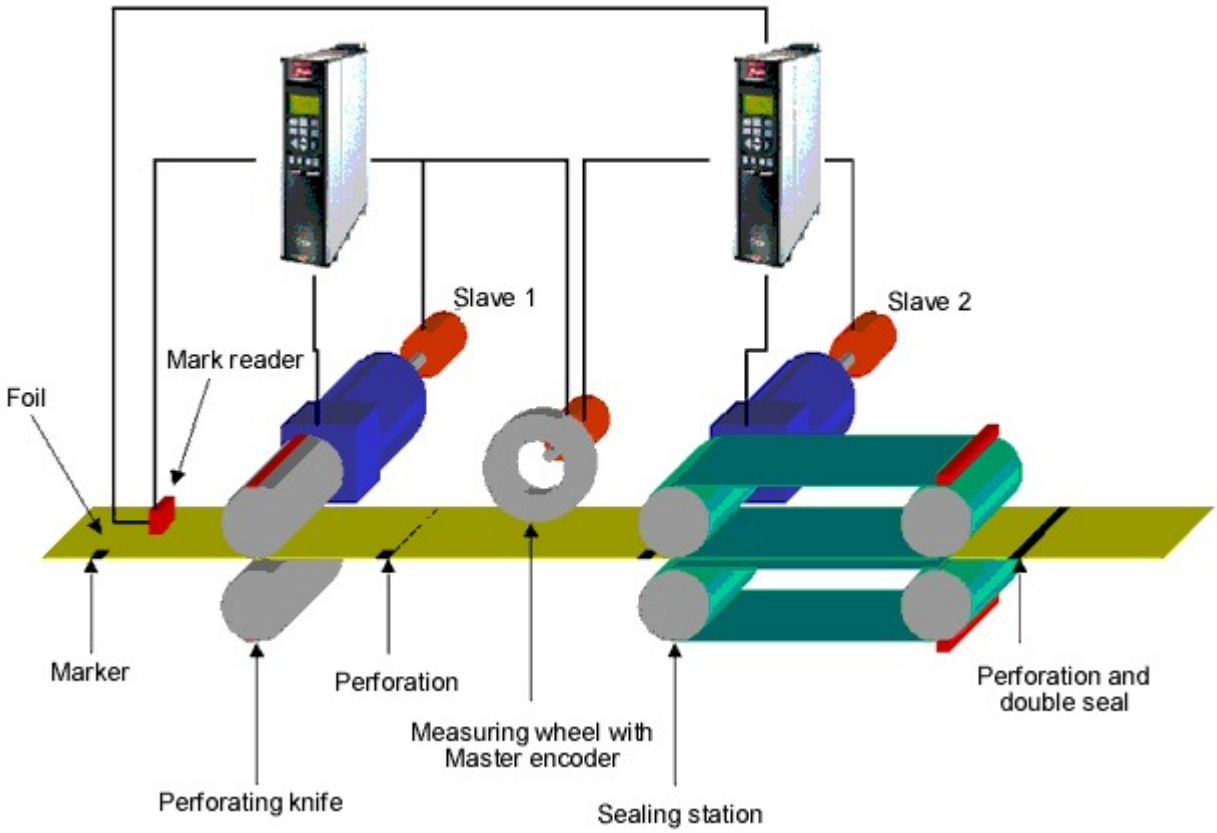
```
// Start synchronization.  The 0 means "engage roller immediately
with the
// set maximum velocity".  This does not cause any movement since
the master
// is idle and (hopefully) the sensor has not been triggered.  The
stamping
// roller will start to move when the conveyor starts to move.

SYNCCSTART 0

// Now the master (conveyor) can be started.  Wait for digital input
1 to
// go high.  This will indicate that the conveyor belt has been
turned on.

PRINT "Ready.  Please start the conveyor belt (Input 1)."

while (not IN 1) do
endwhile

PRINT "Running...  Input 2 signals a box arriving."

// SIMULATOR ONLY: Enable the virtual master.  These commands will
make it
//                 appear as if the conveyor belt is moving.  The
PULSVEL
//                 value of 37236 counts/sec will drive the conveyor
belt
//                 at a velocity of 400 mm/sec (i.e. 60 boxes/sec).
The
//                 PULSACC value of 10000 will cause the conveyor
belt to
//                 take "a few" box lengths to speed up before the
constant
//                 velocity of 400 mm/sec is reached.  Note that the
virtual
//                 master will override the "SET MENCODER 40"
command done
//                 above so encoder 4 will NOT actually be used by
the
//                 Simulator.

PULSACC 5000
PULSVEL 1000

// Wait for digital input 1 to go low.  This will indicate that the
conveyor
// belt has been turned off.  Once this happens, we can turn off the
```

```
stamping
// roller motor.
//
// Note: We are testing for the digital input to go low very quickly
after it
//       has gone high.  If the switch driving the digital input is
"rough" or
//       "bounces", then it may appear as if the input goes on-off-
on.  So wait
//       a short while (100 ms) for the input to stabilize before
starting to
//       test it again.

DELAY 100

while (IN 1) do
endwhile

// The conveyor belt has been turned off.  Turn off the stamping
roller motor.

MOTOR OFF

// SIMULATOR ONLY: Disable the virtual master.

PULSACC 0
PULSVEL 0

PRINT "Finished.  Stamping roller has been stopped."
```

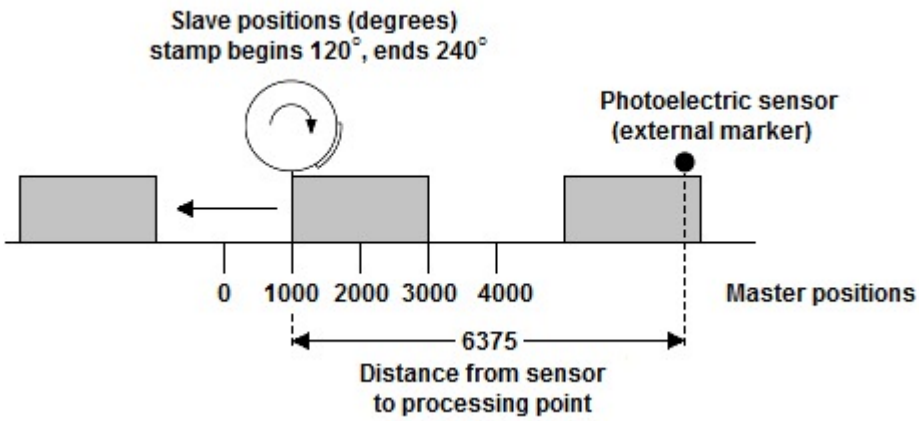
6.5.7.1 Example 2: Using the Simulator

6.5.7.2 Example 2: Large Sensor Distances

In many applications, the marker cannot be placed within one master cycle length, for example in the case of the following equipment for the production of plastic bags:



Since no markers can be installed between the slaves here, there is only a marker reader in this application; the welding station is much farther away than one master cycle length. Since the distance of the sensor is larger than one master cycle length, a buffer is provided for the marker deviation. When the marker appears, the value is entered in the buffer and read out upon the appearance of the next marker:



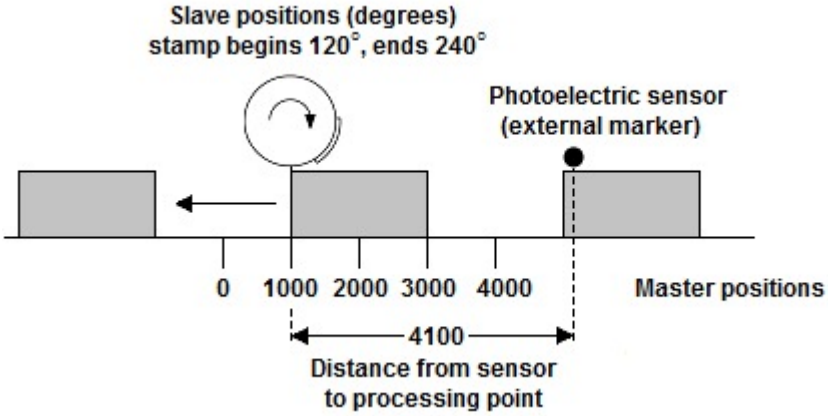
In order to assess in what area corrections may be made, you should subtract the master cycle length as often as necessary until the value is < 1 master cycle length. This is the maximum permitted distance for making corrections. In this example, it is 6375 - 4000 = 2375 and thus the same correction range as in the previous example.

6.5.7.3 Example 2: Problematic Situations

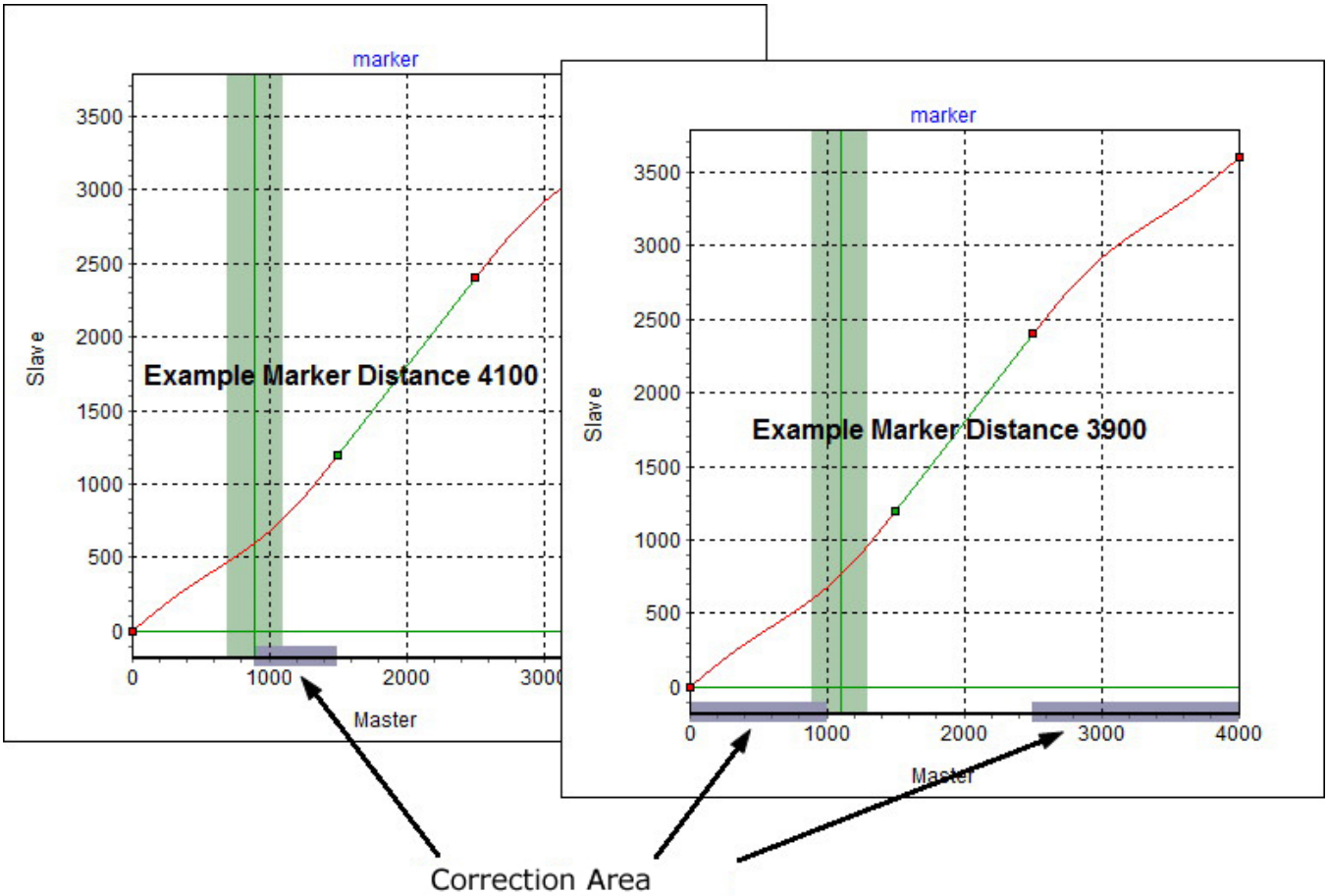
If the marker is mounted so close to the processing point that there is no time left to correct the synchronization

after the marker has been detected, then you can remedy the problem only through mechanical modification of the marker.

On the other hand, the same effect might also occur if the marker distance is larger than the master cycle length and if an insufficient distance likewise remains after the subtraction of this value, for example:



The value is entered into the buffer when the marker appears. The buffer is read out only when the next marker is recognized. This means that the marker is only "recognized" at the master position 900 and that there is little time left in our example to correct the error. It is the same effect as if the sensor had been mounted at the value (distance - master cycle length) or (4100 - 4000), respectively, i.e. only 10 mm in front of the processing point.



Thus, it would be better to install the sensor in such a way that the distance to the processing point is either smaller or substantially larger than one master cycle length; here, for example, at a distance of 3900. Then it is

possible to correct from 2500 to 1000.

Alternatively, the sensor could be installed further away, for example at a distance of 7900; this has the same effect as if the sensor had been installed at distance - master cycle length (7900 - 4000), i.e. 3900 in front of the processing point. This allows enough time to correct the synchronization.

If this cannot be done mechanically, then the values must be manipulated somewhat in order to avoid the solution with the buffer. Please proceed as follows:

Subtract a value x from the actual distance so that the distance becomes less than the master cycle length, for example 4100 - 200 = 3900. You also subtract the value x from the master position, i.e. 1000 - 200 = 800.

Enter both values in the Axis Data Tab and the CAM Curve Tab:

Master Marker Distance SYNCMPULSM = 3900
Master Marker Position = 800

Since no buffer is generated now, it would be possible to correct from 2500 to 800, for example.

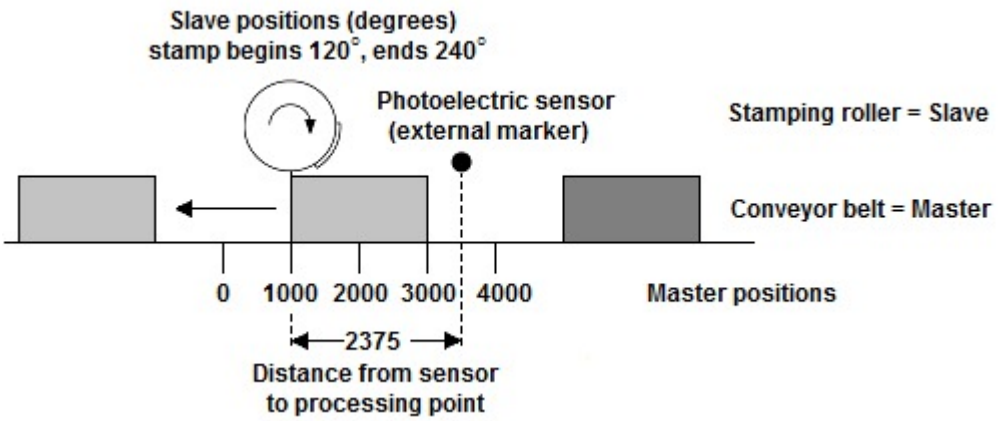
6.5.8 Example 3: Engaging and Disengaging

The following example is based on the previous example (see Example 1: Basic Synchronization). However, it assumes that the boxes are not always spaced at exactly the same distance apart. In this case, markers will be required that can signal the start of a box and correct the synchronization.

The following step-by-step explanation describes how you can adapt the curve of the previous example to this application.

The basic setup of the system is identical to the previous example except that a photoelectric sensor has been added to signal the start of a box as it passes on the conveyor belt. The photoelectric sensor will be connected to the controller by means of one of the controller's digital inputs. The ApossIDE refers to this type of sensor as an "external marker".

The distance from the photoelectric sensor to the processing point (the axis of the stamping roller) is 237.5 mm. Note that the processing point is at position 1000 MU. The sensor will set the digital input high when it is triggered. As in the previous example, the nominal spacing of the boxes will be 400 mm, however the actual spacing may vary by +/-20 mm.



Basically, the controller will operate as follows: The stamping roller will be parked in an "idle" state, waiting for a box. The arrival of the next box will be triggered by the photoelectric sensor. Once the trigger is received, the controller must synchronize the velocity and position of the stamping roller with the box by the time that the incription location arrives at the processing point. This is referred to as "engaging". Once the inscription stamping has been completed, then the stamping roller will return to its "idle" state and wait for the next box. This is referred to as "disengaging".

⚠ For this example to work correctly, the distance from the sensor to the processing point must be *less* than the Master Length of the CAM curve (i.e. the sensor must be *within* the range of the CAM curve). See Example 2: Large Sensor Distances and Example 2: Problematic Situations for how to adapt this example to other cases.

How to Edit the Curve

1. Steps 1 to 10 are the same as in the previous example.

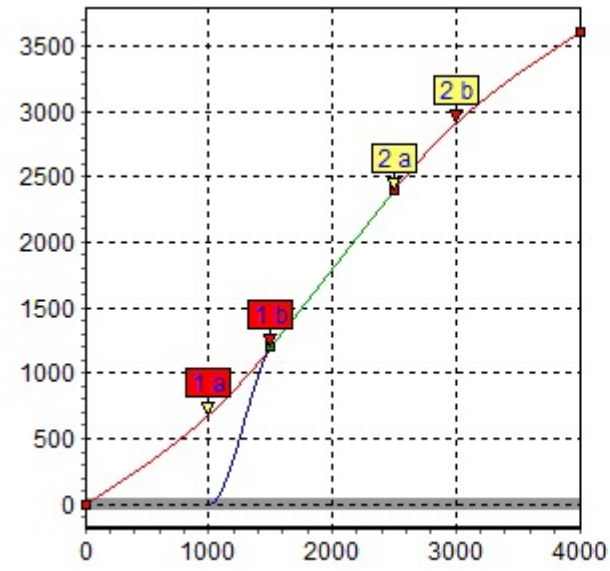
2. Define two sets of **Start Stop Points** in the Start Stop Points Tab. The first will be used for engaging and the second for disengaging. For this example, we will assume that both engaging and disengaging require 500 MU to complete. In a real application, these distances will be determined by the tuning parameters for the motor (i.e. maximum acceleration, etc.). During the engaging period, the stamping roller must synchronize itself with the conveyor belt. This must be complete by the start of the inscription at 1500 MU and so will start at 1000 MU. Similarly, disengaging must not start before the end of the inscription at 2500 MU and can take up until 3000 MU.

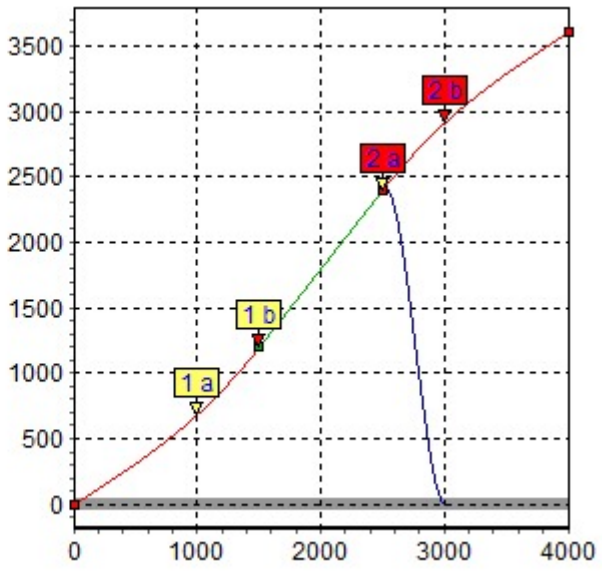
Point	Start	Stop
1	1000	1500
2	2500	3000

3. We will also assume that when the stamping roller is not actively engaging, disengaging, or stamping the inscription, then it will return to the 0 degree position and stop. This will be its "idle" state. This can be specified in the CAM Curve Tab by setting **Slave Stop Position** to 0 UU.

   Slave Stop Position = 0

   Enable ☑ **Slave Stop**, ☑ **Start Stop**, and ☑ **Engage Path** in the Display Tab. Switch to the Start Stop Points Tab and select the first pair of start stop points. The display should be similar to that shown below on the left. The horizontal gray bar shows the Slave Stop Position (i.e. the roller's "idle" position at 0 UU). The red "1a" and "1b" show the start stop points of the selected pair (i.e. the first pair). This is the engage path. The path itself is shown in blue. You can see that the path starts at 0 UU and synchronizes with the CAM curve at 1200 UU, just in time to start the printing. Select the second pair of start stop points (i.e. the disengage points), switch to the Display Tab, disable the **Engage Path**, and enable the ☑ **Disengage Path**. The display should be similar to that shown below on the right. The red "2a", and "2b" show the disengage points and the blue curve is the actual disengage path. You can see that the roller starts to disengage at 2400 UU, just after the printing finishes, and then stops at the "idle" position of 0 UU.

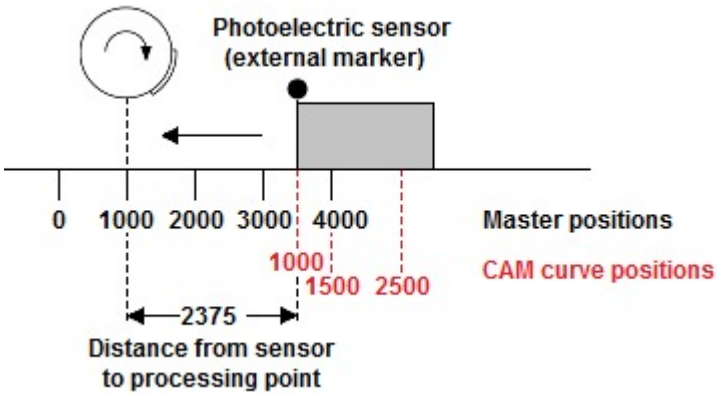




> Look carefully at the engage and disengage paths. When engaging, the roller will rotate forward to go from 0 to 120 degrees. But when disengaging, the roller will rotate *backwards* from 240 to 0 degrees! This is a very different behaviour than the previous example (see Example 1: Basic Synchronization) where the roller is always rotating forward. This may be an important consideration in your own application and you may want to implement the original CAM curve slightly differently in order to have the roller rotate forward when returning to 0 degrees.

4. When the photoelectric sensor (external marker) detects the front edge of a box, then this position will correspond to master position 1000 MU *in the CAM curve* (i.e. where the front edge of the box is positioned in the CAM curve). This is shown below in red. Remember that the inscription will start at 1500 MU. In the CAM Curve Tab, set the **Master Marker Position** to 1000 MU.

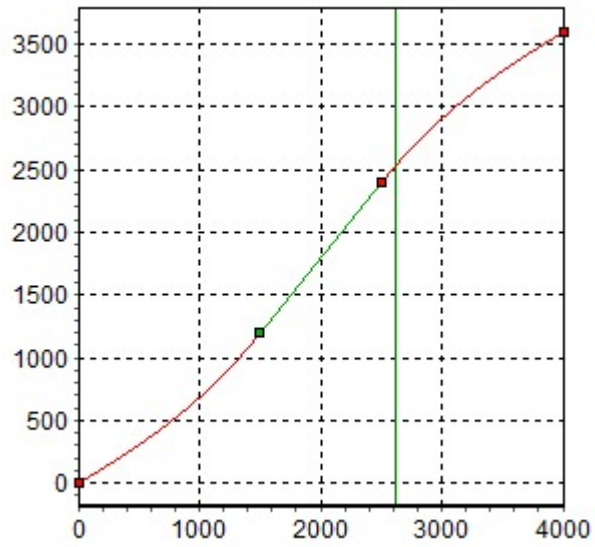
   Master Marker Position = 1000



   The distance from the processing point (1000 MU) to the photoelectric sensor is 237.5 mm or 2375 MU. This is the distance offset that is required to match the original master positions (shown black above) to the corresponding CAM curve positions (shown red above). Enter this value as the **Master Marker Distance** in the Axis Data Tab.

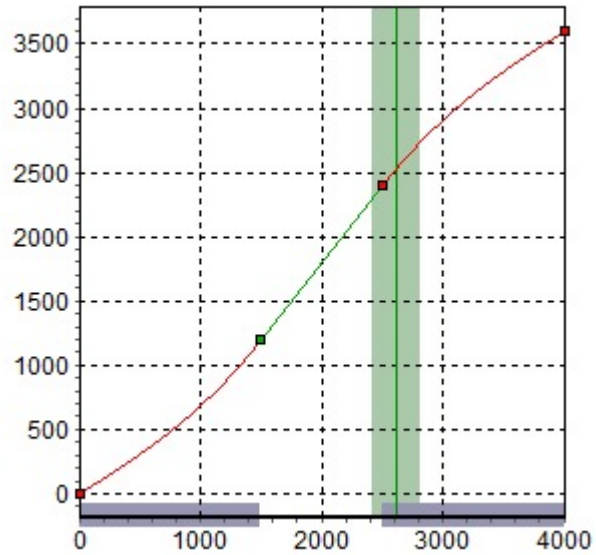
   Master Marker Distance: SYNCMPULSM = 2375

Enable ☑ **Master Marker** in the Display Tab. This will display the master marker as a vertical green line as is shown below.



5. Since the spacing of the boxes can vary by +/-20 mm, set the **Master Marker Tolerance** to 200 MU in the Axis Data Tab. The photoelectric sensor is an external sensor and the signal goes high when triggered, so set the **Master Marker Type** to 2. The display should look similar to that shown below. The green background area shows the tolerance window for the appearance of the master marker.

   Master Marker Tolerance: SYNCMWINM = 200
   Master Marker Type: SYNCMTYPM = 2

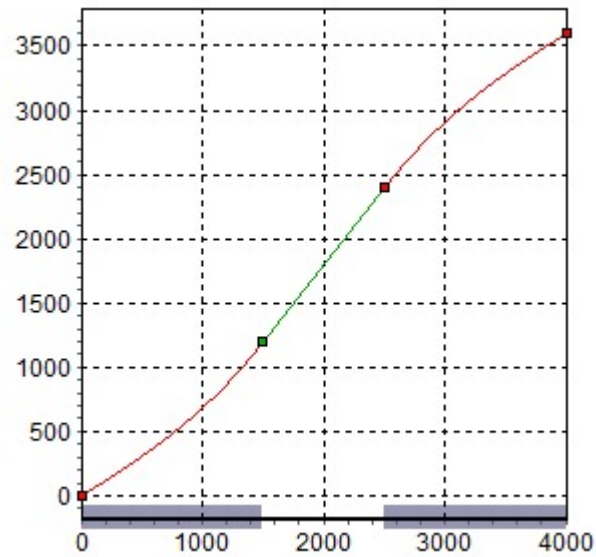


6. Take a look at the diagram above in order to determine when the correction of the synchronization may begin at the earliest and when it must be finished. The vertical green line indicates the master position where the marker is recognized, the light green area shows the tolerance window for the appearance of the master marker. At the earliest, the correction may begin when the printing of a box has been completed since any change of velocity during the printing process would damage the box. The correction must be finished completely when the next cardboard box reaches the processing point. In this example, the master

positions at the end and beginning of a box are quite suitable. Enter the values in the CAM Curve Tab.

Correction Start = 3000
Correction End = 1000

Enable ☑ **Correction** in the Display Tab. The area of correction will be shaded in blue at the bottom of the diagram.



7. Enable ☑ **Velocity** and ☑ **Vel. Limit**. Verify that the velocity of the roller is within the limits.
8. Disable the velocity display and enable ☑ **Acceleration** and ☑ **Acc. Limit**. Verify that the acceleration of the roller is within the limits.
9. You will see the number of array elements required for this CAM curve in the title bar. Remember this number as you will need it later during programming.

10. The CAM curve is now complete. You should save the CAM curve and settings as a zbc file using **File** → **Save**.
11. Download the CAM curve to the controller using **Parameters** → **Restore from file**. Note that only the DIM Arrays need to be downloaded. Also, use the **Create Include File** button in this dialog to create a Parameter Include File. You will need this later during programming (see Initializing Temporary Parameters).

Program Sample

The DIM instruction must appear first in your program:

```
// The CAM Curve will be stored in a DIM array and DIM arrays must
be declared
// first in application programs.  The number of elements in the
array (43)
// comes from the title bar of the CAM Editor.
```

```
DIM marker[43]

// The axis parameter settings should be set before the rest of the
application
// program starts.  These values are taken from the Configuration
ZBC file
// that was saved by the CAM Editor.
//
// Note: After axis parameters are set, the VEL, ACC, and DEC
commands must
//       be executed in order to allow the controller to start using
the
//       new axis parameters.  100 will allow the controller to
drive the
//       motor at up to 100% of its rated values.

sysvar[0x01230002] = 500         // VELMAX
sysvar[0x01230018] = 25          // POSFACT_Z
sysvar[0x0123001B] = 9           // POSFACT_N
sysvar[0x01230020] = 500         // RAMPMIN
sysvar[0x01230032] = 2048        // SYNCFACTM
sysvar[0x01230033] = 55          // SYNCFACTS
sysvar[0x0123003B] = 2375        // SYNCMPULSM
sysvar[0x0123003C] = 0           // SYNCMPULSS
sysvar[0x01230045] = 200         // SYNCMWINM

VEL 100
ACC 100
DEC 100

// This example is slightly different from the Stamping.m example.
The
// Stamping.m example required the conveyor belt to be stopped
before we
// could synchronize the stamping roller.  This example only
requires
// that no boxes arrive and trigger the sensor before we have
initialized
// the stamping roller.  However, to keep things simple, we will
still
// treat digital input 1 as indicating whether or not the conveyor
belt
// is moving.

if (IN 1) then
    PRINT "ERROR: The conveyor belt is already running."
    PRINT "       Please stop the conveyor belt (Input 1) and then
```

```
restart this program."
    EXIT
endif

// SIMULATOR ONLY: Set the encoder type for the stamping roller to
9.
//                 This is a special encoder type that simply copies
//                 the requested encoder position to the actual
encoder
//                 position.  This makes it look as if the stamping
//                 roller is moving even though a motor does not
exist.

SET ENCODERTYPE 9

// Set the master encoder to be an incremental encoder (type 0)
connected
// to encoder input 4.

SET MENCODERTYPE 40

// Set the master marker type.  This will set the master marker to
use
// digital input 3 and to trigger on a rising edge.

SET SYNCMTYPM 32

// SIMULATOR ONLY: This sets marker type for the virtual master
encoder
//                 so that it also uses digital input 2 and triggers
on
//                 a rising signal.

SET VMENCMTYP 22

// Turn the stamping roller motor on.  It will not move unless we do
this.

MOTOR ON

// The stamping roller uses an incremental encoder so it has no
physical
// "home" position.  However, before the system is first started,
the
// stamping roller should have been manually placed into the
position
// that corresponds to 0 degrees.  Define this as the "origin" of
```

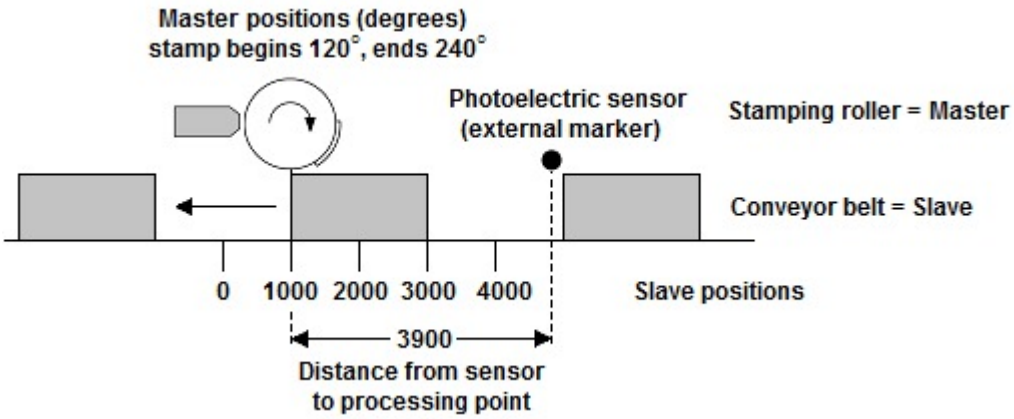
```
the
// roller's encoder (i.e. the position where the encoder reads 0).

DEFORIGIN

// Set the stamping curve.

SETCURVE marker

// When a box triggers the sensor, this corresponds to the position
1000 MU
// on the CAM curve.  When this happens, the processing point is
237.5 mm
// (i.e. SYNCMPULSM) before this point.  Calculate this point and
use it to
// define the initial position of the conveyor belt (i.e. the
master).  This
// position may be negative.

dist = GET SYNCMPULSM
DEFMCPOS (1000-dist)

// Set the starting behaviour for synchronization with marker
correction.
// The 2000 means the master encoder will be ignored until the next
sensor
// trigger is received.  We can do this since the DEFMCPOS command
(above)
// will already have set the master position correctly to coincide
with
// the first trigger event.

SET SYNCMSTART 2000
set synctype 3

// Change into CAM Mode with master marker correction.  The 0 means
that
// we will stay in CAM Mode until we explicitly leave CAM Mode.

SYNCCMM 0

// Start synchronization.  The 0 means "engage roller immediately
with the
// set maximum velocity".  This does not cause any movement since
the master
// is idle and (hopefully) the sensor has not been triggered.  The
stamping
// roller will start to move when the conveyor starts to move.
```

```
SYNCCSTART 0

// Now the master (conveyor) can be started.  Wait for digital input
1 to
// go high.  This will indicate that the conveyor belt has been
turned on.

PRINT "Ready.  Please start the conveyor belt (Input 1)."

while (not IN 1) do
endwhile

PRINT "Running...  Input 2 signals a box arriving."

// SIMULATOR ONLY: Enable the virtual master.  These commands will
make it
//                 appear as if the conveyor belt is moving.  The
PULSVEL
//                 value of 37236 counts/sec will drive the conveyor
belt
//                 at a velocity of 400 mm/sec (i.e. 60 boxes/sec).
The
//                 PULSACC value of 10000 will cause the conveyor
belt to
//                 take "a few" box lengths to speed up before the
constant
//                 velocity of 400 mm/sec is reached.  Note that the
virtual
//                 master will override the "SET MENCODER 40"
command done
//                 above so encoder 4 will NOT actually be used by
the
//                 Simulator.

PULSACC 5000
PULSVEL 1000

// Wait for digital input 1 to go low.  This will indicate that the
conveyor
// belt has been turned off.  Once this happens, we can turn off the
stamping
// roller motor.
//
// Note: We are testing for the digital input to go low very quickly
after it
//       has gone high.  If the switch driving the digital input is
```

```
"rough" or
//       "bounces", then it may appear as if the input goes on-off-
on.  So wait
//       a short while (100 ms) for the input to stabilize before
starting to
//       test it again.

DELAY 100

while (IN 1) do
endwhile

// The conveyor belt has been turned off.  Turn off the stamping
roller motor.

MOTOR OFF

// SIMULATOR ONLY: Disable the virtual master.

PULSACC 0
PULSVEL 0

PRINT "Finished.  Stamping roller has been stopped."
```

6.5.9 Example 4: Slave Synchronization with Marker

In the following example, the conveyor belt is the slave and the stamp roller the master, since the take-up and delivery of the dye must be continuous for a uniform printing process. A maximum of 20 cardboard boxes are transported on the conveyor belt per minute. The distance of the boxes is not larger than one master cycle length. Stamp roller and box must run in sync during the printing operation:



In contrast to the synchronization with master marker correction, here the slave position is being corrected instead of the curve.

How to Edit the Curve

1. Set the controller with the required parameters and save these user parameters with **Parameters** → **Save to**

file with the extension “zbc”.

2. Open this zbc file with **File** → **Open**. This will automatically start the CAM Editor.

3. Determine the gearing factor of the master in MU units.

   Gearing factor = 5/1

   Encoder resolution (Incremental encoder) = 500

   One revolution of the roller is 360 degrees. We are going to work with a resolution of 1/10 degrees. This means that we are dividing one revolution of the roller into 3600 work units:
   Scaling factor = 3600

   $$\frac{\text{Gearing Factor} * \text{Encoder Resolution} * 4}{\text{Scaling Factor}} \text{ qc} = 1 \text{ MU}$$

   Enter these whole number values in the Axis Data Tab:

   Sync factor Master: SYNCFACTM = 25
   Sync factor Slave: SYNCFACTS = 9

4. Enter the gearing factor of the slave in uu units:

   The input should be possible in 1/10 mm resolution.

   The drive is connected with the conveyor belt by means of a gearing of 25:11; i.e. the motor makes 25, the drive pulley 11 revolutions.
   Gearing factor = 25/11

   Incremental encoder directly on the master drive; encoder resolution = 4096

   The drive pulley has 20 teeth/revolution, 2 teeth correspond to 10 mm, thus 1 revolution corresponds to = 100 mm conveyor belt feed or 1000/10 mm.
   Thus, the scaling factor is 1000

   $$\frac{\text{Gearing Factor} * \text{Encoder Resolution} * 4}{\text{Scaling Factor}} \text{ qc} = 1 \text{ UU}$$

   Enter these values in the Axis Data Tab:

   User factor numerator: POSFACT_Z = 2048
   User factor denominator: POSFACT_N = 55

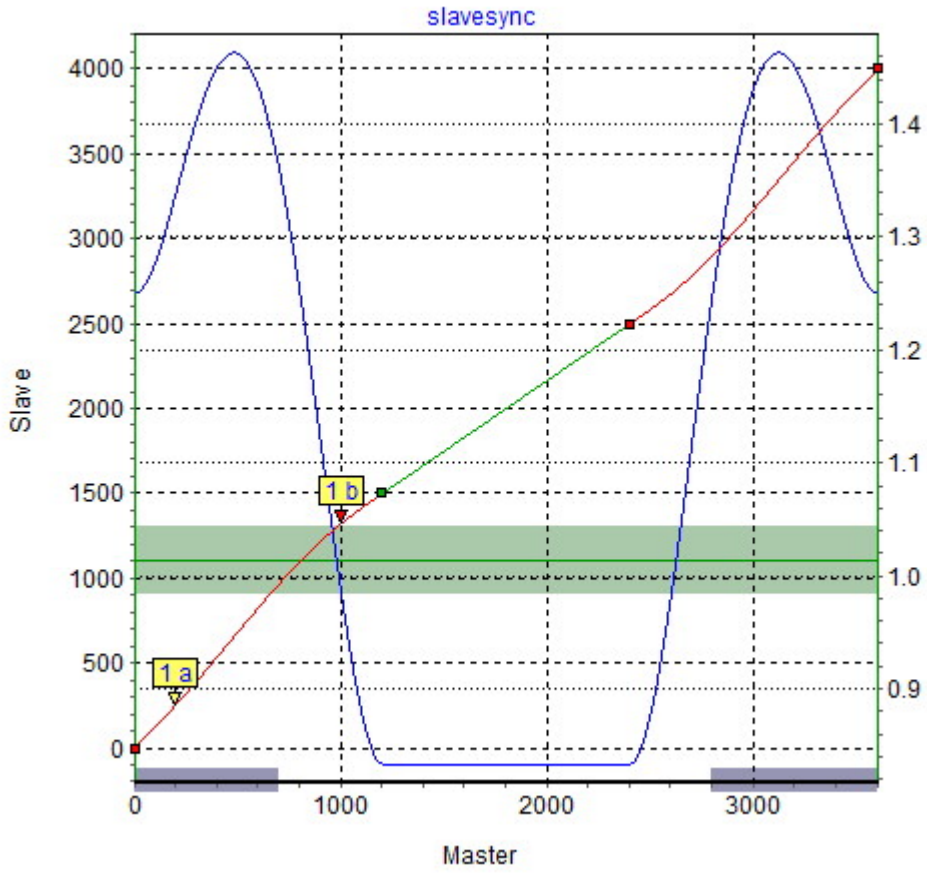
5. Define **Fixpoints** for the roller (slave) and the conveyor belt (master). The function **Snap on Grid** should be activated.

Point	Master	Slave	Type	Λ
1	0	0	C	
2	1200	1500	C	
3	2400	2500	C	
4	3600	4000	-	

6. Master and slave must run synchronously with the same velocity between the master positions 1200 to 2400. For this, you need to have a straight line that is determined with two tangent points. Double-click in the column **Type** for the fixpoint on the left at position 1200 and select **Tangent**.

Point	Master	Slave	Type	Λ
1	0	0	C	
2	1200	1500	T	
3	2400	2500	C	
4	3600	4000	-	

Activate the diagram of the ☑ **Velocity** to see the corresponding velocity curve as it is shown in this figure:



7. Enter the **Cycles/min Master** = 20 in the CAM Curve Tab. This is the (maximum) number of cardboard boxes that can be processed per minute.

8. Verify whether the acceleration of the slave is within the limit. For this purpose, you must activate the illustration of the **Acceleration** and of the **Acc. Limit.**

9. Define in the list **Start-Stop Points** with some safe distance in order to start the synchronization at the beginning. Engaging should take place between 20 and 100 degrees because it must be completed at 120 degrees.

Point	Start	Stop
1	200	1000

10. In the CAM Curve Tab, define the position where the conveyor belt should stop if no other **Slave Stop**

Position is being defined in the program:

The conveyor belt should always stop in position 0: **Slave Stop Position** = 0

11. The photoelectric beam (external marker) has a distance of 390 mm from the processing point (= stamp touches the box) and detects the beginning of the box (corresponds to slave position 1000). Thus, the marker distance is 3900. Enter this value in the Axis Data Tab and define the permitted tolerance for the appearance of the markers and the external marker type = 2 for the slave:

    Slave Marker
    Distance SYNCMPULSS = 3900
    Tolerance SYNCMWINS = 200
    Type SYNCMTYPS = 2

    Enter the value in the CAM Curve Tab:
    Slave-Marker-Position = 1000

12. Take a look at the curve profile in order to determine when the correction of the synchronization may begin at the earliest and when it must be finished. The green horizontal line indicates the master position where the marker is recognized, the light green area shows the tolerance window for the appearance of the master marker. (See PC Software for color.)

    At the earliest, the correction may begin when the printing of a cardboard box has been completed, since any change of velocity during the printing process would damage the printing stamp and/or the box. Also, the correction must have been completed in its entirety when the next box arrives at the processing point. In this example, the slave positions at the end and beginning of a box are quite suitable. Enter the values in the CAM Curve Tab:

    Correction Start = 2800
    Correction End = 750

13. Verify whether the velocity and acceleration of the slave remain within the limit. For this purpose, you must activate the illustration of the **Velocity** and of the **Vel.-Limit** and then the illustration of the **Acceleration** and of the **Acc.-Limit.**

14. **File** → **Save** the zbc file.

15. Load the zbc file with the modified parameters and the - automatically generated - curve arrays into the controller by means of **Parameters** → **Restore from File**.

Program Sample

In order to determine the master position, a switch on the master is required that indicates the zero position. In order to put the slave into the correct position, it will be moved forward to the photoelectric beam. This corresponds to the beginning of the box = 1000. Then the slave will be moved further by 2900 (= marker distance 3900-1000); thus, the slave is exactly in front of the processing point with the beginning of the cardboard box 1000 i.e. at slave position 0.

```
DIM slavesync[108]          // Number of elements from the title bar
of the CAM-Editor
HOME                        // Slave axis performs a home run (switch
for zero position on top)
                            // Afterwards, the slave will be in the
zero position (0 degrees)
```

```
                            // (Omitted if an absolute encoder is
used)
DEFMCPOS 0                  // Curve starts at master position 0
SET SYNCMSTART 2000         // Counting of the master pulse does not
begin
                            // until the next edge comes from the
sensor
SETCURVE slavesync          // Set curve for the slave
synchronization
                            // Go to start
CSTART
CVEL 10                     // Go forward slowly until photoelectric
beam appears
oldi = IPOS                 // oldi = last marker position of the
slave
WHILE (oldi == IPOS) DO     // Wait until box is detected
ENDWHILE
POSA (IPOS + 2900)          // Move box forward by 2900
SYNCCMS 0                   // Synchronize in CAM-Mode
SYNCCSTART 1                // Engage with start-stop point pair
```

6.5.10 Example 5: CAM Box

How a CAM Box with the ApossIDE operates:

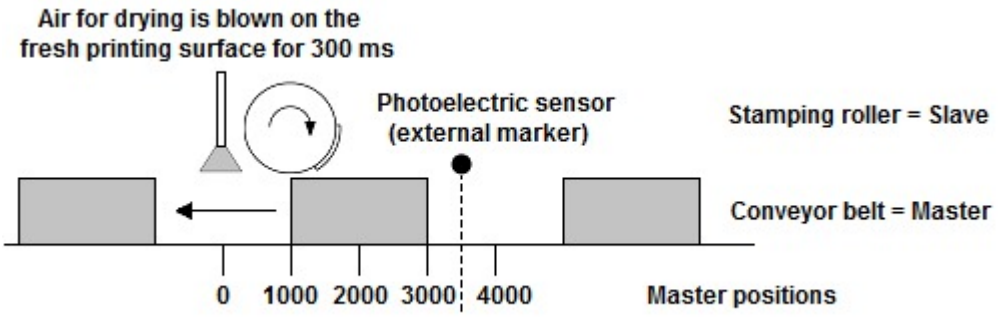
The mechanical camshaft is also reproduced by one (or more) curves. In order to realize CAM box, it must be possible to engage and disengage the slave at specific master positions over and over again.

This is possible with the ApossIDE with the position interrupt commands ON MAPOS .. GOSUB and ON APOS .. GOSUB. It is possible to call up a subprogram whenever a defined master position has been passed (both in the positive or negative direction).

It is possible to realize many applications that are typical for the packaging industry in connection with a curve profile in which several curves have been defined for engaging and disengaging.

Example of a CAM box

After a cardboard box has been printed, the fresh print is to be dried immediately in the air stream:



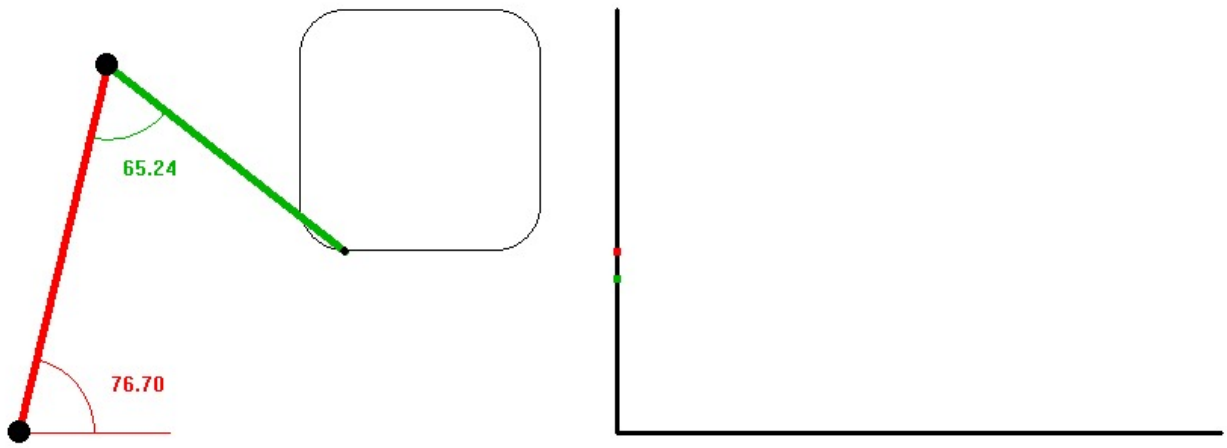

```
// Call up a subprogram when the master position 2500
// is passed in positive direction.
```

```
ON MCPOS 2500 GOSUB drier

SUBMAINPROG
    SUBPROG drier
    OUT 1 1          // Turn on drier
    DELAY 300        // Dry for 300 ms
    OUT 1 0          // Turn off drier
    RETURN
ENDPROG
```

6.6 Working with Kinematics

The zub motor controller provides significant functionality to support robotic applications. Typically, these applications require the synchronization of multiple axes to move a tool along a specific path through space. For example, the 2-dimensional space of a pen in an XY plotter or the 3-dimensional space of a tool in a numerical milling machine, a coordinate measuring machine, a "pick-and-place" machine, etc.

Paths are generally specified using a familiar 3-dimensional Cartesian coordinate system (X,Y,Z). The controller must then follow the path through space and convert these spatial coordinates into the corresponding motor positions which are then used to drive the motors. Shown below for example, the arms of a SCARA (Selective Compliance Assembly Robot Arm) are positioned using angles. Technically, this process of working backwards from (X,Y) coordinates to machine angles, is termed "inverse kinematics" and it can sometimes be a difficult mathematical problem. However, the ApossIDE provides built-in algorithms that completely eliminate this problem for the user, thus allowing the user to work directly in the original Cartesian coordinate system without concern for the details of the inverse kinematics.



See Multi-axis Kinematics for a list of the robot architectures supported by the ApossIDE.

Please read the following topics for background information required before programming a robotic application:

- Basic Coordinate Systems
- Transformations
- Working with Points
- Dynamic Systems

All kinematics features can be controller dependent and is listed in the Controller Dependent Feature List.

6.6.1 Multi-axis Kinematics

The ApossIDE provides the inverse kinematics for the following robot architectures:

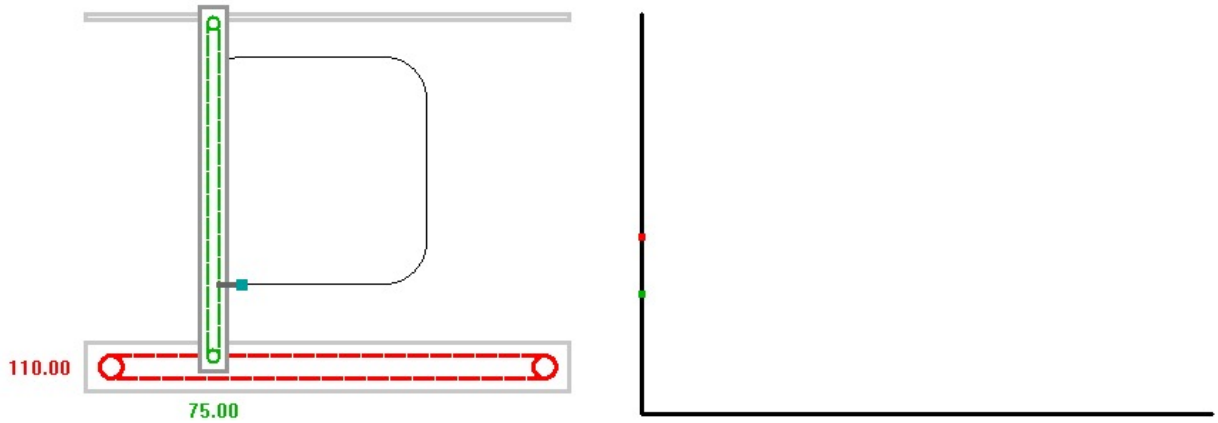
- Cartesian
- H-Bot / CoreXY
- SCARA
- Dual SCARA
- Delta

Examples of how to use each of these types of kinematics can be found by following the above links.

The kinematics to be used in an application, must be specified by the application program using the KinematicsSetup command. This command will return a "handle" value. This handle must then be passed to any subsequent command that needs to refer to that kinematics.

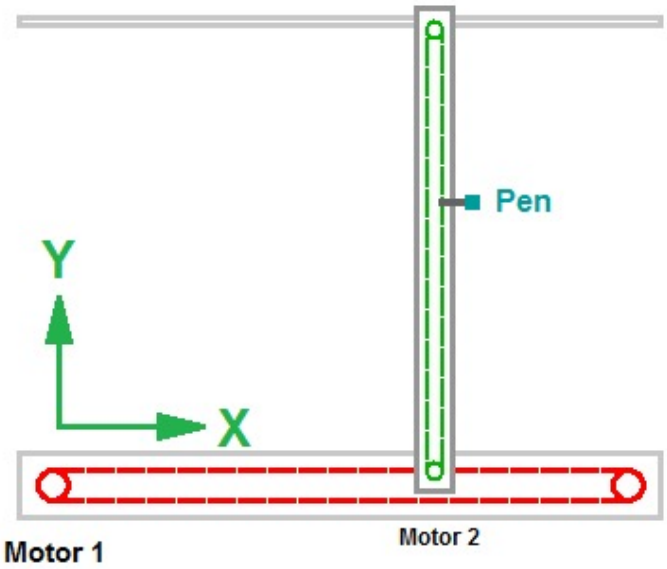
6.6.1.1 Cartesian Kinematics

The basic design and behaviour of a Cartesian robot is shown in the diagram below. This is a very simple linear design where each motor drives an orthogonal axis. Generally, each motor will need to move all "following" motors. For example, Motor 1 moves the entire gantry containing Motor 2. The ApossIDE supports both 2-dimensional and 3-dimension versions of Cartesian kinematics. The 2-dimensional design is typically how pen plotters are constructed. Large gantry cranes are examples of 3-dimensional systems.



Machine Coordinate System

The Cartesian kinematics supported by the ApossIDE requires either two or three motors, depending on whether a 2-dimensional or 3-dimensional system is being driven. For the purposes of this description, they will be called "Motor 1" (on the base driving the X axis) and "Motor 2" (on the gantry driving the Y axis). The X axis of the Machine Coordinate System (see Basic Coordinate Systems) will point towards the right, away from Motor 1. The Y axis will point up, away from Motor 1. If a Z axis is present, then it will point up out of the diagram, away from Motor 1 (i.e. oriented as a right-handed coordinate system). The coordinate system origin will not be defined relative the machine itself. Instead, the origin will be the position of the end effector when the motors are at their zero-positions. The units of the Machine Coordinate System will be identical to the units of the Kinematics Coordinate System (see below).



Kinematics Coordinate System

The Kinematics Coordinate System consists of either two or three coordinates, one for each of the motors. The coordinate for Motor 1 is oriented so that increasing values will move the end effector towards the right, parallel with the X axis. The coordinate for Motor 2 is oriented so that increasing values move it upwards, parallel with the Y axis. If a Z axis is present, then a coordinate for Motor 3 will exist and will be oriented so that increasing values move up out of the diagram, parallel to the Z axis. The coordinate system origin will be the position of the end effector when the motors are at their zero-positions. The units of the Kinematics Coordinate System will be identical to the user units of the motors. Motor directions and the size of user units may be set in the normal way for each axis (see How to Set Up Resolution and Gear Parameters). Note that the encoder resolution of each motor may be different, but the gearing factors *must* be set so that user units are the same for all motors.

Calibrating User Units

The gearing factor for a motor is simply the number of quadcounts required to move the belt one user unit (i.e. quadcounts per user unit). It is very important to have the gearing factors for the two motors set as accurately as possible. Errors in the gearing factors will result in the end effector not moving the expected distance, or in the expected direction, in the real world.

Often the gearing factor can be determined directly from the design parameters of the system. For example, if the number of quadcounts in one rotation of the motor and the circumference of the pulley in user units are both known, then the gearing factor can be determined directly. However in many cases, this may not be true. The pulley circumference may not be know or may be difficult to measure. Pulleys and/or belts may be replaced, which can affect how far the belt moves in one rotation of the motor. Perhaps the mechanics does not even use pulleys and belts. In these cases, the correct gearing factors can be determined by using the following calibration procedure:

1. Manually position the end effector so that it is free to move towards the right (corresponding to Motor 1). Mark this position.
2. Move Motor 1 *forward* a known number of quadcounts (i.e. "n"), keeping Motor 2 fixed. Note that it is not important to move the motor an integral number of rotations; any distance is acceptable as long as the quadcount number is known. This should move the end effector towards the right. Mark this position.
3. Measure the straight-line distance (i.e. "d") between the two marks as accurately as possible *in user units*. Note that as the distance moved is increased, the error in the gearing factor calculation is reduced. So the distance moved should chosen to be as large as is reasonablly possible.
4. The gearing factor for Motor 1 is then the value "n/d".
5. Repeat this same procedure with Motor 2 (positioning the end effector so that it is free to move upwards). Note that the gearing factors may be different for Motor 1 and Motor 2. This does not affect the correct behaviour of the kinematics.

Kinematics Axes and Parameters

The "axis" array required by the KinematicsSetup command, contains either two or three mandatory elements, depending on whether a 2-dimensional or 3-dimensional system is being driven. The first value is the axis module number of the controller axis associated with Motor 1. The second is the axis module number of the axis associated with Motor 2. The third (if needed) is the axis module number of the axis associated with Motor 3. Optional axes may be specified after that.

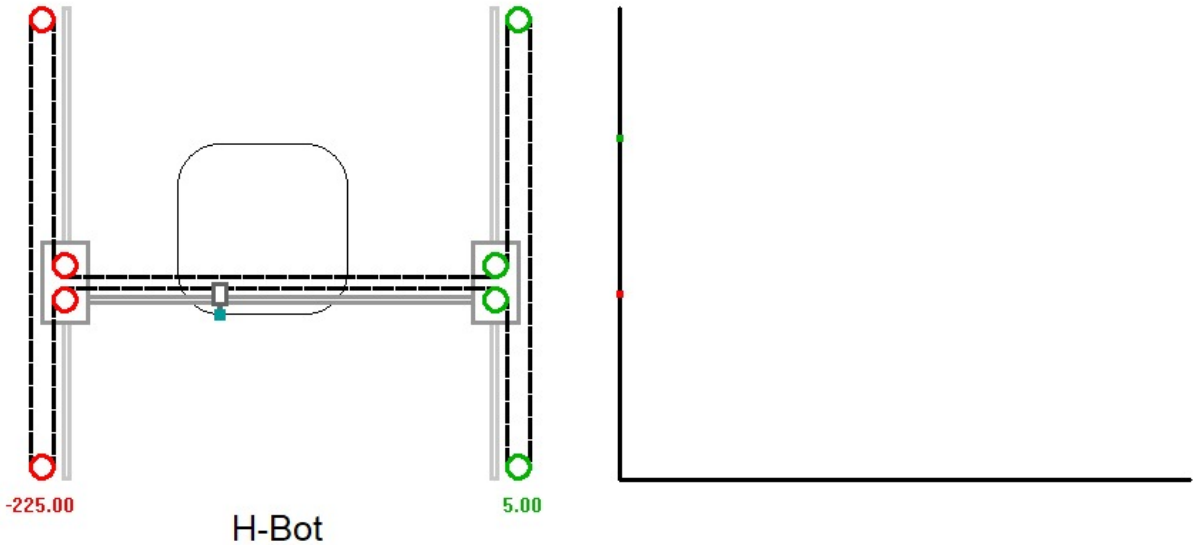
The Cartesian kinematics requires only one parameter: the number of dimensions.

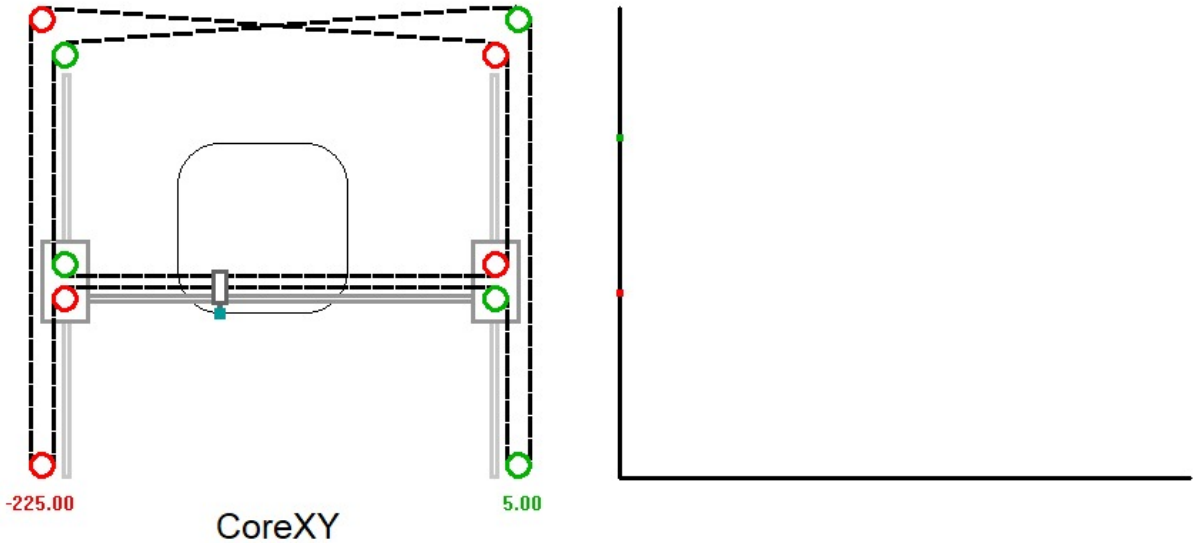
Example

```
long   hndl, axis[3];
double prms[1];

axis[0] = 1;    // Motor 1 uses controller axis module 1.
axis[1] = 0;    // Motor 2 uses controller axis module 0.
axis[2] = 2;    // Motor 3 uses controller axis module 2.

prms[0] = 3;    // 3-dimensional (X,Y,Z) kinematics.

hndl = KinematicsSetup("Cartesian", axis, prms);   // Select the Cartesian kinematics.
```

6.6.1.2 H-Bot/CoreXY Kinematics

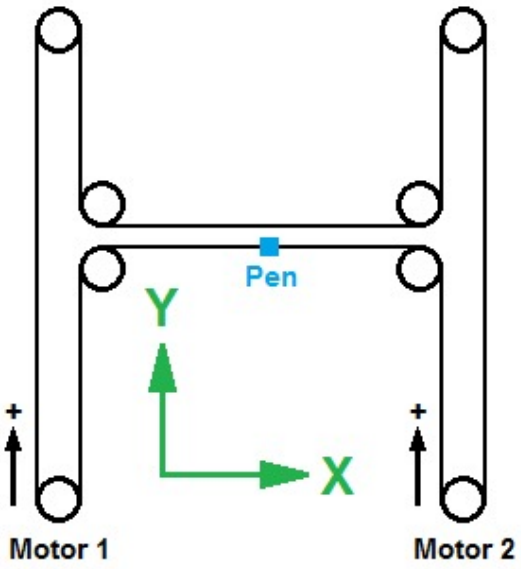
The basic design and behaviour of H-Bot and CoreXY robots is shown in the diagrams below. These are very simple 2-dimensional linear designs similar to Cartesian robots. However, they differ from Cartesian robots in that both motors are fixed, rather than having one motor move the weight of the second motor. This may allow the machine to use less powerful motors.





Machine Coordinate System

The H-Bot and CoreXY kinematics supported by the ApossIDE requires two motors. For the purposes of this description, they will be called "Motor 1" (on the left) and "Motor 2" (on the right). Motor 1 will drive the end effector in the "down and left" direction and Motor 2 will drive it in the "up and left" direction. The X axis of the Machine Coordinate System (see Basic Coordinate Systems) will point in the direction from Motor 1 to Motor 2. The Y axis will point up, away from both motors. The coordinate system origin will not be defined relative the machine itself. Instead, the origin will be the position of the end effector when the two motors are at their zero-positions. The units of the Machine Coordinate System will be identical to the units of the Kinematics Coordinate System (see below).



Kinematics Coordinate System

The Kinematics Coordinate System consists of two coordinates, one for each of the two motors. The coordinate for Motor 1 is oriented so that increasing values (clockwise rotation) will move the end effector in the "down and left" direction. The coordinate for Motor 2 is oriented so that increasing values (clockwise rotation) move it in the "up and left" direction. The coordinate system origin will be the position of the end effector when the two motors are at their zero-positions. The units of the Kinematics Coordinate System will be identical to the user units of the motors. Note that user units of the motors are measured according to the distance that the *belt* moves and *not* to the distance that the end

effector moves (see "Calibrating User Units" below). Motor directions and the size of user units may be set in the normal way for each axis (see How to Set Up Resolution and Gear Parameters). Note that the encoder resolution of each motor may be different, but the gearing factors *must* be set so that user units are the same for both motors.

Calibrating User Units

The gearing factor for a motor is simply the number of quadcounts required to move the belt one user unit (i.e. quadcounts per user unit). It is very important to have the gearing factors for the two motors set as accurately as possible. Errors in the gearing factors will result in the end effector not moving the expected distance, or in the expected direction, in the real world.

Often the gearing factor can be determined directly from the design parameters of the system. For example, if the number of quadcounts in one rotation of the motor and the circumference of the pulley in user units are both known, then the gearing factor can be determined directly. However in many cases, this may not be true. The pulley circumference may not be know or may be difficult to measure. Pulleys and/or belts may be replaced, which can affect how far the belt moves in one rotation of the motor. In these cases, the correct gearing factors can be determined by using the following calibration procedure:

1. Manually position the end effector so that it is free to move in the "down and left" direction (corresponding to Motor 1). Mark this position.
2. Move Motor 1 *forward* a known number of quadcounts (i.e. "n"), keeping Motor 2 fixed. Note that it is not important to move the motor an integral number of rotations; any distance is acceptable as long as the quadcount number is known. This should move the end effector down and left at a 45 degree angle. Mark this position.
3. Measure the straight-line distance (i.e. "d") between the two marks as accurately as possible *in user units*. Note that as the distance moved is increased, the error in the gearing factor calculation is reduced. So the distance moved should chosen to be as large as is reasonablly possible.
4. From the geometry of the system, it can be seen that when the belt of Motor 1 moves forward "b" user units (with Motor 2 fixed), the end effector will move left "b/2" user units and down "b/2" user units. This means the straight-line distance moved will be:

   $d = \text{sqrt}((b/2)^2 + (b/2)^2) = \text{sqrt}(b^2/2) = b/\text{sqrt}(2)$
   $b = d \times \text{sqrt}(2)$.

5. The gearing factor for Motor 1 is then the value "n/(d × sqrt(2))".
6. Repeat this same procedure with Motor 2 (positioning the end effector so that it is free to move up and left). Note that the gearing factors may be different for Motor 1 and Motor 2. This does not affect the correct behaviour of the kinematics.

Kinematics Axes and Parameters

The "axis" array required by the KinematicsSetup command, contains two mandatory elements: The axis module number of the controller axis associated with Motor 1, followed by the axis module number of the axis associated with Motor 2. Optional axes may be specified after that.

No parameters are required for the H-Bot/CoreXY kinematics.

Example

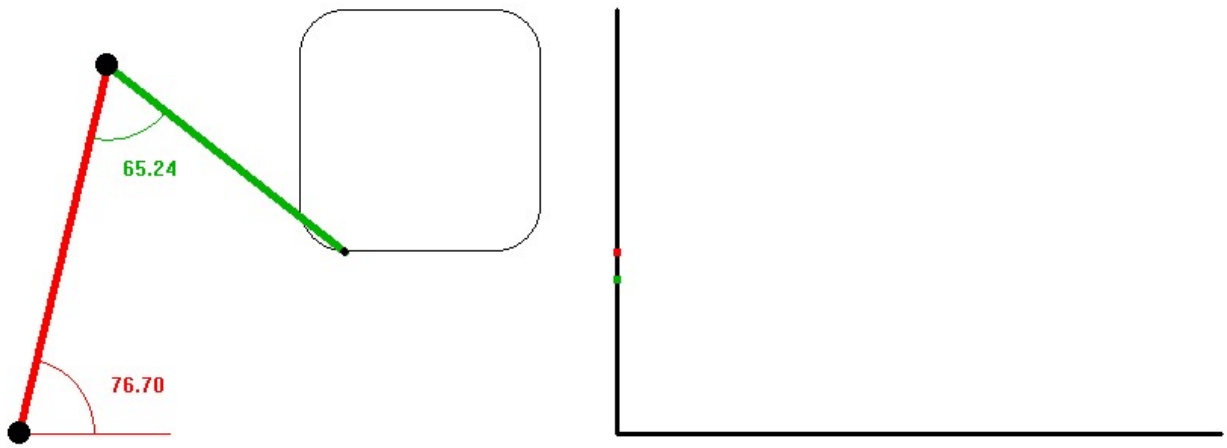
```
long hndl, axis[2];

axis[0] = 0;    // Motor 1 uses controller axis module 0.
axis[1] = 1;    // Motor 2 uses controller axis module 1.

hndl = KinematicsSetup("Hbot", axis);   // Select the H-Bot/CoreXY kinematics.
```

6.6.1.3 SCARA Kinematics

The basic design and behaviour of a SCARA is shown in the diagrams below. This is very similar to a human arm. It uses one fixed motor to adjust the angle of a primary arm. A second motor is attached to the end of the primary arm and adjusts the angle of a secondary arm. The end effector can be positioned anywhere within quite a large 2-dimensional

area. This type of robot has a small mounting "footprint" and it is able to reach fairly easily into confined spaces.

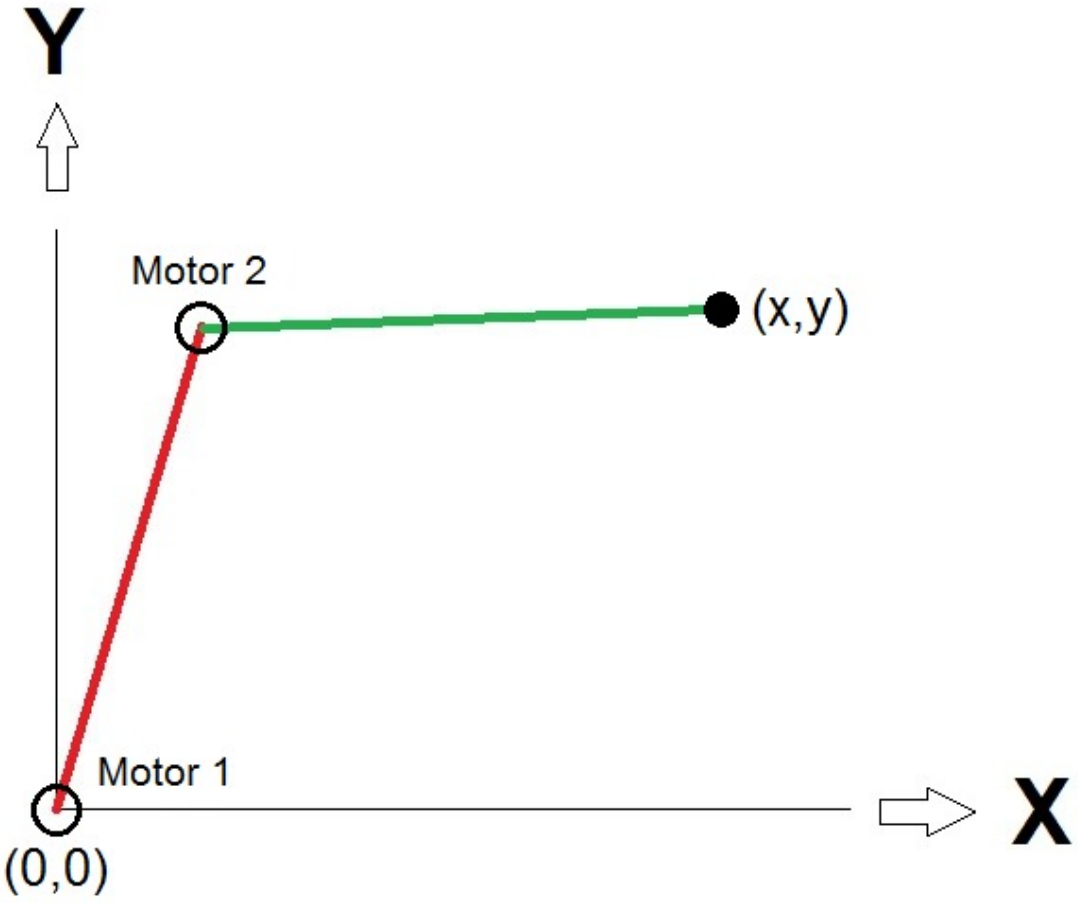
The ApossIDE supports both 2-dimensional and 3-dimension versions of SCARA kinematics. However, in the 3-dimensional version, the Z axis behaves as a simple linear axis that is orthogonal to the X and Y axes (i.e. similar to the Z axis in Cartesian kinematics).



Machine Coordinate System

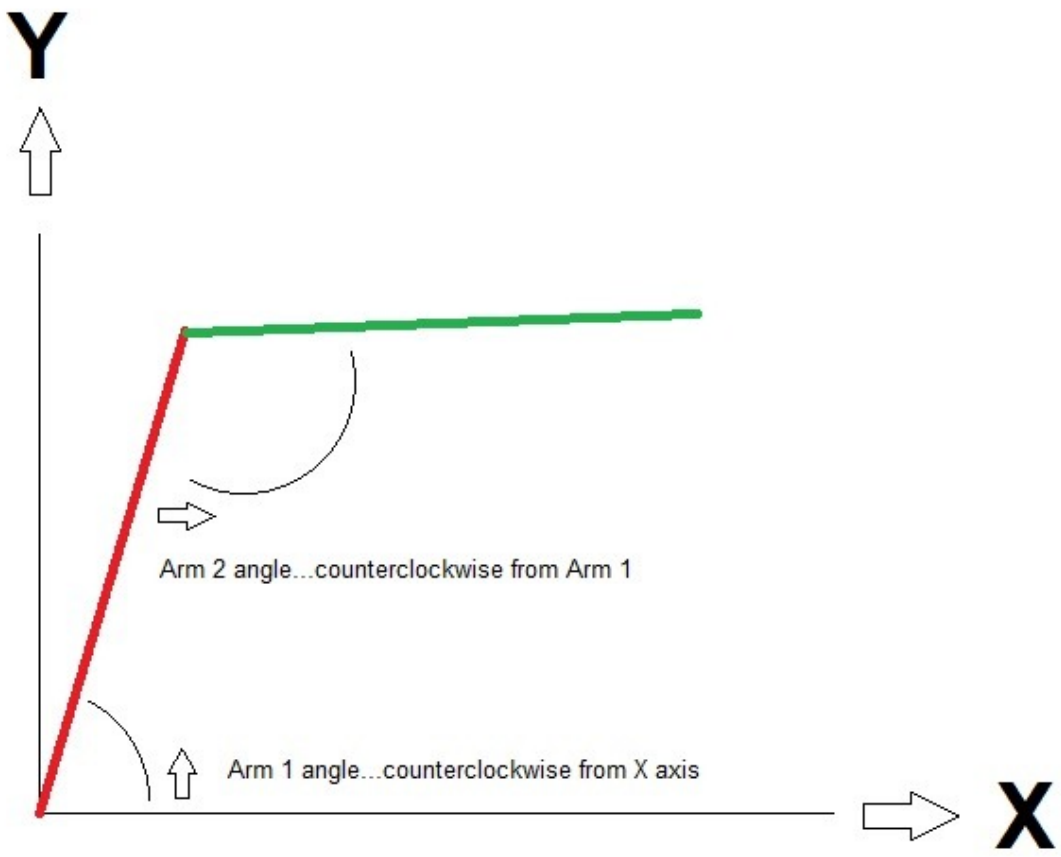
The SCARA kinematics supported by the ApossIDE requires two motors. If the 3-dimensional version is being used, then a third motor is also required. For the purposes of this description, they will be called Motor 1, Motor 2, and Motor 3. Motor 1 will be the motor at the base of the primary arm; Motor 2 will connect the secondary arm to the primary arm. If it exists, then Motor 3 drives the linear Z axis. The coordinate system used by the ApossIDE, is defined so that the origin is positioned at the rotational axis of Motor 1 when Motor 3 is at its zero-position. This will be the Machine Coordinate System of the machine (see Basic Coordinate Systems).

The (x,y) position of the machine is defined to be the end of the secondary arm. If a different position is required by the application, then transformations may be applied to effectively shift the Machine Coordinate System (see Transformations). Similarly, transformations may be used if the origin or orientation of the Machine Coordinate System does not match what the application would like to work with.



Kinematics Coordinate System

The Kinematics Coordinate System consists of either two or three coordinates, one for each of the motors. The kinematics expects Motor 1 to be defined so that in its zero position (origin), the primary arm lies on the positive X axis. The motor direction will be oriented so that positive movements are counterclockwise when looking down at the XY plane from above. Motor 2 is defined so that in its zero position (origin), the secondary arm lies on the primary arm, pointing towards the origin of the primary arm. The motor direction will also be oriented so that positive movements are counterclockwise when looking down at the XY plane from above. Both coordinates will be angular coordinates measured in user units (see FEEDREV and FEEDDIST). If a Z axis is present, then a coordinate for Motor 3 will exist and will be oriented so that increasing values move up out of the diagram, parallel to the Z axis. Motor directions and the size of user units may be set in the normal way for each axis (see How to Set Up Resolution and Gear Parameters). Note that the encoder resolution and user units of each motor may be different. However, if Motor 3 exists, then its user units *must* match the units used to specify the arm lengths.



Determining Coordinate System Units

Determining the units of the coordinate systems is quite simple for SCARA. It consists of three parts, all of which should be determined directly from the design parameters of the system.

1. Angle units: The angles used by the Kinematics Coordinate System are measured in user units determined directly from the FEEDREV and FEEDDIST axis parameters. These two parameters allow the controller to calculate the number of user units per revolution and therefore the actual angle corresponding to any given number of user units. See How to Set Up Resolution and Gear Parameters.
2. Distance units: Distance units in the Machine Coordinate System are derived directly from the units used to measure the SCARA arms. For example, if the user decides to use millimeters to specify arm lengths, then the Machine Coordinate System will use millimeters as units. If the arms are measured in inches, then the Machine Coordinate System will be in inches.
3. If a Z axis is being used, then its user units *must* be defined to be the same units as those used to specify the lengths of the arms.

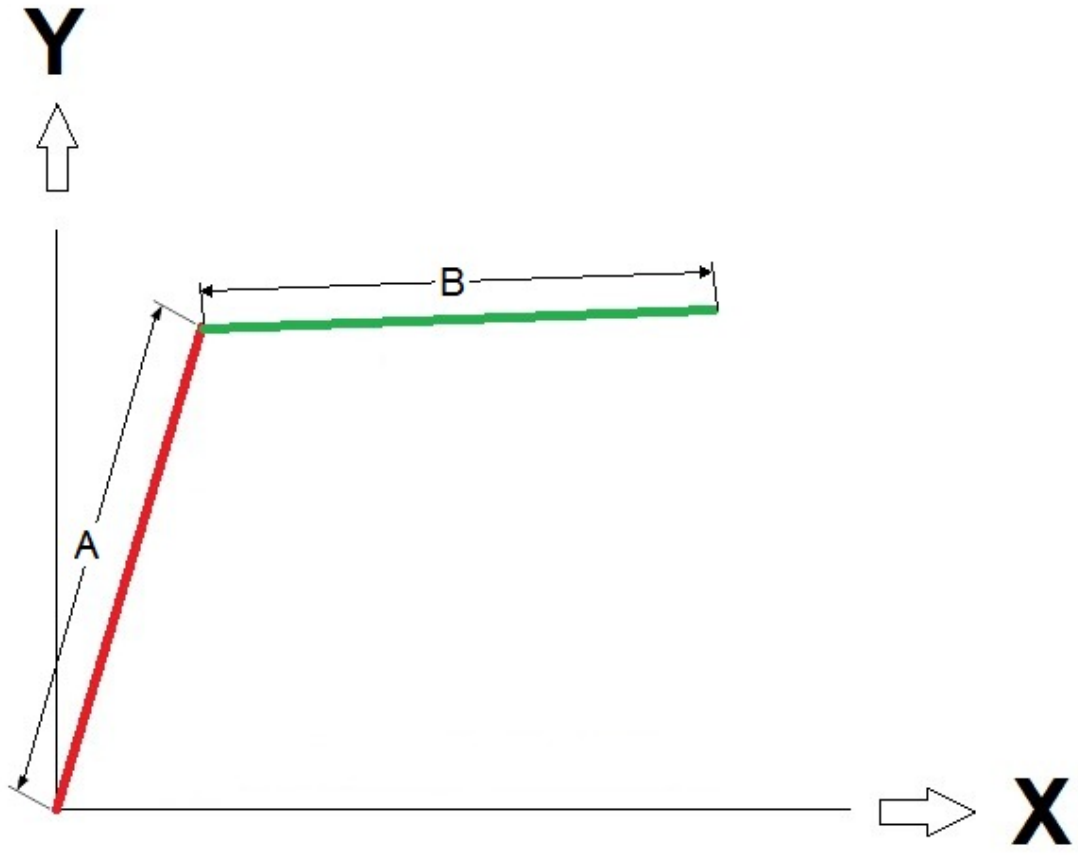
Kinematics Axes and Parameters

The "axis" array required by the KinematicsSetup command, contains two mandatory elements. The first value is the axis module number of the controller axis associated with Motor 1. The second is the axis module number of the axis associated with Motor 2. If a Z axis is being used, then this axis number must be next. Optional axes may be specified after that.

The ApossIDE requires four parameters for the SCARA kinematics:

1. The length of the primary arm, in Machine Coordinate System units ("A" in the diagram below).
2. The length of the secondary arm, in Machine Coordinate System units ("B" in the diagram below).

3. The orientation of the arms: 1 - Left arm oientation, -1 - Right arm orientation.
4. The number of spatial dimensions. Only 2 or 3 dimensions are supported.



Workspace and Limitations

The workspace of the SCARA is defined so the Arm 1 angle is not allowed to be higher than 180 degrees or lower than -180 degrees. The points but also the path in between the points cannot allow such a position. Please also note that when enabling the kinematics the Arm 1 angle cannot be in such a position either or else an error is thrown.

The diagram above shows a "left oriented" arrangement (i.e. like a person's left arm). A "right oriented" arrangement would be oriented like a person's right arm. In both cases, the angles are measured in the same way. This means the angle of Arm 2 will always be less than 180 degrees in a left oriented arrangement and always be greater than 180 degrees in a right oriented arrangement.

Example

```
long   hndl, axis[2];
double prms[4];

axis[0] = 1;    // Motor 1 uses controller axis module 1.
axis[1] = 0;    // Motor 2 uses controller axis module 0.


prms[0] = 1000.0;    // Primary arm is 1000 mm long.
prms[1] =  600.0;    // Secondary arm is 600 mm long.
prms[2] =      1;    // Use a "Left" orientation.
```

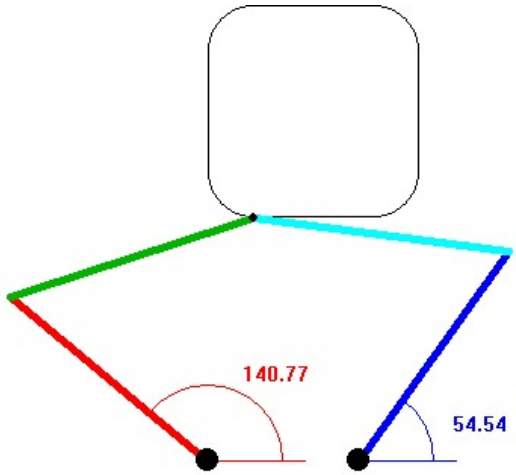
```
prms[3] =      2;    // Use 2-dimensional kinematics.

hndl = KinematicsSetup("Scara", axis, prms);   // Select the SCARA kinematics.
```

6.6.1.4 Dual SCARA Kinematics

The basic design and behaviour of a Dual SCARA is shown in the diagrams below. It uses two fixed motors to adjust the angles of two primary arms. Secondary arms are attached to the ends of the primary arms and also to each other. These secondary arms are "floating"; they are only driven by the positions of the primary arms. Compare this to SCARA Kinematics. The structure of the robot arms need not be as sturdy as SCARA, since the second motor is fixed rather than being moved with the primary arm. But the range of the end effector is much more restricted since *both* arms must be able to reach the target point.

The range is further restricted since the secondary arms are not allowed to become "inverted" (i.e. with the joint pointing down and not up when the target point lies in the positive Y axis region or with the joint pointing up and not down when the target point lies in the negative Y axis region). The ApossIDE supports enabling the kinematics setup for both positive and negative Y axis target points. However, a direct switch from the positive Y axis region to the negative one or vice versa is not allowed, since this will cause the secondary arms to become "inverted". To switch from one Y axis region to the other, the kinematics setup must first be disabled, then the arms are moved to the other region using Axis Commands before enabling the kinematics again.

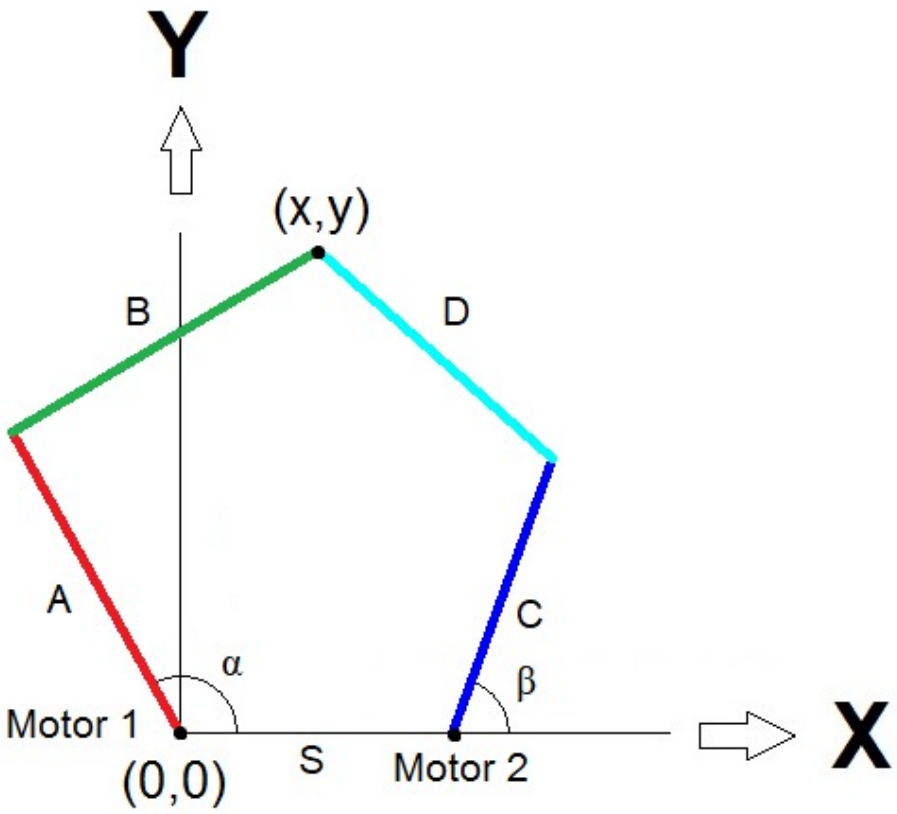
The ApossIDE supports both 2-dimensional and 3-dimension versions of Dual SCARA kinematics. However, in the 3-dimensional version, the Z axis behaves as a simple linear axis that is orthogonal to the X and Y axes (i.e. similar to the Z axis in Cartesian kinematics).



Machine Coordinate System

The Dual SCARA kinematics supported by the ApossIDE requires two motors. If the 3-dimensional version is being used, then a third motor is also required. For the purposes of this description, they will be called Motor 1, Motor 2, and Motor 3. Motor 1 will be the motor at the base of the left primary arm; Motor 2 will be at the base of the right primary arm. If it exists, then Motor 3 drives the linear Z axis. The coordinate system used by the ApossIDE, is defined so that the origin is positioned at the rotational axis of Motor 1 when Motor 3 is at its zero-position. Motor 2 will be positioned on the X axis to the right of Motor 1. This will be the Machine Coordinate System of the machine (see Basic Coordinate Systems).

The (x,y) position of the machine is defined to be the end of the secondary arms where they connect. If a different position is required by the application, then transformations may be applied to effectively shift the Machine Coordinate System (see Transformations). Similarly, transformations may be used if the origin or orientation of the Machine Coordinate System does not match what the application would like to work with.



Kinematics Coordinate System

The Kinematics Coordinate System consists of either two or three coordinates, one for each of the motors. The ApossIDE expects both Motor 1 and Motor 2 to be defined so that in their zero positions (origins), the primary arms lie on the positive X axis. The motor direction will be oriented so that positive movements are counterclockwise when looking down at the XY plane from above. See alpha and beta in the above diagram. Both coordinates will be angular coordinates measured in user units (see FEEDREV and FEEDDIST). If a Z axis is present, then a coordinate for Motor 3 will exist and will be oriented so that increasing values move up out of the diagram, parallel to the Z axis. Motor directions and the size of user units may be set in the normal way for each axis (see How to Set Up Resolution and Gear Parameters). Note that the encoder resolution and user units of each motor may be different. However, if Motor 3 exists, then its user units *must* match the units used to specify the arm lengths.

Determining Coordinate System Units

Determining the units of the coordinate systems is quite simple for Dual SCARA. It consists of three parts, all of which should be determined directly from the design parameters of the system.

1. Angle units: The angles used by the Kinematics Coordinate System are measured in user units determined directly from the FEEDREV and FEEDDIST axis parameters. These two parameters allow the controller to calculate the number of user units per revolution and therefore the actual angle corresponding to any given number of user units. See How to Set Up Resolution and Gear Parameters.
2. Distance units: Distance units in the Machine Coordinate System are derived directly from the units used to measure the Dual SCARA arms. For example, if the user decides to use millimeters to specify arm lengths, then the Machine Coordinate System will use millimeters as units. If the arms are measured in inches, then the Machine Coordinate System will be in inches.
3. If a Z axis is being used, then its user units *must* be defined to be the same units as those used to specify the lengths of the arms.

Kinematics Axes and Parameters

The "axis" array required by the KinematicsSetup command, contains two mandatory elements. The first value is the axis module number of the controller axis associated with Motor 1. The second is the axis module number of the axis associated with Motor 2. If a Z axis is being used, then this axis number must be next. Optional axes may be specified after that.

The ApossIDE requires six parameters for the SCARA kinematics:

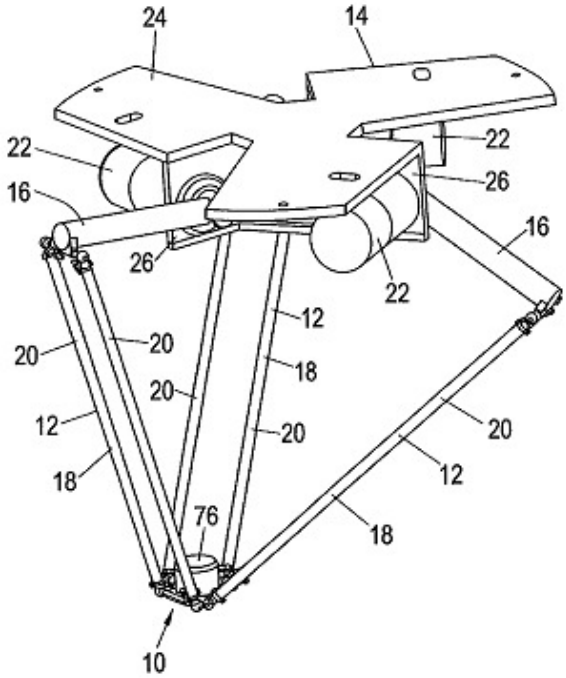
1. The length of the left primary arm, in Machine Coordinate System units ("A" in the diagram above).
2. The length of the left secondary arm, in Machine Coordinate System units ("B" in the diagram above).
3. The length of the right primary arm, in Machine Coordinate System units ("C" in the diagram above).
4. The length of the right secondary arm, in Machine Coordinate System units ("D" in the diagram above).
5. The separation distance between Motor 1 and Motor 2 ("D" in the diagram above).
6. The number of spatial dimensions. Only 2 or 3 dimensions are supported.

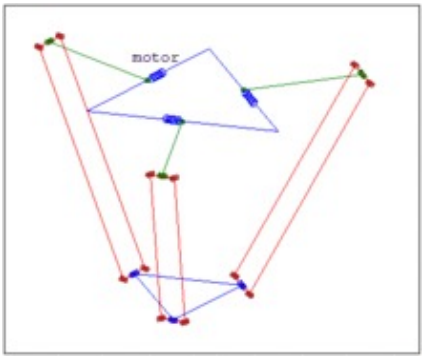
Example

```
long   hndl, axis[2];
double prms[6];

axis[0] = 0;    // Motor 1 uses controller axis module 0.
axis[1] = 1;    // Motor 2 uses controller axis module 1.

prms[0] = 1000.0;    // Left primary arm is 1000 mm long.
prms[1] =  600.0;    // Left secondary arm is 600 mm long.
prms[2] = 1000.0;    // Right primary arm is 1000 mm long.
prms[3] =  600.0;    // Right secondary arm is 600 mm long.
prms[4] =  700.0;    // The motors are separated by 700 mm.
prms[5] =      2;    // Use 2-dimensional kinematics.

hndl = KinematicsSetup("DualScara", axis, prms);   // Select the Dual SCARA
kinematics.
```

6.6.1.5 Delta Kinematics

The basic design and behaviour of a Delta robot is shown in the diagrams below. This uses three fixed motors to adjust the angle of arms that connect to an end effector below the motors. The end effector can be positioned anywhere within a limited 3-dimensional volume. This type of robot has two particularly useful properties: the end effector always maintains an orientation that is parallel to the base or floor, and the arms can be lightweight allowing the robot to move quickly. These properties make Delta robots a good match for "Pick-and-Place" applications.

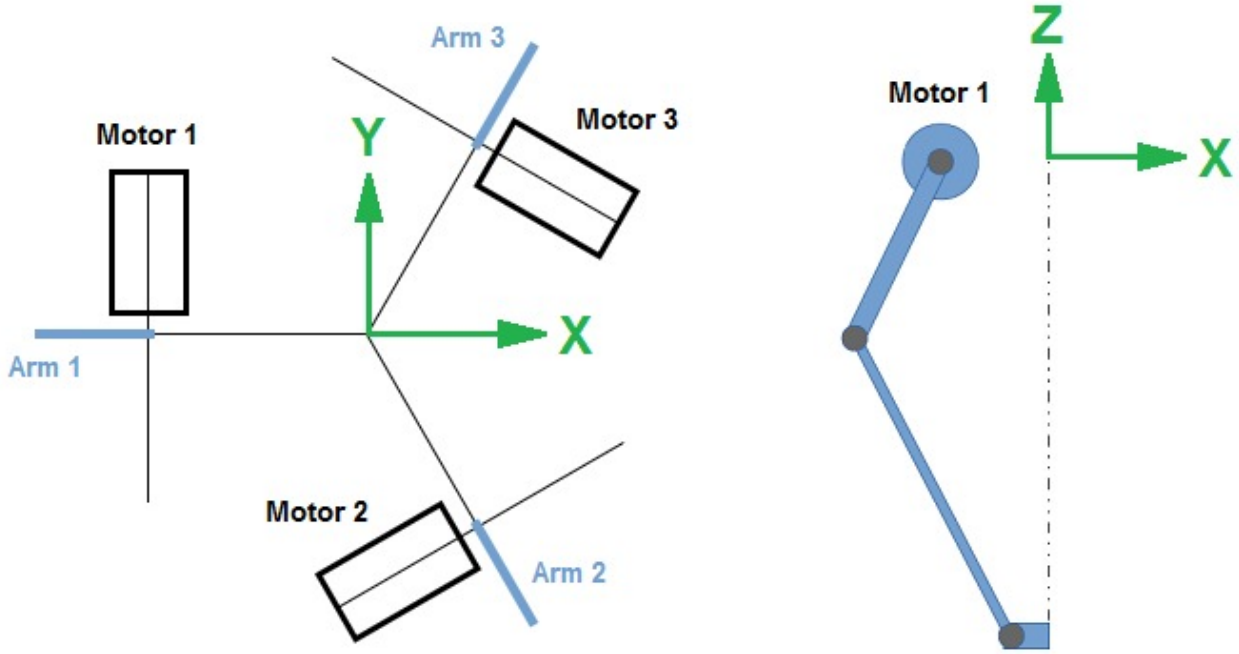




Machine Coordinate System

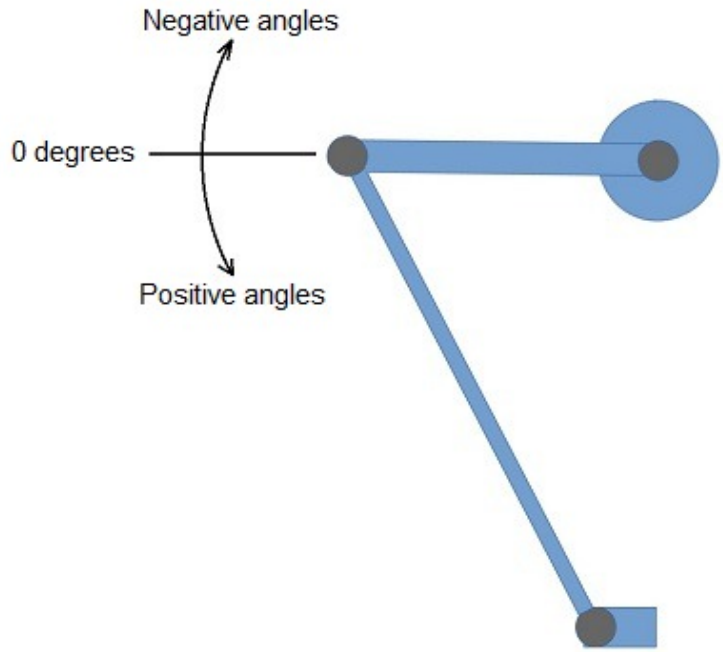
The Delta kinematics supported by the ApossIDE requires three motors. For the purposes of this description, they will be called Motor 1, Motor 2, and Motor 3. The axes of all three motors will be coplanar and the motors will be arranged 1, 2, 3 in the counterclockwise direction when looking *down* at the machine. The coordinate system used by the ApossIDE, is defined so that all three axes lie in the XY plane, the origin is centered between the motors, and the X axis is colinear with the axis of Motor 1. This implies that the Z axis points "up" and that *all* Z coordinates used in positioning the machine will be negative. This will be the Machine Coordinate System of the machine (see Basic Coordinate Systems).

The (x,y,z) position of the machine is defined to be the center point of the bottom plate in the same plane as the connecting joints. If a different position is required by the application (e.g. the end effector is typically below the bottom plate), then transformations may be applied to effectively shift the Machine Coordinate System (see Transformations). Similarly, transformations may be used if the origin or orientation of the Machine Coordinate System does not match what the application would like to work with.



Kinematics Coordinate System

The Kinematics Coordinate System consists of three coordinates, one for each of the motors. The ApossIDE expects each motor to be defined so that in its zero position (origin) the first arm is horizontal (i.e. lying in the XY plane). The motor direction will be oriented so that positive movements are counterclockwise when looking at the "front" of the motor. So a forward movement will cause the arm to move down and a negative movement will cause the arm to move up. All three coordinates will be angular coordinates measured in user units (see FEEDREV and FEEDDIST). Motor directions and the size of user units may be set in the normal way for each axis (see How to Set Up Resolution and Gear Parameters). Note that the encoder resolution and user units of each motor may be different.



Determining Coordinate System Units

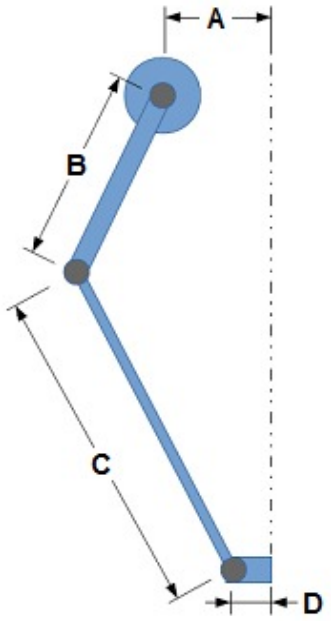
Determining the units of the coordinate systems is quite simple for Delta robots. It consists of two parts, both of which should be determined directly from the design parameters of the system.

1. Angle units: The angles used by the Kinematics Coordinate System are measured in user units determined directly from the FEEDREV and FEEDDIST axis parameters. These two parameters allow the controller to calculate the number of user units per revolution and therefore the actual angle corresponding to any given number of user units. See How to Set Up Resolution and Gear Parameters.
2. Distance units: Distance units in the Machine Coordinate System are derived directly from the units used to measure the Delta arms. For example, if the user decides to use millimeters to specify arm lengths, then the Machine Coordinate System will use millimeters as units. If the arms are measured in inches, then the World Coordinate System will be in inches.

Kinematics Axes and Parameters

The "axis" array required by the KinematicsSetup command, contains three mandatory elements. The first value is the axis module number of the controller axis associated with Motor 1. The second is the axis module number of the axis associated with Motor 2. The third is the axis module number of the axis associated with Motor 3. Optional axes may be specified after that.

The ApossIDE requires six parameters for the Delta kinematics: four lengths, a minimum motor angle, and a maximum motor angle. The lengths are shown in the diagram below. These must be specified in the same units used by the Machine Coordinate System. The minimum angle specifies the angle (in radians) beyond which the motor should not move in the "up" direction. The maximum angle specifies the angle beyond which the motor should not move in the "down" direction.



Example

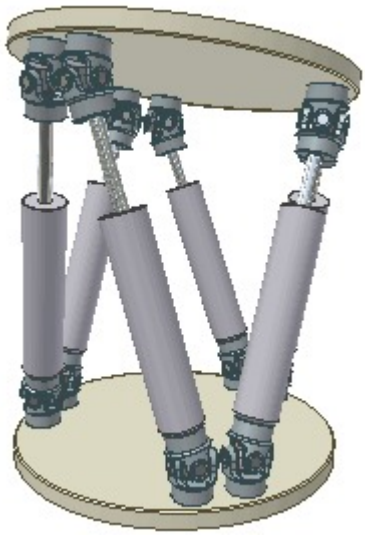
```
long   hndl, axis[3];
double prms[6];

axis[0] = 0;    // Motor 1 uses controller axis module 0.
axis[1] = 1;    // Motor 2 uses controller axis module 1.
axis[2] = 2;    // Motor 3 uses controller axis module 2.

// This block defines the Scaling of the axes (hundredth of degree)
```

```
AXE_PARAM(0, FEEDREV)   = 1;
AXE_PARAM(0, FEEDDIST)  = 36000;  // Motor 1 (axis 0) moves 36000 user units in 1
revolution
AXE_PARAM(1, FEEDREV)   = 1;
AXE_PARAM(1, FEEDDIST)  = 36000;  // Motor 2 (axis 1) moves 36000 user units in 1
revolution
AXE_PARAM(2, FEEDREV)   = 1;
AXE_PARAM(2, FEEDDIST)  = 36000;  // Motor 3 (axis 2) moves 36000 user units in 1
revolution

// For a higher resolution the length can be given in tenth of a millimeter.
prms[0] =  800.0;   // Length A is 80.0 mm.
prms[1] = 1200.0;   // Length B is 120.0 mm.
prms[2] = 1800.0;   // Length C is 180.0 mm.
prms[3] =  300.0;   // Length D is 30.0 mm.
prms[4] = rad(-10.0);   // The arm must not move higher than -10 degrees.
prms[5] = rad( 10.0);   // The arm must not move lower than +10 degrees.

hndl = KinematicsSetup("Delta", axis, prms);   // Select the Delta kinematics.
```

6.6.1.6 Stewart Kinematics

The Stewart kinematics is not yet supported.

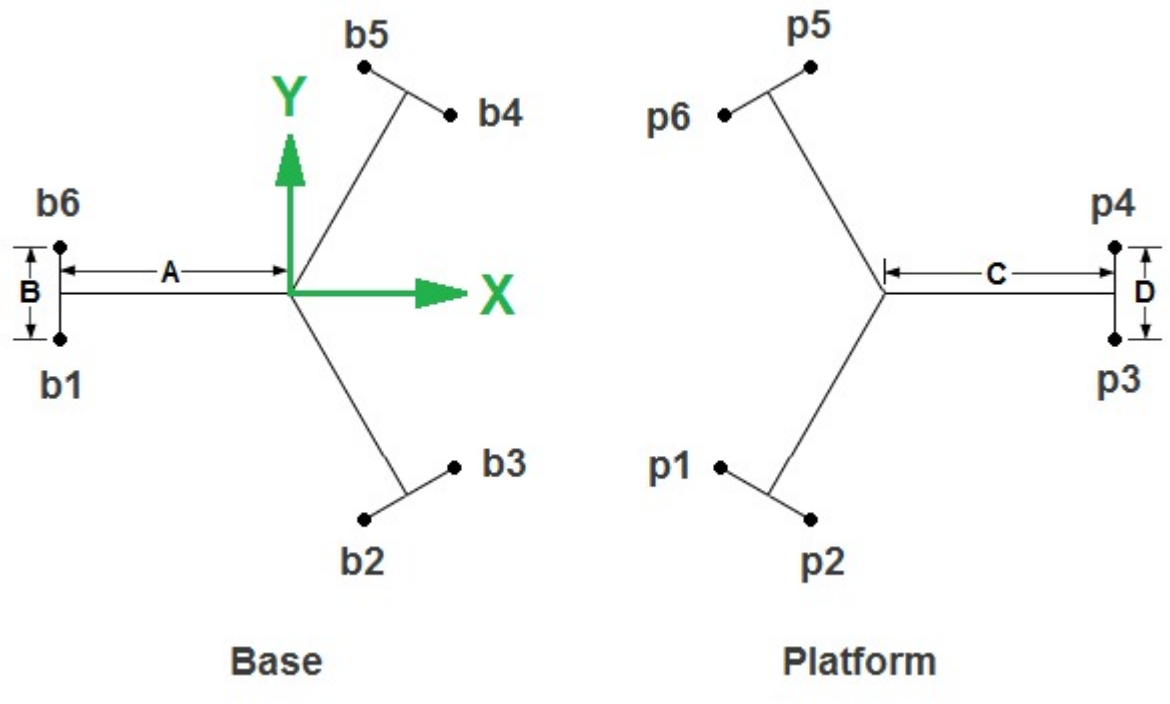
The basic design and behaviour of a Stewart Platform is shown in the diagram below. This uses six linear motors to adjust the lengths of cylinders that connect to a platform containing the end effector. The end effector can be positioned anywhere within a limited 3-dimensional volume and within a limited range of yaw, pitch, roll angles. This type of robot has six degrees of freedom and makes a good match for applications that require significant control over angular alignment. For example, radio telescopes sometimes use this type of mechanics.



Machine Coordinate System

The Stewart Platform kinematics supported by the ApossIDE requires six linear motors. For the purposes of this description, they will be called motors 1 to 6. The bottom ends of these motors will be connected in pairs to a base plate, 120 degrees apart, as shown in the diagram below (labeled b1 to b6). The connection points of all six motors are expected to be coplanar and lie in the XY plane. The origin will be the center point between all three pairs. The upper ends of the motors will be connected to a free-moving platform above the base, also in pairs and 120 degrees apart (labeled p1 to p6). This implies that the Z axis points "up" and that *all* Z coordinates used in

positioning the machine will be positive. This will be the Machine Coordinate System of the machine (see Basic Coordinate Systems).



The (x,y,z) position of the machine is defined to be the center point of the plane between all three pairs of platform joints. If a different position is required by the application (e.g. the end effector is typically above the plane of the platform joints), then transformations may be applied to effectively shift the Machine Coordinate System (see Transformations). Similarly, transformations may be used if the origin or orientation of the Machine Coordinate System does not match what the application would like to work with.

The angular position of the machine (α,β,γ) is defined as follows:

- α The rotation angle (in radians) about the X-axis done in a counterclockwise direction when looking at the origin from a position on the positive X-axis. This rotation is done *first*.
- β The rotation angle (in radians) about the Y-axis done in a counterclockwise direction when looking at the origin from a position on the positive Y-axis. This rotation is done *second*.
- γ The rotation angle (in radians) about the Z-axis done in a counterclockwise direction when looking at the origin from a position on the positive Z-axis. This rotation is done *third*.

Kinematics Coordinate System

The Kinematics Coordinate System consists of six coordinates, one for each of the motors. The ApossIDE expects each motor to be defined so that in its position matches its length. The motor direction will be oriented so that positive movements increase the length. So a forward movement will tend to cause the platform to move up and a negative movement will tend to cause the platform to move down. The units of the Kinematics Coordinate System will be identical to the units of the Machine Coordinate System (i.e. user units and machine units will be the same). Motor directions and the size of user units may be set in the normal way for each axis (see How to Set Up

Resolution and Gear Parameters).

Kinematics Axes and Parameters

The "axis" array required by the KinematicsSetup command, contains six mandatory elements. The first value is the axis module number of the controller axis associated with Motor 1, the second is the axis module number of the axis associated with Motor 2, and so forth. Optional axes may be specified after that.

The ApossIDE requires six parameters for the Stewart Platform kinematics: a minimum motor length, a maximum motor length, and the four dimensions (labeled A, B, C, and D) shown in the diagram above. All of these must be specified in the same units used by the Machine Coordinate System.

Example

```
long   hndl, axis[6];
double prms[6];

axis[0] = 0;    // Motor 1 uses controller axis module 0.
axis[1] = 1;    // Motor 2 uses controller axis module 1.
axis[2] = 2;    // Motor 3 uses controller axis module 2.
axis[3] = 3;    // Motor 4 uses controller axis module 3.
axis[4] = 4;    // Motor 5 uses controller axis module 4.
axis[5] = 5;    // Motor 6 uses controller axis module 5.

prms[0] =  800.0;   // Minimum length is 800 mm.
prms[1] = 1200.0;   // Maximum length is 1200 mm.
prms[2] =  300.0;   // Length A is 300 mm.
prms[3] =   30.0;   // Length B is 30 mm.
prms[4] =  200.0;   // Length C is 200 mm.
prms[5] =   30.0;   // Length D is 30 mm.

hndl = KinematicsSetup("Stewart", axis, prms);   // Select the Stewart Platform
kinematics.
```

6.6.2 Basic Coordinate Systems

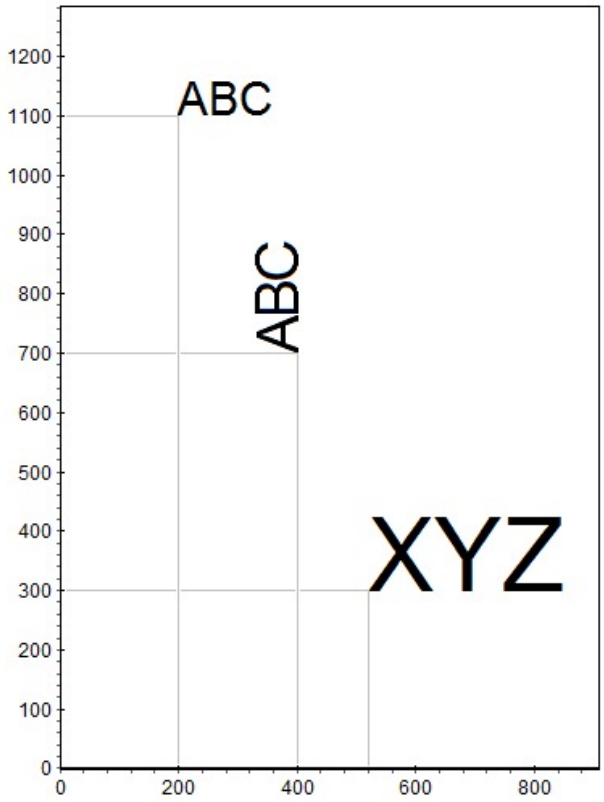
Users must be aware of four basic coordinate systems when developing applications for robots:

- Work Coordinate System
- Machine Coordinate System
- Kinematics Coordinate System
- Axis Coordinate System

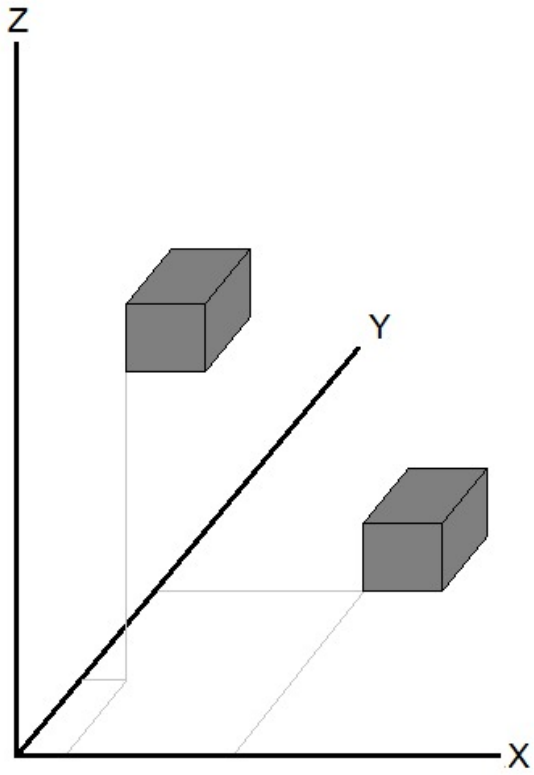
If Path Curves are being used, then two additional coordinate systems are used. Please see Additional Coordinate Systems.

Work Coordinate System

The Work Coordinate System is the primary coordinate system in which the user will work. This is a familiar Cartesian coordinate system using standard (X,Y,Z) coordinates. Note that all coordinate systems used by the ApossIDE are *right-handed* (see below). The Work Coordinate System provides a fixed coordinate system into which the objects on which work is being done are placed. For example, it may represent a page on a plotter where characters are placed at different positions on the page or drawing is done somewhere on the page. Or it may represent the 3-D space into which an object is placed. The application can move the end effector of the robot anywhere within this space simply by telling the robot to move to a specified (X,Y,Z) location. Moves are done using the PathMove command.

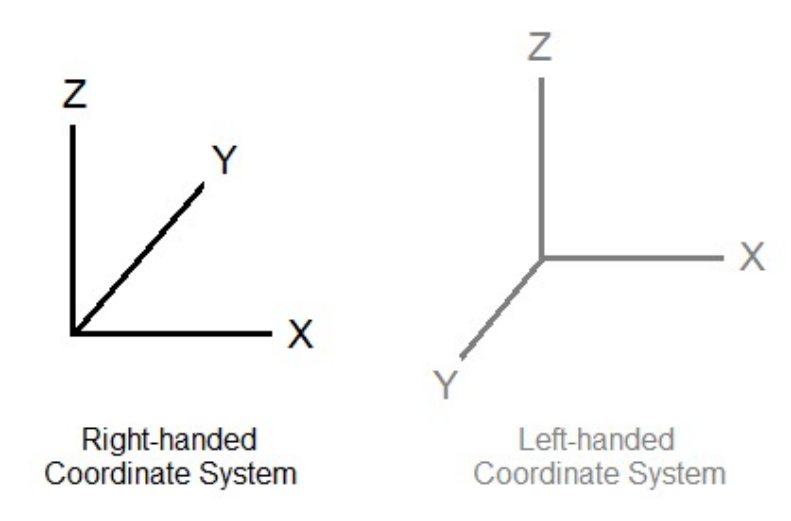


2-D Work Coordinate System



3-D Work Coordinate System

All spatial coordinate systems used by the ApossIDE are *right-handed*. The following illustration shows the difference between right-handed and left-handed coordinate systems.

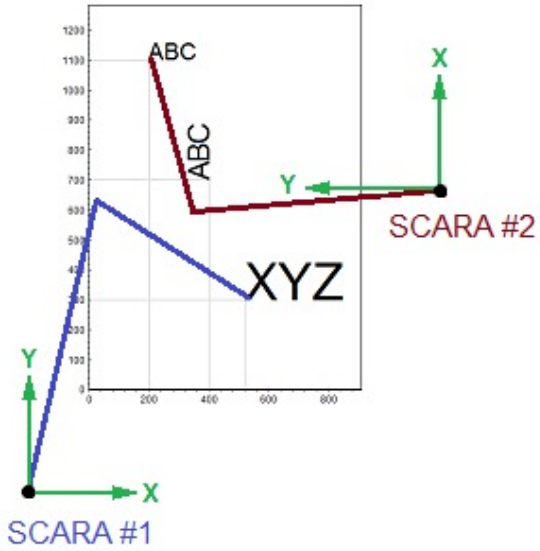


Machine Coordinate System

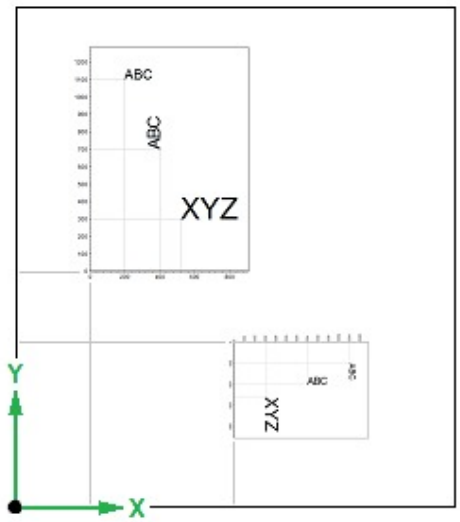
The Machine Coordinate System is the coordinate system of the robot itself. This is fixed in relationship to the robot and it is specific to the kinematics being used by the robot. For example, the origin of the Machine Coordinate System for a SCARA will be centered at the base of the primary arm.

The ApossIDE uses a simple user-specified transformation (see Transformations) to map the Work Coordinate System onto the Machine Coordinate System of the robot. This allows the robot to be placed anywhere in relationship to the

Work Coordinate System (as on the left of the following drawing which shows two possible locations). An alternative view is that it allows the Work Coordinate System to be placed anywhere within the Machine Coordinate System of the robot (as on the right of the following drawing). In *both* cases, the application need only move the robot to (X,Y,Z) positions within the original Work Coordinate System. Subsequent transformations to the Machine Coordinate System and beyond, are handled automatically by the controller.



Machine Coordinate Systems placed around a Work Coordinate System



Work Coordinate Systems placed within a Machine Coordinate System

The transformation from the Work Coordinate System to the Machine Coordinate System is specified in application programs using the KinematicsWorkToMc command.

Kinematics Coordinate System

In order to be able to follow a path in the real world, two or more physical movements must be synchronized. How this is done depends on the kinematics of the machine being driven. For example, in a simple 2-dimensional pen plotter (see Cartesian Kinematics), one motor may drive a gantry horizontally while a second motor drives the pen vertically. A similar application using a SCARA (see SCARA Kinematics) will require two arm angles to be set. The Kinematics Coordinate System will represent the coordinate system of the moveable elements of the machine (i.e. the position of the plotter motors or the angles of the SCARA motors). Each position in the Machine Coordinate System will correspond to a position in the Kinematics Coordinate System.

While the Machine Coordinate System is Cartesian (with orthogonal axes), the Kinematics Coordinate System may or may not be Cartesian. For example, the plotter Kinematics Coordinate System will most likely be Cartesian since the basic mechanics of the plotter are Cartesian. On the other hand, the SCARA kinematics, consisting of angles, is clearly not Cartesian. The actual Kinematics Coordinate System used will depend entirely on the design of the mechanics in the machine.

To follow a path, the controller must convert machine coordinates into kinematics coordinates that are then used to drive the actual motors. For certain types of kinematics, this conversion process can be quite difficult mathematically. However, the controller contains support to do this automatically for several common types of kinematics.

The Kinematics Coordinate Systems supported by the ApossIde are designed to be as generic as possible. For example, they contain no intrinsic measurement units. Measurement units are typically inherited from the Machine Coordinate System or determined by the user units used in the lower-level Axis Coordinate System. This allows kinematic coordinates to be adapted to a wide variety of machine designs.

See Multi-axis Kinematics for a detailed description of the different Kinematics Coordinate Systems provided by the ApossIDE and for how coordinate system units are determined.

Axis Coordinate System

The Axis Coordinate System corresponds to the actual positions of each individual motor axis in the controller, as measured by user units. These are the coordinates used by commands such as AxisPosAbsStart. Each coordinate of the Kinematics Coordinate System will correspond to a unique coordinate (or controller axis) in the Axis Coordinate System. The axis gearing factors (see How to Set Up Resolution and Gear Parameters) are used to convert from kinematics coordinates directly to axis coordinates. These gearing factors determine the final correspondence between the units of the Machine Coordinate System and the quadcount values used to drive the motor positions.

The complete sequence of transformations is as follows:

Work coordinates (Cartesian, user-specific)
→ Machine coordinates (Cartesian, kinematics-specific)
→ Kinematics coordinates (kinematics-specific)
→ Axis coordinates (user units)
→ Motor (quadcounts)

6.6.3 Transformations

Transformations are used to define the mapping from the Work Coordinate System to the Machine Coordinate System (see Basic Coordinate Systems). They are also used with Path Curves to allow the basic shape of a prespecified path to be arbitrarily scaled and positioned before it is used in the "real" world. Transformations will only affect the spatial coordinates (X, Y, and Z) of path curves. They can be used with both 2-dimensional and 3-dimensional path curves.

To use transformations, follow these steps:

1. Start by declaring one or more `transform` variables. They are declared using syntax similar to how all other variables are declared.

   ```
   transform var1, var2;
   ```

   TRANSFORM variables are automatically initialized to the identity transformation (i.e. no transformation) when they are declared.

   > Arrays of transform variables are not currently supported.

2. Add the required transformations to the variables using the TransTranslate, TransScale, TransStretch, TransRotateXY, TransRotateYZ, TransRotateZX, and TransReflect commands. Multiple transformations may be incorporated into each variable.

   ```
   TransScale(var1, 2.5);
   TransRotateXY(var1, rad(45));
   TransTranslate(var1, 500, 400, 0);
   ```

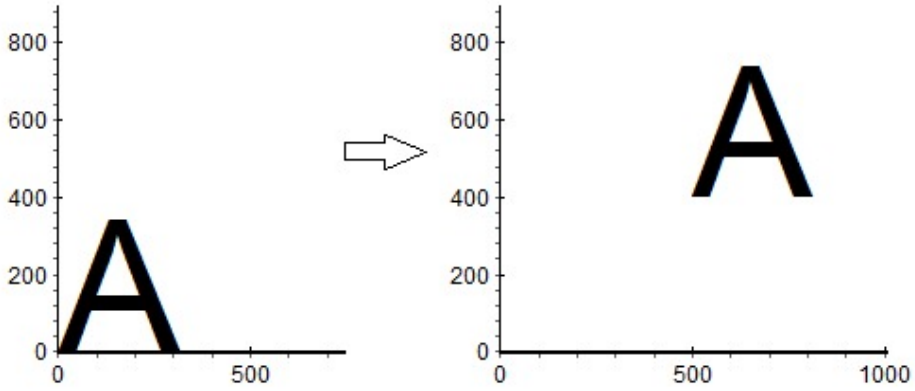
3. The `transform` variable may then be used with various commands such as KinematicsWorkToMc, PathStart, PathNext, etc.

Translations

Translations are done using the TransTranslate command and simply shift the curve in space without affecting its size or orientation.

Example:

```
TransTranslate(var1, 500, 400);
```

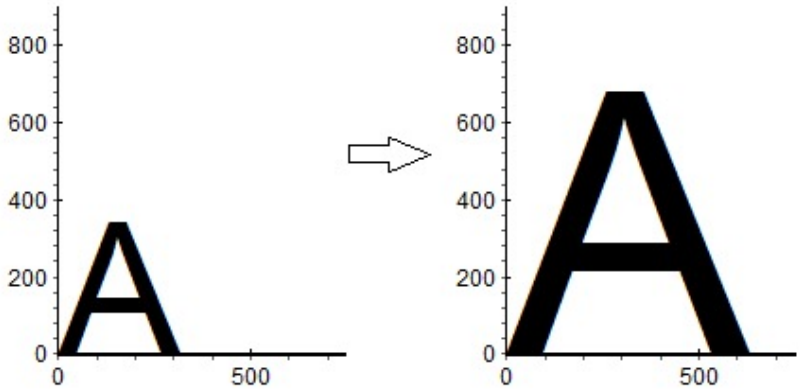


Scaling

Scaling is done using the TransScale command and will resize the curve without affecting its orientation. Note that scaling is done relative to the origin of the coordinate system. This means that if the curve is not somehow based at the origin, then scaling the curve will also effectively translate the curve either further away, or closer to, the origin.

Example:

```
TransScale(var1, 2.0);
```

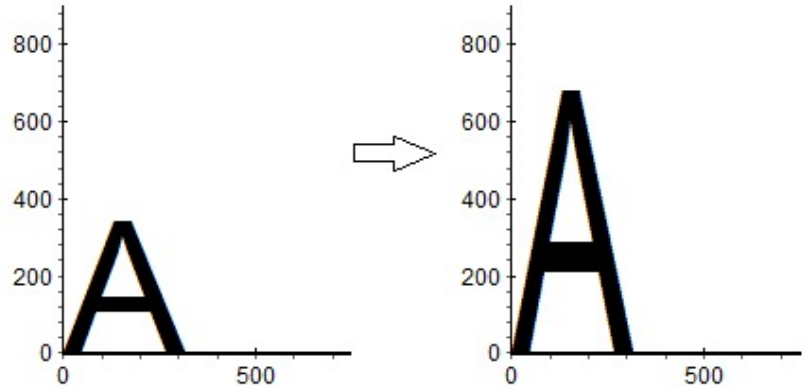


Stretching

Stretching is done using the TransStretch command and will stretch the curve without affecting its orientation. Stretching is similar to scaling except that it allows the curve to be scaled differently in each dimension. Note that stretching is done relative to the origin of the coordinate system. This means that if the curve is not somehow based at the origin, then stretching the curve will also effectively translate the curve either further away, or closer to, the origin.

Example:

```
TransStretch(var1, 1.0, 2.0);
```

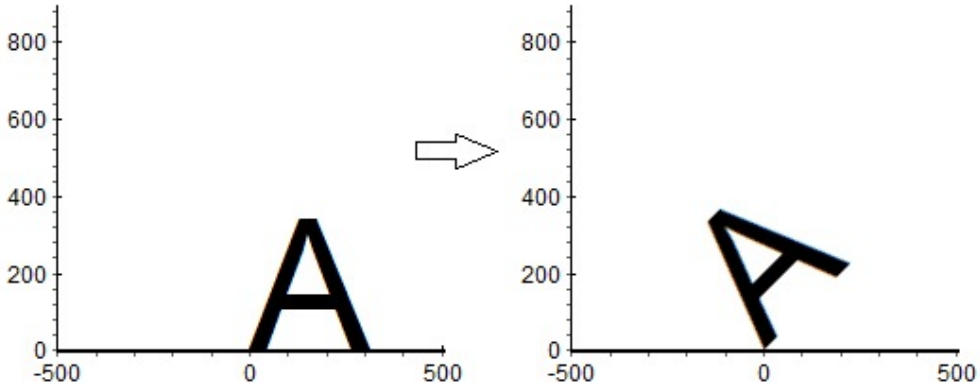


Rotations

Rotations are done using the TransRotateXY, TransRotateYZ, and TransRotateZX commands. Each of these commands will rotate the curve about a different coordinate system axis without affecting its size. Positive rotation angles will rotate the curve in a counter clockwise direction. Note that, as with scaling, the rotation is done relative to the origin of the coordinate system. This may effectively move the curve in space.

Example:

```
TransRotateXY(var1, rad(45));
```

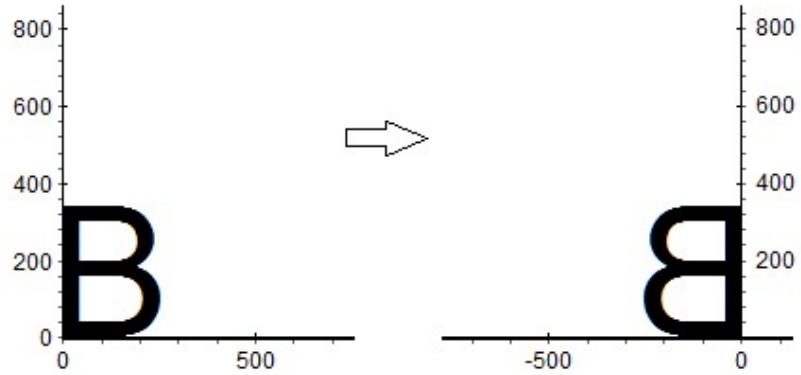


Reflections

Reflections are done using the TransReflect command. This command will reflect the curve across the YZ plane (i.e. X coodinates will be negated). In the simpler 2-dimensional case, this just reflects the curve across the Y axis. Reflections across other planes can be done by applying appropriate pre- and post-rotations to align the required plane with the YZ plane.

Example:

```
TransReflect(var1);
```



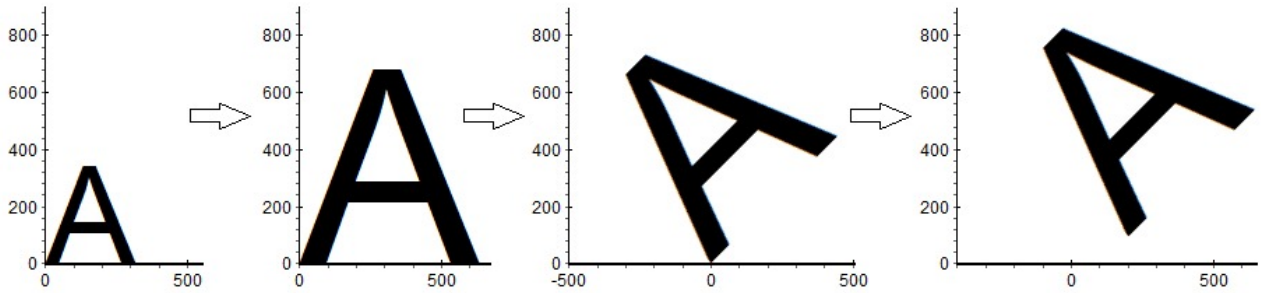
Multiple Transformations

Multiple transformations may be chained together in a single variable to create general transformations. Note that the order in which the transformations are done is important. For example, scaling and then translating has a different effect that translating and then scaling.

Example 1:

This sequence will create a single transformation that will double the size of a curve, rotate it to 45 degrees, and then translate it so that its origin is at (200,100). This would be a common transformation used when drawing character strings.

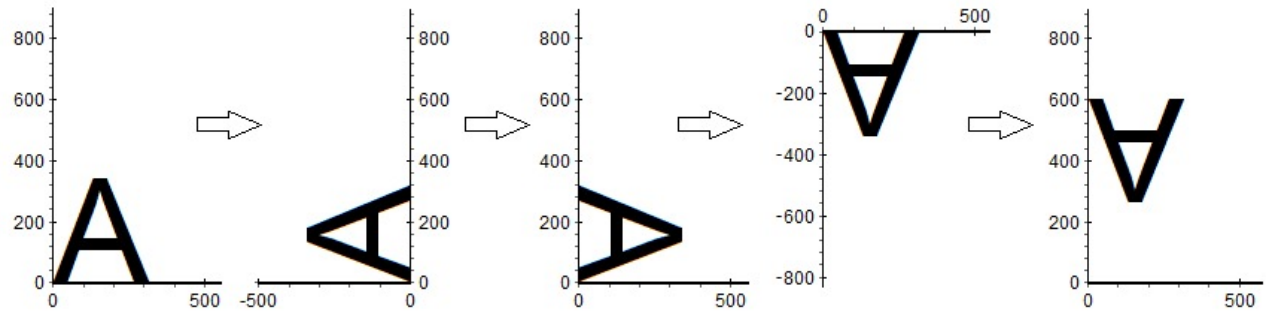
```
transform ChrToPaper  // Transformation from character to paper.

TransIdent(ChrToPaper);                // Initialize the transformation.
TransScale(ChrToPaper, 2);             // Scale by 2.
TransRotateXY(ChrToPaper, rad(45));    // Rotate to 45 degrees.
TransTranslate(ChrToPaper, 200, 100); // Move start position to (200,100).
```



Example 2:

This sequence will create a single transformation that will mirror a curve top-to-bottom and then translate it so that its origin is at (0,600).

```
transform ChrToPaper  // Transformation from character to paper.

TransIdent(ChrToPaper);                // Initialize the transformation.
TransRotateXY(ChrToPaper, rad(90));    // Rotate by 90 degrees.
TransReflect(ChrToPaper);              // Reflect across Y axis.
TransRotateXY(ChrToPaper, rad(-90));   // Rotate back by -90 degrees.
TransTranslate(ChrToPaper, 0, 600);    // Move start position to (0,600).
```



6.6.4 Working with Points

Since applications will generally be working with points in the multi-dimensional Work Coordinate System, ApossC provides a **dpoint** variable type to simplify the handling of points. A single **dpoint** variable can be configured to simultaneously hold all the coordinates required by any kinematics selected with the KinematicsSetup command. For example, when working with a Delta robot in 3-dimensional space, a **dpoint** variable will hold values for each of the X, Y, and Z coordinates.

dpoint variables are not limited to the Work Coordinate System. They may also be used to store points in the Machine Coordinate System, Path Coordinate System, etc. See Basic Coordinate Systems and Additional Coordinate Systems.

Declaring `dpoint` Variables

dpoint variables are declared using syntax similar to how all other variables are declared.

```
dpoint var1, var2;
```

> Arrays of **dpoint** variables are not currently supported.

Initializing `dpoint` Variables

dpoint variables may be initialized by assigning another **dpoint** variable to them or by assigning values to the induvidual coordinates using the '.' character followed by 'x', 'y', or 'z'.

```
dpoint pnt1, pnt2;

pnt1.x = 10.0;    // Init X coordinate with 10.
pnt1.y = 20.0;    // Init Y coordinate with 20.
pnt1.z = 30.0;    // Init Z coordinate with 30.

pnt2 = pnt1;
```

> The values *must* be integer or floating point constants. Expressions are not supported.

Accessing Individual Coordinates

Individual coordinates within a **dpoint** variable can be accessed using the '.' character followed by 'x', 'y', or 'z'.

```
dpoint pnt;

pnt.x = 10.0;    // Assign 10 to the X coordinate.
pnt.y = 20.0;    // Assign 20 to the Y coordinate.
pnt.z = 30.0;    // Assign 30 to the Z coordinate.

print("(X,Y,Z) = (", pnt.x, ",", pnt.y, ",", pnt.z, ")");
```

When using this syntax to assign individual coordinates, expressions are fully supported.

Non-spatial Coordinates

If the kinematics selected with the KinematicsSetup command contains additional non-spatial dimensions, then the number of coordinates stored in **dpoint** variables must be increased. This is done using the PNT_LENTGH #pragma. Note that the coordinate count must include a Z coordinate even if 2-dimensional kinematics are being used and Z is not needed. Hence, PNT_LENGTH must always be at least 3. PNT_LENGTH must be specified *before* the first **dpoint** variable is declared. For more information on **dpoint** storage structure, please see Understanding Point Storage Structure.

The additional coordinates may be initialized using the "[...]" syntax, simply by including the additional coordinates after the Z coordinate (see below).

The individual non-spatial coordinates may be accessed (as above) using the letters 'a' through 'g'.

```
// 3-D Delta kinematics will be used with one extra non-spatial coordinate.

#pragma PNT_LENGTH 4

dpoint pnt;

pnt = [10, 20, 30, 0.5];    // X = 10, Y = 20, Z = 30, A = 0.5

pnt.x = 10.0;    // Assign 10 to the X coordinate.
pnt.y = 20.0;    // Assign 20 to the Y coordinate.
pnt.z = 30.0;    // Assign 30 to the Z coordinate.
pnt.a = 0.5;     // Assign 1/2 to the non-spatial coordinate.
```

Transforming `dpoint` Variables

dpoint variables may be transformed by any transform stored in a **transform** variable (see Transformations). The TransPoint command will transform the point and the TransInverse command will transform the point by the inverse of the transform. Note that only the spatial coordinates (i.e. X, Y, and Z) will be transformed. Any non-spatial coordinate stored in the **dpoint** variable will remain unchanged.

```
dpoint    pnt;
transform trns;

pnt.x = 10.0;  // Assign 10 to the X coordinate.
pnt.y = 20.0;  // Assign 20 to the Y coordinate.
pnt.z = 30.0;  // Assign 30 to the Z coordinate.

// The position is now (10,20,30).

TransIdent(trns);
TransScale(trns, 10);

TransPnt(trns, pnt);

// The position is now (100,200,300).

TransInv(trns, pnt);

// The position is now (10,20,30) again.
```

Converting between Kinematics and Machine Coordinates

Application programs may convert back and forth between the Kinematics Coordinate System (i.e. user units) and the Machine Coordinate System, by using the PathUuToMc and PathMcToUu commands. See Basic Coordinate Systems for information about coordinate systems.

These functions are particularly useful for applications that rely on movements being "taught by example". This is where the robot is manually moved from point to point with the sequence being stored. The robot will then move by "replaying" the sequence of saved points.

> Points may be converted back and forth between Machine coordinates and Work coordinates by using the dynamic "Work-to-Machine" transform along with the TransInverse and TransPoint commands. See Basic Coordinate Systems and the KinematicsWorkToMc command for more information.

```
long      kinhdl;
double    curpos[3];
dpoint    pnt;
transform worktomachine;

kinhdl = Kinematics(....);

// Set up the dynamic Work-to-Machine transformation.

TransIdent(worktomachine);
....
DefWrkToMch(kinhdl, worktomachine);

// Get the current axis positions in user units.

curpos[0] = Apos(0);
curpos[1] = Apos(1);
curpos[2] = Apos(2);

// Convert the position to Machine coordinates.

PathUuToMc(kinhdl, curpos, pnt);

print("Machine (X,Y,Z) = (", pnt.x, ",", pnt.y, ",", pnt.z, ")");

// Convert the position to Work coordinates.

TransInv(worktomachine, pnt);

print("Work (X,Y,Z) = (", pnt.x, ",", pnt.y, ",", pnt.z, ")");
```

Querying the Current Position

Application programs can query the current position of the robot by using the KinematicsCpoint command. This command will return the current "command" position (see Cpos) in any of the available coordinate systems being used by the robot. The KinematicsCpoint command can be used to simplify applications that rely on movements being "taught by example".

```
long      kinhdl;
dpoint    pnt;
transform worktomachine;

kinhdl = Kinematics(....);

// Set up the dynamic Work-to-Machine transformation.
```

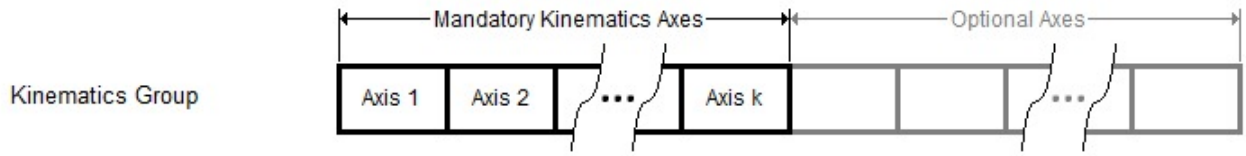
```
TransIdent(worktomachine);
....
DefWrkToMch(kinhdl, worktomachine);

// Get the current position in Path coordinates.

cpoint(kinhdl, pnt, PATHCOORD_WORK);

print("Work (X,Y,Z) = (", pnt.x, ",", pnt.y, ",", pnt.z, ")");
```

6.6.5 Understanding Point Storage Structure

When working with multi-axis kinematics, it is important to understand the correspondence between kinematics axes as they are defined by the KinematicsSetup command, point coordinates as they are stored in DPOINT variables, and path coordinates as they are defined by Path Curves. This is especially true when optional axes and non-spatial coordinates are being used.

Kinematics Axes

The kinematic axes defined by the KinematicsSetup command are ordered into two types: Mandatory axes related directly to the kinematics model first, then followed by any optional axes that have been defined by the user. This is show below:

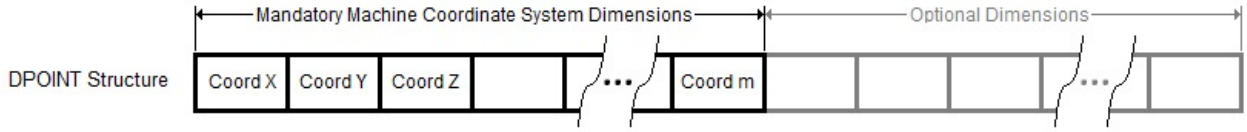


In this example, there are "k" mandatory axes. See Multi-axis Kinematics for descriptions of the mandatory axes required by the different kinematics models.

DPOINT Coordinates

dpoint variables are used to store points in any of the Machine Coordinate System, Work Coordinate System, Path Coordinate System, etc. See Basic Coordinate Systems and Additional Coordinate Systems. However, the basic structure of a dpoint variable is determined by the Machine Coordinate System of the kinematics model that the dpoint variable will be used with. Points in the Work Coordinate System and Path Coordinate System will have an identical structure.

The coordinate values stored in dpoint variables, are also ordered into two types: Mandatory Machine Coordinate System coordinates first, then followed by any optional coordinates that have been defined by the user. The mandatory machine coordinates always include three spatial axes (for X, Y, and Z), followed by zero or more other axes that depend on the kinematics model. Note that Z coordinates always exist in dpoint variables, even if the kinematics model does not use a Z coordinate. This is show below:

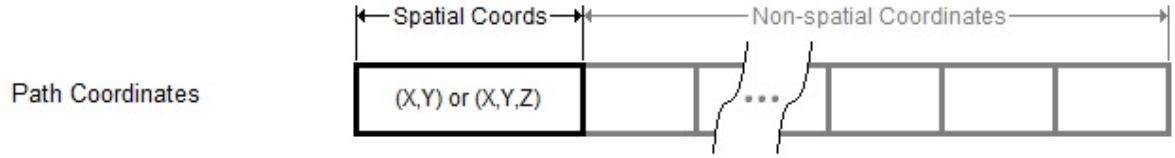


In this example, there are "m" mandatory machine coordinates. See Multi-axis Kinematics for descriptions of the mandatory machine coordinates required by the different kinematics models.

Path Curve Coordinates

The coordinate values originating from Path Curves, are also ordered into two types: Spatial coordinates first, then followed by any non-spatial coordinates that have been defined by the user. Spatial coordinates correspond only to the X, Y, and Z coordinates of points. If points require rotational coordinates such as

yaw, pitch, and roll, then these are considered to be non-spatial coordinates. Note that for 2-dimensional paths, the Z coordinate is *omitted* and does not exist in the Path Array containing the path curve. Path coordinates are show below:



Kinematics Axis and DPOINT Correspondence

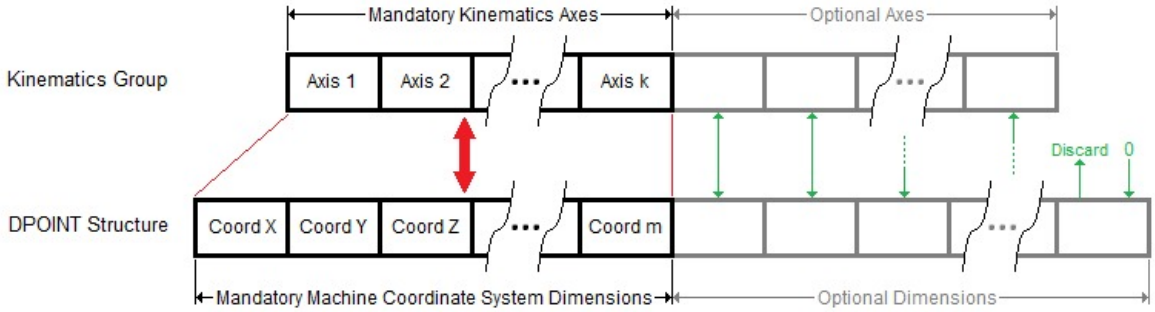
When points are converted from Machine Coordinate System DPOINT values to kinematics axis positions, the conversion is done as follows:

- Inverse kinematics mathematics is used to convert the mandatory machine coordinates into the mandatory kinematic axis positions.
- Optional coordinates are copied directly (1-to-1) from the DPOINT value to the corresponding axis position. If there are more optional DPOINT values than optional axis position values, then the excess DPOINT values are discarded. Note that the length of DPOINT variables is set so that there are never more optional axes than there are optional coordinates in DPOINT variables.

When points are converted from kinematics axis positions to DPOINT values, the conversion is done as follows:

- Forward kinematics mathematics is used to convert the mandatory kinematic axis positions into the mandatory machine coordinates.
- Optional coordinates are copied directly (1-to-1) from the axis position to the corresponding DPOINT value. If there are more optional DPOINT values than optional axis position values, then the excess DPOINT values are set to 0.

Note that depending on the kinematics model, the number of mandatory kinematics axes may be different from the number of mandatory machine coordinates. This is particularly true for 2-dimensional kinematics that work only with X and Y coordinates, where there will be 2 kinematic axes and 3 machine coordinates. In this case, the Z coordinate in the DPOINT variable will typically be ignored in the "inverse" direction and set to 0 in the "forward" direction.



Since all DPOINT variables in an application program share the same length (i.e. number of coordinates), the length of DPOINT variables must be defined to be long enough for any kinematics model that the application program will be using. For example, if one kinematics model requires 3 mandatory machine coordinates and 4 optional coordinates (for a total of 7), and a second model requires 4 mandatory machine coordinates and 2 optional coordinates (for a total of 6), then the DPOINT length must be set to 7. The length of DPOINT variables can be set using the PNT_LENTGH #pragma. See Working with Points.

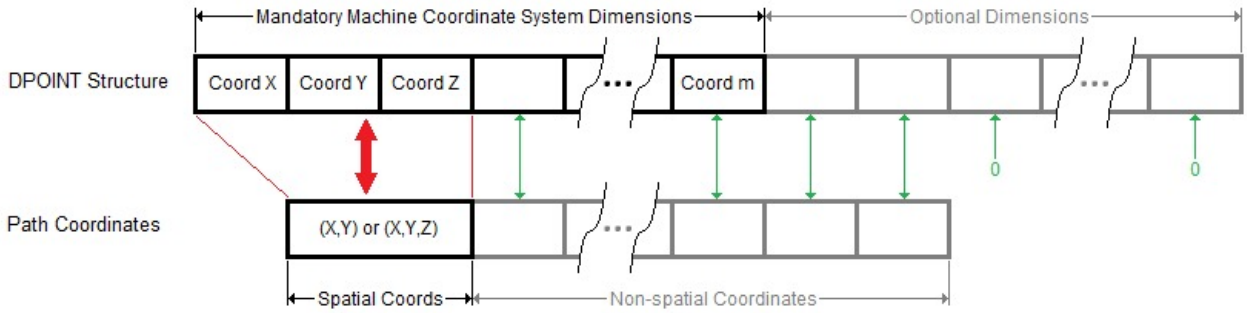
While it is possible to assign one DPOINT variable to another DPOINT variable, this should only be

done with great care if the two DPOINT variables are being actively used with different kinematics models. The controller in general has no understanding of the meaning of non-spatial and optional coordinates and therefore provides *no* support for converting these DPOINT values from one kinematics model to a different kinematics model.

DPOINT and Path Coordinate Correspondence

When points derived from path curves (e.g. while a path curve is being followed) are converted to dpoint values, the conversion is done as follows:

- Spatial coordinates (i.e. X, Y, and Z) from the path are transformed from the Path Coordinate System to the Machine Coordinate System and then placed directly into the mandatory machine coordinates of the dpoint variable. If the path contains no Z coordinate, then the path Z coordinate is set to 0 *before* the point is transformed.
- Optional path coordinates are copied directly (1-to-1) into the remaining DPOINT values. Note that this includes the non-spatial mandatory machine coordinates. If there are more remaining dpoint values than optional path coordinates, then the excess dpoint values are set to 0. If there are more optional path coordinates than remaining DPOINT values, then the excess path coordinates are discarded.



The handling of coordinate count mismatches between dpoint variables and path coordinates (i.e. discarding or setting to 0), is designed to facilitate the use of 2-dimensional path curves with 3-dimensional kinematics models (and vice versa) where this makes sense. For example, a single 2-dimensional path for a character font could be used with both 2-dimensional and 3-dimensional kinematics models.

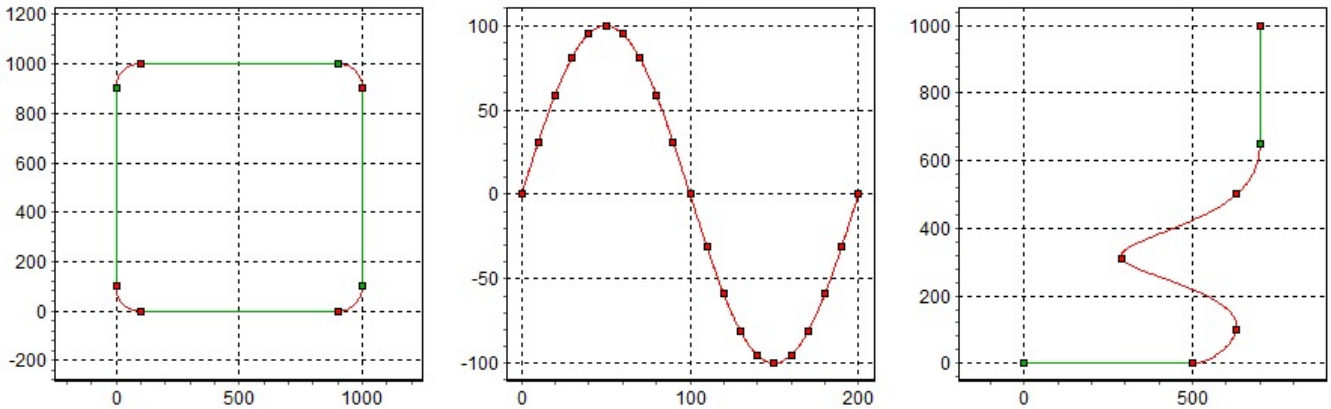
6.6.6 Path Curves

Many motor controller applications require the synchronization of multiple axes to move a tool along a specific curve through space. For example, the 2-dimensional space of a pen in an XY plotter or the 3-dimensional space of a tool in a numerical milling machine or a coordinate measuring machine. The ApossIDE supports this using multi-dimensional “path curves”.

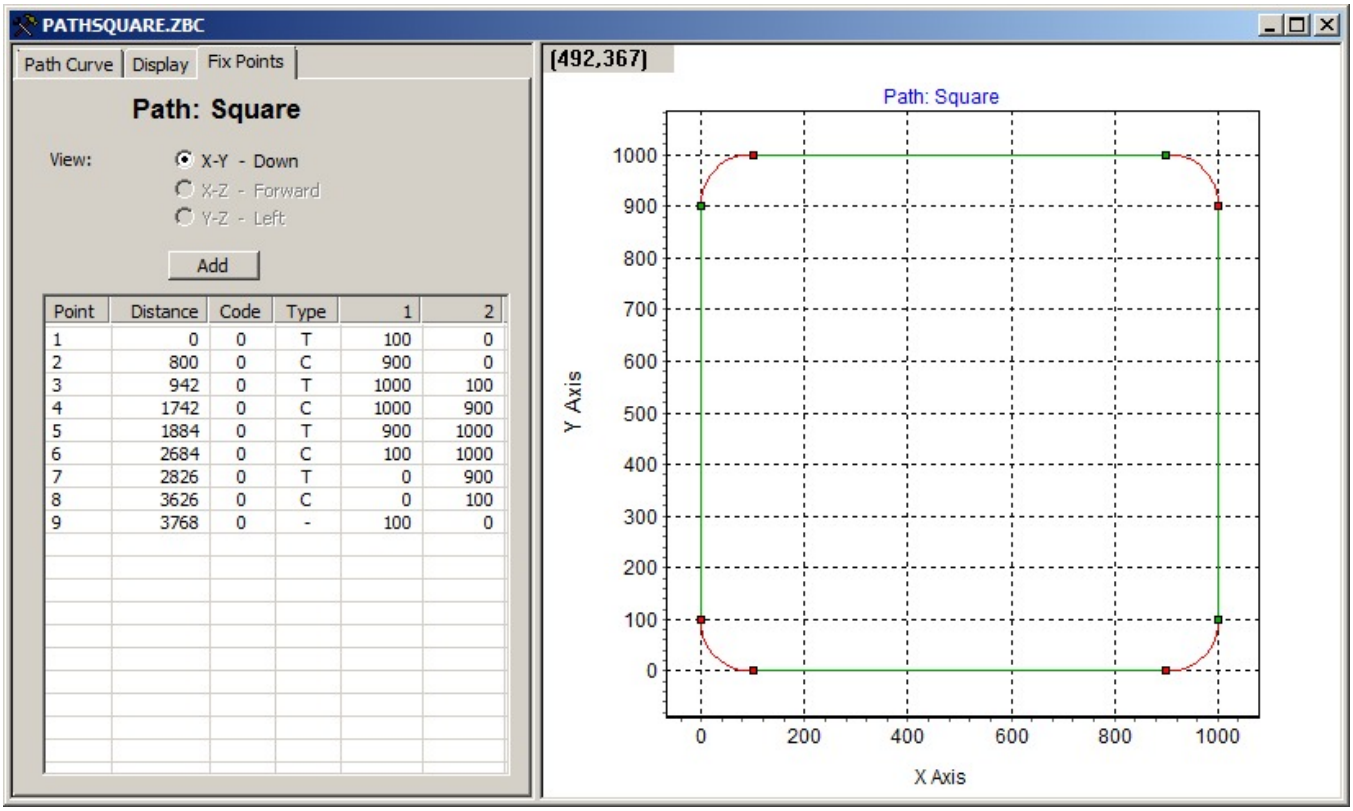
In the ApossIDE, the path is defined by a series of “fixed points” that are placed along the desired path at user-specified positions. The ApossIDE will then generate a smooth path that will move exactly through each of the fixed points. “Smooth” means that at each fixed point, both the velocity and acceleration (i.e. first and second derivatives) when arriving at the fixed point are identical to the velocity and acceleration when leaving the fixed point. Mathematically, the velocity and acceleration are continuous functions. Note that the “jerk” (i.e. third derivative) will not be a continuous function; it will have a discontinuity or “jump” at fixed points. The ApossIDE creates the smooth curve by fitting a cubic spline through the fixed points.

Paths may either be cyclic or noncyclic. Cyclic paths are intended to be repeated and the ApossIDE will facilitate this by ensuring that the velocity and acceleration match at the beginning and end of the path. Note that cyclic paths can be either closed or open. In a closed path, the first and last fixed point will be identical making the path

loop back on itself. In an open path, the first and last fixed points are different and the path will represent one cycle of a repeated pattern (e.g. a sine wave). A noncyclic path does not have any of these constraints and the end point positions, velocities, and accelerations can all be different. Simple 2-dimensional examples of these paths are shown below.



The multi-dimensional path is then separated into multiple synchronized 1-dimensional movements, one movement for each axis. This is done internally in the controller; the user need not be concerned with the mechanics of this separation process. These movements are implemented using standard CAM curves. The CAM curves are generated in such a way that they all use the same master coordinate as input; the system will stay synchronized and follow the original curve as long as the same master coordinate is used to drive each CAM curve. The master coordinate used for the CAM curves is an approximation of the arc length distance along the original multi-dimensional path. The example below illustrates a simple cyclic 2-dimensional path along with its fixed points. The "Distance" column represents the master coordinates that will be used for the CAM curves. Note that this is a closed cyclic path since the first and last fixed points are identical.



Point	Distance	Code	Type	1	2
1	0	0	T	100	0
2	800	0	C	900	0
3	942	0	T	1000	100
4	1742	0	C	1000	900
5	1884	0	T	900	1000
6	2684	0	C	100	1000
7	2826	0	T	0	900
8	3626	0	C	0	100
9	3768	0	-	100	0

The ApossIDE supports paths that consist of both spatial (i.e. X, Y, Z) and non-spatial (e.g. rotation) components. Currently, a total of up to 10 dimensions may be specified. However, the ApossIDE supports only 2 and 3 spatial dimensions; the remaining dimensions are assumed to be non-spatial. Only the spatial dimensions will be included in the calculation of arc length for the master coordinates of the CAM curves.

The coordinate system used for the fixed points is completely arbitrary; any measurement unit may be used as long as all spatial dimensions use the same units. However, the coordinates of fixed points are restricted to integer values. As a result, the measurement units will need to be scaled in order to achieve the positioning accuracy required by the end application. For example, fixed point coordinates based on millimeters will need to be scaled by 100 if 0.01 mm accuracy is required. Note that the CAM curve master coordinate (which is an arc length approximation) will then also be measured using this scale factor. The end user application must take this scale factor into account when driving the CAM curves.

Path curves can be created and managed either with the CAM Editor within the ApossIDE or with an external user-supplied application. User-supplied applications can gain access to the path curve functionality through the ZbPath DLL. Functionality for downloading the resulting curve data arrays to the controller is available using the ZbMoc DLL.

The basic steps for using a path curve with the CAM Editor are as follows:

1. In the CAM Editor, create the path curve by defining the fixed points that will represent that curve.
2. Save the curve in a Configuration ".zbc" file.
3. Download the configuration to the controller.
4. Create, download, and execute the application program that will drive the axes based on the path curves.

The basic steps for using a path curve generated by an external user-supplied application are as follows:

1. In the user application, generate the fixed points that will represent that curve.
2. Call the various functions in the ZbPath DLL to convert the path curve fixed points into one or more DIM arrays.
3. Call the ZbMoc DLL functions to connect to the controller and download the DIM arrays.
4. Create, download, and execute the application program that will drive the axes based on the path curves.

The ApossIDE supports two methods of handling the DIM arrays required for path curves:

1. The path curve can be converted to a single "path array" and downloaded as a single DIM array. This will require less DIM memory and the application program which uses the path will be significantly easier to implement. However, this requires a controller with firmware version 8.1.07 or later.
2. The path curve can be converted to multiple CAM arrays and downloaded as multiple DIM arrays. This requires more DIM memory and the application program which uses the path will require significantly more programming experience to implement. This method may also be used only for Cartesian kinematics. It is recommended that newer controller firmware be used when applications will be using path curves.

6.6.6.1 Additional Coordinate Systems

When using Path Curves in an application, two *additional* coordinate systems are involved in the transformation of the original path as defined in the Path Editor, to the final position of the path in the real world:

- Path Coordinate System
- Cyclic Coordinate System

See Basic Coordinate Systems for a description of the other coordinate systems.

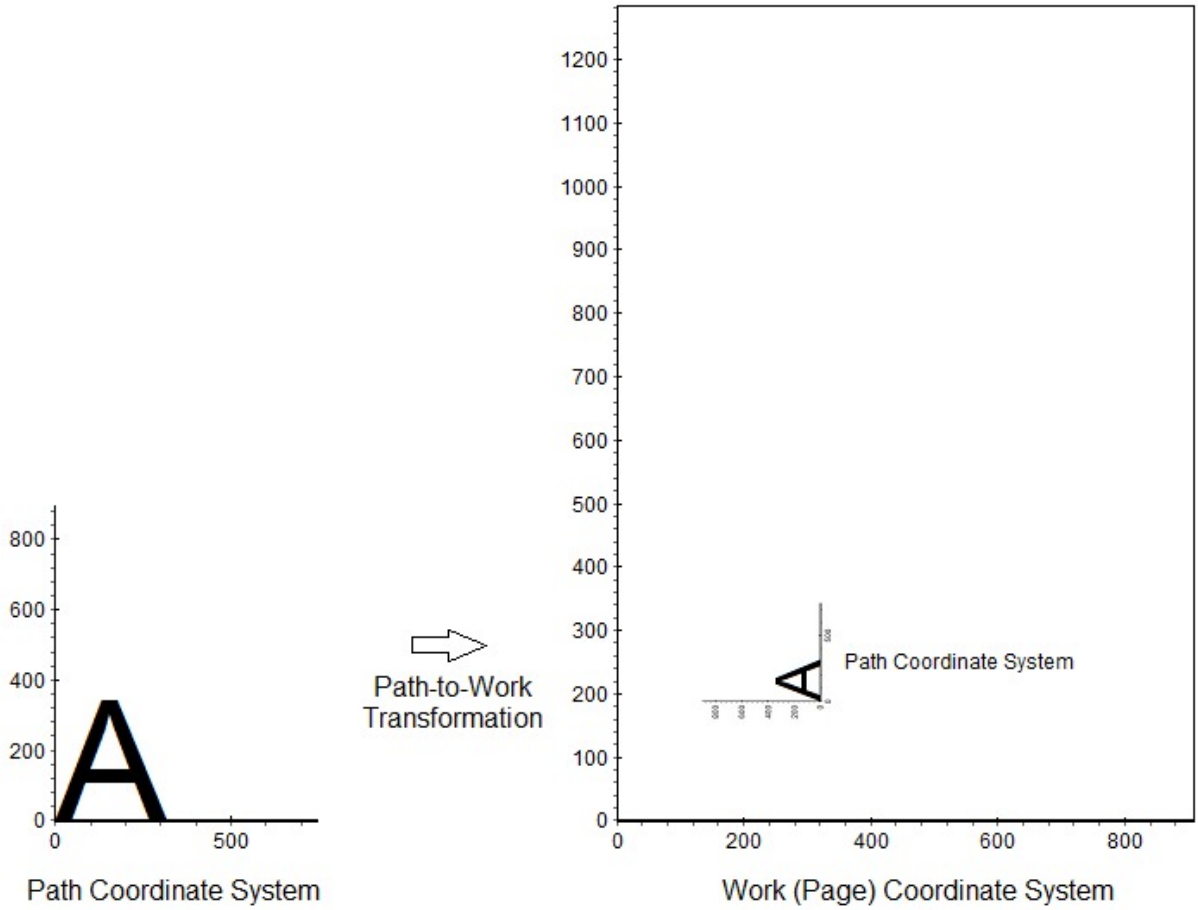
Path Coordinate System

This is the coordinate system that is used in the Path Editor to define the basic shape of the curve. This may have either

two or three spatial axes. The user is free to choose whatever units are appropriate for the path curve. For example, if font characters are being defined, then typographical "point" units (i.e. 1/72 inch) could be used. However, the units of all spatial axes must be the same. This is necessary for correct arc lengths to be calculated.

Typically, when a path is placed into the Path Coordinate System, it will somehow be placed at a position that is "natural" relative to the origin. Often, this will be the start of the path. For example, if the path represents the font strokes for the character "A", then the bottom-left corner of the "A" might be placed at (0,0). However, this is not necessary; the path may be placed anywhere in the Path Coordinate System.

If no other requirements needed to be considered, then a single transformation would be required to map the character into the Work Coordinate System. For example, consider a page of paper and the character "A" is to be drawn somewhere on the page. Then the Work Coordinate System would match the page of paper and there would be a transformation that specified how the path for the character "A" (defined in the Path Coordinate System) mapped onto the page. The transformation would be something similar to "Scale to the desired font size, rotate to the desired text angle, and then move the character to the desired location on the page." This transformation will be called the Path-to-Work transformation. See Transformations for a description of how to create transformations in application programs.

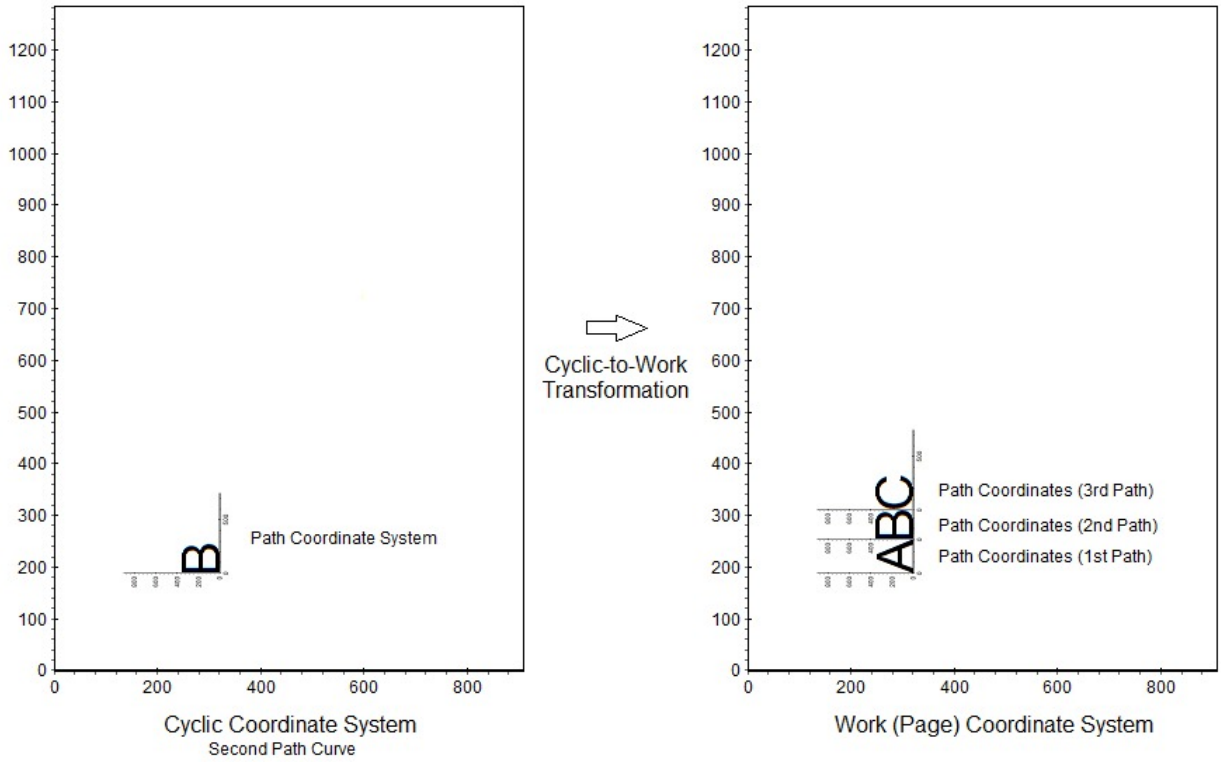


Cyclic Coordinate System

The Cyclic Coordinate System is important only when open cyclic path curves or sequential non-cyclic paths are being used (see Path Curves for a description of these type of curves). In both of these cases, the path curves will "walk along" through the real world as the path curves cycle. For example, consider individual path curves for each character of the alphabet, and the string "ABC" is to be drawn on a page of paper. As each new character is drawn, the character's path must be shifted by the appropriate character spacing. This "shift" is done by splitting the Path-to-Work transformation into a Path-to-Cyclic transformation and a Cyclic-to-Work transformation. This effectively creates an intermediate coordinate system between the Path Coordinate System and the Work Coordinate System.

The Path-to-Cyclic transformation specifies where the curve would be in the Work Coordinate System if no cyclic shift is required. This is the same as the Path-to-Work transformation described above. It is the user's responsibility to define this transform. The Cyclic-to-Work transformation accounts for all the changes resulting from cyclic shifts. This transformation is handled automatically by the path handling functions in the controller.

The Cyclic-to-Work transformation is initialized to an identity transformation (i.e. no transformation) when the first path curve in a sequence is drawn. The first path will be drawn by the PathStart command. The Cyclic-to-Work transformation is then recalculated at the beginning of each new path curve cycle. The Cyclic-to-Work transformation is calculated so that the start point of the next path matches the end point of the previous path. This is done so that the path as a whole is continuous and does not contain gaps. Note that the Path-to-Cyclic transformation of each individual character must be specified as if that character is the first character in the string. For this example, this means all character path curves will share the same Path-to-Cyclic transformation. The Cyclic-to-Work transformation will then be used to "walk" the sequence of paths along the page.



The complete sequence of transformations from the Path Coordinate System to the motors is as follows:

Path coordinates (Cartesian, path-specific)
→ Cyclic coordinates (Cartesian, internal)
→ Work coordinates (Cartesian, user-specific)
→ Machine coordinates (Cartesian, kinematics-specific)
→ Kinematics coordinates (kinematics-specific)
→ Axis coordinates (user units)
→ Motor (quadcounts)

6.6.6.2 Path Fixed Points

Fixed points are used to define the basic shape of the path curve. The CAM Editor will fit a mathematical spline through these fixed points in order to create a smooth curve. Note that you should not define more fixed points than are necessary to adequately produce the final desired curve. Adding more points than necessary will consume controller resources without a corresponding increase in benefit. For paths that already contain a dense

set of fixed points, the CAM Editor is able to automatically ignore unnecessary fixed points (see Thinning Fixed Points). This is common for path curves that have been created by other path generatation tools and then imported into the CAM Editor.

Each fixed point used to define a path, will consist of a set of attributes for the fixed point, followed by one coordinate corresponding to each of the synchronized axes. The spatial coordinates must precede the non-spatial coordinates. The attributes will consist of a "code", a "type", and other optional information related to the fixed point. The 2-dimensional example in Path Curves contains the following 9 fixed points (4 values per fixed point):

```
Point #1: (0, T,  100,    0)
Point #2: (0, C,  900,    0)
Point #3: (0, T, 1000,  100)
Point #4: (0, C, 1000,  900)
Point #5: (0, T,  900, 1000)
Point #6: (0, C,  100, 1000)
Point #7: (0, T,    0,  900)
Point #8: (0, C,    0,  100)
Point #9: (0, -,  100,    0)
```

The fixed point "code" values are maintained by the ApossIDE but (except for the "Segment Count" shown in the Path Curve Tab) the ApossIDE assigns no meaning to the values. The values are downloaded to the controller along with the path curve data and are available to the application program for whatever purpose the program may require of them. For example, they might be used to indicate “pen up” and “pen down” positions. These values are 3-byte unsigned numbers in the range 0-0x00FFFFFF (16777215).

The fixed point "type" is used to determine the type of the single path segment that follows the fixed point (i.e. that goes from this fixed point to the next fixed point). The following fixed point types are supported:

C	Curve segment.
T	Tangent (straight line) segment.
CZ	Trapezoidal curve segment.
C3	Third order (cubic) curve segment.

These fixed point types affect the shape of the path curve in exactly the same way as they affect the shape of CAM curves. See the CAM curve Fix Points Tab for a more detailed description of the effects of the different fixed point types.

A path will normally consist of a sequence of curve (C) and tangent (T) segments. Tangent segments are fixed straight lines while curve segments are “flexible”. In order to produce a smooth curve, the ApossIDE will adjust the curvature of the curve segments; tangents are fixed and will not be adjusted in any way. The requirement for a smooth path imposes some restrictions on fixed points and the sequence of segment types:

1. Adjacent fixed points must not specify the same point (i.e. zero-length segments are not allowed).
2. Any number of curve (C) segments may be placed sequentially.
3. If two tangent segments exist, then they must be separated by at least one curve segment (which allows the controller to make the path smooth when connecting the two tangents). Any of the three curve segment types may be used.
4. Trapezoidal (CZ) and third order (C3) curves may only be used as single segments connecting adjacent tanget segments.
5. For a cyclic path, the last segment is assumed to follow the first segment. This means that the first and last segments must not both be tangents (or rule #2 would be violated). It also means that trapezoidal and third order segments can be used at the ends of a path as long as the adjacent segments are tangents.
6. For a noncyclic path, both the first and last segments may be tangents since there is no requirement that velocities and accelerations match. As well, the first and last segments may be either trapezoidal or third

order curves as long as the adjacent segment is a tangent.

The ApossIDE will ensure smooth paths by fitting cubic splines to any sequence of 2 or more curve (C) segments. However, in the case where there is a single curve segment connecting two tangent segments, it is not generally possible to connect the tangents smoothly by using a single cubic spline segment. In this specific situation, the ApossIDE gives the user more control over the shape of the connecting curve by providing the following segment types for this curve segment:

1. Curve (C): The connecting curve will be a single 5th order polynomial. This polynomial is capable of smoothly matching both velocity and acceleration at both ends of the segment where it connects to the adjacent tangents. However, depending on the situation, 5th order polynomials can have some undesirable side effects (such as excessive oscillations in position). In these cases, it may be helpful to select one of the other two curve types.
2. Trapezoidal (CZ): The connecting curve will be handled as a *pair* of 3rd order polynomials. This curve is also capable of smoothly matching both velocity and acceleration at both ends of the segment.
3. Third order (C3): The connecting curve will be a single 3rd order polynomial. This polynomial will smoothly match velocity at both ends of the segment but it will *not* match accelerations.

6.6.6.3 Start and End Directions and Velocities

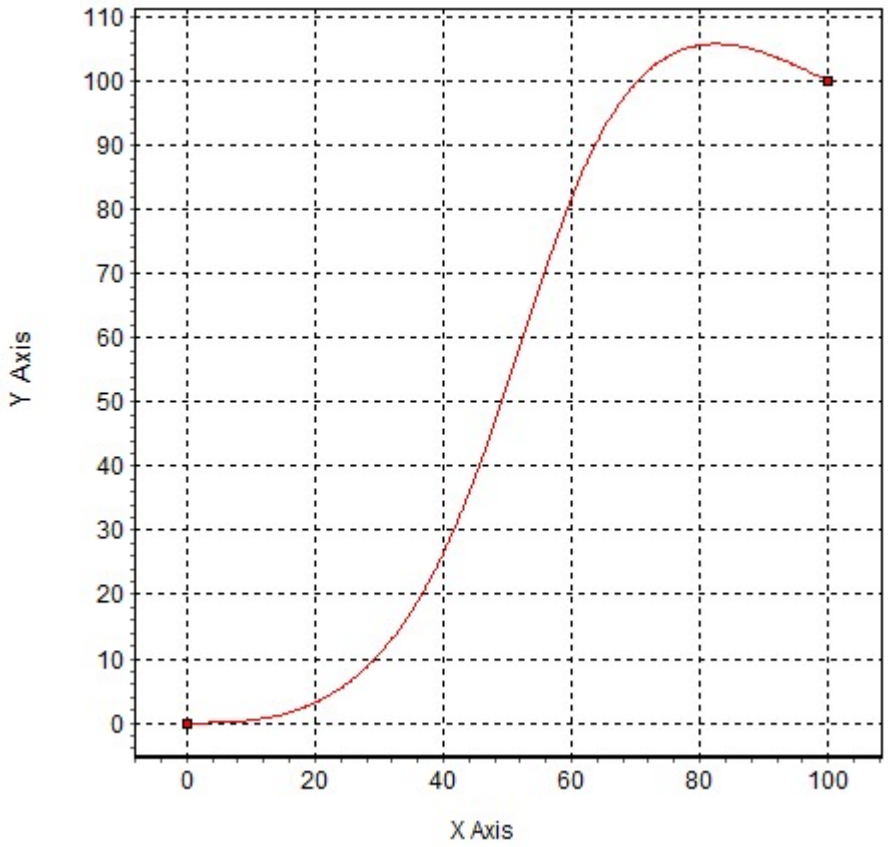
For noncyclic paths, the ApossIDE needs the starting and ending directions of the path to be specified. This is done using directions vectors that the user can set; one vector for the starting direction and one vector for the ending direction. Each coordinate of the direction vector will correspond to one of the spatial coordinates of the path. For example, a starting vector of (1,1) indicates that the path starts upwards and to the right at a 45 degree angle. The magnitude of the vector is unimportant; (2,2) is equivalent to (1,1).

Direction vectors can also be set to all 0's. This indicates that the path starts or ends with a "floating" (or "low energy") direction that is generally oriented in the direction of the path. Mathematically, this is done by starting or ending with a 0 acceleration rather than a specific direction.

For non-spatial coordinates, starting and ending conditions must be specified as velocities where each velocity will consist of an integer numerator and a non-0 integer denominator. In certain circumstances (specifically, when a path starts with a single curve segment followed by a tangent segment or ends with a tangent segment followed by a single curve segment or when the entire path consists of a single curve segment), an acceleration is also required. In these cases, an acceleration of 0 is used

Note that paths that start with a tangent do not need starting directions or velocities to be specified. The starting direction and velocity will be fully determined by the tangent. Similarly, paths that end in a tangent do not need ending directions or velocities to be specified.

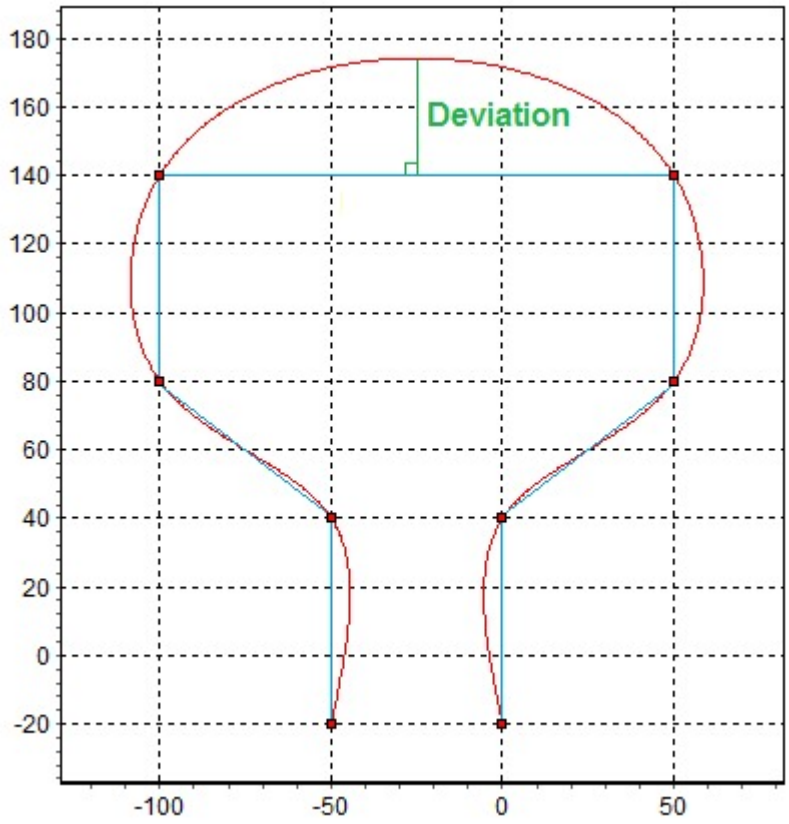
As an example, the following is a simple path consisting of a single curve segment where the starting direction is set to (1,0) and the ending direction is set to (2,-1).



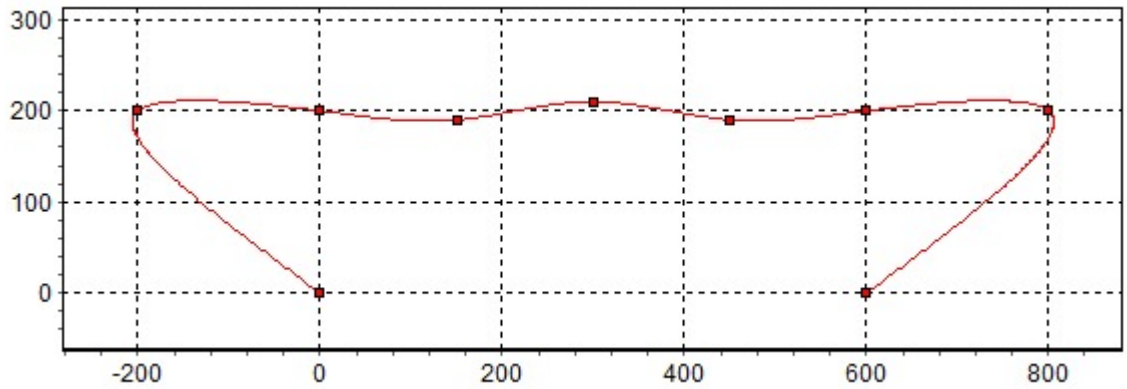
6.6.6.4 Thinning Fixed Points

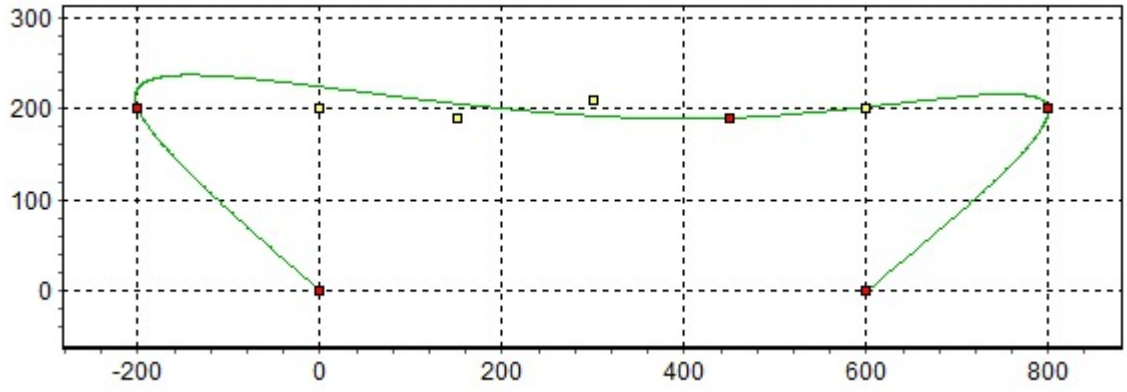
The CAM curves created for path curves will normally pass exactly through every fixed point. For some applications that define the path curve using a large number of closely spaced fixed points, this can result in very large CAM curves, very large DIM arrays that must be downloaded and saved on the controller, and an unnecessary waste of resources on the controller. In these cases, the original fixed points can be automatically "thinned". This process will exclude fixed points from the CAM curves as long as the final CAM curves still follow the original non-thinned fixed point path "closely enough".

"Closely enough" is determined by a user-specified "maximum allowable deviation" value. The deviation value will be the maximum perpendicular distance from any point on the path curve to the "piece-wise" straight line connecting all the fixed points. This is shown below. The red line is the path curve, the blue line is the piece-wise straight line, and the green distance is the deviation. The actual maximum deviation for the curve is calculated by the ApossIDE and shown in the Path Curve Tab.

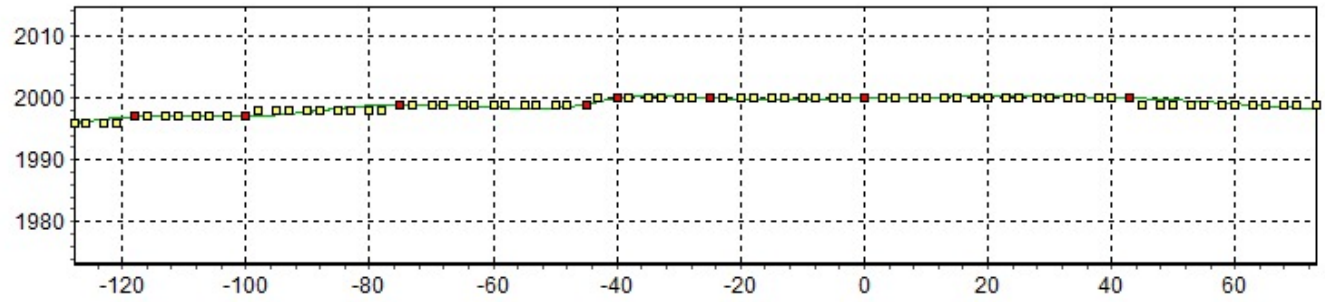


The two illustrations below show a path curve that passes through every fixed point (red curve) and the same path (green curve) after it has been thinned to a maximum deviation of 70. The yellow points show which fixed points have been thinned. Note that the green curve no longer passes exactly through these fixed points. No point on the green curve is further away than 70 from the piece-wise straight line connecting the all the fixed points (including the yellow fixed points).





The illustration below shows a more realistic example of thinning. This is a small portion of a very densely sampled circular path. The maximum allowable deviation has been set to 1 (the minimum). Even with this small deviation, the number of fixed points has been reduced by approximately 90%. The red fixed points have been included in the final path and the yellow fixed points have been thinned and are no longer included in the final path.

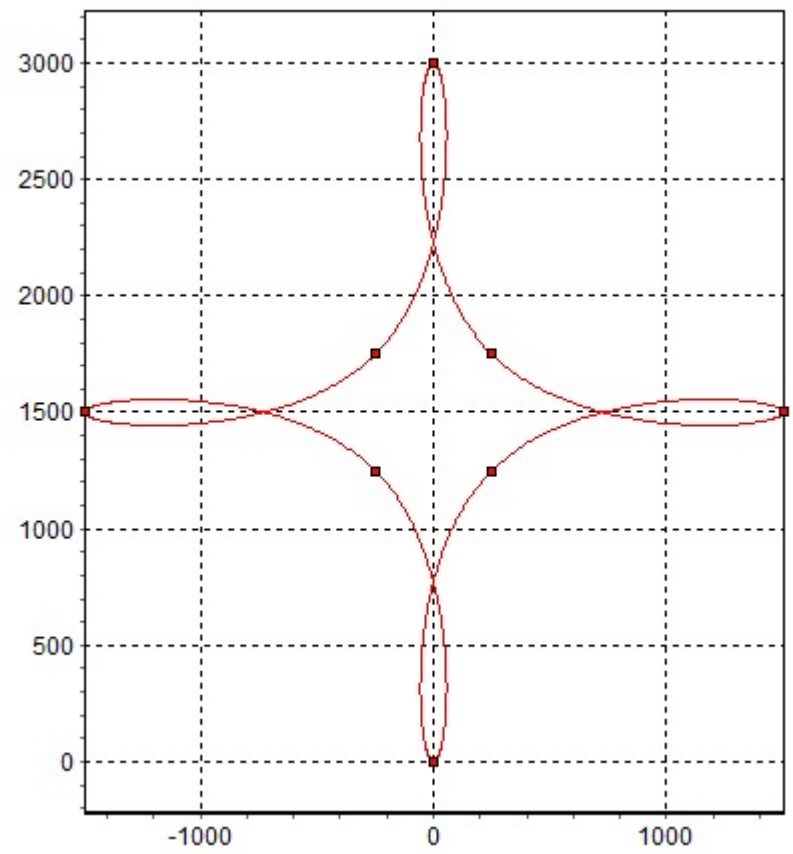


Note that since paths will contain curved segments and the deviation is calculated based on piece-wise straight lines, paths will have a non-0 deviation even when all fixed points have been included. Hence, reducing the maximum allowable deviation to a value less than this deviation may reduce the number of fixed points in other parts of the curve but it will not reduce the actual maximum deviation (see Path Curve Tab). This will normally happen if fixed points are spaced too far apart. It should be possible to solve this problem by inserting more fixed points within the path at the positions where the deviation is the worst.

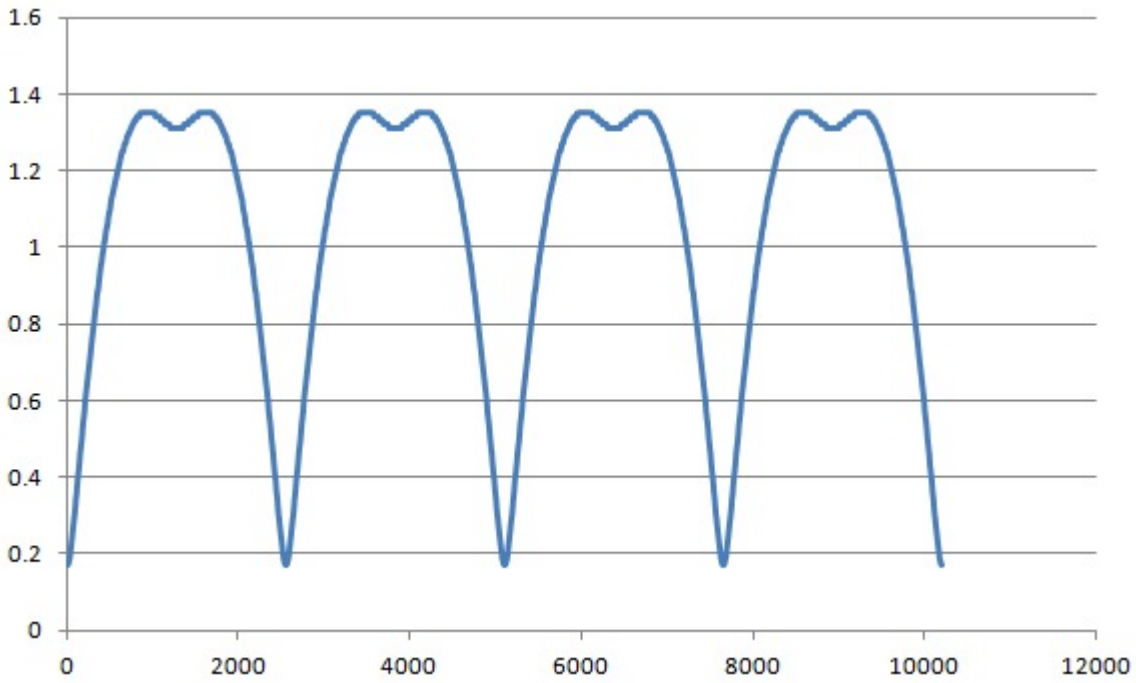
6.6.6.5 Arc Length CAM Curves

As stated previously, the CAM curves that result from a path will have master coordinates that approximate arc length along the curve. However, an artifact of the mathematics used to create the CAM spline curves causes the velocity within each of the individual spline segments to vary. This variation can sometimes be substantial, depending on how the curvature of the path changes. This can be extremely important for certain applications that must follow a path with constant velocity.

For example, the following illustration shows a simple path that contains an area of very high curvature at the outside point of each of the four lobes. The path starts at the bottom of the lower lobe at (0,0) and goes counterclockwise around the shape. The approximate arc length of this path is 10200 units, so the master length of the CAM curves will be 10200 mu.

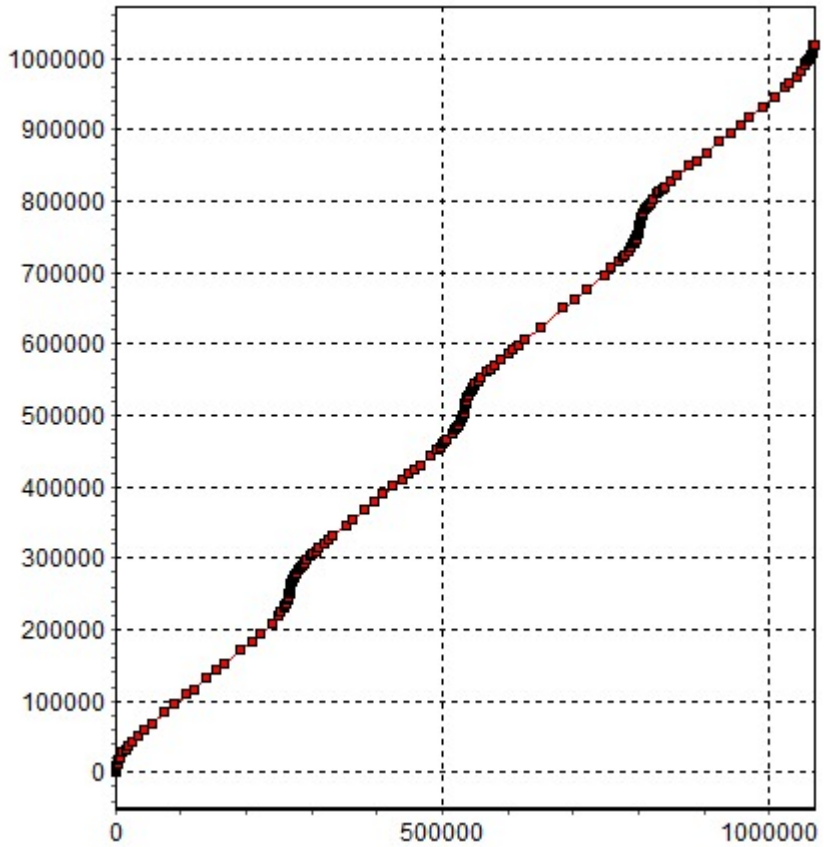


If the master coordinate of these CAM curves were to be followed at a constant rate (in mu/sec), then the actual linear speed along the path (in units/sec) will be shown in the chart below. Note that the linear speed is quite low where the path curvature is high. Also note that the linear speed is higher than 1 (i.e. faster than the master coordinate rate) in the areas of low curvature. In a way, these areas of higher speed compensate for the areas of low speed so that the average speed within each spline segment is more or less uniform from segment to segment.

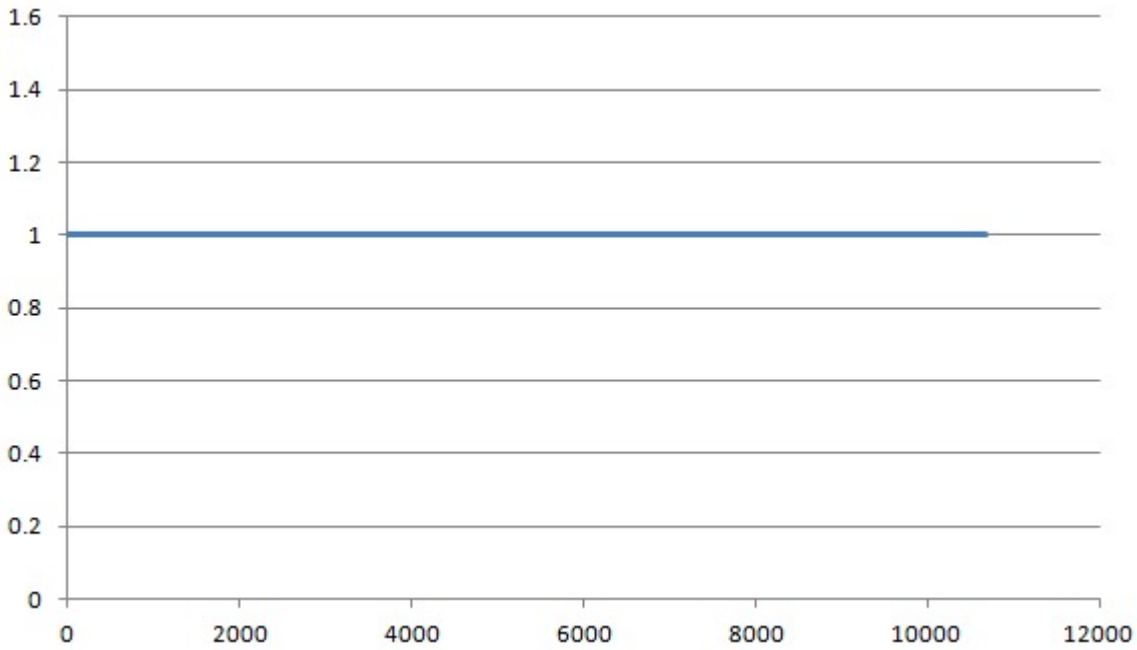


This effect can be “corrected” by creating an auxiliary “arc length CAM curve”. This is an additional CAM curve that is then downloaded along with the path curve. Constant velocity along the path curve can then be achieved by using the arc length CAM curve to drive a virtual axis. The output of the virtual axis is then used to drive the CAM curves for the real axes.

The following illustrations shows the arc length CAM curve that would be created for the above curve. The scaling factor for this curve was set to 100 (i.e. both the master and slave coordinates have been scaled by a factor of 100). This scaling is necessary in order to provide enough accuracy. The actual arc length of the original path is 10681.42 units (slightly longer than the approximate length of 10200). Hence, the master length of this CAM curve will be 1068142 and the slave length will be 1020000. Note that arc length CAM curves can have very dense fixed points. As with path curves, the ApossIDE gives the user the opportunity to thin the fixed points. However, there is a trade-off since fewer fixed points usually results in larger velocity variations. Displaying the arc length CAM curve in the ApossIDE allows the user to investigate these trade-offs.



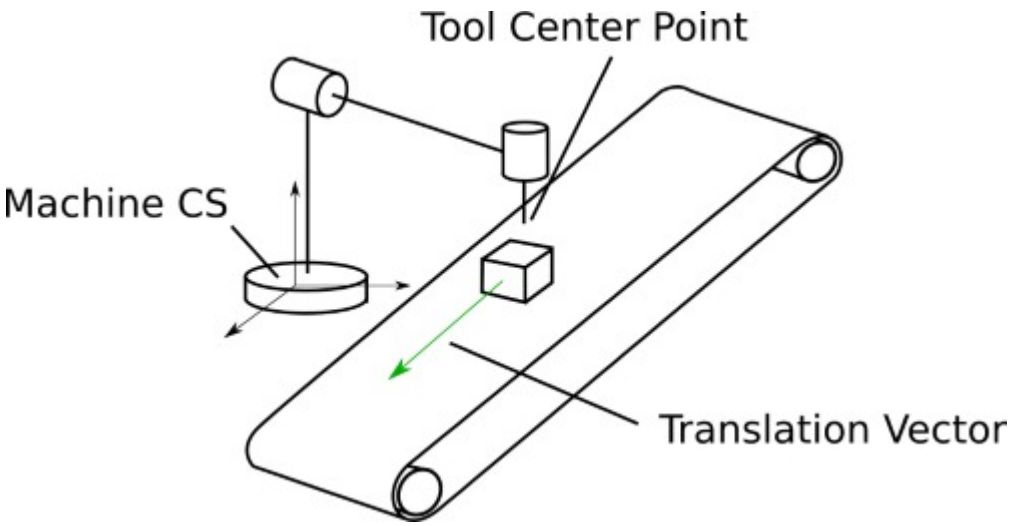
If the master coordinate of the arc length CAM curve were to be followed at a constant rate (in mu/sec) and the slave output values of that CAM curve used as input master coordinate values to the path CAM curves, then the actual linear speed along the path (in units/sec) will be shown in the chart below. Note that this is very close to constant linear speed along the path (i.e. 1 unit/sec). Exactly how constant this speed is, will be determined by the scaling factor and thinning chosen by the user when the arc length CAM curve is created.



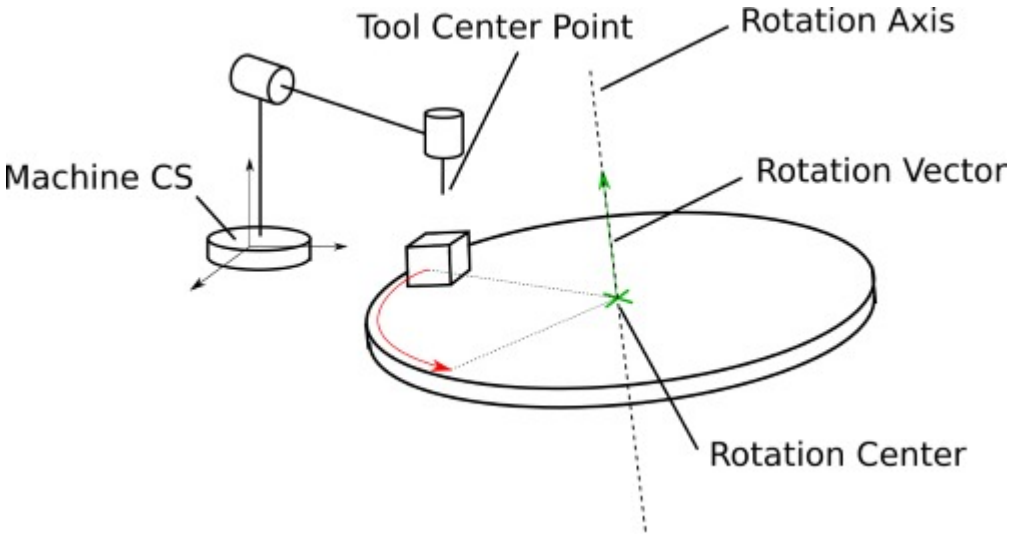
6.6.6.6 Kinematics synchronization

It is possible to synchronize a path to a translation or a rotation. This can be used if a path needs to be executed on a conveyer belt or a rotary table. The synchronization mode can be switched between translational and rotational with the machine parameter KIN_SYNC_TYPE (1=translational, 2=rotational). As long as the synchronization is active and a path is executed, every movement of the conveyer belt or rotary table is added to the movement of the tool center point of the machine.

Translational mode



Rotational mode



The master of these synchronization movements is a virtual counter module, which can be set with the machine

parameter MACHINE_PISRC_VIRTCOUNT.

If synchronization is in translational mode, the direction of the movement can be described with the machine parameters KIN_SYNC_VEC[X/Y/Z]. These values will be directly multiplied with the quad counts of the master encoder input and are a scaling factor between quad counts and machine units. Since these are integer values they are divided by KIN_SYNC_FPSCALE in order to provide the possibility to fractional scaling.

If synchronization is in rotational mode, the rotation axis can be described with a rotation vector, and a rotation center. These values are the KIN_SYNC_VEC[X/Y/Z] and KIN_SYNC_ROTC[X/Y/Z]. The rotation center point can be anywhere on the axis. In this mode the quad counts are mapped by the norm of the rotation vector divided by KIN_SYNC_FPSCALE.

To turn the synchronization on and off you need to set two flags: KIN_SYNC_STARTFLAG and KIN_SYNC_STOPFLAG. The KIN_SYNC_STARTFLAG indicates that the next path that is executed will be synchronized. The KIN_SYNC_STOPFLAG indicates that the synchronization stops with the stop of the next path. Whenever the synchronization is started or stopped the respective flag is set to 0. IMPORTANT: the STOPFLAG should always be set with the STARTFLAG since it is not allowed to make a new PATHSTART movement when still being synchronized. However there can be special cases where the synchronization is still needed after the path is finished.

Example

This example shows a kinematics synchronisation with a turntable which has a variable movement around Z.

```
MACHINE_PARAM(0, MACHINE_PISRC_VIRTCOUNT) = 4     // set the master of the
movment as the virtual counter module number 4
MACHINE_PARAM(0, KIN_SYNC_TYPE)  = 2              // select the rotary sync mode
MACHINE_PARAM(0, KIN_SYNC_VECX)  = 1              // set the x-component of the
rotation vector
MACHINE_PARAM(0, KIN_SYNC_VECY)  = 0              // set the y-component of the
rotation vector
MACHINE_PARAM(0, KIN_SYNC_VECZ)  = 0              // set the z-component of the
rotation vector
MACHINE_PARAM(0, KIN_SYNC_ROTCX) = 0              // set the x-component of the
rotation center
MACHINE_PARAM(0, KIN_SYNC_ROTCY) = 0              // set the y-component of the
rotation center
MACHINE_PARAM(0, KIN_SYNC_ROTCZ) = -10            // set the z-component of the
rotation center
```

6.6.7 Working with Paths

This is intended to describe what paths are and how to use the various PATH commands in a user program.

General Path Information

A path is a sequence of points in a 2-dimensional or 3-dimensional space. The points can be connected in diffrent ways to shape the path. The points can be linked together as splines, tangents or corner points. A combination of all the diffrent point types allows the user to design a path of their needs. The CAM-Editor tool inside the ApossIDE offers a graphical way to design paths. See CAM Editor. Beside the coordinates of the points, point types and the calculated distance between the points, it is also possible to add a user code to the path. This code can be read back while the path is executed in the controller and the application program can react accordingly.

The path is stored as an array structure and when downloaded is also available in an **DIM** array inside the MACS controller. The final path movement is calculated at real time with the information of the running path array. so it is also possible to manipulate the path array in the application program.

The most benefit is when paths are used together with the kinematics (as Carteasian, scara, dual scara and delta) that is also provided by the MACS firmware.

Path Array

Paths are stored in a given sturcure in the array. See CAM Array Structure for a complete description of the structure of Path curve arrays. There can be more than one path in one array. That allows the user to store for example a whole font in one array. The ApossC command PathArrayIndex can help to find the correct start index of a path or segment in an array.

Kinematics

Paths should be used together with the kinematics (as Carteasian, scara, dual scara and delta) that is also provided by the MACS firmware. Once you have set-up a kinematics you can simply use the PathStart command to run a predefined path stored in a array. Learn more about kinematics here Working with Kinematics

6.6.7.1 Program Access to Paths

In various cases, it can be useful for application programs to directly access the contents of the path arrays created by the Path Editor. For example, a program may need to "look ahead" at the fixed point codes as a path is being followed so that it can handle "pen up" and "pen down" functions.

The SysDef.mi include file provides the following #define statements that programs may use to allow easier access to the different elements within the array:.

PathSegType(type)

Return the actual type of this segment within the given path segment.

PathSegCode(type)

Return the user code associated with this segment.

PathSegCorner(type)

Return the if the start of this segment is a "corner" point.

PathSegStop(type)

Return the if the start of this segment is a "stop" point.

Furthermore there is a PathArrayIndex function. This PathArrayIndex function is designed to allow the application to easily locate the various data blocks within the Path array structure. The application may then use normal array access to read and modify the attributes of a Path curve.

Example

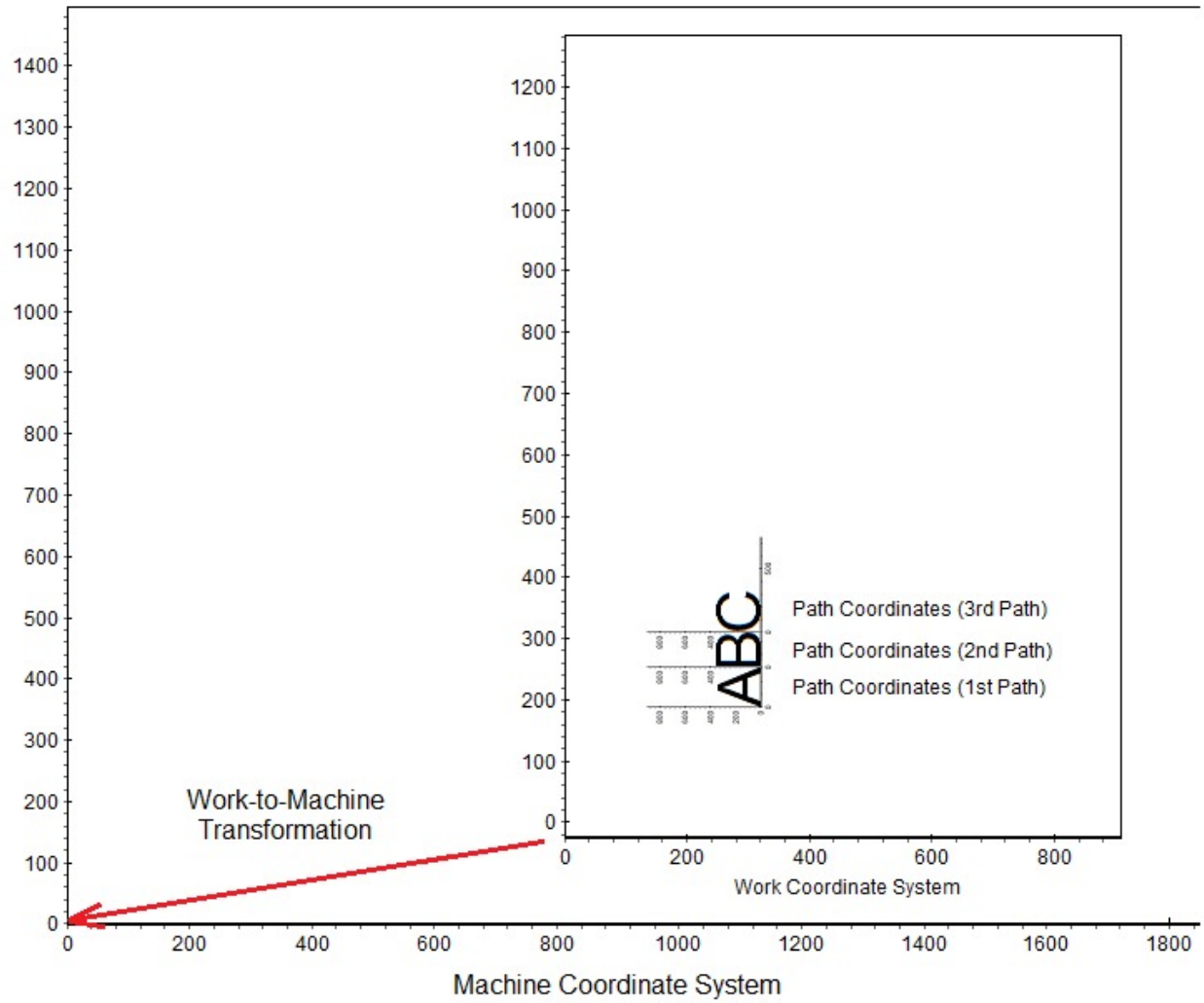
This example checks to see if the start of the *next* segment is a "stop" point.

```
dim long PthCrv[85];

long NextStop(long curcrv, long curseg)
{
    // Index of NEXT segment block.
    long seg = PathArrayIndex(PATH_INDEX_SEGMENT, PthCrv, curcrv, curseg+1);

    // Segment type of next segment.
    long type = PthCrv[seg+PTH_SEG_TYPE];

    return(PathSegStop(type) ? 1 : 0);
}
```

6.6.8 Dynamic Systems

Imagine that the entire Work Coordinate System is somehow moving in the real world. For example, the Work Coordinate System may represent a page of paper that is moving on a conveyor belt or somehow being rolled along. In contrast, the Machine Coordinate System is associated with the robot itself and will generally be fixed in time and space; the conveyor belt will move in front of the robot. So at each point in time, the page (i.e. the Work Coordinate System) will need to be placed "further along" within the Machine Coordinate System. The Work-to-Machine transformation has become a dynamic, time-dependent, transformation.

In this case, the Work-to-Machine transformation must be updated by the application and passed to the path handling functions as often as is necessary for the controller to track the movement of the path in the real world. The Work-to-Machine transformation is passed to the path handling functions using the KinematicsWorkToMc command. The KinematicsWorkToMc command is designed to be executed *while* a path is being followed.



The diagram above illustrates multiple path curves (i.e. for 'A', 'B', and 'C') being drawn on a page as it moves. Please see Path Curves for a description of how path curves can be used to draw character strings.

Example

Assume that an application is to draw a specified character string at a specified position on top of each box of a series of boxes that is moving past on a conveyor belt. The sequnce of steps the application would need to perform are as follows:

1. The path curve array containing the character fonts is downloaded to the controller.
2. The character string and position of the string on the box are somehow specified to the application.
3. The application uses this information to create the Path-to-Work transformation.
4. The path curve corresponding to the first character in the string is located.
5. The application then waits for a box to arrive.
6. When a box is detected, then the Work-to-Machine transformation is calculated and passed to the path handling functions using the KinematicsWorkToMc command.
7. The PathStart command is executed to start drawing the first character. Note that this command takes the Path-to-Work transformation as a parameter.
8. The path curve for the next character in the string is located and the PathNext command is used to pass it to the path handling functions. This can be done immediately after the PathStart command is executed.
9. As the drawing progresses, the application continuously monitors the movement of the conveyor belt. As often as is necessary, the KinematicsWorkToMc command is used to update the transformation for the path handling functions.

10. When the first charcter has been completed, the path handling functions will automatically start drawing the next character (with the appropriate Path-to-Work offset). At this time, the application must locate the path curve for the next character and set it using the PathNext command.
11. Steps 9 and 10 are then repeated until all characters in the string have been drawn.
12. The application then goes back to step 5 and waits for the next box to arrive.

6.6.9 Paths and State Machines

It is strongly recomended to use the State Machines syntax for working with paths and kinematics because there are a few events that are generated automatically in firmware. All those events can be used easily in a state machine application programm, just by setting up the SmSystem with the parameter value PFGEVENT.

For details on `SmSystem PFGEVENT` please see Profile Generator Events.

For a working sample of a state machine that handles path, please see Path Curve Sample.

6.7 Internal Amplifier

This section shows the different application possibilities with the internal MACS amplifiers. It is important to note that not all MACS controllers offer the same possible configuration. In the section Controller Types the restrictions are shown. Documentation about wiring can be found in hardware manuals of the controllers.

The following different amplifier controls types are available for MACS products. One Controllers can also support a combination of different control types (Example: One MiniMACS6 can support 3 DC motors and 2 EC motors).

Amplifier Control Type	
DC Motor	Control of two-phase DC motors.
EC Motor	Commutation of three-phase EC motors (Brushless DC motor).
Stepper Motor	Commutation of two-phase bipolar stepper motors.
Twin Mode	To drive higher currents, DC, BLDC and stepper motors can be operated in twin mode.
Brake Chopper	A chopper is designed to limit the power supply voltage.
General Topics	
Amplifier Control Loop Design	If an internal amplifier is used, the entire controlled system consists of cascading position, speed and current control. The position control is described in section Position Control Loop Design.
Overcurrent Protection (OVC)	In order to protect the connected motors and the controller from a too high continuous current load, it is possible to define an I2T protection in the application.
Current Ripple and Motor Chokes	The internal amplifier and its load can produce disturbing current ripples. What can be done?

Example

```
 HWAMP_PARAM(axis, HWAMP_COMMTYPE) = HWAMP_COMMTYPE_BLDC;   // Set commutation
type always as the first parameter
 HWAMP_PARAM(axis, HWAMP_MODE)     = HWAMP_MODE_POS_CUR;    // Set controller
principle
 HWAMP_PARAM(axis, HWAMP_POLES)    = 8;               // Number of pole pairs
 HWAMP_PARAM(axis, HWAMP_MAXCUR)   = 4000;            // Max current in mA
 HWAMP_PARAM(axis, HWAMP_ENCRES)   = 1024;            // Given in qc
 HWAMP_PARAM(axis, HWAMP_MAXRPM)   = 2000;            // Given in RPM

// Setup of current controller
HWAMP_PARAM(AxisNo, HWAMP_CURKPROP) = …
```

6.7.1 EC Motor

Commutation types

EC (**E**lectronically **C**ommuted, or **E**lectronically **C**ontrolled) motors are electric motors which have permanent magnets on the rotor and use electronics to control the voltage and current applied to the motor.

MACS controllers offer two different types of commutation for EC Motors. These two types of commutation are described in the following sections.

Sinusoidal Commutation	Sinusoidal commutation (or so-called FOC "Field Oriented Control) applies a "sinusoidal" winding current based on the accurate information about the position of the motor's rotor. The rotor position is derived from the encoder information. Finally this type of commutation results in low torque ripple and lowest motor noise.
Block Commutation	Block commutation is the most simple way of commutation. Block commutation just switches the windings stepwise on / off based on the signal pattern of the three hall sensors.

Comparing the two types of commutation, the following statements can be made.

- Sinusoidal commutation demands for a **high-resolution sensor** like an analog- or digital incremental-encoder
- With sinusoidal commutation , an **alignment** must be executed (manually, with sensors or without sensors)
- **Max. continuous current** is about 15% higher with sinusoidal commutation.
- **Torque constant** is about 10% smaller with sinusoidal commutation.
- **Max. continuous torque** is about 5% higher with sinusoidal commutation.
- Hence, the **continuous operation range** is enlarged by about 5%.

EC Motor have to align between magnets and windings. Several techniques are offered.

6.7.1.1 Sinusoidal Commutation

An EC Motor with sinusoidal commutation requires alignement between magnets (rotor) and windings (stator). This process is sometimes also called calibration or teaching. Several technics are offered.

```
HWAMP_PARAM(Axis, HWAMP_COMMTYPE) = HWAMP_COMMTYPE_PMSM; // For Alinment with hall sensor ->
```

```
HWAMP_COMMTYPE_HALL_PMSM
```

Alignment	Features	ApossC Implementation

Open Loop Alignment
Alignment with forward and backwards movement
	• Encoder has to be connected
	• Parameter HWAMP_ELPOL is set automatically during alignment
	• Before using motor, alignment must be started explicitly in application
		`MotorAlignStart(AxisNo, 1);`
		`…`
		`AxisControl(AxisNo, ON);`
		`…`

Open Loop Alignment
Alignment with a movement in one direction
	• Encoder has to be connected
	• Parameter HWAMP_ELPOL is set automatically during alignment
	• Before using motor, alignment must be started explicitly in application
		`MotorAlignStart(AxisNo, 4);`
		`…`
		`AxisControl(AxisNo, ON);`
		`…`

Absolute Alignment Alignment with absolute encoder	• Comming soon, not yet implemented	

HALL Alignment
Alignment with HALL sensores
	• Easy to use in application, since motor can be moved at once in any direction
	• HALL must be wired and configured
	• Parameter HWAMP_ELPOL must be set by application
	• May be inaccurate if first move of motor has a very high acceleration
		`// Note: Different COMMTYPE`
		`HWAMP_PARAM(axis, HWAMP_COMMTYPE) = HWAMP_COMMTYPE_HALL_PMSM;`
		`HWAMP_PARAM(axis, HWAMP_ELPOL) = -1;`
		`…`
		`AxisControl(AxisNo, ON);`
		`…`

6.7.1.2 Block Commutation

Block commutation works always with HALL sensor. In some cases the HALL pattern has to be determinated first. The HALL sensors are often shifted counterclockwise by 120°. Alternatively, the pattern can be detected once during commissioning with the function HallDetectStart().

```
HWAMP_PARAM(Axis, HWAMP_COMMTYPE) = HWAMP_COMMTYPE_BLDC;            //
HWAMP_COMMTYPE should be set as first HWAMP_PARAM
HWAMP_PARAM(Axis, HWAMP_HALL_ALIGNMENT) = HALL_ALIGNMENT_300060180; // 120° CCW:
H1 at 300°, H2 at 060°, H3 at 180°
```

6.7.2 DC Motor

A direct current (DC) motor is a type of electric machine that converts electrical energy into mechanical energy. DC motors take electrical power through direct current, and convert this energy into mechanical rotation.

DC motors use magnetic fields that occur from the electrical currents generated, which powers the movement of a rotor fixed within the output shaft. The output torque and speed depends upon both the electrical input and the design of the motor.

```
HWAMP_PARAM(Axis, HWAMP_COMMTYPE) = HWAMP_COMMTYPE_DC; // HWAMP_COMMTYPE should
be set as first HWAMP_PARAM
```

6.7.3 Stepper Motor

A **stepper motor**, also known as **step motor** or **stepping motor**, is a brushless DC electric motor that divides a full rotation into a number of equal steps. A stepper motor has to be wired to the controller as **2-phase bipolar stepper motor**, meaning the stepper motor has a four wire connection to the controller. The wiring of the motor to the controller depends on the controller and is explained in the hardware reference manual for each controller.

Controllers support microstepping. **Microstepping** is a way of moving the stator flux of a stepper more smoothly than in full- or half-step drive modes. This results in less vibration, and makes noiseless stepping possible. It also makes smaller step angles and better positioning possible. There are 1024 microsteps implemented (given in ApossIDE with the macro HWAMP_ENCRES_MICROSTEP_RES).

MACS controllers offer two different types of commutation for stepper Motors. They are described in the following sections.

Open Loop	Control of a stepper motor without the use of an additional feedback signal.
Closed Loop	Control of a stepper motor with the use of an encoder as feedback signal and a position control.

In open and closed loop it is recommended to use a current controller.

6.7.3.1 Stepper Motor - Open Loop

The motor can be controlled based on the defined steps. In this way, a position can be reached or hold. No position feedback is required for this. It is mandatory that the motor has been parameterized correctly (torque and speed). However, steps can be lost. This mode is referred as **open loop**.

Example

```
// Definition of the type of operation
HWAMP_PARAM(AxisNo, HWAMP_COMMTYPE) = HWAMP_COMMTYPE_STEP;

// Set mode to current control. Alternatively no current control is
possible (HWAMP_MODE_PFG_PWM),
// but not recommended, since all controller support current control.
HWAMP_PARAM(AxisNo, HWAMP_MODE) = HWAMP_MODE_PFG_CUR_PWM;

// Motor has 100 [Steps/rev]. For 2-phase stepper: Poles = Steps per
revolution / 4
```

```
HWAMP_PARAM(AxisNo, HWAMP_POLES) = 25;

// Setup encoder for PFG and amplifier, note that both are a
multiplication of
// poles and microsteps. HWAMP_ENCRES_MICROSTEP_RES is a constant.
HWAMP_PARAM(AxisNo, HWAMP_ENCRES) = HWAMP_PARAM(AxisNo, HWAMP_POLES) *
HWAMP_ENCRES_MICROSTEP_RES;
AXE_PARAM(AxisNo, POSENCQC) = HWAMP_PARAM(AxisNo, HWAMP_POLES) *
HWAMP_ENCRES_MICROSTEP_RES;

// Setup current controller
HWAMP_PARAM(AxisNo, HWAMP_MAXCUR) = 4000;
HWAMP_PARAM(AxisNo, HWAMP_CURKPROP) = 2000;
HWAMP_PARAM(AxisNo, HWAMP_CURKINT) = 100;
HWAMP_PARAM(AxisNo, HWAMP_CURKILIM) = 5000;

// Suppress tracking error
VIRTCOUNTIN_PARAM(AxisNo, VIRTCNTIN_PISRC_COUNTER) =
AXE_PROCESS_SRCINDEX(AxisNo, REG_COMPOS);
```

6.7.3.2 Stepper Motor - Closed Loop

The controller supports **closed loop** as well, merging the benefits of stepper and servo motor technology. They run more smoothly and have a lower resonance than stepper motors, provide position feedback and control, feature short settling times, and exhibit no step loss at all.

It needs an encoder for the position feedback. A calibration sequence is necessary to align the rotor to the stator (see MotorAlignStart(AxisNo, 3)).

Example

```
// Definition of the type of operation
HWAMP_PARAM(AxisNo, HWAMP_COMMTYPE) = HWAMP_COMMTYPE_STEP;
HWAMP_PARAM(AxisNo, HWAMP_MODE) = HWAMP_MODE_POS_CUR;

// Motor has 100 [Steps/rev]. For 2-phase stepper: Poles = Steps per
revolution / 4
HWAMP_PARAM(AxisNo, HWAMP_POLES) = 25;

// Setup an encoder with 4096 [qc/rev]
HWAMP_PARAM(AxisNo, HWAMP_ENCRES) = 4096;
AXE_PARAM(AxisNo, POSENCQC) = 4096;

// Setup current controller
HWAMP_PARAM(AxisNo, HWAMP_MAXCUR) = 4000;
HWAMP_PARAM(AxisNo, HWAMP_CURKPROP) = 6100;
HWAMP_PARAM(AxisNo, HWAMP_CURKINT) = 210;
HWAMP_PARAM(AxisNo, HWAMP_CURKILIM) = 6200;
```

```
// Perform an rotor/stator alignment similar to EC motor (PMSM)
MotorAlignStart(AxisNo, 3);
while((MotorAlignStatus(AxisNo)) == 0)
{
 print("Aligning... ");
 Delay(500);
}
if(MotorAlignStatus(AxisNo) < 0)
 print("Aligning failed");
else
 print("Aligning succeeded");
```

6.7.4 Twin Mode

In Twin Mode two half bridges are parallel. This rises the current limitation up to 180%. Almost all commutation types offer a twin mode.

The wiring of the **parallel half bridges** has to be done carefully, since the current must be divided as equal as possible between the two half bridges. If the impedance of the motor is low (about 0.4 Ohm or less), the differences of wire impedance can become more significant and the current of parallel half bridges may not be distributed equally. Consequently over current errors (Overcurrent Protection (OVC)) may be raised, even if the aforementioned total current limitation of 180% was never reached, because one of the two parallel half bridge got too much current while the other got less.

6.7.5 Brake Chopper

A brake chopper is designed to limit the power supply voltage (amplifier supply voltage). Threshold voltages can be defined. If these thresholds are exceeded, the chopper will intervene and reduce the excess energy via an external resistor. A brake chopper is also known as a shunt regulator or just chopper.

In normal operation, the value of the supply voltage is appointed by the power supply. 4-quadrant amplifiers can feed back brake energy into the supply and therefore work like generators. Thus, a long braking process can cause the supply voltage to rise due to the fed back energy. Overvoltage errors may be risen consequently or even destruction of hardware elements may occur. The task of the chopper is to limit the voltage increase up to a permissible value and to transform the excess energy into heat.

How to design a chopper?

1. Is a chopper necessary? Are there situations when the power supply voltage can rise above limits of the controller or the power supply? Could the rise of the supply voltage prevented by softer braking? Could the surplus energy be saved in a capacity?
2. Determinate HWAMP_CHOP_UDCMIN and HWAMP_CHOP_UDCMAX . HWAMP_CHOP_UDCMIN should be set higher than power supply voltage and lower than HWAMP_CHOP_UDCMAX . Set HWAMP_CHOP_UDCMAX to a level that no hardware can be damaged.
3. Determinate the external resistor: R = Max chopper voltage / Max current (We assume 100% duty cycle when we reach HWAMP_CHOP_UDCMAX)
4. Determinate the power dissipation of the resistor. The surplus energy of a worst case scenario has to be found. The general rule is that the larger the mechanical load and the faster its deceleration, the more power that will need to be dissipated. On the other hand allow some resistors a power overload of factor 25 for a duration of 1s. Consult the resistor's data sheet for details. It is strongly recommended to conduct some tests to verify a correct operation. The destruction of the resistor may result in the destruction of other hardware elements.

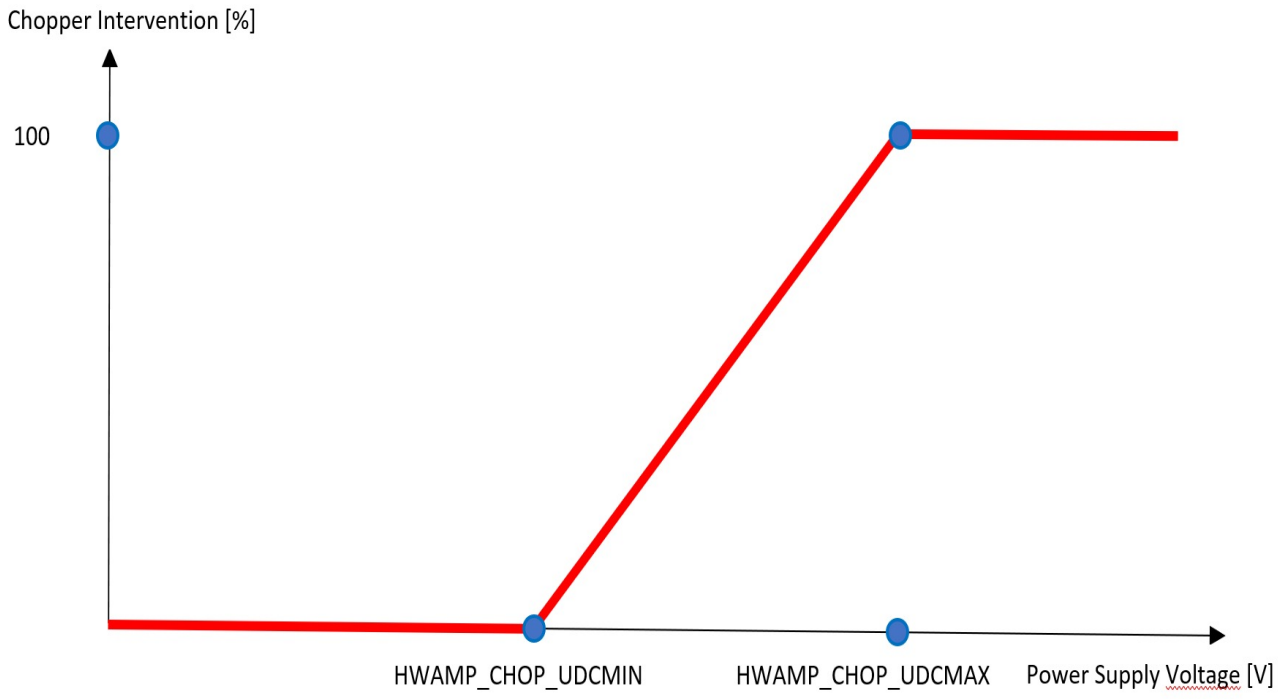
5. Define the mode. HWAMP_MODE_INT_PWM (without current control) and HWAMP_MODE_INT_CUR_PWM (with current control) are possible. Details more below.
6. It is strongly recommended to add an inductivity to the resistor. In case of active current control (HWAMP_MODE_INT_CUR_PWM) it it is mandatory, otherwise no current may be detected and the control loop makes no sense. In case of inactive current control (HWAMP_MODE_INT_CUR_PWM), it is recommended, because over current protection (OVC) needs information about the current.
7. Some choppers recommend a diode over an external resistor. This is already build-in for all controllers which support chopper.

How does a chopper work?

The diagram below shows the relationship of 'Power Supply Voltage' and 'Chopper Intervention' in percent. Up to HWAMP_CHOP_UDCMIN chopper does not intervene. If Power Supply Voltage rises higher, chopper starts intervening linearly. Depending on the mode, chopper intervention is used differently:

- **Open Loop**, without current control (HWAMP_MODE_INT_PWM): Chopper intervention is PWM Duty Cycle, both have the same unit. In other words: Power Supply Voltage controls Chopper Intervention. Chopper intervention is equal to PWM's Duty Cycle.
- **Close Loop**, with current control (HWAMP_MODE_INT_CUR_PWM): Current depends on Chopper Intervention. In other words: Power Supply Voltage controls Chopper Intervention. Chopper intervention and HWAMP_MAXCUR controls current. Calculate the current with the formula below.

$$\text{Current [mA]} = \frac{\text{Chopper Intervention [\%]* HWAMP\_MAXCUR [mA]}}{100}$$



Note:

- 100% duty cycle is not possible. Depending on the controller hardware, a duty cycle of about max 90% is

possible.
- Open and close loop refers to the current loop which is inactive or active. On the other hand overcurrent Protection (OVC) is always active.

Error handling

In case of any error, the chopper is disabled and stops working, since it is handled like any axis. The controller and its power supply are protected from the higher voltage, since the generator, which produces the surplus energy, is in most cases disconnected as well. Nevertheless: In some applications, this behavior may not be desired. In this case, an error handler could restart the chopper and makes the chopper continuing its job or parameter ERRCOND could define a different behavior. Otherwise, a simple error may lead to the destruction of hardware, even if a chopper is implemented.

Example

This example shows how a close loop brake chopper can be set up. The aforementioned error handling is omitted.

It is assumed that the power voltage is nominal 24V. The intervention starts at 28V and rises to 10A at 35V. At 40V, the power supply could be damaged. Therefore, an error would rise at 38V, which disconnects all motors and the chopper. To the controller connected and running motor would coast.

- R = 35/10 =3.5Ohm, selected: 3.3Ohm
- P = I^2/R = 10^2/3.3 = 30W, Note: This is the continuous case, but in many cases the load is applied just a short time.
- L =U/4/Ipp/fpwm = 35/4/2/48000 = 90uH, Note: Ipp is the current peak-peak ripple, fpwm is the PWM's frequency (In case of MiniMACS56 48kHz)

```
// Definition of the type of operation
HWAMP_PARAM(AxisNo, HWAMP_COMMTYPE) = HWAMP_COMMTYPE_CHOPPER;
HWAMP_PARAM(AxisNo, HWAMP_MODE) = HWAMP_MODE_INT_CUR_PWM;
HWAMP_PARAM(AxisNo, HWAMP_MAXCUR) = 10000;

// Chopper intervention starts at 28V and is 100% at 35V
HWAMP_PARAM(AxisNo, HWAMP_CHOP_UDCMIN) = 28000/4
HWAMP_PARAM(AxisNo, HWAMP_CHOP_UDCMAX) = 35000/4;

// The following line is included to raise error 78 in case power
// voltage exceeds its limits. If error 78 occurs, the chopper is not
// designed, wired or configurated correctly.
HWAMP_PARAM(AxisNo, HWAMP_VOLT_MAX) = 38000 / 4;

// Define the behavior in case of error. Coast (0) is the default, but other behavior could
be defined
// here and in an error handler. We define here that all motors coast, but the chopper will
continue its
// job in case of any error.
ERRCOND(AxisNo, 5);
ERRCOND(MotAxis1, 0);
ERRCOND(MotAxis2, 0);

// Only necessary in case of HWAMP_MODE_INT_CUR_PWM
HWAMP_PARAM(AxisNo, HWAMP_CURKPROP) = 1300;
HWAMP_PARAM(AxisNo, HWAMP_CURKINT) = 220;

// Configure all axis before switching chopper on
```

```
AxisControl(AxisNo, ON);
```

6.7.6 General

6.7.6.1 Overcurrent Protection (OVC)

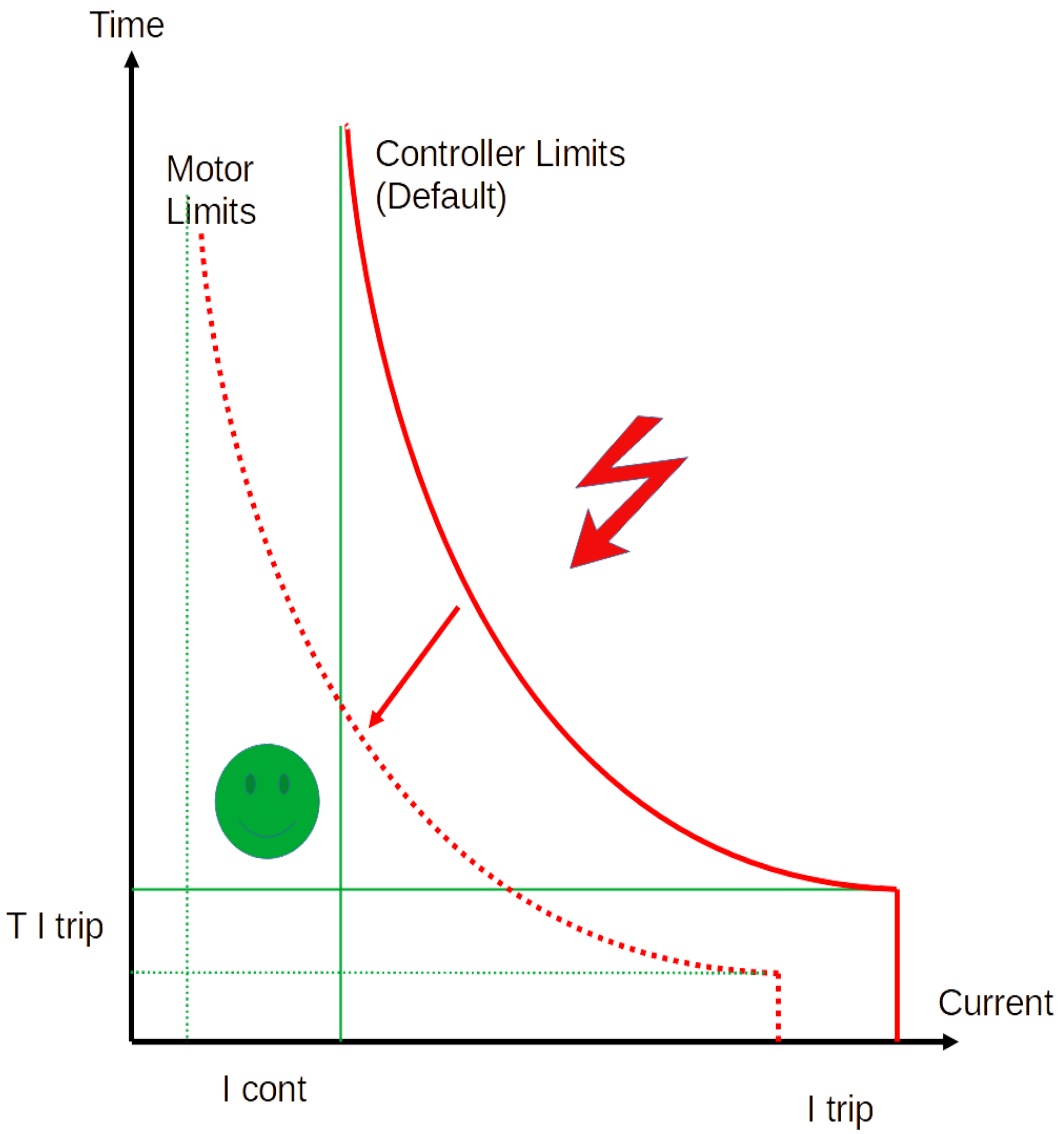
In order to protect the connected motors and the controller from a too high current load, it is possible to define an overcurrent protection in the application. The default protection depends on the controller hardware, but can be adapted to be more restrictive for smaller motors.

With most motor manufacturers you will find a diagram showing the typical operating range. A classification is made between continuous operation and short time operation. For thermal reasons, a motor may only be permanently loaded with the maximum continuous current. However, higher currents are permitted for short periods. As long as the winding temperature of the motor is below the critical value, it will not be damaged. The thermal time constant of the winding can be a few seconds for small motors and up to several minutes for larger motors.

As you can see in the section module description, MACS products provide a virtual layer which connects the hardware with the ApossIDE functionalities. If an I2T protection needs to be generated, this protection can be generated on the internal amplifier or in the virtual layer. If an external amplifier is used, it can also be protected in the virtual layer.

Internal Amplifier's OVC Parameter

The graph below shows the behavior of the OVC protection. The maximum **Controller Limits** depend on the Controller Type and can be shifted to the left and/or down by setting OVC parameters.



To implement a more sensitive OVC protection following values must be known.

- Axis number
- Maximum continuous current I cont
- Maximum short term current I trip
- Maximum time for short term current T I trip

- Controller's internal constants (see example)

Example

The following code section shows a minimum application which set OVC parameter to lower values than the default of **MiniMACS6** controller.

```
// Internal constants which depend on controller. Only for MiniMACS6
#define INIT_OVC_I2TDIVSHIFT 17
#define FREQ_Hz 24000 // Frequency of current control loop
#define I_RES 0.002 // Internal resolution in [A/lsb]

/* Set parameters for internal amplifier's over current protection (OVC). Use
this function to set limits
lower (!) than controller's limits to protect motor.
axis axis 0-indexed
I_cont_A  continues current in ampere
I_trip_A  trip current in ampere
T_Itrip_sec max time permitted for I_trip_A in seconds
Note 1: When any parameter's limit are exceeded, error 101 will be raised within
this function.
Note 2: This code has to be changed a bit for MACS5 controllers, since sub
indexes are different.
*/
void setOvcParamInternalAmplifier(long axis, double I_cont_A, double I_trip_A,
double T_Imax_sec)
{
HWAMP_PARAM(axis, HWAMP_I2T_CONT) = (long) pow(I_cont_A/I_RES,
2)/(1<<INIT_OVC_I2TDIVSHIFT);
HWAMP_PARAM(axis, HWAMP_I2T_TRIPVALUE) =
(long) (pow(I_trip_A/I_RES, 2) - pow(I_cont_A/I_RES,
2))*T_Imax_sec*FREQ_Hz/(1<<INIT_OVC_I2TDIVSHIFT);
HWAMP_PARAM(axis, HWAMP_TRIP_CURRENT) = (long) I_trip_A*1000;
}

long main(void)
{
double I_cont_A;
double I_trip_A;
double T_Itrip_sec;
long axis = 0;
// By setting HWAMP_COMMTYPE, OVC Parameter are set to default values
HWAMP_PARAM(axis, HWAMP_COMMTYPE) = HWAMP_COMMTYPE_PMSM;

// Setup OVC to new values lower than controller's limits
I_cont_A = 5;
I_trip_A = 12;
T_Itrip_sec = 1;
// Set parameter
setOvcParamInternalAmplifier(axis, I_cont_A, I_trip_A, T_Itrip_sec);

// Setup other stuff
```

```
HWAMP_PARAM(axis, HWAMP_MAXCUR) = 8000;
// ...
return 0;
}
```

HWAMP_MAXCUR, HWAMP_TRIP_CURRENT and HWAMP_I2T_TRIPVALUE

- HWAMP_MAXCUR limits the set current in mA of the current controller. Note: The current controller can be disabled (Depending on HWAMP_MODE) and the value of HWAMP_MAXCUR has no effect. Keep also in mind, that the measured current may exceed the value set with HWAMP_MAXCUR. This depends on the current controller's parameters (HWAMP_CURKPROP, HWAMP_CURKINT, HWAMP_CURKILIM) and the load.
- When HWAMP_TRIP_CURRENT is exceeded, Error 97 is thrown. It is **not** possible to disable HWAMP_TRIP_CURRENT, it is just possible to set a lower value than default. The unit is mA.
- When HWAMP_I2T_TRIPVALUE is exceeded, Error 99 is thrown. It is **not** possible to disable HWAMP_I2T_TRIPVALUE, it is just possible to set a lower value than default.
- In case of Twin Mode: Each leg of a parallel half bridge is supervised by the two limits HWAMP_TRIP_CURRENT and HWAMP_I2T_TRIPVALUE individually. As a consequence it is important to wire the parallel half bridges carefully, the current must be distributed equally. The current controller (if active) controls on the other hand a sum current.

Error

The controller throws errors if limits are exceeded: Error 99 or Error 97

I2T Protection - Virtual Amplifier Parameter

The virtual amplifier has an I2T protection. Its limits are independent from the internal amplifier's limits since the virtual amplifier may support any (external) amplifier. The differences to the OVC Protection are:

- It is possible to switch off virtual amplifier's I2T protection (By default it is switched off). On the other hand it is never possible to switch off internal amplifier's OVC protection.
- Internal amplifier's OVC protection has a trip current (HWAMP_TRIP_CURRENT) which is an absolute limit for the current. It is also possible to define an accepted time for trip current (T Imax). The virtual amplifier has no such limits.

Example

The following code section shows a minimum application of the I2T protection with the Virtual Amplifier Parameter.

```
#define AXIS        0        // Axis
#define I_CONT_A    8.000    // Continuous current in A
#define I2T_TIME_MS 30000    // I2T Time in ms

VIRTAMP_PARAM(AXIS, VIRTAMP_I2TLIMIT)= I_CONT_A*I_CONT_A*1000;
VIRTAMP_PARAM(AXIS, VIRTAMP_I2TTIME) = I2T_TIME_MS;
```

6.7.6.2 Amplifier Control Loop Design

MACS controllers offer the possibility to adapt the controller cascade to the desired requirement. The cascade can be defined with the parameter HWAMP_MODE. In the lower section two frequently used controller cascades are shown. The following controllers are part of this cascade.

Position Control The position control can also be used with an external amplifier and is explained in detail in the Position Control Loop Design section.

Velocity Control The velocity control loop serves two purposes. It enables the system to respond to changing velocity commands and it enables the system to withstand high frequency load disturbances.

However, the velocity control loop alone cannot guarantee that the system will maintain a particular position for long periods of time.

To tune a velocity controller, different velocity steps can be executed.

Current Control Current control is an essential part of motor control. The parameters of the current control depend on the conditions of the connected motor (R/ L).

With the help of the Current Regulator Calculator these can be set approximately. Afterwards the fine tuning can be done with current steps.

Controller Cascade: Position -> Velocity -> Current

```
HWAMP_PARAM(axis, HWAMP_MODE)  = HWAMP_MODE_POS_VEL_CUR;
```

[redacted]

Controller Cascade: Position -> Current

```
HWAMP_PARAM(axis, HWAMP_MODE)  = HWAMP_MODE_POS_CUR;
```

[redacted]

6.7.6.3 Current Ripple and Motor Chokes

All internal amplifiers use PWM (pulse width modulation) as motor power output which leads to so called current ripples in case of low inductance. Current ripples can lead to the following problems:

- Destruction of the controller because the controller is not able to measure the current and therefore its controller or overcurrent protection is useless or inaccurate
- Some electronic components like capacities (in the controller or in the power supply) may experience higher wear with higher current ripples
- A milder form of the problems above is the instability of the current control loop
- The current ripple can be one factor of heating up of the motor windings

The following links help to calculate an additional inductance, also called **motor chokes**, as counteract these problems:

- Some basics about current ripple: PWM, PWM scheme (2 level, 3 level), Current ripple, Motor heating – maxon Support (maxongroup.com)
- More details about the calculation of motor chokes: PWM power stage: Current ripple & Motor chokes

6.8 Internal Encoder

This section shows the different application possibilities with the internal MACS encoders. It is important to note that not all MACS controllers offer the same possible configuration. In the section Controller Types the restrictions are shown. Documentation about wiring can be found in hardware manuals of the controllers.

The scaling of the encoder values in UserUnits is described in the section How to Set Up Resolution and Gear Parameters.

The following different encoder types are available for MACS products. One controller can also support a combination of different encoder types (Example: One MiniMACS6 can support one incremental, two SSI and one

SinCos encoder).

Encoder Types	
Incremental	Incremental Encoder Input
Signal Generator	Incremental Encoder Output
SinCos	Sin/Cos Encoder
SSI	SSI Encoder active/ passiv clock

6.8.1 Incremental Encoder

Coming soon

6.8.2 Signal Generator

Coming soon

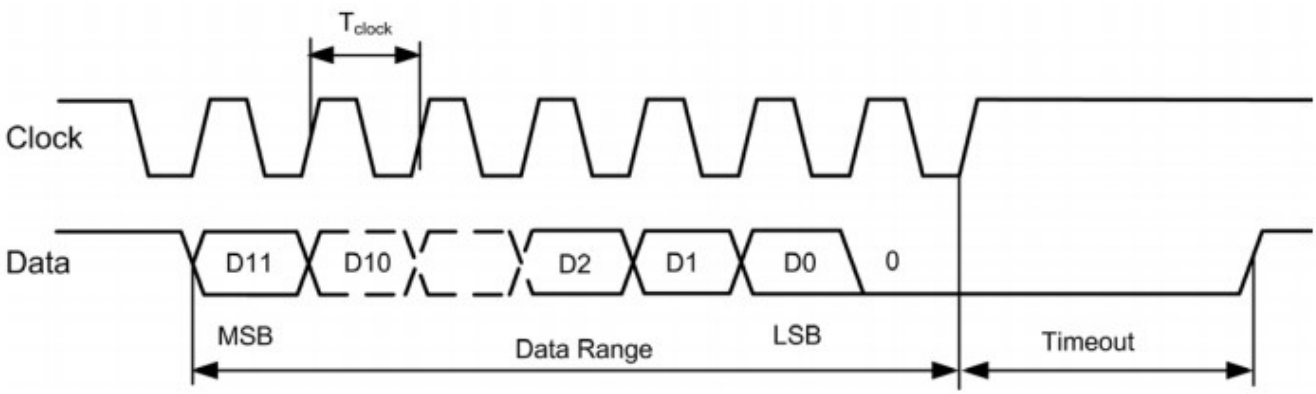
6.8.3 Sin Cos Encoder

Coming soon

6.8.4 SSI Encoder

SSI Concept

The graphic below shows the transmission mode of an SSI encoder. Per sample cycle the data is transmitted bit by bit.



Clock frequency settings

The clock must be configured appropriately for the system. This means that it must be fast enough so that the data can be processed in one readout cycle. In addition, it must be noted that high clock frequencies are more susceptible to interference than low frequencies. The maximum encoder input frequency of the controller must not be exceeded.

If the encoder is used for commutation of the motor, a higher sampling frequency

(HWENCODER_FAST_UPDATE) must be used. The sampling rate can be found in the respective HW manual.

Example of an encoder with 12 bit data length and an encoder timeout of 15 µs.

$$t_{Sample} > n_{bit} * \frac{1}{f_{clock}} + t_{EncoderTimeout}$$

$$\frac{1}{24'000\ Hz} > 12 * \frac{1}{1'000'000\ Hz} + 0.000015$$

$$41.66\ \mu s > 27\ \mu s$$

Examples

The following examples only set up the encoder. The user units and the amplifier must be parameterized additionally.

Example Singleturn SSI Encoder

Example: SSI Singleturn Encoder												
Start: 1 Bit	Singleturn: 12 Bit											
1	12	11	10	9	8	7	6	5	4	3	2	1
not used	Singleturn used											
not used	HWAMP_ENCRES/ POSENCQC = 2^12 = 4096											
HWENCODER_DATLEN = 13												

```
HWENC_PARAM(encPort, HWENCODER_MODE)        = HWENCODER_MODE_SSI_ACTIVE;    //
Setup SSI Encoder active clock
HWCOUNTABS_PARAM(encPort, HWCNTABS_CODING)  = HWCNTABS_CODING_GREY;           //
SSI is gray coded
HWENC_PARAM(encPort, HWENCODER_DATLEN)      = 13;                           //
Total number of data bits
HWENC_PARAM(encPort, HWENCODER_CLOCKFREQ)   = 2000000;                      //
Clock frequency for SSI encoder


// Configures the SSi encoder update rate - Sample frequency (e.g. 8 kHz or 24
kHz on MiniMACS6)
HWENC_PARAM(encPort, HWENCODER_FAST_UPDATE) = HWENCODER_FAST_UPDATE_ENABLE;

//Defines how the source value is handled
VIRTCOUNTIN_PARAM(axis, VIRTCNTIN_MODE)   =
VIRTCNTIN_MODE_ABSOLUTE_DIRECT_ENDLESS;

// Setup virtual counter module
VIRTCOUNTIN_PARAM(axis, VIRTCNTIN_PISRC_COUNTER) =
HWCOUNTABS_PROCESS_SRCINDEX(encPort,PO_HWCNTABS_VALUE);

// Value where counter should produce an overflow - Encoder resolution
VIRTCOUNTIN_PARAM(axis, VIRTCNTIN_OVFL_VALUE) = 4096;
```

Example Multiturn SSI Encoder

Example: SSI Multiturn Encoder			
Start: 8 Bit	Multiturn: 16 Bit	Singleturn: 17 Bit	
1 2 3 4 5 6 7 8	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1	17 16 15 14 13 12 11 10 9 8 7 6 5 4	3 2 1
Start	Multiturn used	Singleturn used	Trail
not used	17 = 2^17 = 131'072	HWAMP_ENCRES/ POSENCQC = 2^14 = 16'384	not used
not used	HWENCODER_POSBITS = 30 2^30 = 1'073'741'824 (internal Overflow)		TRAILBITS = 3
HWENCODER_DATLEN = 41			

```
HWENC_PARAM(encPort, HWENCODER_MODE)          = HWENCODER_MODE_SSI_ACTIVE;   //
Setup SSI Encoder active clock
HWCOUNTABS_PARAM(encPort, HWCNTABS_CODING)  = HWCNTABS_CODING_NONE;          //
SSI is Binary
HWENC_PARAM(encPort, HWENCODER_DATLEN)       = 41;                           //
Total number of data bits
HWENC_PARAM(encPort, HWENCODER_TRAILBITS)    = 3;                            //
Total number of trail bits
HWENC_PARAM(encPort, HWENCODER_POSBITS)      = 30;                           //
Total number of position bits
HWENC_PARAM(encPort, HWENCODER_CLOCKFREQ)    = 800000;                       //
Clock frequency for SSI encoder

// Configures the SSi encoder update rate - Sample frequency (e.g. 8 kHz or 24
kHz on MiniMACS6)
HWENC_PARAM(encPort, HWENCODER_FAST_UPDATE) = HWENCODER_FAST_UPDATE_DISABLE;

//Defines how the source value is handled
VIRTCOUNTIN_PARAM(axis, VIRTCNTIN_MODE)   =
VIRTCNTIN_MODE_ABSOLUTE_DIRECT_ENDLESS;

// Setup virtual counter module
VIRTCOUNTIN_PARAM(axis, VIRTCNTIN_PISRC_COUNTER) =
HWCOUNTABS_PROCESS_SRCINDEX(encPort,PO_HWCNTABS_VALUE);

// Value where counter should produce an overflow - Encoder resolution 2^30 →
POSBITS
VIRTCOUNTIN_PARAM(axis, VIRTCNTIN_OVFL_VALUE) = pow(2,30);
```

ExampleBISS-C Singleturn Encoder

Example: BISS-C Singleturn Encoder		
Start: 8 Bit	Singleturn: 12 Bit	Data: 16 Bit
1 2 3 4 5 6 7 8	12 11 10 9 8 7 6 5 4 3 2 1	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Start	Singleturn used	Trail
not used	HWAMP_ENCRES/ POSENCQC = 2^12 = 4096	not used
not used	HWENCODER_POSBITS = 12	HWENCODER_TRAILBITS = 16
HWENCODER_DATLEN = 36		

```
HWENC_PARAM(encPort, HWENCODER_MODE)          = HWENCODER_MODE_SSI_ACTIVE;    //
Setup SSI Encoder active clock
HWCOUNTABS_PARAM(encPort, HWCNTABS_CODING)  = HWCNTABS_CODING_GREY;           //
is gray coded
HWENC_PARAM(encPort, HWENCODER_DATLEN)       = 36;                            //
Total number of data bits
HWENC_PARAM(encPort, HWENCODER_TRAILBITS)    = 16;                            //
Total number of trail bits
HWENC_PARAM(encPort, HWENCODER_POSBITS)      = 12;                            //
Total number of position bits
HWENC_PARAM(encPort, HWENCODER_CLOCKFREQ)    = 1000000;                       //
Clock frequency for SSI encoder
// Configures the SSi encoder update rate - Sample frequency (e.g. 8 kHz or 24
kHz on MiniMACS6)
```

```
HWENC_PARAM(encPort, HWENCODER_FAST_UPDATE) = HWENCODER_FAST_UPDATE_DISABLE;

//Defines how the source value is handled
VIRTCOUNTIN_PARAM(axis, VIRTCNTIN_MODE)   =
VIRTCNTIN_MODE_ABSOLUTE_DIRECT_ENDLESS;

// Setup virtual counter module
VIRTCOUNTIN_PARAM(axis, VIRTCNTIN_PISRC_COUNTER) =
HWCOUNTABS_PROCESS_SRCINDEX(encPort,PO_HWCNTABS_VALUE);

// Value where counter should produce an overflow - Encoder resolution
VIRTCOUNTIN_PARAM(axis, VIRTCNTIN_OVFL_VALUE) = 4096;
```

6.9 Homing

«Homing» describes the procedure according to which the drive seeks the home position (also called reference point or zero point).
There are various Homing Methods to achieve this using limit switches at both ends of travel. Some of the methods use the index (zero) pulse train of an incremental encoder.

The controllers firmware is capable of exeuting the homing in the background. The user starts the procedure using the AxisHomeStart command.
Now the homing state machine runs in the firmware and the ApossC program can execute other commands during the completion. The state of the homing state machine can be polled with the AxisHomeStatus command.

In case the procedure shall be stopped, the command AxisHomeStop can be used.

There are two possible kinds of errors during the homing:

- Homing Procedure Faults: There was a problem with the homing sequence (e.g. no encoder index was found after two revolutions). The homing will be stopped and AxisHomeStatus will return a specific error number in that case.
- General Faults (see Error Reference): When a general problem occurrs (like e.g. a motor power loss), the homing procedure will be stopped. AxisHomeStatus will return HOMING_STATE_ERR_AXIS in that case.

> ⚠ While the homing is active, do not execute any other Axis commands (like e.g. AxisStop). This will result in failure of the homing.

There are numerous homing methods. See the specific article for the descriptions.

A number of axis parameters is used for the configuration of the homing:

Axis Parameter	Description
HOME_TYPE	Selection of the homing method
HOME_VEL	Velocity for "switch search"
HOME_ZEROVEL	Velocity for "zero search"
HOME_RAMP	Ramp setting during homing
HOME_CURTHRESHOLD	Motor current threshold to detect mechanical limit
HOME_OFFSET	Offset to move away from a detected position
HOME_FORCE	To block any move commands before homing is done
I_NEGLIMITSW	Configuration of negative limit switch

I_POSLIMITSW	Configuration of positive limit switch
I_REFSWITCH	Configuration of reference switch

The homing function can be controlled by the following commands:

ApossC command	Description
AxisHomeStart	Start the homing procedure
AxisHomeStatus	Returns the status of the homing procedure
AxisHomeStop	Stopps the homing procedure

⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

There is an example program in ApossIDE (See Menu File->Sample->Movement and Positioning->Homing.mc

6.9.1 Homing Methods

The homing methods listed below are supported by MACS Cotroller. In addition, with the help of the functions DefOrigin(), Apos(), Ipos(), DigInput(), ... an own homing procedure can be realized.

1	Negative Limit Switch & Index
2	Positive Limit Switch & Index
7	Home Switch Positive & Index
11	Home Switch Negative & Index
17	Negative Limit Switch
18	Positive Limit Switch
23	Home Switch Positive Speed
27	Home Switch Negative Speed
33	Index negative Speed
34	Index positive Speed
37	Home at actual position
-1	Current Threshold Positive Speed & Index
-2	Current Threshold Negative Speed & Index
-3	Current Threshold Positive Speed
-4	Current Threshold Negative Speed

6.9.1.1 Homing Method 1 (Negative Limit Switch & Index)

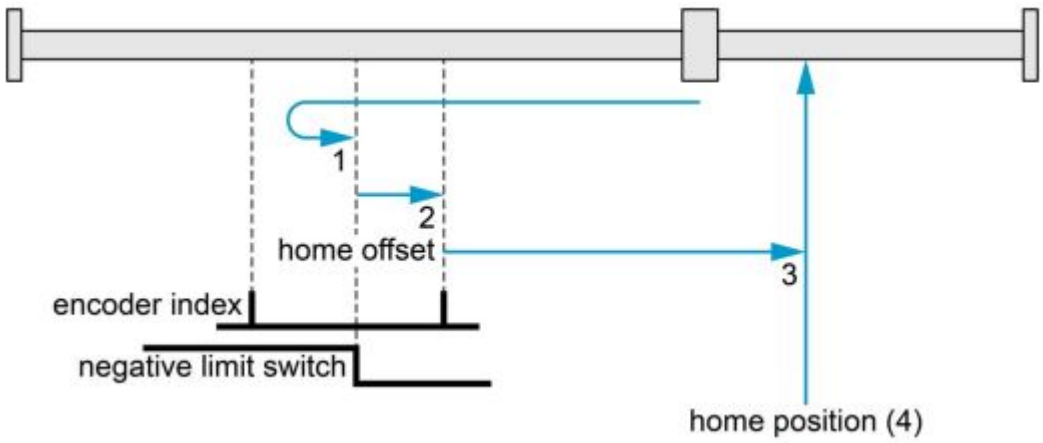
The initial direction of the movement is negative (here to the left) if the negative limit switch is inactive (here

shown as low).

a) The axis moves with HOME_VEL Speed to the positive edge of the I_NEGLIMITSW limit switch (1).

b) The axis moves with HOME_ZEROVEL Speed to the negative edge of the I_NEGLIMITSW limit switch and further to the first encoder index pulse after the negative edge of the I_NEGLIMITSW limit switch (2).

c) The axis moves the HOME_OFFSET move distance (3) in positive direction with HOME_VEL Speed. This point will be used as new home position (4).



Example

```
AXE_PARAM(0, HOME_TYPE)                = 1;            // Select "Negative
Limit Switch & Index"
AXE_PARAM(0, HOME_VEL)                 = 30;           // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;           // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;         // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;           // Set homing ramp to
50%
AXE_PARAM(0, I_POSLIMITSW)             = 2;            // Select digital input
2 as positive limit switch
AXE_PARAM(0, I_NEGLIMITSW)             = 3;            // Select digital input
3 as negative switch
```

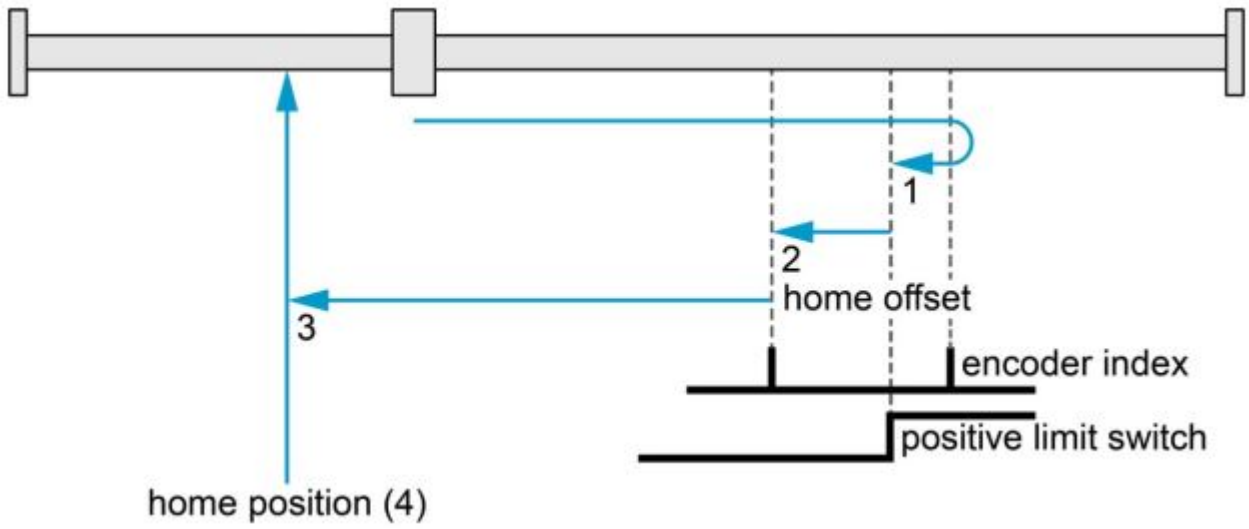
6.9.1.2 Homing Method 2 (Positive Limit Switch & Index)

The initial direction of the movement is positive (here to the right) if the positive I_POSLIMITSW limit switch is inactive (here shown as low).

a) The axis moves with HOME_VEL Speed to the positive edge of the I_POSLIMITSW limit switch (1).

b) The axis moves with HOME_ZEROVEL Speed to the negative edge of the I_POSLIMITSW limit switch and further to the first encoder index pulse after the negative edge of the I_POSLIMITSW limit switch (2).

c) The axis moves the HOME_OFFSET move distance (3) in negative direction with HOME_VEL Speed. This point will be used as new home position (4).



Example

```
AXE_PARAM(0, HOME_TYPE)                = 2;            // Select "Positive
Limit Switch & Index"
AXE_PARAM(0, HOME_VEL)                 = 30;           // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;           // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000 ;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;           // Set homing ramp to
50%
AXE_PARAM(0, I_POSLIMITSW)             = 2;            // Select digital input
2 as positive limit switch
AXE_PARAM(0, I_NEGLIMITSW)             = 3;            // Select digital input
3 as negative switch
```

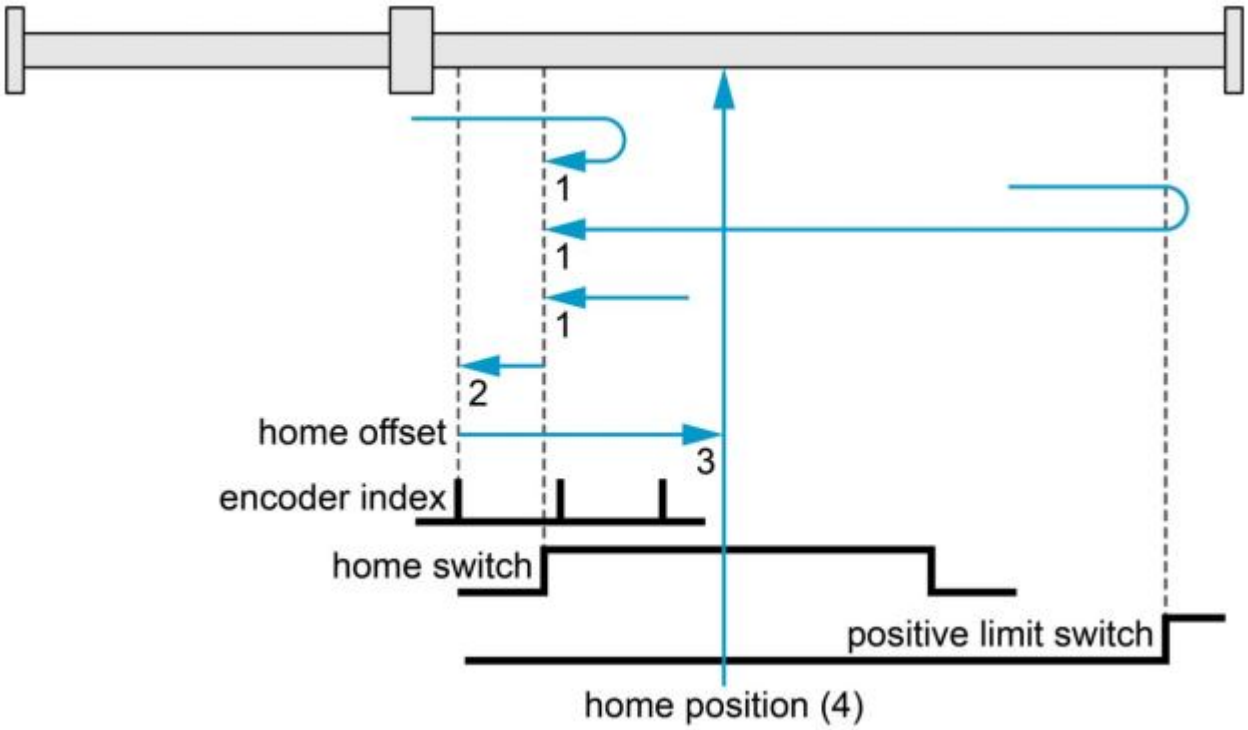
6.9.1.3 Homing Method 7 (Home Switch Positive Speed & Index)

The method uses a I_REFSWITCH home switch, which is active only during part of the movement. In effect, the switch acts as the position of the axis sweeps past the switch. The initial direction of the movement is positive (here to the right) except when the I_REFSWITCH home switch is already active upon start of the movement.

a) The axis moves with HOME_VEL Speed to the positive edge of the I_REFSWITCH home switch (1).

b) The axis moves with HOME_ZEROVEL Speed to the encoder index pulse (2).

c) The axis moves the HOME_OFFSET move distance (3) with HOME_VEL Speed. This point will be used as new home position (4).



⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

```
AXE_PARAM(0, HOME_TYPE)                = 7;           // Select "Home Switch
Positive & Index"
AXE_PARAM(0, HOME_VEL)                 = 30;          // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;          // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;          // Set homing ramp to
50%
AXE_PARAM(0, I_REFSWITCH)              = 1;           // Select digital input
1 as home switch
AXE_PARAM(0, I_POSLIMITSW)             = 2;           // Select digital input
2 as positive limit switch
AXE_PARAM(0, I_NEGLIMITSW)             = 3;           // Select digital input
3 as negative switch
```

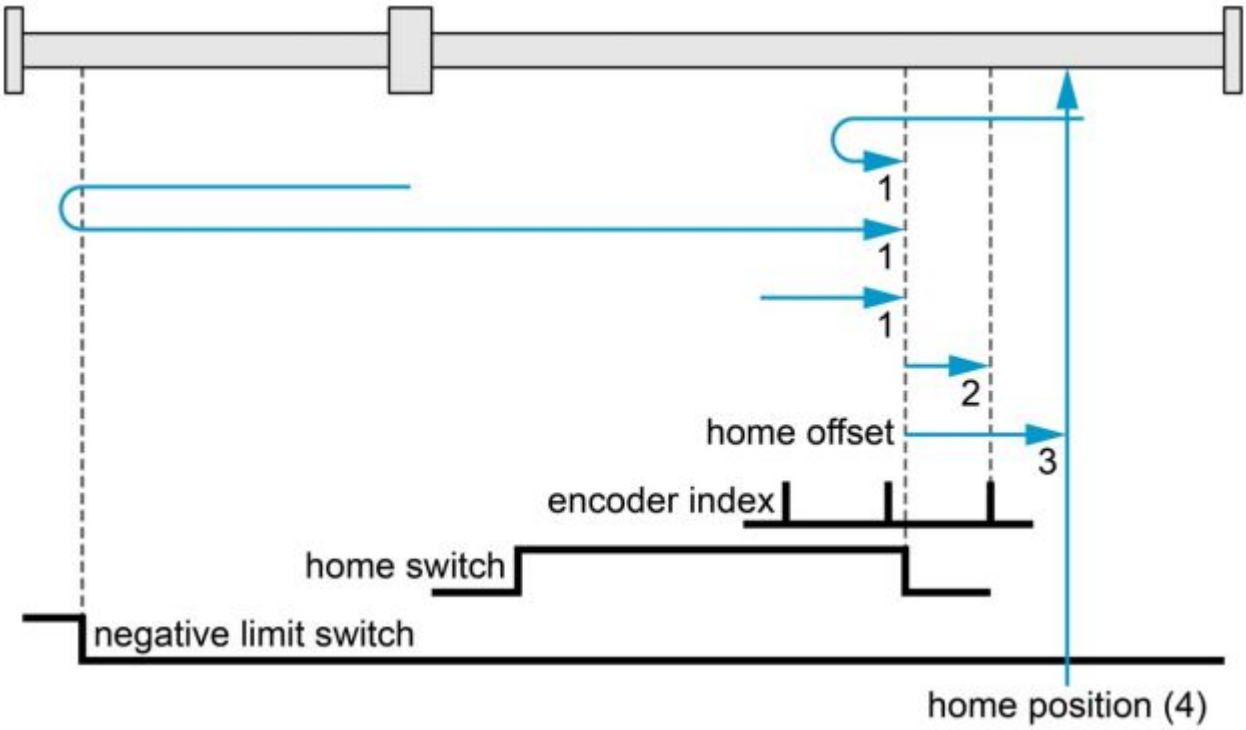
6.9.1.4 Homing Method 11 (Home Switch Negative Speed & Index)

The method uses a I_REFSWITCH home switch, which is active only during part of the movement. In effect, the switch acts as the position of the axis sweeps past the switch. The initial direction of the movement is negative (here to the left) except when the I_REFSWITCH home switch is already active upon start of the movement.

a) The axis moves with HOME_VEL Speed to the positive edge of the I_REFSWITCH home switch (1).

b) The axis moves with HOME_ZEROVEL Speed to the encoder index pulse (2).

c) The axis moves the HOME_OFFSET move distance (3) with HOME_VEL Speed. This point will be used as new home position (4).



⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

```
AXE_PARAM(0, HOME_TYPE)                = 11;          // Select "Home Switch
Negative & Index"
AXE_PARAM(0, HOME_VEL)                 = 30;          // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;          // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;          // Set homing ramp to
50%
AXE_PARAM(0, I_REFSWITCH)              = 1;           // Select digital input
1 as home switch
AXE_PARAM(0, I_POSLIMITSW)             = 2;           // Select digital input
2 as positive limit switch
AXE_PARAM(0, I_NEGLIMITSW)             = 3;           // Select digital input
3 as negative switch
```

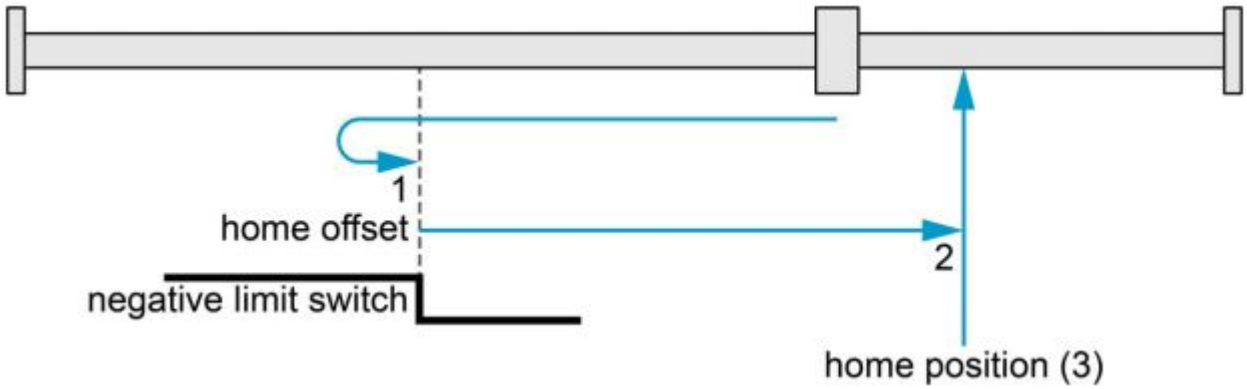
6.9.1.5 Homing Method 17 (Negative Limit Switch)

The initial direction of the movement is negative (here to the left) if the negative I_NEGLIMITSW limit switch is inactive (here shown as low).

a) The axis moves with HOME_VEL Speed to the positive edge of the I_NEGLIMITSW limit switch (1).

b) The axis moves with HOME_ZEROVEL Speed to the negative edge of the I_NEGLIMITSW limit switch.

c) The axis moves the HOME_OFFSET move distance (2) in positive direction with HOME_VEL Speed. This point will be used as new home position (3).



Example

```
AXE_PARAM(0, HOME_TYPE)                = 17;          // Select "Negative
Limit Switch"
AXE_PARAM(0, HOME_VEL)                 = 30;          // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;          // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;          // Set homing ramp to
50%
AXE_PARAM(0, I_POSLIMITSW)             = 2;           // Select digital input
2 as positive limit switch
AXE_PARAM(0, I_NEGLIMITSW)             = 3;           // Select digital input
3 as negative switch
```

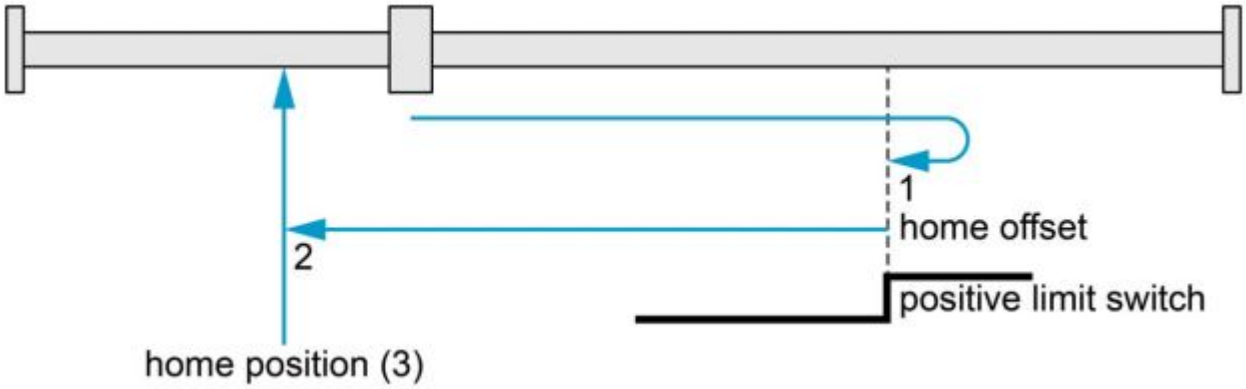
6.9.1.6 Homing Method 18 (Positive Limit Switch)

The initial direction of the movement is negative (here to the left) if the negative I_POSLIMITSW limit switch is inactive (here shown as low).

a) The axis moves with HOME_VEL Speed to the positive edge of the I_POSLIMITSW limit switch (1).

b) The axis moves with HOME_ZEROVEL Speed to the negative edge of the I_POSLIMITSW limit switch.

c) The axis moves the HOME_OFFSET move distance (3) in positive direction with HOME_VEL Speed. This point will be used as new home position (3).



Example

```
AXE_PARAM(0, HOME_TYPE)                 = 18;           // Select "Positive
Limit Switch"
AXE_PARAM(0, HOME_VEL)                  = 30;           // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)               = 10;           // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)               = 1000;         // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                 = 50;           // Set homing ramp to
50%
AXE_PARAM(0, I_POSLIMITSW)              = 2;            // Select digital input
2 as positive limit switch
AXE_PARAM(0, I_NEGLIMITSW)              = 3;            // Select digital input
3 as negative switch
```

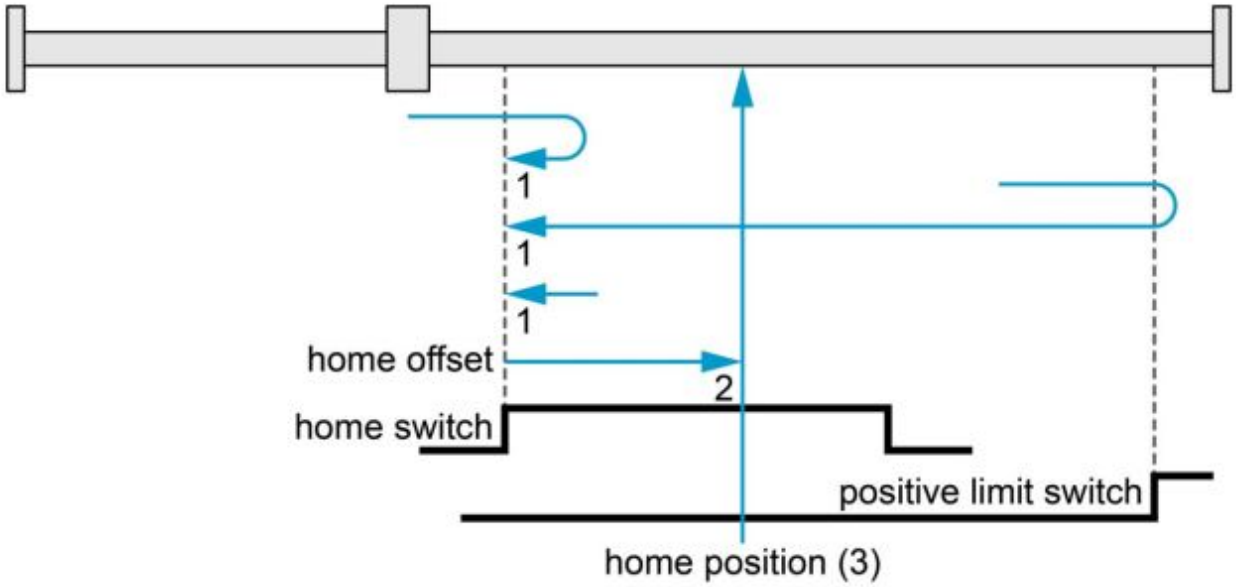
6.9.1.7 Homing Method 23 (Home Switch Positive Speed)

The method uses a I_REFSWITCH home switch, which is active only during part of the movement. In effect, the switch acts as the position of the axis sweeps past the switch. The initial direction of the movement is positive (here to the right) except when the home switch is already active upon start of the movement.

a) The axis moves with HOME_VEL Speed to the positive edge of the I_REFSWITCH home switch (1).

b) The axis moves with HOME_ZEROVEL Speed in negative direction to the negative edge of the I_REFSWITCH home switch

c) The axis moves the HOME_OFFSET move distance (2) with HOME_VEL Speed. This point will be used as new home position (3).



⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

```
AXE_PARAM(0, HOME_TYPE)                 = 23;           // Select "Home Switch
Positive Speed"
AXE_PARAM(0, HOME_VEL)                  = 30;           // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)               = 10;           // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)               = 1000;         // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                 = 50;           // Set homing ramp to
50%
AXE_PARAM(0, I_REFSWITCH)               = 1;            // Select digital input
1 as home switch
AXE_PARAM(0, I_POSLIMITSW)              = 2;            // Select digital input
2 as positive limit switch
AXE_PARAM(0, I_NEGLIMITSW)              = 3;            // Select digital input
3 as negative switch
```

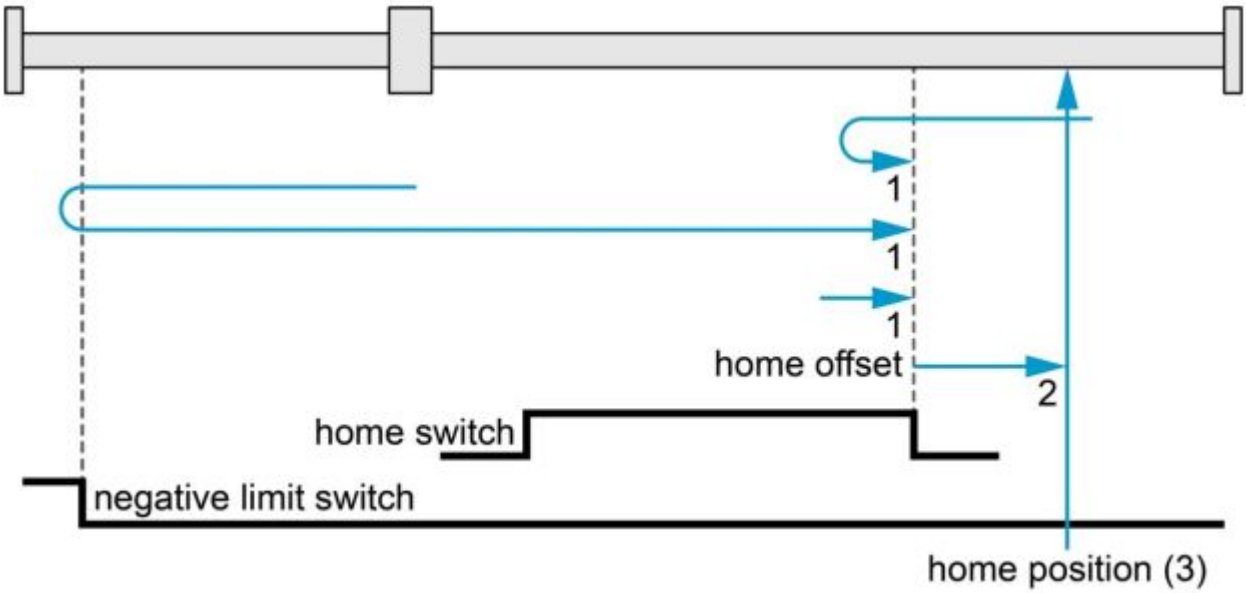
6.9.1.8 Homing Method 27 (Home Switch Negative Speed)

The method uses a I_REFSWITCH home switch, which is active only during part of the movement. In effect, the switch acts as the position of the axis sweeps past the switch. The initial direction of the movement is negative (here to the left) except when the home switch is already active upon start of the movement.

a) The axis moves with HOME_VEL Speed to the positive edge of the I_REFSWITCH home switch (1).

b) The axis moves with HOME_ZEROVEL Speed in positive direction to the negative edge of the home switch

c) The axis moves the HOME_OFFSET move distance (2) with HOME_VEL Speed. This point will be used as new home position (3).



⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

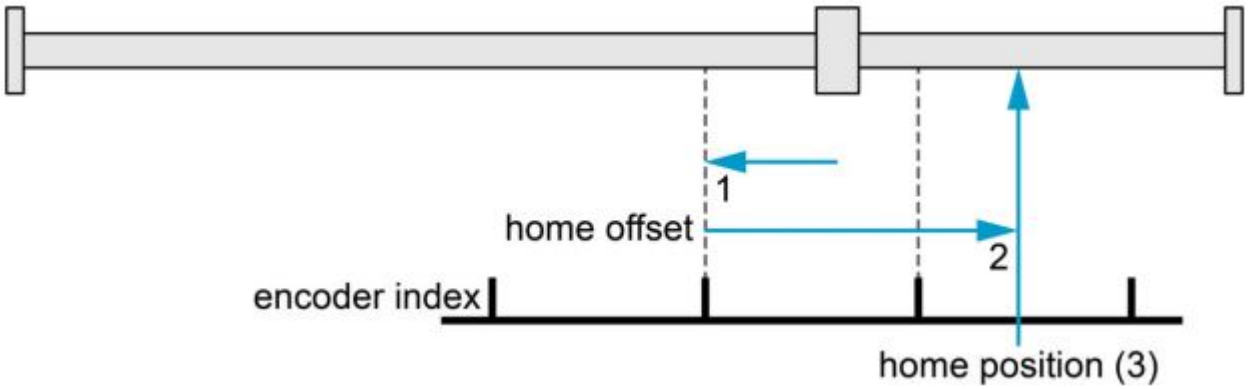
```
AXE_PARAM(0, HOME_TYPE)                = 27;          // Select "Home Switch
Negative Speed"
AXE_PARAM(0, HOME_VEL)                 = 30;          // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;          // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;          // Set homing ramp to
50%
AXE_PARAM(0, I_REFSWITCH)              = 1;           // Select digital input
1 as home switch
AXE_PARAM(0, I_POSLIMITSW)             = 2;           // Select digital input
2 as positive limit switch
AXE_PARAM(0, I_NEGLIMITSW)             = 3;           // Select digital input
3 as negative switch
```

6.9.1.9 Homing Method 33 (Index Negative Speed)

The direction for homing is negative (here to the left).

a) The axis moves with HOME_ZEROVEL Speed to the next encoder index pulse (1).

b) The axis moves the HOME_OFFSET move distance (2) with HOME_VEL Speed. This point will be used as new home position (3).



⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

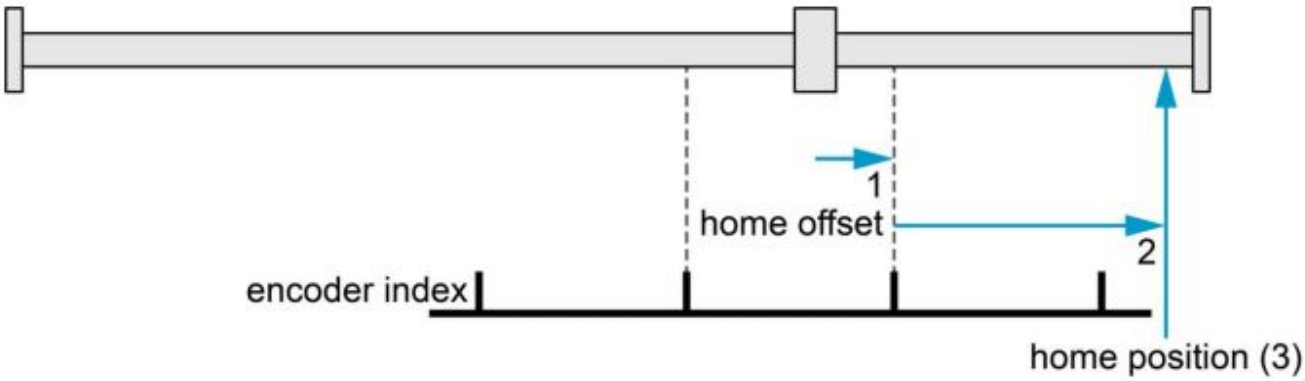
```
AXE_PARAM(0, HOME_TYPE)                = 33;          // Select "Index
negative Speed"
AXE_PARAM(0, HOME_VEL)                 = 30;          // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;          // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;          // Set homing ramp to
50%
```

6.9.1.10 Homing Method 34 (Index Positive Speed)

The direction for homing is positive (here to the right).

a) The axis moves with HOME_ZEROVEL Speed to the next encoder index pulse (1).

b) The axis moves the HOME_OFFSET move distance (2) with HOME_VEL Speed. This point will be used as new home position (3).



⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

```
AXE_PARAM(0, HOME_TYPE)                 = 34;           // Select "Index
positive Speed"
AXE_PARAM(0, HOME_VEL)                  = 30;           // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)               = 10;           // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)               = 1000;         // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                 = 50;           // Set homing ramp to
50%
```

6.9.1.11 Homing Method 37 (Actual Position)

The actual position is considered as the future home position (1). An additional positions HOME_OFFSET can be specified.

Example

```
AXE_PARAM(0, HOME_TYPE)                 = 37;           // Select "Home at
actual position"
AXE_PARAM(0, HOME_OFFSET)               = 1000;         // Set homing offset to
1000 qc
```

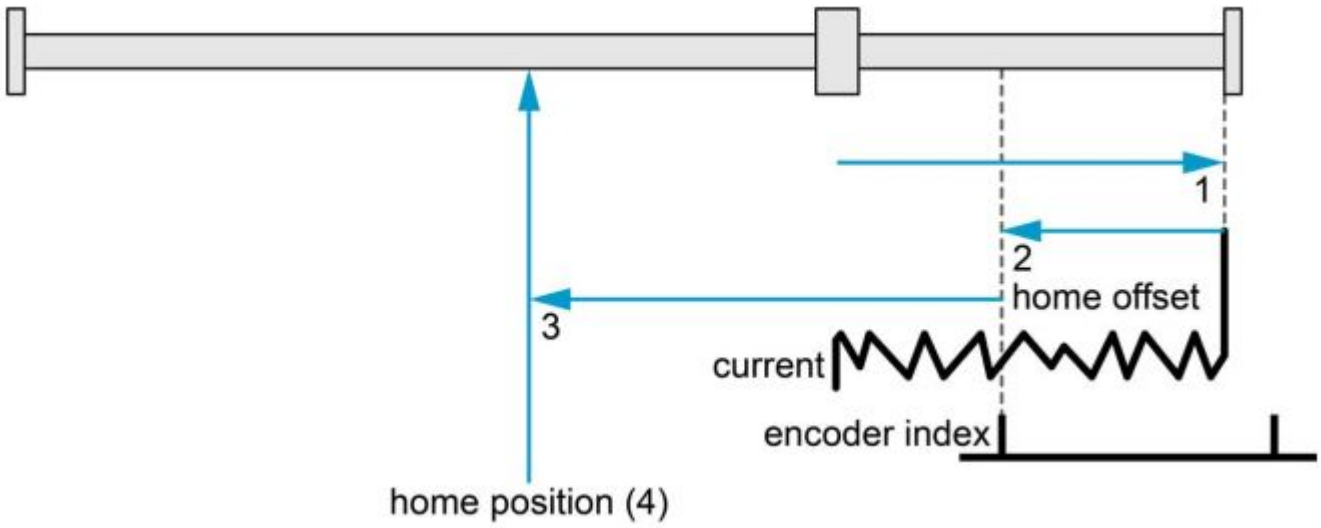
6.9.1.12 Homing Method -1 (Current Threshold Positive Speed & Index)

The method uses a mechanical end stop on the right (positive) side. The edge is detected when the averaged output current rises above the HOME_CURTHRESHOLD parameter value.

a) The axis moves with HOME_VEL Speed to the mechanical end stop (1).

b) The axis moves with HOME_ZEROVEL Speed to the next encoder index pulse (2).

c) The axis moves the HOME_OFFSET move distance (3) to the negative side with HOME_VEL Speed. This point will be used as new home position (4).



⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

```
AXE_PARAM(0, HOME_TYPE)                = -1;          // Select "Current
Threshold Positive Speed & Index"
AXE_PARAM(0, HOME_VEL)                 = 30;          // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;          // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;          // Set homing ramp to 50
AXE_PARAM(0, HOME_CURTHRESHOLD)        = 1000;        // Set current threshold
to +/- 1000 mA
```

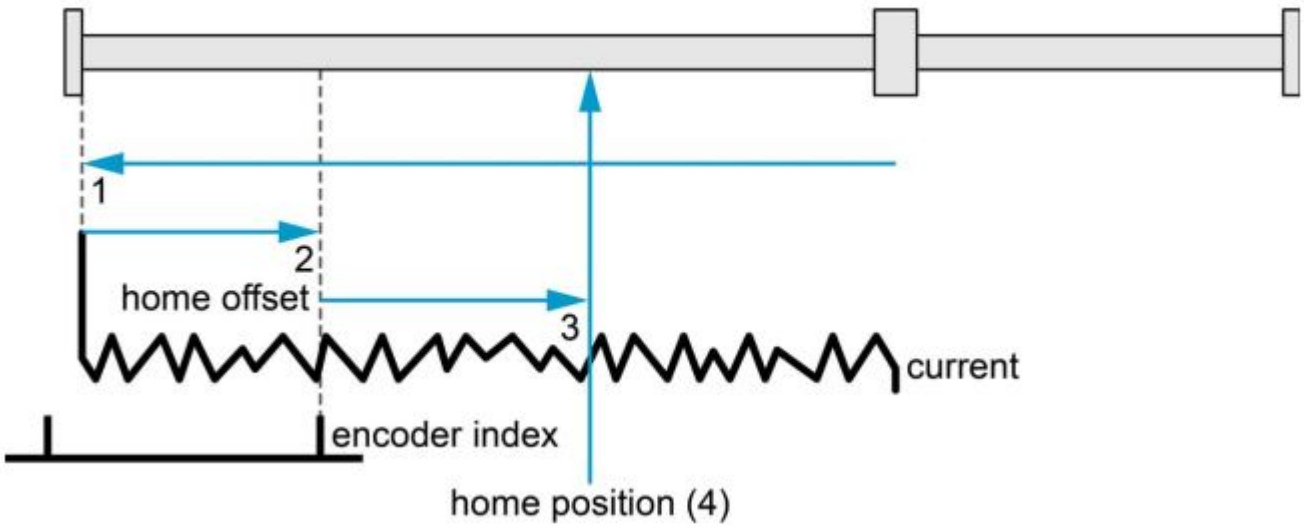
6.9.1.13 Homing Method -2 (Current Threshold Negative Speed & Index)

The method uses a mechanical end stop on the left (negative) side. The edge is detected when the averaged output current rises above the HOME_CURTHRESHOLD parameter value.

a) The axis moves with HOME_VEL Speed to the mechanical end stop (1).

b) The axis moves with HOME_ZEROVEL Speed to the next encoder index pulse (2).

c) The axis moves the HOME_OFFSET move distance (3) to the positive side with HOME_VEL Speed. This point will be used as new home position (4).



⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

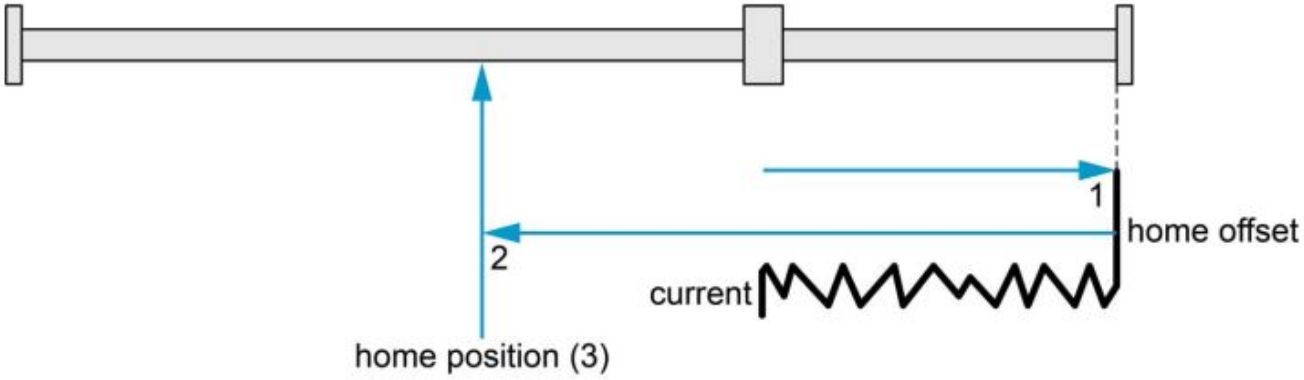
```
AXE_PARAM(0, HOME_TYPE)                = -2;          // Select "Current
Threshold Negative Speed & Index"
AXE_PARAM(0, HOME_VEL)                 = 30;          // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;          // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;          // Set homing ramp to 50
AXE_PARAM(0, HOME_CURTHRESHOLD)        = 1000;        // Set current threshold
to +/- 1000 mA
```

6.9.1.14 Homing Method -3 (Current Threshold Positive Speed)

The method uses a mechanical end stop on the right (positive) side. The edge is detected when the averaged output current rises above the HOME_CURTHRESHOLD parameter value.

a) The axis moves with HOME_VEL Speed to the mechanical end stop (1).

b) The axis moves the HOME_OFFSET move distance (3) to the negative side with HOME_VEL Speed. This point will be used as new home position (4).



⚠ Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

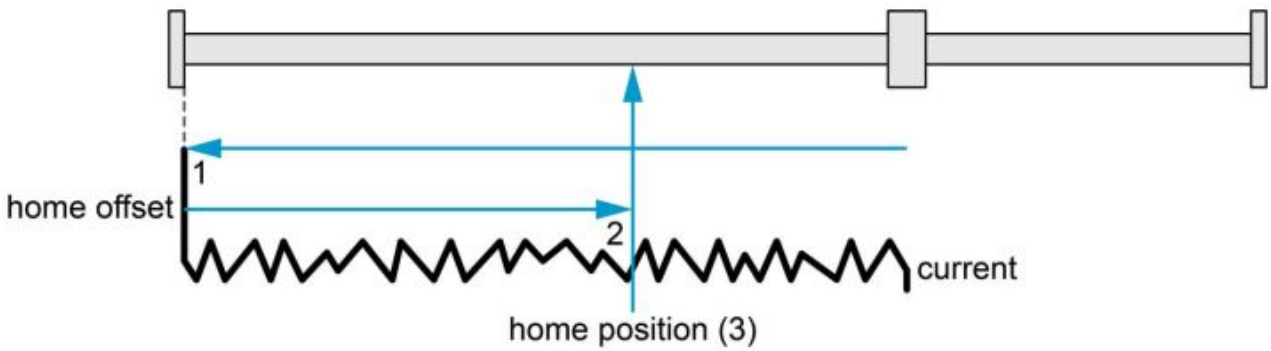
```
AXE_PARAM(0, HOME_TYPE)                = -3;          // Select "Current
Threshold Positive Speed"
AXE_PARAM(0, HOME_VEL)                 = 30;          // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;          // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;          // Set homing ramp to 50
AXE_PARAM(0, HOME_CURTHRESHOLD)        = 1000;        // Set current threshold
to +/- 1000 mA
```

6.9.1.15 Homing Method -4 (Current Threshold Negative Speed)

The method uses a mechanical end stop on the left (negative) side. The edge is detected when the averaged output current rises above the HOME_CURTHRESHOLD parameter value.

a) The axis moves with HOME_VEL Speed to the mechanical end stop (1).

b) The axis moves the HOME_OFFSET move distance (3) to the positive side with HOME_VEL Speed. This point will be used as new home position (4).



Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Example

```
AXE_PARAM(0, HOME_TYPE)                = -4;          // Select "Current
Threshold Negative Speed"
AXE_PARAM(0, HOME_VEL)                 = 30;          // Select 30% of VELMAX
as Speed for Switch search
AXE_PARAM(0, HOMEZEROVEL)              = 10;          // Select 10% of VELMAX
as Speed for Index serach
AXE_PARAM(0, HOME_OFFSET)              = 1000;        // Set homing offset to
1000 qc
AXE_PARAM(0, HOME_RAMP)                = 50;          // Set homing ramp to 50
AXE_PARAM(0, HOME_CURTHRESHOLD)        = 1000;        // Set current threshold
to +/-1000 mA
```

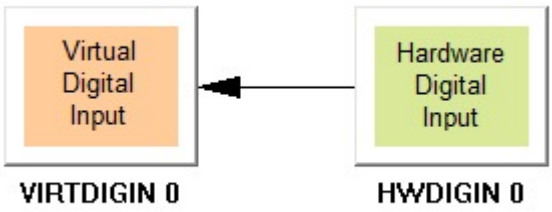
7 Controller Reference

7.1 Modules Description

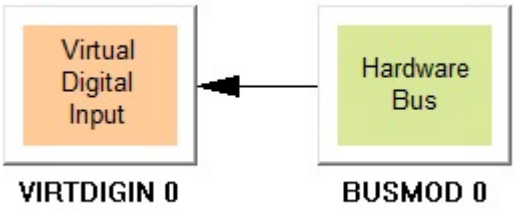
Some functionality dependent on the motion controller. This chapter describes the most important ones.

Function Modules

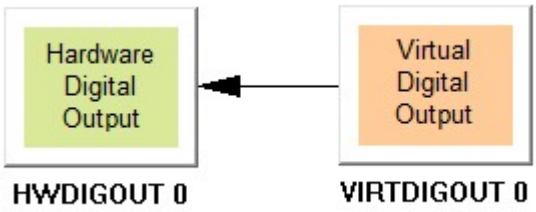
MACS controllers offer the possibilities to have function modules for specific functions inside the MACS firmware (MOCON). The modules can be linked together in a flexible way that allows to configure the system that fits best to the user's demands. For some modules the source of data input can be changed and specified with a parameter. Furthermore, there is a link layer for accessing the MACS functionality out of the application program. These are the virtual modules. This means that the application commands access the virtual modules instead of hardware modules directly. For example, when you read a digital input in the program with the command "DigInput(1)", the first input bit in the VIRTDIGIN 0 (virtual digital input module 0) is read. By default, VIRTDIGIN 0 is linked to the module HWDIGIN 0 (hardware digital input module 0) and therefore reading the first digital input on the MACS controller.



The user can re-link the VIRTDIGIN 0 to another source. For example, to a byte that is coming over the CAN bus using the BUSMOD 0 (HW bus module 0). With the same ApossC command "DigInput(1)" the input from a CAN IO module is read.

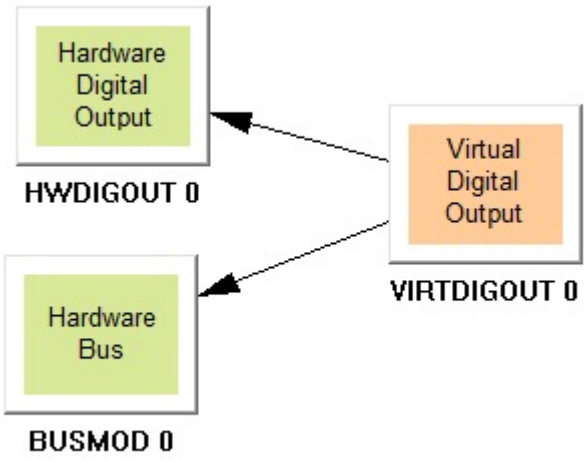


The same is possible the other way around. The ApossC command DigOutput(1, 1); sets the first digital output in the VIRTDIGOUT 0 (virtual digital output module). By default, the VIRTDIGOUT 0 module is linked to the HWDIGOUT 0 (hardware digital output module) and the first digital output on the MACS controller is set.



The VIRTDIGOUT 0 could forward this information to another module than HWDIGOUT 0. Even this information

can be split to two different modules.



The most common modules are:

Module Name	Short Description
Global parameters	Module for global settings (global parameters and system process data)
User parameters	Module for user parameters (no links to this module available, only links from this module)
Axis modules	Module for axis parameter and axis process data. Every axis in the system has its own module.
Virtual Master Modules	Virtual master module allows to setup a virtual master counter
Virtual Counter Input Modules	Virtual Counter Input module. This module is connected to the ApossC command "Apos()" (for reading the actual position)
Virtual Latch Modules	Virtual Latch Module has all the function used for position latching
Virtual Amplifier Modules	Virtual amplifier modules are used for controlling an internal or external axis (amplifier)
Virtual Digital Input Modules	Virtual Digital Input Modules for reading digital inputs
Virtual Digital Output Modules	Virtual Digital Output Modules for setting digital outputs
Virtual Analog Input Modules	Virtual Analog Input Modules for reading analog inputs
Virtual Analog Output Modules	Virtual Analog Output Modules for setting analog outputs
HW Bus Modules	Modules for bus communication, using PDO over CAN and EtherCAT bus
HW Digital Inputs	Modules for hardware digital inputs that are available on the MACS controller
HW Digital Outputs	Modules for hardware digital outputs that are available on the MACS controller

HW Encoder Modules	HW encoder modules for encoder settings
HW Incremental Counter Modules	HW incremental counter for incremental encoders

More modules are available!

Counting the modules

For many functions there are more than one module of the same kind. The number of modules depends on the controller type and on the number of axes that are supported.

The counting of the modules is zero based. So, the first module is module 0 and increments by one. For example, if there are 4 hardware encoder modules we have:

HWENC(0)
HWENC(1)
HWENC(2)
HWENC(3)

Parameters and Process data

Most modules have parameters and process data. Parameter allow to do some settings or configuration to the module. Process data are the output of a module.

Module parameters and process data are accessible through SDO objects with index and subindex

For example:

Sysvar[0x012C4001] ➔ virtual counter input value of the first module.

For having nice names for the modules are defines in the include file sysdef.mi or sysdef.mh. The same value can be read with the syntax:

VIRTCOUNTIN_PROCESS(0,PO_VIRTCNTIN_VALUE)

Virtual modules layer

The idea behind the virtual modules is that the application program is less hardware dependent. Instead of accessing the hardware module directly a virtual module is accessed, and the virtual module is linked to the hardware module. This structure has the advantage that controlling axis, inputs, outputs and counters in the same way from your application. It doesn't matter if the motor is connected to an internal amplifier or a CANopen drive over CAN bus, the inputs and outputs are on the controller or connected over a bus.

Another scenario could be that the virtual counter input module which is related to the actual position information "Apos()" could be linked to an analog input module. That allows to have the position feedback from a potentiometer. By default, the virtual counter input module is linked to the hardware incremental counter. In both cases the actual position in the application program is read with the same command.

The virtual layer in the MACS controller offer a flexible way to configure the system.

From your application code it is recommended that functions are accessed via the virtual module layer.

Axes and virtual counter input modules assignment

An axis in the MACS controller has slave and master information. Both, master and slave have its own virtual counter input module (VIRTCOUNTERIN) for position counter. The number of virtual counter input modules available in the system is two times the number of axis available.

This means the virtual counter modules are assigned by default to the existing axes as following:

VIRTCOUNTERIN n	n = 0 to (NumOfAxes-1)	assigned to slave position
VIRTCOUNTERIN n	n = NumOfAxes to ((2* NumOfAxes)-1)	assigned to master position

An example for a MACS controller with 6 axes.

VIRTCOUNTERIN 0	... VIRTCOUNTERIN 5	for slave
VIRTCOUNTERIN 6	... VIRTCOUNTERIN 11	for master

This default assignment can be changed by using the module AXE_PARAM. In this module the linking is done by giving the module number of the virtual counter input module in the following parameter.

PISRC_SVIRTCOUNT	Setting for linked virtual counter module number for slave
PISRC_MVIRTCOUNT	Setting for linked virtual counter module number for master

Actual status of the linking can be read in the process data:

REG_SVIRTCOUNT	Information about linked virtual counter number for slave
REG_MVIRTCOUNT	Information about linked virtual counter number for master

The same kind of assignment is valid for the virtual latch modules using:

PISRC_SVIRTLATCH	Setting for linked virtual latch module number for slave
PISRC_MVIRTLATCH	Setting for linked virtual latch module number for master

REG_SVIRTCOUNT	Information about linked virtual latch number for slave
REG_MVIRTCOUNT	Information about linked virtual latch number for master

Linking modules

For many modules the user can give the input source. This is done over the parameter that accepts the process input source **PISRC** (**p**rocess **i**nput **s**ou**rc**e).

The linking is done always on the module that takes data from somewhere. There are specific parameters in the parameter section of a module to give the process input source. For most of the modules the source is given with INDEX, SUBINDEX and LENGTH (bytes):

Sysvar[0x0129000A] = 0x4A000104;

whereas: 0x4A00 is index ; 0x01 is subindex and 0x04 is the length in **bytes**.

when including <sysdef.mi> in your project you can use the pre-defined names:

VIRTDIGIN_PARAM(0, VIRTDIGIN_PISRC_MAP1) = 0x4A000104;

or:

VIRTDIGIN_PARAM(0, VIRTDIGIN_PISRC_MAP1) = ((HWDIGIN_PROCESS_SRCINDEX(0, PO_HWDIGIN_VALLONG) <<8) +0x04);

There is a second variant to give the source for a module without specifying the data length. For example:

VIRTCOUNTIN_PARAM(0,VIRTCNTIN_PISRC_COUNTER) = 0x00488001;

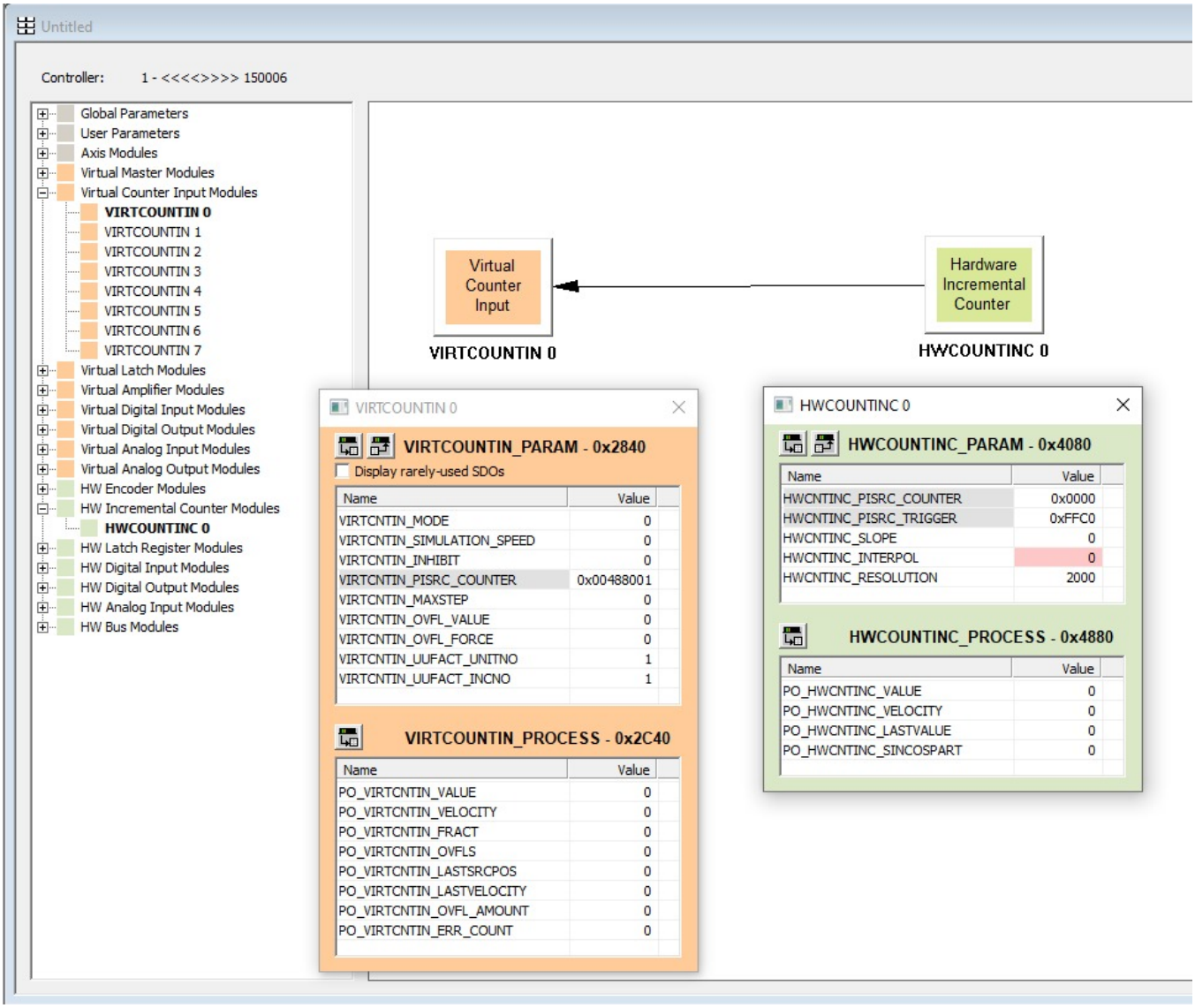
whereas: 0x4880 is index ; 0x01 is subindex

The modules are linked by default that makes sense for most users. Virtual modules are linked to their corresponding hardware modules. There is the possibility to re-link the modules.

Note: The module editor in the ApossIDE could be a could help for re-linking modules.

Module Editor

ApossIDE offers a Module Editor. This allows to display and check the existing module configuration. Furthermore, changes can be made directly in the module editor and write the changes directly to the controller.



Function modules are listed in the left area. By double clicking a module from that list the module is displayed on the right area and the corresponding module window is opened. In the module window you see the parameter and the process data section.

For more information on module editor, please see Module Editor.

8 Function Reference

The ApossIDE provides a library of built-in standard functions. These correspond roughly to the familiar functions available in standard C. These can be used anywhere in an application program without having to declare them.

> Some functions are supported only by newer versions of firmware. If connected to a controller with old firmware and an unsupported operator or function is used in an application program, then an error message will be displayed either during the compile or prior to the program being downloaded. These restrictions are described where appropriate in the following sections.

Integer Functions

Function	Description
min (expr,expr)	Return the minimum of the two expressions. If *both* expressions yield **long** values, then the result of the function is a **long** value.
max (expr,expr)	Return the maximum of the two expressions. If *both* expressions yield **long** values, then the result of the function is a **long** value.
abs (expr)	Return the absolute value of the expression.
rand (expr)	Return a psuedo random number in the range 0-32767 (0x7FFF). This function relies on an initial "seed" value being specified. Do this by setting "expr" to a non-0 number on the first call. Thereafter, set "expr" to 0 return subsequent random numbers. Given the same "seed" value, the function will always return the same sequence of random numbers.
rnd (x, y)	Executes the division "x/y" (y ≠ 0), rounding the result to the nearest integer.

Double Functions

> ⚠ These functions have only 32-bit precision on some hardware (See Controller Types).

Function	Description
min (expr,expr)	Return the minimum of the two expressions. If *either* expression yields a **double** value, then the result of the function is a **double** value.
max (expr,expr)	Return the maximum of the two expressions. If *either* expression yields a **double** value, then the result of the function is a **double** value.
fabs (expr)	Absolute value of the expression (firmware ≥ 6.6.04).
sqrt (expr)	Square root of the expression.
sin (expr)	Sine of the expression (expr in radians).

Function	Description
cos (expr)	Cosine of the expression (expr in radians).
tan (expr)	Tangent of the expression (expr in radians).
asin (expr)	Arcsine of the expression (result in radians).
acos (expr)	Arccosine of the expression (result in radians).
atan (expr)	Arctangent of the expression (result in radians).
grad (expr)	Convert the expression from radians to degrees.
rad (expr)	Convert the expression from degrees to radians.
log (expr)	Natural logarithm of the expression.
pow (x, y)	"x" raised to the power of "y".
round (expr)	Round the expression to the nearest integer value (result is **double**).

String Functions

Function	Description

chr (value)	Print the ascii character corresponding to the specified value. This function can only be used with the print command. For example:
	`print("Semicolon = ",chr(59));`
	will print:
	`Semicolon = ;`

radixstr (value,radix)	Format a value using a radix (firmware >= 7.0.90). This function can only be used with the print command. For example:
	`print("hex: ",radixstr(31,16));`
	`print("bin: ",radixstr(31,2));`
	will print:
	`hex: 1F`
	`bin: 11111`

Array Functions

Function	Description
arraylen (arr)	Determine the length of the array.
arraycpy (dest,src) **arraycpy** (dest,src,cnt)	Copy the "src" source array into the "dest" destination array.
arraycat (dest,src)	Concatenate the "src" source array onto the end of the

Function	Description
arraycat (dest,src,cnt)	"dest" destination array.
arrayset (dest,value) **arrayset** (dest,value,cnt)	Set values in the destination array.
ansilen (arr)	Determine the length of an array containing ANSI characters.
ansicpy (dest,src) **ansicpy** (dest,src,cnt)	Copy the "src" source array containing ANSI characters into the "dest" destination array.

8.1 ansicpy

Summary Copy ANSI text strings from one array to another array.

Syntax

```
length = ansicpy(dst, src);
length = ansicpy(dst, src, cnt);
```

Parameters

`dst` Name of destination array. **long**, **double**, and **wchar** arrays are supported. The array may include zero or more array indices. Not all array indices need to be specified. Arrays may be specified using either of the following methods:

```
arrayname
arrayname[ndx]...
```

`src` Name of **long** source array containing the ANSI string. See above for a description of how array names may be specified.

`cnt` The maximum number of characters (including any NULL-terminator) to be copied. If this is 0, then all characters up to the end of the array are copied. See below for a more detailed description of when the copy stops. This value will default to 0 (i.e. all characters) if it is not specified.

`ndx` Array index of start position within array. If an index is omitted, then that index will default to the begining of the index range for that index. Note that this is the index of a **long** element within the array; it is not possible to index individual characters within a **long** element.

Return Value The number of characters copied. This count will include the NULL-termintor if one is copied.

Description In the 1-dimensional case, the **ansicpy** routine will copy characters from the source array to the destination array and it will stop either at the specified maximum number of characters (including any NULL-termintor), at the end of the shorter array, or at the first NULL-termintor found in the source. The NULL-termintor, if found, will be *included* in the copy.

ANSI characters will be converted to Unicode characters as they are being copied.

> ANSI text strings cannot be printed directly with the print or printf commands. **ansicpy** should be used to first copy the ANSI string to a normal **wchar** array. Then the **wchar** array can be printed.

Summary Copy ANSI text strings from one array to another array.

For multi-dimensional arrays, the *entire* array is treated as a single 1-dimensional array with values stored "by row" (see Arrays). The copy does *not* stop at row boundaries, but continues through all elements in all rows.

> ⚠ Programmers must exercise care when using **ansicpy** with 2-dimensional arrays so that this "row wraparound" behaviour does not introduce bugs into application programs.

8.2 ansilen

Summary Find the length of an ANSI text string.

Syntax
```
length = ansilen(array);
```

Parameters

`array` Name of **long** array containing the ANSI string. The array may include zero or more array indices. Not all array indices need to be specified. Arrays may be specified using either of the following methods:

```
arrayname
arrayname[ndx]...
```

`ndx` Array index of start position within array. If an index is omitted, then that index will default to the begining of the index range for that index. Note that this is the index of a **long** element within the array; it is not possible to index individual characters within a **long** element.

Return Value The number of characters found up to, but not including, the first NULL-terminator.

Description The **ansilen** routine counts the number of ANSI characters found in the specified array. The search will stop either at the end of the array or at the first NULL character. If a NULL character is found, then this is *not* included in the count. If a start position is specified (by including one or more array indices), then the counting begins at this array element; all preceding characters are ignored (even if they are NULL).

For multi-dimensional arrays, the *entire* array is treated as a single 1-dimensional array with values stored "by row" (see Arrays). The search does *not* stop at row boundaries, but continues through all elements in all rows. If necessary, an application program can determine the individual row and column indices based on the count returned by **ansilen** and the declared dimensions of the array.

8.3 arraycat

Summary Concatenate array elements from one array onto the end of another array.

Syntax

```
length = arraycat(dst, src);
length = arraycat(dst, src, cnt);
length = arraycat(dst, "text");
length = arraycat(dst, "text", cnt);
```

Parameters

`dst` Name of destination array. **long**, **double**, and **wchar** arrays are supported. The array may include zero or more array indices. Not all array indices need to be specified. Arrays may be specified using either of the following methods:

```
arrayname
arrayname[ndx]...
```

`src` Name of source array. **long**, **double**, and **wchar** arrays are supported. See above for a description of how array names may be specified.

`cnt` The maximum number of elements to be copied. If this is 0, then all elements up to the end of the array are copied. See below for a more detailed description of when the copy stops. This value will default to 0 (i.e. all elements) if it is not specified.

`ndx` Array index of start position within array. If an index is omitted, then that index will default to the begining of the index range for that index.

`text` Text string.

Return Value The number of elements copied. In the case of a **wchar** or **string** array or text string, this count will include the NULL-termintor if one is copied.

Description In its simplest form (concatenating 1-dimensional **long** or **double** arrays), the **arraycat** routine will copy elements from the source array to the destination array and it will stop either at the specified maximum number of elements or at the end of the shorter array, whichever comes first. If the source array is longer than the destination array, then the excess elements in the source array are ignored. If the source array is shorter than the destination array, then the extra elements in the destination array will remain unchanged.

If *either* the destination or source array is a **wchar** array or text string, then *in addition* to the above stopping criteria, the copy will *also* stop at the first NULL-termintor found in the source. The NULL-termintor, if found, will be *included* in the copy.

The position where the first value will be copied into the destination array, will be determined by an implicit execution of the arraylen function.

Mixed-mode copies are also supported. For example, **long** arrays may be copied to **double** arrays, etc. Values will be automatically converted from the source type to the destination type during the copy.

For multi-dimensional arrays, the *entire* array is treated as a single 1-dimensional array with values stored "by row" (see Arrays). The copy does *not* stop at row boundaries, but continues through all elements in all rows.

> ⚠ Programmers must exercise care when using **arraycat** with 2-dimensional arrays

Summary Concatenate array elements from one array onto the end of another array.

> so that this "row wraparound" behaviour does not introduce bugs into application programs.

8.4 arraycpy

Summary Copy array elements from one array to another array.

Syntax

```
length = arraycpy(dst, src);
length = arraycpy(dst, src, cnt);
length = arraycpy(dst, "text");
length = arraycpy(dst, "text", cnt);
```

Parameters

`dst` Name of destination array. **long**, **double**, and **wchar** arrays are supported. The array may include zero or more array indices. Not all array indices need to be specified. Arrays may be specified using either of the following methods:

```
arrayname
arrayname[ndx]...
```

`src` Name of source array. **long**, **double**, and **wchar** arrays are supported. See above for a description of how array names may be specified.

`cnt` The maximum number of elements to be copied. If this is 0, then all elements up to the end of the array are copied. See below for a more detailed description of when the copy stops. This value will default to 0 (i.e. all elements) if it is not specified.

`ndx` Array index of start position within array. If an index is omitted, then that index will default to the begining of the index range for that index.

`text` Text string.

Return Value The number of elements copied. In the case of a **wchar** array or text string, this count will include the NULL-termintor if one is copied.

Description In its simplest form (copying 1-dimensional **long** or **double** arrays), the **arraycpy** routine will copy elements from the source array to the destination array and it will stop either at the specified maximum number of elements or at the end of the shorter array, whichever comes first. If the source array is longer than the destination array, then the excess elements in the source array are ignored. If the source array is shorter than the destination array, then the extra elements in the destination array will remain unchanged.

If *either* the destination or source array is a **wchar** array or text string, then *in addition* to the above stopping criteria, the copy will *also* stop at the first NULL-termintor found in the source. The NULL-termintor, if found, will be *included* in the copy.

Mixed-mode copies are also supported. For example, **long** arrays may be copied to **double** arrays, etc. Values will be automatically converted from the source type to the destination type during the copy.

Summary Copy array elements from one array to another array.

> Copying ANSI text strings is supported but must be done using the ansicpy function.

For multi-dimensional arrays, the *entire* array is treated as a single 1-dimensional array with values stored "by row" (see Arrays). The copy does *not* stop at row boundaries, but continues through all elements in all rows.

> Programmers must exercise care when using **arraycpy** with 2-dimensional arrays so that this "row wraparound" behaviour does not introduce bugs into application programs.

8.5 arraylen

Summary Find the length of an array or text string.

Syntax

```
length = arraylen(array);
length = arraylen("text");
```

Parameters

`array` Name of array. **long**, **double**, and **wchar** arrays are supported. The array may include zero or more array indices. Not all array indices need to be specified. Arrays may be specified using either of the following methods:

```
arrayname
arrayname[ndx]...
```

`ndx` Array index of start position within array. If an index is omitted, then that index will default to the begining of the index range for that index.

`text` Text string.

Return Value The return value depends of the type of parameter:

long The number of elements up to, but not including, the first 0 element.

double The number of elements up to, but not including, the first 0 element.

> The count stops only when an *exact* 0.0 value is found. Since numerical error in floating point calculations often results in "close to 0" values rather than exactly 0.0, application programs should *not* rely on finding 0 values resulting from calculations.

wchar The number of characters found up to, but not including, the first NULL-terminator.

`text` The number of characters found up to the end of the string.

Description The **arraylen** routine counts the number of non-0 elements found in the specified array. The search will stop either at the end of the array or at the first 0 element. If a 0 element is found, then this element is *not* included in the count. If a start position is specified (by including one or more array indices), then the counting begins at this array element; all preceding array elements are ignored (even if they are 0). Note that it

Summary Find the length of an array or text string.

is not necessary to specify all array indices when specifying a start position.

For multi-dimensional arrays, the *entire* array is treated as a single 1-dimensional array with values stored "by row" (see Arrays). The search does *not* stop at row boundaries, but continues through all elements in all rows. If necessary, an application program can determine the individual row and column indices based on the count returned by **arraylen** and the declared dimensions of the array.

> Finding the length of ANSI text strings is supported but must be done using the ansilen function.

8.6 arrayset

Summary Set array elements to a specific value.

Syntax

```
length = arrayset(dst, value);
length = arrayset(dst, value, cnt);
```

Parameters

`dst` Name of destination array. **long**, **double**, and **wchar** arrays are supported. The array may include zero or more array indices. Not all array indices need to be specified. Arrays may be specified using either of the following methods:

```
arrayname
arrayname[ndx]...
```

`value` Value to be assigned to array elements.

`cnt` The maximum number of elements to be set. If this is 0, then all elements up to the end of the array are set. This value will default to 0 (i.e. all elements) if it is not specified.

`ndx` Array index of start position within array. If an index is omitted, then that index will default to the begining of the index range for that index.

Return Value The return value will be the number of elements set.

Description The **arrayset** routine will assign the specified value to elements in the destination array. The value will be cast to the appropriate data type before being assigned. The routine will start at the specified array position and stop either at the specified maximum number of elements or at the end of the array, whichever comes first.

For multi-dimensional arrays, the *entire* array is treated as a single 1-dimensional array with values stored "by row" (see Arrays). The routine does *not* stop at row boundaries, but continues through all elements in all rows.

> Programmers must exercise care when using **arrayset** with multi-dimensional arrays so that this "row wraparound" behaviour does not introduce bugs into application programs.

9 Command Reference

In all of the reference pages that describe commands that refer to axes, the syntax descriptions will include one of the following:

(axis,...)
(axis, value,...)

"..." indicates that the command will accept multiple axes. If a "value" is required, then the "axis" and "value" parameters *must* be specified in *pairs* and each "value" will correspond to the preceeding "axis". For these commands, it is also possible to specify the keyword **AXALL** for "axis" and specify a *single* "value" (if required). If a "value" is specified, then the same value is used for all axes. For example:

Syntax: `AxisPosAbsStart(axis, position,...);`

```
// Move axis 0 to 1000.
AxisPosAbsStart(0,1000);

// Move axis 2 to 0 and move axis 1 to 1000.
AxisPosAbsStart(2,0,1,1000);

// Move all axes to 1000.
AxisPosAbsStart(AXALL,1000);
```

(axis)
(axis, value)

If the command syntax does not include "...", then it will accept only a single axis. If any other parameters are required, then they will apply only to the specified axis. For example:

Syntax: `res = Apos(axis);`

```
pos = Apos(0);     // Position of axis 0.
```

All references to axes in AposssC, are made using **axis module numbers**. Axis modules numbers are **0-based**, so the first axis will be axis module 0, the second will be module 1, and so on.

All available commands are organized below into general functional groups:

Amplifier	Command concerning the internal or external power stage
Axis and Information	Commands to control the axis or get information of it
Communication (CAN Basic)	Commands to send and receive CAN messages
Communication (CANopen)	Commands to send and receive CANopen SDOs and PDOs
Communication (EtherCAT)	Commands to configure and control the EtherCAT masters.
Communication (Serial)	Commands to handle the RS485 and RS233 interfaces.
Communication (Ethernet)	Commands to handle the Ethernet interface.

Data Recording	Commands to set speed and velocity, to start, stop, and to position.
Input/Output	Commands to set and read inputs and outputs.
Interrupt Functions	Commands to manage interrupts.
Kinematics	Commands for multi-axis control using selected robot kinematics.
Memory	Commands for data transfer and for saving of permanent data
State Machine	Commands related to state machines.
Synchronization	Commands to synchronize a slave with a master.
System	Various system commands
Debug	Various debug commands

9.1 Axis and Information Commands

Commands to initialize the axes and the control unit start up and define the zero point(s):

Acc	Set the acceleration for motion commands.
Apos	Query the current actual position of an axis.
AposDiff	Overflow handling of incremental encoders in applications.
Avel	Query the actual velocity of axis.
AxisControl	Select the axis control (ON, OFF ...).
AxisCvelStart	Start constant velocity mode.
AxisCvelStop	Stops the drive in constant velocity mode.
AxisHomeStart	Starts the integrated homing function.
AxisHomeStop	Stops (aborts) the integrated homing function
AxisHomeStatus	Returns the state of the integrated homing function.
AxisLinAbsStart	Simultaneous absolute positioning of several axes.
AxisLinRelStart	Simultaneous relative positioning of several axes.
AxisMoving	Test whether or not an axis is moving.
AxisPosAbsStart	Positioning in relation to actual zero point.
AxisPosAbsStartTvel	Position to the specified target point, ending with a specified target velocity.
AxisPosRelStart	Positioning in relation to actual position.
AxisStatus	Query axis and control status.

AxisStatusClear	Clear the slave and master index flags.
AxisStop	Stop any axis movement with Dec ramp.
AxisTargetReached	Test whether or not a controlled movement is active.
AxisWaitReached	Wait until the target position is reached.
Cpos	Query the current command position of an axis.
CposDiff	Overflow handling of incremental encoders in applications.
CurvePos	Retrieve slave curve position that corresponds to the current master position of the curve.
Cvel	Set the velocity for constant velocity controlled motor movements.
Dec	Set the deceleration for motion commands.
DefCposOrigin	Set the command position as zero point.
DefMasterCpos	Define initial position of the master.
DefMasterOrigin	Set the current master position as the zero point for the master.
DefOrigin	Set the current position as zero point.
Ipos	Query the last index or marker position of the slave.
IposDiff	Overflow handling of incremental encoders in applications.
JerkFinVel	Calculate the final velocity for a jerk-limited stop with maximum acceleration/deceleration.
JerkStopDist	Calculate the necessary distance for a jerk-limited stop with maximum deceleration.
Mapos	Query the current actual position of the master.
MaposDiff	Overflow handling of incremental encoders in applications.
Mavel	Query the actual velocity of the master.
Mipos	Query last index or marker position of the master.
MiposDiff	Overflow handling of incremental encoders in applications.
ResetOrigin	Erase temporary zero point.
SetMasterOrigin	Set any position as the zero point for the master.
SetOrigin	Set absolute position as temporary zero point.
TrackError	Query the the actual tracking error of an axis.
Tvel	Set the target velocity for the AxisPosAbsStartTvel

command.

9.2 Amplifier Commands

Commands for the internal amplifier

AmpErrorClear	Clears an error in the amplifier
HallDetectStart	Starts the hall detection procedure
HallDetectStatus	Polls the hall detection procedure status
MotorAlignStart	Starts the motor alignment procedure
MotorAlignStatus	Polls the motor alignment procedure status

9.3 Input/Output Commands

Commands to set and read inputs and outputs:

AnalogInput	Reads analog input and process data objects (PDO) of CAN objects.
AnalogOutput	Sets analog output (freely available) and an output, which is initialized as a drive.
DigInput	Reads input bit-by-bit (individually).
DigInputByte	Reads input by bytes (units of 8).
DigOutput	Sets digital outputs bit-by-bit (single).
DigOutputByte	Sets digital outputs by bytes (units of 8).

9.4 Interrupt Functions

Commands for managing interrupts:

InterruptSetup	Sets up interrupt handler functions.
InterruptDelete	Deletes interrupt handler functions.
InterruptEnable	Enables interrupt handling.
InterruptDisable	Disables interrupt handling.

9.5 Communication CANopen Commands

All global and axis parameters with a parameter code can be set and read with these commands:

LinkPdo	Link the system variable with PDO and copy in the internal parameters.
LinkSdo	Copying an internal object in the PDO.
SdoRead	Reads SDO of a connected CANopen device.

SdoReadSeg	Segmented read of SDOs (unpacked).
SdoReadSegp	Segmented read of SDOs (packed).
SdoStatus	Checks the result of an active communication.
SdoWrite	Sets an SDO of a connected CANopen device and uses the data exchange without defined number of valid data bytes (= "Data set size is not indicated").
SdoWriten	Sets an SDO and specifies the number of valid data bytes (= "Data set size is indicated").
SdoWriteSeg	Segmented write of SDOs (only for EtherCAT master).

9.6 Communication CAN Basic Commands

Commands for executing basic CAN functions:

CanDelete	Erases all or single CAN objects.
CanIn	Reads an object.
CanInit	Initializes the necessary objects (PDOs) for data exchange of CANopen nodes or enables extended CANINIT, CANIN function.
CanOpenRestart	Initializes the CANopen functionality.
CanOut	Sends message (active).
DefCanIn	Defines a receive object.
DefCanOut	Defines a transmit object in the CAN controller.
CANINPUT	Calls an interrupt handler routine when a CAN telegram arrives.

9.7 Communication EtherCAT Master Commands

Commands for handling an EtherCAT master:

ECatMasterCommand	Controls the EtherCAT master.
ECatMasterConfig	Configures the EtherCAT master.
ECatMasterInfo	Reads information from the EtherCAT master.

CAN over EtherCAT CoE is also supported. Therefore, any kind of SDO communication works also for the EtherCAT master.

Please check the list: Communication CANopen Commands

9.8 Ethernet Commands

Ethernet commands:

EthernetGetConnectionStatus	Get the current ethernet connection status.
EthernetOpenClient	Opens a socket connection to a socket server..
EthernetOpenServer	Opens a socket server, which waits for a connection from a client.
EthernetReceiveTelegram	Receives a telegram if there is one in the buffer.
EthernetSendTelegram	Sends a telegram.
EthernetClose	Close a socket connection.

General return values

ETHSOCK_RETVAL_OK = 0, // No error, everything OK

LWIP_ERR_MEM = -1, // Out of memory error.

LWIP_ERR_BUF = -2, // Buffer error.

LWIP_ERR_TIMEOUT = -3, // Timeout.

LWIP_ERR_RTE = -4, // Routing problem.

LWIP_ERR_INPROGRESS = -5, // Operation in progress

LWIP_ERR_VAL = -6, // Illegal value.

LWIP_ERR_WOULDBLOCK = -7, // Operation would block.

LWIP_ERR_USE = -8, // Address in use.

LWIP_ERR_ISCONN = -9, // Already connected.

LWIP_ERR_ABRT = -10,// Connection aborted.

LWIP_ERR_RST = -11,// Connection reset.

LWIP_ERR_CLSD = -12,// Connection closed.

LWIP_ERR_CONN = -13,// Not connected.

LWIP_ERR_ARG = -14,// Illegal argument.

LWIP_ERR_IF = -15,// Low-level netif error

ETHSOCK_RETVAL_INVALIDHANDLE = -21,// Invalid handle has been passed

ETHSOCK_RETVAL_NOFREESOCKETS = -22,// Limit of socket connection exceeded

ETHSOCK_RETVAL_NOTCONNECTED = -23,// The socket is not connected

ETHSOCK_RETVAL_TGTOOLONG = -24,// Too much data has been attempted to send

ETHSOCK_RETVAL_TXOVERFLOW = -25,// Previous data not yet sent

ETHSOCK_RETVAL_RXOVERFLOW = -26,// Previously received data has not been read and new data has arrived

ETHSOCK_RETVAL_RXTOOLONG = -27,// Received telegram is too long for the internal buffer

ETHSOCK_RETVAL_NOTSUPPORT = -28,// This controller does not support the socket feature

ETHSOCK_RETVAL_PROTNOTSUPPORT= -29 // The requested protocol type is not supported

9.9 Synchronization Commands

Commands to synchronize a slave with a master:

SyncCam	Enter CAM mode synchronization and wait in an "idle" state.
SyncCamMarkerMaster	Synchronization in CAM mode with master marker correction.
SyncCamMarkerSlave	Synchronization in CAM mode with slave marker correction.
SyncCamStart	Start slave for synchronization in CAM mode.
SyncCamStop	Stop slave after the CAM synchronization.
SyncDefOrigin	Defines master-slave relation for the next SyncPos or SyncMarker command.
SyncError	Queries actual synchronization error of the slave.
SyncSetCurve	Set a CAM curve.
SyncShiftOrigin	Relative shifting of the origin of synchronization.
SyncMarker	Synchronization of angle/position with marker correction.
SyncMarkerRestart	Resets a marker or resets marker handling.
SyncPos	Synchronization of angle/position.
SyncStatus	Queries flag for synchronization status.
SyncStatusClear	Resetting of the flags MERR and MHIT.
SyncVel	Synchronization of velocity.

Commands to access elements within a CAM array:

CamArrayIndex	Determine the array index of various data blocks within a CAM curve array.
CamArrayLen	Calculate the required length of a CAM array.

9.10 Kinematics Commands

Commands for multi-axis control using selected kinematics:

KinematicsCpoint	Query the current kinematics position.
KinematicsWorkToMc	Set a work-to-machine transformation that will be used

	with the selected kinematics.
KinematicsSetup	Select the multi-axis kinematics model to be used with a robot.
PathAcc	Set the acceleration to be used when following a path.
PathContinue	Continue following a path that has been paused.
PathDec	Set the deceleration to be used when following a path.
PathJerk	Set the limited-jerk values to be used when following a path.
PathMcToUu	Convert machine coordinates to axis coordinates.
PathMove	Move to a specified point, using the selected kinematics.
PathNext	Set the next multi-axis path curve to be used when the current path ends.
PathStart	Start following a multi-axis path curve, based on the selected kinematics.
PathStop	Stop a running path movement.
PathUuToMc	Convert axis coordinates to machine coordinates.
PathVel	Set the velocity to be used when following a path.
TransIdent	Set a transformation to the identity transformation (i.e. no transformation).
TransInverse	Transform a dpoint variable by the inverse of a transformation.
TransPoint	Transform a dpoint variable by a transformation.
TransReflect	Reflect a transformation across the YZ plane (i.e. X = 0).
TransRotateXY	Rotate a transformation in the XY plane (i.e. about the Z axis).
TransRotateYZ	Rotate a transformation in the YZ plane (i.e. about the X axis).
TransRotateZX	Rotate a transformation in the ZX plane (i.e. about the Y axis).
TransScale	Scale a transformation.
TransStretch	Stretch a transformation.
TransTranslate	Translate a transformation.

Commands to access elements within a Path array:

PathArrayIndex	Determine the array index of various data blocks within a Path curve array.

9.11 Memory Commands

Command for managing controller memory:

Delete	Delete all arrays in RAM defined by the DIM command.
MemoryDump	Save SDO data on the Micro-SD card.
MemoryDumpStatus	This command can be used to poll the state/result of a MemoryDump command.
Save	Save DIM arrays, axis parameters, global parameters, or user parameters in permanent memory.
DimArray	Get the reference of the DIM array from its number.
DimArrayNum	Get the array number from the array name.

9.12 Communication Serial Commands

Commands for handling serial communication:

SerialSetup	Initializes the serial interface handler.
SerialReadComm	Checks for a completely received telegram on the serial interface.
SerialWriteComm	Sends a complete telegram over the serial interface.
SerialReadChar	Reads a single character from the serial interface.
print / printcontrol	The print command can also be used to print to the serial interface (see sample ISR_Keypressed.mc)

9.13 Data Recording Commands

Commands for configuring and using arrays for recording test results:

RecordDest	Defines the memory section for saving recorded data.
RecordIndex	Specifies system variables for data recording.
RecordTime	Defines the sample interval (i.e. the amount of cycles between samples).
RecordTimeBase	(Optional) Defines the time base (i.e. the cycle time).
RecordType	Defines “one time” or “cyclic” recording.
RecordStart	Starts the data recording.
RecordStop	Stops the data recording.

9.14 State Machine Commands

State machine commands:

SmClear	Clear the event queue.
SmConfig	Define the state machine configuration parameters.
SmEvent	Define an event and its parameters.
SmInput	Request an event when a digital input changes.
SmMachine	Define a state machine variable.
SmParam	Request an event when a parameter value changes.
SmPeriod	Start a periodic timer.
SmPosition	Request an event when the motor passes a specified position.
SmPost	Create an event and post it to the end of the event queue.
SmPublish	Create an event and publish for all state machines.
SmRun	Start state machines running.
SmState	Define a state machine state, including its event handlers and substates.
SmStop	Stop running state machines.
SmSubscribe	Subscribe to receive published events.
SmTimer	Start a "once only" timer.
SmUnsubscribe	Request to stop receiving published events.
SmUrgent	Create an event and post it to the head of the event queue.

9.15 System Commands

Various system commands:

Delay	Time delay.
Delayus	Time delay in microseconds.
DimArray	Pass a DIM array to functions by referencing the array number of the DIM array rather than the array name.
ErrorAxis	Determine the axis causing the last error.
ErrorClear	Clear the last error.
ErrorInfo	Query additional information about the current error.
ErrorNo	Query the actual error code.
Time	Query the system time.
TimeDiffus	Calculate an elapsed time.

TimeHw	Query the system timer.
UserStatus	Set the user status.
WdActivate	Activate the watchdog.
WdReset	Reset the watchdog.

9.16 Acc

Summary Set the acceleration for motion commands.

Syntax `Acc(axis, value,...);`

Parameter
axis = Axis module number (see Command Reference).
value = Scaled acceleration value.

Return Value -

Description This command defines the acceleration for the next motion command (speed control, synchronizing, or positioning). The value will remain valid until a new acceleration value is set using the **Acc** command. The value is related to the parameters ***Maximum Acceleration*** RAMPMIN and ***Maximum Velocity*** VELMAX as well as ***Velocity Resolution*** VELRES.

> If acceleration has not been defined previous to a motion command, then the maximum acceleration will be the setting of parameter DFLTACC.

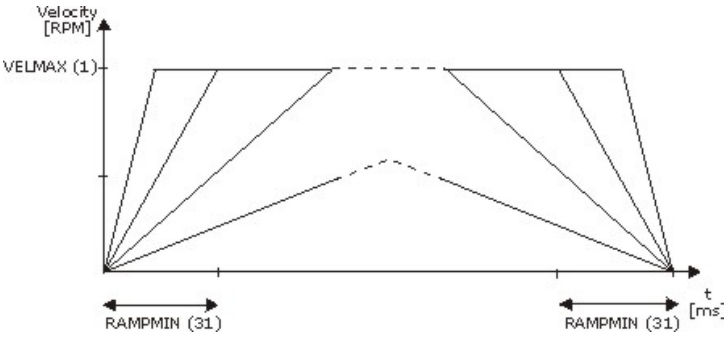
Default The default value is 100.

Command Group Axis and Information Commands

Syntax Examples

```
Acc(1, 10);           // Axis 1: 10
Acc(1, 50, 2, 10);    // Axis 1: 50, Axis 2: 10
Acc(AXALL, 30);       // All Axes: 30
```

Example Minimum acceleration time: 1000 ms
Maximum velocity: 1500 RPM (25 Rev/s)
Velocity resolution: 100



Program Sample See **File** → **Sample**: BasicPositionControl.mc

9.17 AmpErrorClear

Summary Clear amplifier error without doing any other function.

Syntax `AmpErrorClear(axis,...);`

Parameter
axis = Axis module number (see Command Reference).

Return Value -

Description This command can be used to clear a axis error in the internal amplifier. It also can be used to clear errors in a

Summary Clear amplifier error without doing any other function.

CANopen or EtherCAT drive as well.

Internally the AmpErrorClear is linked to the REG_CONTRLWORD in the axis process data. This can be used in the virtual amplifier CMDWORD. If the CMDWORD is forwarded to a drive using the CANopen control word (0x6040) over CAN or EtherCAT the AmpErrorClear can also clears an error in the drive.

Command Group Amplifier Commands

9.18 AnalogInput

Summary Reads an analog input.

Syntax `res = AnalogInput(n);`

Parameter n = Number of the analog input (1, 2, 3, ...)

Return Value Analog value.

Scaling see in controller type list: Controller Types

Description The **AnalogInput** command reads the value of the analog inputs. Depending on the hardware this can be either unipolar or bipolar and the resolution can also vary accordingly.

By default the internal available analog inputs are read.

> The **AnalogInput** is used to read a value from the virtual analog input module. The virtual analog input module can also be linked to a analog input device that is connected over CAN or EtherCAT bus.

Command Group Input/Output Commands

Syntax Example

```
an1 = AnalogInput(1);
print("Analog input 1 = ",an1);
```

9.19 AnalogOutput

Summary Sets analog output.

Syntax `AnalogOutput(n, value);`

Parameter n = Output number

value = Value to which the output should be set. Scaling see in controller type list: Controller Types

Return Value -

Description With the **AnalogOutput** command it is possible to set the freely available analog outputs.

By default the internal available analog outputs are used.

> The **AnalogOutput** is used to set a value in for the virtual analog output module. The virtual analog output module can also be linked to a analog output device that is connected over CAN or EtherCAT bus.

Command Group Input/Output Commands

Syntax Example

```
AnalogOutput(1, 12);  // Analog output 1 is set to 12
```

9.20 Apos

Summary Query the current actual position of an axis.

Syntax `res = Apos(axis);`

Parameter axis = Axis module number (see Command Reference).

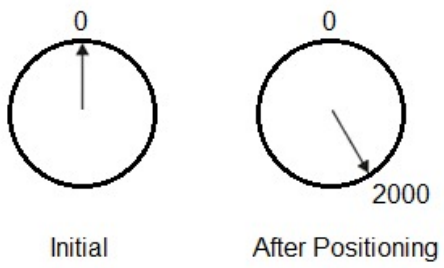
Return Value Current actual position relative to the current zero point.

Description The **Apos** command queries the current actual position of the axis relative to the current zero point and returns the value in user units (uu). This may be either a positive or negative value, depending on whether the current position is after, or before, the zero point. See How to Set Up Resolution and Gear Parameters for a description of user units.

> If a temporary zero point has been set using SetOrigin, then the position value is relative to this zero point.

> The final **Apos** value returned after a AxisPosAbsStart command has been completed, may differ slightly from the target position specified by the AxisPosAbsStart command. The error or deviation is normally due to the mechanics involved and to the numerical resolution of user units.

Example:




```
AxisPosAbsStart(0, 2000);
AxisWaitReached(0);
print("Actual Position Reached = ",Apos(0));
```

Output:

```
Actual Position Reached = 2000
```

Example with SetOrigin:

In the following, the SetOrigin command sets a new zero point to be at position 2000 relative to the original zero point. The AxisPosAbsStart command then moves the drive to a position 2000 beyond the new zero point.

```
SetOrigin(0, 2000);
AxisPosAbsStart(0, 2000);
AxisWaitReached(0);
print("Actual Position Reached = ",Apos(0));
```

Output:

```
Actual Position = 2000
```

Summary Query the current actual position of an axis.

Command Group Axis and Information Commands

Program Samples See **File** → **Sample**: BasicPositionControl.mc, BasicVelocityControl.mc

9.21 AposDiff

Summary Overflow handling of incremental encoders in applications.

Syntax
```
res = AposDiff(axis, oldpos);
```

Parameter

axis = Axis module number (see Command Reference).

oldpos = Apos of the specified axis at a previous time.

Return Value Returns difference between Apos and oldpos (i.e. "Apos(...) - oldpos") in uu.

Description This command simplifies overflow handling of incremental encoders in applications. For example, if the user saves an actual position in the program and needs to calculate a difference in position at some later time, then he must normally account for the possibility of an overflow of the position. Instead, this command may be used.

This command will usually return the simple mathematical difference between the old and current actual positions (i.e. "Apos(...) - oldpos"). However, if it is determined that a position overflow occured between the two positions, then the difference is adjusted to account for the overflow so that the correct position difference is returned.

See Encoder Position Values for a description of position overflow.

> This command is designed to work only in the case where the gearing factor is 1 (i.e. 1 uu = 1 qc). See How to Set Up Resolution and Gear Parameters for a description of gearing factors.

Command Group Axis and Information Commands

Syntax Example
```
oldpos = Apos(0);
...
diff = AposDiff(0, oldpos);
```

9.22 Avel

Summary Query the actual velocity of axis.

Syntax
```
res = Avel(axis);
```

Parameter

axis = Axis module number (see Command Reference).

Return Value Actual signed velocity of axis in uu/s.

Description This function returns the actual velocity of the corresponding axis in User Units per second.

The velocity value is filtered over a standard duration of 20 ms. However, this duration may be changed by the user with the process data value REG_VELSMPT for each axis. If the duration is changed, then the new duration will apply to all future **Avel** querries. For example, the setting:

```
AXE_PROCESS(axis,  REG_VELSMPT) = 100;
```

sets the duration of the measurement to 100 ms. This will result in a considerably better resolution of the velocity returned by **Avel** and Mavel. However, quick velocity changes will be "smoothed"; they will appear smaller and last longer.

Command Group Axis and Information Commands

9.23 AxisControl

Summary Turn the motor on or off.
Turn motor control on with a user defined reference position.
Turn motor control on with a user defined reference current.
Turn motor control on with a user defined reference velocity.

Syntax `AxisControl(axis, cmd,...);`

Parameter

axis = Axis module number (see Command Reference).

cmd = Axis command: ON, OFF, USERREFPOS, USERREFCUR or USERREFVEL.

Return Value -

Description ON

When an application program is started, motors are turned off. This means the motors are free to move and the controller is not trying to drive them. Motors must be turned on using the **AxisControl** command with the ON parameter before commands such as AxisPosAbsStart, etc., can be used to move them. When the ON parameter is executed, the motor's command position is set to the actual position, the command velocity is set to 0 (i.e. the motor remains at the current actual position), and the positioning error is reset.

> Because the ON parameter sets the motor velocity to 0, the motor is expected to be stationary when the command is executed. If the motor is *not* stationary, then it will be stopped immediately *using the maximum possible torque*.

Normally, the ON parameter is expected to be used at the beginning of an application program or after the OFF parameter has been executed. However, it may actually be used at any time, even when the motor is already turned on and being actively driven. In this case, as described above, the motor will be stopped immediately *using the maximum possible torque*. Using the ON parameter in this way is not recommended. Instead, the AxisStop should be used to stop the motors in a more controlled manner.

> The ON parameter cannot be used for any axis that is currently in kinematics mode (see KinematicsSetup). Motors should be turned on *before* kinematics mode is enabled and motors in kinematics mode that are moving, should be stopped using the PathStop command.

> The ON parameter is not suitable for re-activation of position control following an error. For this purpose, the ErrorClear command must be used.

OFF

The OFF parameter switches off the motor control ("Coasting" state).
After OFF is executed, the drive axis can be moved freely, as long as there is no motor brake. Note that the motor will be still monitored and the Apos command may continue to be used to query the actual position.

> If OFF is used to switch off *any* motor that is currently in kinematics mode (see KinematicsSetup), then *all* motors in that kinematics group will *first* be stopped in a controller manner. Once the motors are stopped, then kinematics mode will be *disabled* and motor control will be switched off for all of the motors. The drive axes will then be able to move freely. OFF cannot be used to switch kinematics motors directly from a controlled moving state to a freely moving "motor-off" state since this has the potential to damage the robot's mechanics.

> For a restart of a motion process after OFF, the ON command must be used. Only the command AxisStop automatically activates ON.

USERREFPOS

This command is intended to allow an external device to have complete control over the motor's positioning functions. It bypasses the controller's own internal position-generating functionality and relies *entirely* on a stream of "reference" positions generated by the external device and fed to the controller. Typically, this is done by linking incoming PDO messages to the REG_USERREFPOS axis process value.

> ⚠ When using this command, it is critical that the incoming PDO messages are correctly synchronized with the controller's internal operating cycle. Otherwise, it will not be possible to produce smooth motions. Please contact zub machine control AG for further information.

Summary Turn the motor on or off.
Turn motor control on with a user defined reference position.
Turn motor control on with a user defined reference current.
Turn motor control on with a user defined reference velocity.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

The command will enable the motor and immediately move it to the position (in qc) currently stored in the REG_USERREFPOS axis process value (SDO 0x2500/40). REG_USERREFPOS will then be used for all future reference positions. This means that each time REG_USERREFPOS is updated, the motor will be moved to the new position. Note that the motor will stop at this position unless a new position is received.

The motor can be disabled again with OFF.

Example

```
// Turn motor control on with a user defined reference position
AxisControl(axis, USERREFPOS);

// Position step in qc
AXE_PROCESS(axis,REG_USERREFPOS)=1024;
```

USERREFCUR

This command is similar to USERREFPOS (see above) except that it allows an external device to control the motor current rather than the position. In a first step, the speed control circuit must also be bypassed.

> ⚠ When using this command, it is critical that the incoming PDO messages are correctly synchronized with the controller's internal operating cycle. Otherwise, it will not be possible to produce smooth motions. Please contact zub machine control AG for further information.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

The command will enable the motor and immediately set an current (in mA) currently stored in the REG_USERREFCUR axis process value (SDO 0x2500/46). REG_USERREFCUR will then be used for all future reference positions. This means that each time REG_USERREFCUR is updated, the motor will set the actual output current.

The motor can be disabled again with OFF

Example

```
// Set the controller loop
HWAMP_PARAM(axis, HWAMP_MODE) = HWAMP_MODE_POS_CUR;

// Turn motor control on with a user defined reference current
AxisControl(axis, USERREFCUR);

// Current Step (mA)
AXE_PROCESS(axis,REG_USERREFCUR)=1000;
```

USERREFVEL

This command is similar to USERREFPOS (see above) except that it allows an external device to control the motor velocity rather than the position.

> ⚠ When using this command, it is critical that the incoming PDO messages are correctly synchronized with the controller's internal operating cycle. Otherwise, it will not be possible to produce smooth motions. Please contact zub machine control AG for further information.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

The command will enable the motor and immediately start to drive it at the velocity (in qc/sec) currently stored in the REG_USERREFVEL axis process value (SDO 0x2500/27). REG_USERREFVEL will then be used for all future reference velocities. This means that each time REG_USERREFVEL is updated, the motor velocity will be changed to the new velocity.

Summary Turn the motor on or off.
Turn motor control on with a user defined reference position.
Turn motor control on with a user defined reference current.
Turn motor control on with a user defined reference velocity.

The motor can be disabled again with OFF.

Example

```
// Set the controller loop
HWAMP_PARAM(axis, HWAMP_MODE) = HWAMP_MODE_POS_VEL_CUR;

// Turn motor control on with a user defined reference velocity
AxisControl(axis, USERREFVEL);

// Velocity Step
AXE_PROCESS(axis,REG_USERREFVEL) =
vel_rpm * 0x4000 / HWAMP_PARAM(axis, HWAMP_MAXRPM);
```

Command Group Axis and Information Commands

Syntax Example

```
AxisControl(0, ON);                    // Switch on controller Axis Mod 0.
AxisControl(AXALL, ON);                // Switch on controller all axes.

AxisControl(0, OFF);                   // Switch off controller Axis Mod 0.
AxisControl(AXALL, OFF);               // Switch off controller all axes.

AxisControl(0, ON, 1, OFF);            // Switch axis 0 on and axis 1 off.
```

Program Sample See **File** → **Sample**: MotorControl.mc

9.24 AxisCvelStart

Summary Start constant velocity mode.

Syntax `AxisCvelStart(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description The **AxisCvelStart** command is starting the drive in speed control mode. Acceleration/deceleration, as well as the speed should be set via the Acc, Dec, and Cvel commands prior to starting.

AxisCvelStart does not contain the command AxisControl(ON) which turns on the motor control. When using **AxisCvelStart** an explicit execution of AxisControl(ON) is necessary after a previous use of AxisControl(OFF).

> If no speed value has been defined via Cvel before the beginning of **AxisCvelStart**, then the default velocity 0 is used. This means that the motor will not rotate, but the PID controller is active.

All Cvel commands following the start of speed mode will be carried out immediately, i.e. a corresponding speed change will take place immediately with the defined acceleration or deceleration (Acc/Dec).

Command Group Axis and Information Commands

Syntax Example

```
AxisCvelStart(0);        // RPM mode Axis Module 0 start
AxisCvelStart(0, 2);     // RPM mode Axis Modules 0 and 2 start
AxisCvelStart(AXALL);    // RPM mode for all axes start
```

Program Sample See **File** → **Sample**: BasicVelocityControl.mc

9.25 AxisCvelStop

Summary Stops the drive in constant velocity mode.

Syntax `AxisCvelStop(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description Via the **AxisCvelStop** command, the speed mode is terminated and the positioning mode is started, whereby a still rotating axis is stopped with the deceleration defined with Dec and the motor is held in the stop position.

> Abruptly braking an axis in motion can lead to mechanical damage, and thus to instability in the regulation.

> The **AxisCvelStop** command should only be used on non-moving axes, which have been brought to velocity 0 via the Cvel command.
> The **AxisCvelStop** command carried out in the positioning mode can also cause an abrupt termination of the positioning procedure.

Command Group Axis and Information Commands

Syntax Example

```
AxisCvelStop(0);          // RPM mode Axis Module 0 stop
AxisCvelStop(0, 2);       // RPM mode Axis Modules 0 and 2 stop
AxisCvelStop(AXALL);      // RPM mode for all axes stop
```

Program Sample See **File** → **Sample**: BasicVelocityControl.mc

9.26 AxisHomeStart

Summary Starts the integrated homing function.

Syntax `AxisHomeStart(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description The homing-related axis parameters have to be set before.

Please read the homing description article for more details.

AxisHomeStart does not contain the command AxisControl(ON) which turns on the motor control. When using **AxisHomeStart** an explicit execution of AxisControl(ON) is necessary after a previous use of AxisControl(OFF).

> Please note that the POSDRCT parameter will not have any influence on the rotational direction of the motor, and that the unit of HOME_OFFSET is in [qc], not [UU].

Compatiblity From controller firmware 9.0.39

Command Group Axis and Information Commands

9.27 AxisHomeStatus

Summary Returns the state of the integrated homing function.

Syntax `res = AxisHomeStatus(axis);`

Parameter axis = Axis module number (see Command Reference).

Return Value

Value	Name	Description
0	HOMING_STATE_IDLE	Idle
1	HOMING_STATE_START	Starting state
2	HOMING_STATE_WAIT_NEGLIM_RISING	Wait for negative limit switch rising edge
3	HOMING_STATE_WAIT_NEGLIM_FALLING	Wait for negative limit switch falling edge
4	HOMING_STATE_WAIT_POSLIM_RISING	Wait for positive limit switch rising edge

Summary Returns the state of the integrated homing function.

5	HOMING_STATE_WAIT_POSLIM_FALLING	Wait for positive limit switch falling edge
6	HOMING_STATE_WAIT_HOMESW_RISING	Wait for home switch rising edge
7	HOMING_STATE_WAIT_HOMESW_FALLING	Wait for home switch falling edge
8	HOMING_STATE_WAIT_INDEX	Wait for encoder index pulse
9	HOMING_STATE_WAIT_MOTORCUR	Wait for motor current over CUR_THRESHOLD
10	HOMING_STATE_WAITSTOP	Wait for motor movement to finish
100	HOMING_STATE_FINISHED	Homing is finished
-1	HOMING_STATE_ERR_INDEXSEARCH	Encoder index pulse has not been found within two encoder revolutions
-2	HOMING_STATE_ERR_AXIS	General axis error occured (see Error reference)
-3	HOMING_STATE_ERR_INVALIDTYPE	Invalid HOME_TYPE value
-4	HOMING_STATE_ERR_AXISON	Axis is not in control (ON) state

Description This function returns the current state of the homing procedure. The homing has to be started previously by the AxisHomeStart command.

Compatiblity From controller firmware 9.0.39

Command Group Axis and Information Commands

9.28 AxisHomeStop

Summary Stops (aborts) the integrated homing function.

Syntax `AxisHomeStop(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description Please read the homing description article for more details about the homing function.

Compatiblity From controller firmware 9.0.39

Command Group Axis and Information Commands

9.29 AxisLinAbsStart

Summary Simultaneous absolute positioning of several axes.

Syntax `AxisLinAbsStart(axis, pos,...);`

Parameter axis = Axis module number (see Command Reference).

pos = Target position.

Return Value -

Description The **AxisLinAbsStart** command enables two or more drives to be operated so that all axes involved reach their absolute target position simultaneously. The axis with the longest drive time will be considered the lead axis and all other axes will be driven with a corresponding low velocity. The **AxisLinAbsStart** command compensates for different accelerations and decelerations so that a **AxisLinAbsStart** command in an orthogonal system will result in a straight line movement.

AxisLinAbsStart does not contain the command AxisControl(ON) which turns on the motor control. When using **AxisLinAbsStart** an explicit execution of AxisControl(ON) is necessary after a previous use of AxisControl(OFF).

> The position input is made in user units (uu) and corresponds to the standard setting of the number of quad counts (parameter ***User factor*** = 1,000).

If a temporary zero point (set and activated via SetOrigin) exists, then the position is relative to this zero point.

Summary Simultaneous absolute positioning of several axes.

If an acceleration and/or velocity have not been defined at the time of the **AxisLinAbsStart** command, then the values of the parameters DFLTVEL and DFLTACC are used for moving.

> ⚠ For the **AxisLinAbsStart** command to function correctly, the following must be true for all axes involved:
>
> 1. All axes *must* be stationary when the command is started.
> 2. All axes *must* have the *same* RAMPTYPE value selected.
> 3. All axes *must* have the *same* PROFTIME value selected.

> ⚠ Executing the **AxisLinAbsStart** command will cause the values of the four JERKMIN axis parameters to be modified. If it is important to maintain the current values, then they should be saved by the application program and restored after the **AxisLinAbsStart** command has been completed.

Command Group Axis and Information Commands

Syntax Example

```
AxisLinAbsStart(1, 10000, 2, 50000);   // Position axes 1 and 2
AxisLinAbsStart(AXALL, 50000);         // Position all axes
```

Program Sample See **File** → **Sample**: SimpleSynchronization.m

9.30 AxisLinRelStart

Summary Simultaneous relative positioning of several axes.

Syntax

```
AxisLinRelStart(axis, pos,...);
```

Parameter

axis = Axis module number (see Command Reference).

pos = Target positions.

Return Value -

Description Using the **AxisLinRelStart** command, several drives can be operated so that all involved axes can reach their target position simultaneously relative to their starting position. The axis with the longest drive time will be considered the lead axis, and all other axes will be driven with a correspondingly low velocity. The **AxisLinRelStart** command compensates for different accelerations and decelerations so that a **AxisLinRelStart** command in an orthogonal system will result in a straight line.

> The position input is made in User Units (uu). The position input is scaled with the 'position scaling' axis parameter, which corresponds in its standard setting to the number of quad counts (parameter ***User factor*** = 1,000).

If acceleration and/or velocity have not been defined at the time of the **AxisLinRelStart** command, then the values of the parameters DFLTVEL and DFLTACC are used for moving.

> ⚠ For the **AxisLinRelStart** command to function correctly, the following must be true for all axes involved:
>
> 1. All axes *must* be stationary when the command is started.
> 2. All axes *must* have the *same* RAMPTYPE value selected.
> 3. All axes *must* have the *same* PROFTIME value selected.

> ⚠ Executing the **AxisLinRelStart** command will cause the values of the four JERKMIN axis parameters to be modified. If it is important to maintain the current values, then they should be saved by the application program and restored after the **AxisLinRelStart** command has been completed.

Command Group Axis and Information Commands

Syntax Example

```
AxisLinRelStart(1, 10000, 2, 50000);   // Position axes 1 and 2
AxisLinRelStart(AXALL, 50000);         // Position all axes
```

Program Sample See **File** → **Sample**: SimpleSynchronization.m

9.31 AxisMoving

Summary Test whether or not an axis is moving.

Syntax `res = AxisMoving(axis);`

Parameter axis = Axis module number (see Command Reference).

Return Value 0 - The axis is not moving.
1 - The axis is moving.

Description This command can be used for determining whether or not an axis is moving.

Command Group Axis and Information Commands

9.32 AxisPosAbsStart

Summary Positioning in relation to actual zero point.

Syntax `AxisPosAbsStart(axis, pos,...);`

Parameter axis = Axis module number (see Command Reference).

pos = Position in user units (uu) absolute to the actual zero point.

Return Value -

Description The axis can be moved to a position absolute to the actual zero position.

When the **AxisPosAbsStart** command exceeds the software limit switch NEGLIMIT or POSLIMIT the program continues with the next command after an error.

AxisPosAbsStart does not contain the command AxisControl(ON) which turns on the motor control. When using **AxisPosAbsStart** an explicit execution of AxisControl(ON) is necessary after a previous use of AxisControl(OFF).

> If a temporary zero point, set via SetOrigin exists and is active, then the position result refers to this zero point.

> If an acceleration and/or velocity have not been defined at the time of the **AxisPosAbsStart** command, then the procedure will take place with the values of parameter DFLTVEL and DFLTACC.

> When several axes are started simultaneously, the axes are *not* guaranteed to arrived at their target positions simultaneously. Each axis will arrive at its target position at a time determined solely by its own velocity, acceleration, etc., settings.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Axis and Information Commands

Syntax Example

```
AxisPosAbsStart(1, 50000);              // Move axis 1 to position 50000
AxisPosAbsStart(1, 20000, 2, 30000);    // Position axis 1 and 2
AxisPosAbsStart(AXALL, 50000);          // Position all axes to 50000
```

Program Sample See **File** → **Sample**: BasicPositionControl.m

9.33 AxisPosAbsStartTvel

Summary Position to the specified target point, ending with a specified target velocity.

Syntax `AxisPosAbsStartTvel(axis, pos,...);`

Parameter axis = Axis module number (see Command Reference).

pos = Position in user units (uu) with respect to the actual zero point.

Return Value -

Description The **AxisPosAbsStartTvel** command is similar to the AxisPosAbsStart command in that it will move the axis to the specified absolute position. However, whereas AxisPosAbsStart ends the movement with a 0 velocity,

Summary Position to the specified target point, ending with a specified target velocity.

AxisPosAbsStartTvel will end with the (usually non-0) velocity specified by the Tvel command. When the movement ends, the axis will automatically enter Cvel (continuous velocity) mode. In addition, the movement done by **AxisPosAbsStartTvel** will *always* be a limited-jerk movement. See Limited Jerk for more information about limited-jerk movements. In all other respects (i.e. limit switch handling, origins, default velocities, etc.), the **AxisPosAbsStartTvel** command behaves in the same manner as the AxisPosAbsStart command.

Since a limited-jerk movement is done, the maximum velocities, accelerations, etc., are specified by the Vel, Acc, and Dec commands and by the four JERKMIN parameters.

As with the AxisPosAbsStart command, when the target position is reached, bit 3 (0x0004) of the PFG_FLAGS status flags (SDO 0x2500/123) will be set. This can be used by application programs to identify when the next movement may be started.

There are several cases which the **AxisPosAbsStartTvel** command is unable to handle. These include the following:

1. The target position is ahead of the current position, the target velocity is forward, and the current velocity is very fast forwards. If this causes the axis to overshoot the target, then **AxisPosAbsStartTvel** is not able to reverse and re-approach the target a second time.
2. The target position is ahead of the current position but quite close, the target velocity is forward and very fast, and the current velocity is slow. In this case, it may not be possible for the axis to accelerate quickly enough to match the target velocity before it reaches the target position. **AxisPosAbsStartTvel** is not able to reverse and take a longer approach run towards to the target position.
3. The target position is ahead of the current position, the target velocity is reversed, and the current velocity is very fast forwards. Here there is a possibility that the axis will overshoot the target position, reverse direction, and then reach constant reverse velocity *before* arriving back at the target position. If this happens, then **AxisPosAbsStartTvel** will enter the final constant velocity Cvel mode too soon and be unable to complete the movement using a second limited-jerk movement.
4. All the same movements when the target position is behind the current position and the velocities are reversed.

If any of these cases occur, then bit 16 (0x8000) of the PFG_FLAGS status flags (SDO 0x2500/123) will be set as soon as the **AxisPosAbsStartTvel** command is executed. The **AxisPosAbsStartTvel** command will then be ignored and the axis will continue whatever movement was active before the **AxisPosAbsStartTvel** command was executed.

> ⚠ If there is any possibility that one of these cases may occur, then the application program should use this flag to trap the error and to take some appropriate error recovery action. It is important to note that in these cases, the position reached staus flag (bit 3) will never be set. Status flags can be trapped using the STATBIT interrupt. For trapping flags in state machine programs, see Detecting Axis Status Changes.

Command Group Axis and Information Commands

9.34 AxisPosRelStart

Summary Positioning in relation to actual position.

Syntax `AxisPosRelStart(axis, dist,...);`

Parameter

axis	=	Axis module number (see Command Reference).
dist	=	Distance to actual position in user units (uu).

Return Value -

Description One or more axes can be moved to a position relative to the actual position by use of the **AxisPosRelStart** command.

AxisPosRelStart does not contain the command AxisControl(ON) which turns on the motor control. When using **AxisPosRelStart** an explicit execution of AxisControl(ON) is necessary after a previous use of AxisControl(OFF).

> If acceleration and/or velocity have not been defined at the time of the **AxisPosRelStart** command, then the procedure will take place with the values of parameter DFLTVEL and DFLTACC.

> When several axes are started simultaneously, the axes are *not* guaranteed to arrived at their target positions simultaneously. Each axis will arrive at its target position at a time determined solely by its own velocity, acceleration, etc., settings.

Summary Positioning in relation to actual position.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Axis and Information Commands

Syntax Example

```
AxisPosRelStart(1, 50000);            // Move axis 1 relative 50000 UU
AxisPosRelStart(1, 20000, 2, 30000);  // Relative position axis 1 and 2
AxisPosRelStart(AXALL, 50000);        // Move all axes relative 50000 UU
```

Program Sample See **File** → **Sample**: BasicPositionControl.m

9.35 AxisStatus

Summary Query axis and control status.

Syntax `res = AxisStatus(axis);`

Parameter axis = Axis module number (see Command Reference).

Return Value Axis and Control status (4-Byte value):

Bit	#define (SysDef.mh)	Meaning
Byte 0		
0	AXSTAT_INDEXMASTER	Index has occurred on master encoder.
1	AXSTAT_INDEXSLAVE	Index has occurred on slave encoder.
2	AXSTAT_POSCTRLOFF	Pos. control switched off (= AxisControl(..., OFF) state).
3	AXSTAT_VOLTAGERANGE	Supply voltage of power stage is out of range (note 4).
5	AXSTAT_STARTACTIVE	Start switch active.
6	AXSTAT_REFACTIVE	Reference switch active.
7	AXSTAT_LIMITACTIVE	Limit switch active.
Byte 1	Power amplifier bits	
8	AXSTAT_I2TTHRESHOLD	I2T treshold or overcurrent error at amplifier.
9	AXSTAT_OVERVOLTAGE	Overvoltage at the amplifier.
10	AXSTAT_TEMPERATURE	Overtemperature threshold (note 6).
Byte 2	Position control status	
18	AXSTAT_POSREACHED	Position reached (note 5).
23	AXSTAT_FLOATING	Position control switched off (identical to AXSTAT_POSCTRLOFF).
Byte 3	Position control status	
24	AXSTAT_MOVING	Axis is moving. See also AxisMoving.
25	AXSTAT_OVERFLOWS	Overflow slave encoder.
26	AXSTAT_OVERFLOWM	Overflow master encoder.
28	AXSTAT_POSLIMERR	Positive software position limit reached (note 2).
29	AXSTAT_NEGLIMERR	Negative software position limit reached (note 2).
30	AXSTAT_CURVE_ERR	Error during execution of SyncCam (note 1).
31	AXSTAT_OVERSHOOT	AxisPosAbsStartTvel command will overshoot the target.

Summary Query axis and control status.

Note 1: PFG_LASTERROR (SDO 0x2500/163) gives more information about the type of error for axis 1.

Note 2: These bits are set, if the corresponding software limit was activated and there was no movement in the opposite direction in the meanwhile. If the motor is moved in the opposite direction, the bit is cleared again.

Note 4: This bit is just valid in case of MACS control units with integrated power amplifiers. If the bit is set, this means that the discrete supply voltage of the integrated power amplifiers is not present or out of the specified range. The bit is 0, if the supply voltage is correct, or a MACS control unit without integrated power amplifiers is in use.

Note 5: See parameter TESTWIN how to define the information "position reached".

Note 6: Pre-warning before temperature switch-off limit.

Description The **AxisStatus** command reports the actual status of the axis control unit as well as that of the queried axis. For example, whether the axis controller shuts down, ends the motion or the end switch is active.

The status consists of a total of four bytes.

> The reported status is dependent on the hardware.

Command Group Axis and Information Commands

Syntax Example

```
print(AxisStatus(0));         // print axis mod 0 status word

if (AxisStatus(0) & AXSTAT_POSCTRLOFF)
    print("Position control is switched off");

if (AxisStatus(0) & AXSTAT_POSREACHED)
    print("Target position reached");

if (AxisStatus(0) & AXSTAT_MOVING)
    print("Motor is in controlled motion");
```

Program Sample See **File** → **Sample**: Information.mc

9.36 AxisStatusClear

Summary Clears AxisStatus Bits.

Syntax `AxisStatusClear(axis, value,...);`

Parameter axis = Axis module number (see Command Reference).

value = Bit to clear

Bit	#define (SysDef.mh)	Meaning
0	AXSTAT_INDEXMASTER	Clear Master Encoder Index Flag
1	AXSTAT_INDEXSLAVE	Clear Slave Encoder Index Flag

Return Value -

Description This command clears the Slave and Master index flags.

Command Group Axis and Information Commands

9.37 AxisStop

Summary Stop the motor.

Syntax `AxisStop(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Summary Stop the motor.

Return Value -

Description By using the **AxisStop** command, a drive in positioning, speed, or synchronizing mode can be decelerated and stopped using programmed deceleration.

The command can also be used to "catch" a running motor and stop it when the axis is currently in AxisControl OFF state ("coasting" motor).

> ⚠ If **AxisStop** is used to stop *any* motor that is currently in kinematics mode (see KinematicsSetup), then *all* motors in that kinematics group will be stopped in a controlled manner. Once the motors are stopped, then kinematics mode will be *disabled*.

> After **AxisStop** is executed, the application will continue to run; the braking process runs in the background. The application should wait for the drive, or drives, to slow down to a speed of zero before any new positioning command is given.

Command Group Axis and Information Commands

Syntax Example

```
AxisStop(0);        // Stop Axis Module 0.
AxisStop(AXALL);    // Stop all axes.
```

9.38 AxisTargetReached

Summary Test whether or not a controlled movement is active.

Syntax

```
res = AxisTargetReached(axis);
```

Parameter axis = Axis module number (see Command Reference).

Return Value 0 - Controlled movement is not active.
1 - Controller movement is active.

Description This command can be used for determining whether or not a movement has terminated. For example, when the target position is reached and the positioning command (e.g. AxisPosAbsStart, AxisPosRelStart) has actually been completed, then this will return to indicate that controlled movement is no longer active.

Command Group Axis and Information Commands

9.39 AxisWaitReached

Summary Wait until the target position is reached.

Syntax

```
AxisWaitReached(axis,...);
```

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description The **AxisWaitReached** command will cause the application program to wait until the specified axes have reached their target positions.

There is a difference in the behaviour, depending on the setting of the axis parameter TESTWIN

Command Group Axis and Information Commands

Syntax Examples

```
AxisWaitReached(0);     // Wait until Axis mod 0 has ended motion
AxisWaitReached(0, 1);  // Wait until Axis mods 0 and 1 have ended motion
AxisWaitReached(AXALL); // Wait until all Axes have ended motion
```

9.40 CamArrayIndex

Summary Determine the array index of various data blocks within a CAM curve array.

Syntax
```
res = CamArrayIndex(CAM_INDEX_INFO, arrayname);
res = CamArrayIndex(CAM_INDEX_GRADIENT, arrayname);
res = CamArrayIndex(CAM_INDEX_SEGMENT, arrayname, num);
res = CamArrayIndex(CAM_INDEX_STARTSTOP, arrayname, num);
```

Parameter

arrayname = Name of an existing array that contains a CAM curve. Dimensions must not be specfied. The array must be a **long** array and it must be declared with only a single dimension. Multi-dimensional arrays are not supported.

num = Curve segment number (for CAM_INDEX_SEGMENT) or start/stop pair number (for CAM_INDEX_STARTSTOP).

Return Value Start index of the requested data block within the array. -1 is returned if the block cannot be found (e.g. the segment does not exist).

> ⚠ If the CAM curve array is corrupt (i.e. it contains an invalid CAM array structure), then the index returned by this function may or may not be correct. It is the user's resposibility to ensure that the array is valid.

Description Some types of applications may require access to the actual internal values of a CAM curve array. For example, it may be necessary for the application to adjust the fixed points dynamically. The **CamArrayIndex** function is designed to allow the application to easily locate the various data blocks within the CAM array structure. The application may then use normal array access to read and modify the attributes of a CAM curve.

This functions is intended to be used with the CAM array index offsets that are defined in "SysDef.mh". See below for an example of how the function may be used.

See CAM Array Structure for a complete description of the structure of CAM curve arrays. See "SysDef.mh" for a complete list of the #defines that may be used to access the array.

Command Group Synchronization Commands

Syntax Examples This example modifies an existing CAM curve by adjusting the master cycle length.

```
// Declare a DIM array long enough for 20 curve segments.
dim long CamCrv[CamArrayLen(20)];

void UpdateCamLength(long delta)
{
    long inf = CamArrayIndex(CAM_INDEX_INFO); // Offset to info block.
    long cnt = CamCrv[inf+CAM_INF_FIXNO];     // Number of fixed points.
    long fix = CamArrayIndex(CAM_INDEX_SEGMENT, cnt-1); // Offset to last fixed point.

    CamCrv[fix+CAM_SEG_MASTER] += delta;      // Adjust the last master position.
    CamCrv[inf+CAM_INF_MCLEN]  += delta;      // Adjust the master cycle length.

    return;
}
```

9.41 CamArrayLen

Summary Calculate the required length of a CAM array.

Syntax
```
res = CamArrayLen(segmax);
res = CamArrayLen(segmax, sspmax);
```

Parameter

segmax = Maximum number of curve segments. This must be an integer constant not less than 2.

sspmax = Maximum number of start/stop pairs. This must be an integer constant not less than 0.

Return Value res = Length of CAM array.

Summary Calculate the required length of a CAM array.

Description This function provides a convenient method of determining the required length of a CAM array. The function parameters must be integer constants and the function will return an integer constant. This allows the function to be used to declare the dimension of arrays that are intended to contain CAM curves. It will return a length that is *at least* long enough to hold the specified maximum number of curve segments and start/stop pairs.

Command Group Synchronization Commands

Syntax Examples

```
// Declare a DIM array long enough for up to
// 20 curve segments and 2 start/stop pairs.
dim long CamCrv[CamArrayLen(20,2)];
```

9.42 CanDelete

Summary Deletes all or single CAN objects.

Syntax `CanDelete(objno);`

Parameter

objno = Object number which is returned during the definition of the object.
-1 deletes all objects (except the standard objects).

Return Value -

Description With the **CanDelete** command, CAN objects which were previously created with DefCanIn or DefCanOut can be deleted.

> See the $N command (see “Communication Reference” in “Chap-IV_EMCA.pdf”, Download manual, www.zub.ch).

Command Group Communication CAN Basic Commands

9.43 CanIn

Summary Reads an object via the CAN bus.

Syntax `status = CanIn(objno, timeout, control, byte03, byte47);`

Parameter

objno = object number which is returned during the definition of the object.

timeout -1 = does not wait for data
0 = waits until data arrives
>0 = waits for the data in *time-out* [ms]

control 0 = Checks whether the new data has arrived. The new data is subsequently copied to the variables.
1 = Sends a remote frame and waits for data in dependence on *time-out*.

byte03 Bytes 0 to 3 of the CAN object data. This must be a simple variable of type "long". Byte order on the bus is swapped. The least-significant byte will be the fourth byte on the bus.

byte47 Bytes 4 to 7 of the CAN object data. This must be a simple variable of type "long". Byte order on the bus is swapped. The least-significant byte will be the eigth byte on the bus.

Return Value status
-1 = no data has arrived
0 = o.k.

Description The **CanIn** command copies the data (if present) of the CAN object 'objno' to the variables 'byte03' and 'byte47'. If 'control = 1', then the data is requested first.

Special Feature It is possible to gather all Transmit-PDOs of digital input modules or CAN-Drive status by using only one CAN telegram. This feature is restricted to the Master-bus. This feature must use the CAN-Object 15 internally which reduces the number of usable CAN objects on the master bus by 1.

Summary Reads an object via the CAN bus.

To enable this feature, the command CanInit 999 must be used

This command enables reception of TxPDOs by interrupt and stores every received PDO in a buffer with a depth of 1. This works for all IDs from 1 to 127. That means if any IO-module in that range sends a TxPDO it is captured and stored it in the buffer. The next TxPDO from the same module of course overwrites the first one. To read out of the buffer, the following command can be used:

```
result = CanIn((id * 100), timeout, 0, byte03, byte47);
```

This command returns -1 if no new information was in the buffer, otherwise it returns 0. You can use the timeout normally (-1 does not wait, 0 waits for ever for new information, n waits n ms).

The result is returned in byte03 and byte47, but we already arrange byte ordering so that they can be used as if you read them with Pdo[]. That means if byte47 contains a 32-bit number then you can use it right away without reordering the bytes.

Whenever you use a new CanInit command or you start a new program, this feature is disabled.

Using this feature of course loads the processor if there is heavy PDO traffic on the BUS.

> It is not recommended that this feature be used together with other CAN PDO commands on the same bus. This may lead to unwanted results.
>
> This is because two CAN objects will exist with the same ID. In that case, the processor only serves the first one.

Command Group Communication CAN Basic Commands

Syntax Example

```
long byte03;                        // Define variables
long byte47;
rx1  = DefCanIn(500, 8);            // a RX object is created
stat = CanIn(rx1, 1000, 0, byte03, byte47);  // wait 1s for data
// byte order on the bus 11 22 33 44 55 66 77 88
// will lead to values 0x11223344 in byte03 and 0x55667788 in byte47
```

Program Sample See **File** → **Sample**: ISR_CanIn.mc

9.44 CanInit

Summary Initializes the necessary objects (PDOs) for data exchange of CANopen nodes or enables extended **CanInit**, CanIn function.

> Direct access to CAN IO devices and CAN drives is no longer supported.
>
> It is strongly recommended to set-up drives and IO modules with help of bus modules (see: Controller Reference) .
>
> Use the **CanInit** command only to collect TxPDO1 telegrams.

Syntax ApossC:

```
CanInit(999);         //Enable objects on CAN1
CanInit(100999);      //Enable objects on CAN2 (deletes objects on CAN1)
CanInit(999, 100999); //Enable objects on both CAN1 and CAN2
CanInit(0);           //Delete objects
```

Parameter

0	Deletes any objects, which were formerly assigned using the **CanInit** command.
999	Enables reception of all TPDO1s (0x181-0x1FF) from Master-bus by interrupt and stores every received PDO in a buffer with a depth of 1.
100999	Enables reception of all TPDO1s (0x181-0x1FF) from Slave-bus by interrupt and stores every received PDO in a buffer with a

Summary Initializes the necessary objects (PDOs) for data exchange of CANopen nodes or enables extended **CanInit**, CanIn function.

> Direct access to CAN IO devices and CAN drives is no longer supported.
>
> It is strongly recommended to set-up drives and IO modules with help of bus modules (see: Controller Reference) .
>
> Use the **CanInit** command only to collect TxPDO1 telegrams.

depth of 1.

Return Value -

An Error 89 can be issued. The INFO value of the error can be one of the following:

- BUSMOD_ERR_NOMORE_MODULES -4: No more bus modules available
- BUSMOD_ERR_COMMFAILED -5: Bus module could not be activated. Reason can be wrong CAN id or baudrate.

Description The **CanInit** command is used to support special features.

It is possible to gather TxPDO1s (0x181, 0x182, ...) on both buses. This can be done by using either **CanInit**(999) for CAN1 or **CanInit**(100999) for CAN2 or both in one statement. The readout of those telegrams work as today by using CanIn with a number of (Node_ID * 100) for CAN1 and (Node_ID + 128)*100 for CAN2.

Command Group Communication CAN Basic Commands

9.45 CanOpenRestart

Summary Initializes the CANopen functionality with the set CAN number (id).

Syntax `CanOpenRestart();`

Parameter -

Return Value -

Description The command starts the CAN Open Slave functionality and initializes the corresponding objects.

Afterwards, a CANopen Master can establish the communication and activate the node via the corresponding NMT messages (simple operating mode) or set variables and objects via SDO commands.

Thus, you can also copy a drives profile (e.g. DSP-402) in the application program.

This command can also be used to re-initalize the CAN bus after the CANBAUD or CANID parameters have changed, without the need of a power cycle.

> ⚠ The CAN bus may not be constantly busy during this command. Otherwise the CAN controller cannot be initialized and an Error 89 will be issued.

Command Group Communication CAN Basic Commands

9.46 CanOut

Summary Sends message with an internal number.

Syntax `CanOut(objno, byte03, byte47);`

Parameter

objno	=	Object number which is returned during the definition of the object (Defcanout command).
byte03	=	Bytes 0 to 3 of the CAN object data. Byte order on the bus is swapped. The least-significant byte will be the fourth byte on the bus.
byte47	=	Bytes 4 to 7 of the CAN object data. Byte order on the bus is swapped. The least-significant byte will be the eighth byte on the bus.

Return Value -

Summary Sends message with an internal number.

Description A CAN message is sent with this command from a sending object defined by DefCanOut. The values *valhi* and *vallo* are 4 bytes long.

The command will issue an Error 89, if a previous message is not fully sent yet. Error 89 can be suppressed by passing a negative (inverted) value for *objno*.

Command Group Communication CAN Basic Commands

Syntax Example

```
byte03 = 0x11223344;
byte47 = 0x55667788;
// The byte order on the bus will be 11 22 33 44 55 66 77 88
tx1    = DefCanOut(500, 8);   // TX object: Id=500, length=8 bytes
CanOut(tx1, byte03, byte47);  // Send CAN message.
CanOut(tx1, byte03, byte47);  // Send next CAN message.
CanDelete(tx1);               // Delete object
```

Program Sample See **File** → **Sample**: ISR_CanIn.mc

9.47 Cpos

Summary Query the current command position of an axis.

Syntax `res = Cpos(axis);`

Parameter axis = Axis module number (see Command Reference).

Return Value res = Current command position relative to the current zero point.

Description The **Cpos** command queries the current command position of the axis relative to the current zero point and returns the value in user units (uu). This may be either a positive or negative value, depending on whether the current position is after, or before, the zero point. See How to Set Up Resolution and Gear Parameters for a description of user units.

> If a temporary zero point has been set using SetOrigin, then the position value is relative to this zero point. See Apos for more information.

The command position is the current target position of the positioning control. It is re-calculated continuously during a movement and is where the positioning control would like the axis to be at that specific time. Compare this with Apos which is the actual position of the axis at that specific time. Normally, these two values will not be the same, but will be very close since the controller is continuously driving the axis from its actual position toward its command position.

There are several reasons why these values may drift apart. For example, if the maximum velocity of the motor is too low, then it may not be able to keep up with the command position. Or if there is a heavy load and the torque is too high, then the motor may also not be able to keep up. If the actual position drifts too far from the command position, then a positioning error (Error 8) is triggered.

Command Group Axis and Information Commands

Program Sample See **File** → **Sample**: Information.mc

9.48 CposDiff

Summary Overflow handling of incremental encoders in applications.

Syntax `res = CposDiff(axis, oldpos);`

Parameter axis = Axis module number (see Command Reference).

oldpos = Cpos of the specified axis at a previous time.

Return Value Returns difference between Cpos and oldpos (i.e. "Cpos(...) - oldpos") in uu.

Description This command simplifies overflow handling of incremental encoders in applications. For example, if the user saves a command position in the program and needs to calculate a difference in position at some later time, then he must normally account for the possibility of an overflow of the position. Instead, this command may be used.

Summary Overflow handling of incremental encoders in applications.

This command will usually return the simple mathematical difference between the old and current command positions (i.e. "Cpos(...) - oldpos"). However, if it is determined that a position overflow occured between the two positions, then the difference is adjusted to account for the overflow so that the correct position difference is returned.

See Encoder Position Values for a description of position overflow.

> This command is designed to work only in the case where the gearing factor is 1 (i.e. 1 uu = 1 qc). See How to Set Up Resolution and Gear Parameters for a description of gearing factors.

Command Group Axis and Information Commands

Syntax Example

```
oldpos = Cpos(1);
...
diff = CposDiff(1, oldpos);
```

9.49 CurvePos

Summary Retrieve slave curve position that corresponds to the current master position of the curve.

Syntax

```
res = CurvePos(axis);
```

Parameter axis = Axis module number (see Command Reference).

Return Value Slave position in CAM units (uu) absolute to the current zero point.

Description **CurvePos** returns the slave value which corresponds to the actual curve master position.

The position can be retrieved independently of the operating status (position control at standstill, positioning procedure, velocity control or synchronization).

CMASTERCPOS and **CurvePos** are now updated even if SyncCam is no longer active. The update of these values starts after a SyncSetCurve command (if SYNCMSTART is < 2000) or after SyncCam and the first master marker (if SYNCMSTART = 2000).

After the SyncCam command is stopped, the update of these values will be continued if SYNCMSTART < 2000.

> The position is only defined if a SyncSetCurve has been set before.

> If a temporary zero point exists which has been set with SetOrigin and is active, the position value will refer to this zero point.

> DefMasterCpos and DefMasterOrigin can still modify this position.

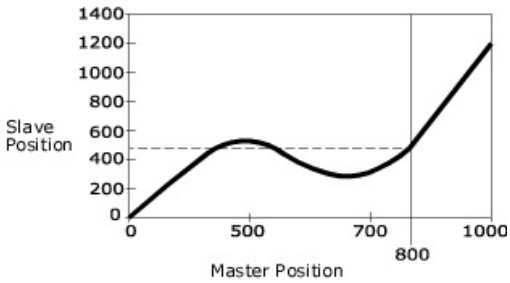
> This command is useful when used in conjunction with AxisPosAbsStart. It will move the slave to the position on the curve that corresponds to the actual master position. For example:
>
> ```
> AxisPosAbsStart(0, CurvePos(0));
> ```

Command Group Axis and Information Commands

Sample Fixed points of a curve:

Master	Slave
0	0
500	500
700	300
1000	1200

Summary Retrieve slave curve position that corresponds to the current master position of the curve.



Say the current master position is 800. Then the **CurvePos** returns the corresponding slave position of 450.

Case 1: Current Master Position is 800 and current slave position is 200.
CurvePos will return the value 450.

Case 2: Current Master Position is 800 and current slave position is 700.
CurvePos will return the value 450.

Hence **CurvePos** is independent of the slave position.

9.50 Cvel

Summary Set the velocity for constant velocity controlled motor movements.

Syntax `Cvel(axis, vel,...);`

Parameter

axis = Axis module number (see Command Reference).

vel = Velocity value (negative value for reversing).
Command Velocity [RPM] = vel * ***Maximum velocity*** VELMAX
Velocity Resolution VELRES

Return Value -

Description The **Cvel** command will set the velocity to be used when the next speed controlled motor movement is started. Speed controlled motor movements are started using the AxisCvelStart command. The velocity value will remain valid until a subsequent **Cvel** command sets a new velocity. If a velocity has not been defined before AxisCvelStart is used to start a speed controller motor movement, then the default velocity will be 0.

If the **Cvel** command is executed after AxisCvelStart has already started speed controlled motor movement, then the speed will be adjusted immediately using the acceleration or deceleration specified by the Acc or Dec command. The new velocity setting will not be postponed to the next speed controlled motor movement.

The specified velocity value will be scaled by the parameters ***Maximum velocity*** VELMAX and ***Velocity Resolution*** VELRES to determine the final command velocity. See Vel for more information.

> ⚠ The **Cvel** command *overwrites* the internal value set by the last Vel command. As a result, the Vel command should be re-executed after speed controlled motor movement has been terminated and before the next motor movement is started.

Command Group Axis and Information Commands

Syntax Examples

```
Cvel(0, 100);        // velocity: axis module 0: 100
Cvel(0, 50, 1, 10);  // velocity: axis mod 0: 50, axis mod 1: 10
Cvel(AXALL, 30);     // velocity: all axes: 30 */
```

Program Sample See **File** → **Sample**: BasicVelocityControl.m

9.51 Dec

Summary Set the deceleration for motion commands.

Syntax `Dec(axis, value,...);`

Summary Set the deceleration for motion commands.

Parameter
axis = Axis module number (see Command Reference).
value = Scaled deceleration value.

Return Value -

Description The **Dec** command defines the deceleration for the next motion command (speed control, synchronization or positioning). The value will remain valid until a new deceleration value is set with another **Dec** command. The value is related to the parameters ***Maximum Acceleration*** RAMPMIN and the ***Maximum Velocity*** VELMAX as well as ***Velocity Resolution*** VELRES.

> If deceleration is not defined previous to the positioning command then deceleration will be the setting (negative) of parameter ***Default Acceleration*** DFLTACC.

Command Group Axis and Information Commands

Example
minimum acceleration time: 1000 ms
maximum velocity: 1500 RPM
velocity resolution: 100

Program Sample See **File** → **Sample**: BasicPositionControl.m

9.52 DefCanIn

Summary Defines a receive object.

Syntax
```
objno = DefCanIn(id, dlen);
```

Parameter
id = CAN-identification number.
dlen = Data length of the object in bytes (max. 8 bytes).

Return Value >0 - Object number to be used with other ApossIDE CAN commands.
<0 - An error has occurred.

Description This command defines an incoming communication object in the CAN-Controller.

This command can also be used with the offset for the slave bus (telegram ID 100000).

So it is possible to define messages for slave and master bus for controllers with two CAN buses. These objects can now be deleted one by one via the command CanDelete objno, where *objno* is the number returned by **DefCanIn** or DefCanOut.

Command Group Communication CAN Basic Commands

Program Sample
```
var1 = 0;                       // Declare variables
var2 = 0;
rx1  = DefCanIn(500, 8);        // RX object with Id=500 and
                                // data length=8 bytes
CanIn(rx1, 0, 0, var1, var2);   // A CAN message is read
```

9.53 DefCanOut

Summary Defines a transmit object in the CAN controller.

Syntax
```
objno = DefCanOut(id, dlen);
```

Parameter
id = CAN-identification number.
dlen = Data length of the object in bytes (max. 8 bytes).

Return Value >0 - Object number to be used with other ApossIDE CAN commands.
<0 - An error has occurred.

Description This command defines an object in the CAN-Controller. This object is an outgoing object with a length of n bytes and the CAN identification of (id). Thus, objno is an internal number of the object Id and is used by the CanOut command.

Summary Defines a transmit object in the CAN controller.

This command can also be used with the offset for the slave bus (telegram ID 100000).

So it is possible to define messages for slave and master bus for controllers with two CAN buses.
These objects will stay defined in the controller, until they are deleted via the command CanDelete objno, where *objno* is the number returned by DefCanIn or **DefCanOut**.
Please note that there is a limited amount of available objects per bus, depending on the controller type. The command will return an error -7 if no object is available anymore.

Command Group Communication CAN Basic Commands

9.54 DefCposOrigin

Summary Set the command position as zero point.

Syntax `DefCposOrigin(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description **DefCposOrigin** sets the command position as zero point. All absolute positioning commands (AxisPosAbsStart etc.) refer to this zero point from now on.

In doing so Cpos is set to zero and Apos is set in that way, that the difference is remained.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Axis and Information Commands

Syntax Example

```
AxisPosAbsStart(0, 80000);  // Absolute positioning
AxisWaitReached(0); // Wait until position is reached
DefCposOrigin(0);   // Define command position as zero point
```

9.55 DefMasterCpos

Summary Define initial position of the master.

Syntax `DefMasterCpos(pos, axis,...);`

Parameter axis = Axis module number (see Command Reference).

pos = Position in Master Units (MU).

Return Value -

Description **DefMasterCpos** defines the initial position of the master (in MU) in the CAM-Mode and thus the point where the curve begins as soon as the master pulses are being counted.

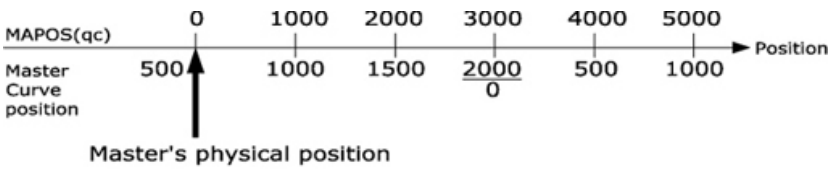
> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Axis and Information Commands

Syntax Example

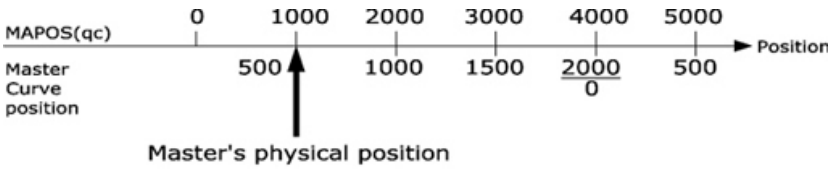
```
DefMasterCpos(1000, 0);   // Set internal MU counter to 1000
DefMasterCpos(1000, 0, 1, 2);
DefMasterCpos(1000, AXALL);
```

Sample **DefMasterCpos** positions Master's current physical position to the master curve position indicated irrespective of what the Mapos is.



Summary Define initial position of the master.

When a **DefMasterCpos** 500 is issued, master's physical position is made as the position 500 of the curve.



When a **DefMasterCpos** 500 is issued, master's physical position is made as the position 500 of the curve.

9.56 DefMasterOrigin

Summary Set the current master position as the zero point for the master.

Syntax `DefMasterOrigin(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description **DefMasterOrigin** defines the current master position as the zero point for the master. The master position (Mapos) refers to this zero point until a redefinition takes place using **DefMasterOrigin** or SetMasterOrigin.

> This command cannot be used with absolute encoders.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Axis and Information Commands

9.57 DefOrigin

Summary Set the current position as zero point.

Syntax `DefOrigin(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description With the **DefOrigin** command the current position is set as the zero point. All absolute positioning commands (AxisPosAbsStart) then refer to this zero point.

The actual position reached in a positioning command is the Target position plus some error which is not compensated automatically while using a **DefOrigin**.

> This command cannot be used with absolute encoders or absolute position counters.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Axis and Information Commands

Syntax Example
```
AxisPosAbsStart(0, 80000); // Absolute positioning
AxisWaitReached(0);  // Wait until target is reached
DefOrigin(0);   // Mod 0: define actual position as zero point
```

Syntax Example
```
axe = 0;
DefOrigin(axe)     // Axis Mod 0: define actual position as zero
DefOrigin(0, 2);   // Axis Mod 0, 2: define real zero point
DefOrigin(AXALL);  // All axes: define real zero point
```

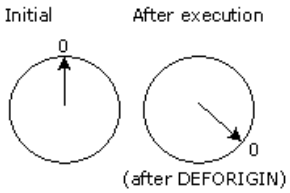
Sample
```
AxisPosAbsStart(0, 2000);
AxisWaitReached(0);  // Wait until target is reached
```

Summary Set the current position as zero point.

```
print("Position before new origin", Apos(0));
DefOrigin(0);
print("Position after defining new origin", Apos(0));
```

Output:

```
Position before new origin 2000
Position after defining new origin 0
```



9.58 Delay

Summary Time delay.

Syntax `Delay(time);`

Parameter time = Time delay in milliseconds (maximum MLONG).

Return Value -

Description The **Delay** command causes the application program to wait a defined program delay. The parameter gives the delay time in milliseconds.

If an interrupt occurs during the delay time, then wait will be continued following the processing of the interrupt procedure. The remaining wait time is still based on the time at which the **Delay** command was originally executed. Thus, the **Delay** command gives a constant delay time, independent of whether various interrupts need to be processed during the programmed delay time.

If an interrupt requires more processing time than remains for the delay, then the interrupt procedure will be executed to completion, the wait is then terminated immediately, and the application program execution continues.

Command Group System Commands

Program Sample See **File** → **Sample**: BasicPositionControl.m

9.59 Delayus

Summary Time delay in microseconds.

Syntax `Delayus(time);`

Parameter time = Time delay in microseconds (maximum MLONG).

Return Value -

Description The **Delayus** command leads to a defined program delay. The parameter gives the delay time in microseconds.

If an interrupt occurs during the delay time, then wait will be continued following the processing of the interrupt procedure. The remaining wait time is still based on the time at which the **Delayus** command was originally executed. Thus, the **Delayus** command gives a constant delay time, independent of whether various interrupts need to be processed during the programmed delay time.

If an interrupt requires more processing time than remains for the delay, then the interrupt procedure will be executed to completion, the wait is then terminated immediately, and the application program execution continues.

Command Group System Commands

9.60 Delete

Summary Delete all arrays in RAM defined by the DIM command.

Syntax `Delete(ARRAYS);`

Parameter -

Return Value -

Description With this command, all arrays in the RAM defined by the DIM command can be deleted without also deleting the parameters. On the other hand, the menu command **Controller** → **Reset** → **Arrays** deletes also DIM arrays saved persistently in flash memory.

> If you then execute a Save command to save arrays, the arrays in the flash memory are also overwritten!

> If Delete(ARRAYS) is carried out after DIM Arrays have been declared in the program, it is then no longer possible to access the DIM array elements.

> If a program contains a Delete(ARRAYS) command, there are no more arrays in the RAM after the program is exited.

Command Group Memory Commands

9.61 DigInput

Summary Reads status of digital input.

Syntax `res = DigInput(n);`

Parameter n = Input number (1, 2, 3, ...).

Return Value input status
0 = Low-level or undefined
1 = High-level

Description The status of a digital input can be read with the **DigInput** command. Depending on the signal level, a 0 or 1 will be given.

> The definition of a high or low level, as well as the input circuit, can be taken from the hardware manual.

> Use the virtual digital input modules to map any source for the digital inputs that are read with the **DigInput** command. This allows to map inputs from an I/O module which is connected over CAN or EtherCAT bus. Hardware inputs from the MACS controller are mapped by default to the virtual digital input module(s).

Command Group Input/Output Commands

Syntax Example
```
in4     = DigInput(4);        // Get the state of input 4
```

Program Sample See **File** → **Sample**: ISR_Input.mc

9.62 DigInputByte

Summary Reads one byte from digital inputs.

Syntax `res = DigInputByte(n);`

Parameter n = Input byte (0, 1, 2, ...).

> Numbering of the bytes begins with 0; this is in contrast to the numbering of the individual inputs, which starts with 1.

Return Value Value of the input byte (0 **...** 255)

The least significant bit corresponds to the condition of input 1.

Description The condition of the digital input can be read with single byte selection via the **DigInputByte** command.

Summary Reads one byte from digital inputs.
The values reflect the condition of the individual inputs.

> The definition of the high and low level, as well as the input circuit, can be taken from the Hardware manual.

> Use the virtual digital input modules to map any source for the digital inputs that are read with the **DigInputByte** command. This allows to map inputs from an I/O module which is connected over CAN or EtherCAT bus. Hardware inputs from the MACS controller are mapped by default to the virtual digital input module(s).

Command Group Input/Output Commands

Syntax Example
```
in     = DigInputByte(0);     // Store the condition of the first 8 inputs
```

Program Sample See **File** → **Sample**: ISR_Input.mc

9.63 DigInputWait

Summary Wait for defined input condition.

Syntax
```
DigInputWait(n, s);
```

Parameter n = input number (1, 2, 3, ...)

s = expected condition:
ON = High-Signal
OFF = Low-Signal

Return Value -

Description The **DigInputWait** command waits before continuing the processing until the specified input has got the desired signal level.

> If the expected input condition does not occur, then the program will remain 'stuck' at this point.

A minimal signal length is required for the sure identification of a signal condition!

Please see the corresponding hardware manual for information about the circuit and technical data for the inputs.

Command Group Input/Output Commands

Syntax Example
```
DigInputWait(4, ON);     // Wait until High level reached input 4
DigInputWait(6, OFF);    // Wait until Low level reached input 6
```

Program Sample See **File** → **Sample**: OtherWaitTypes.m

9.64 DigOutput

Summary Set or reset digital outputs.

Syntax
```
DigOutput(n, s);
```

Parameter n = output number (1, 2, 3, ...)

s = Condition: 0 = OFF, 1 = ON

Return Value -

Description One of the digital outputs can be set or re-set by using the **DigOutput** command.

> If an illegal combination or a pin number is used for X/n, which cannot be set, then Error 71 will be reported.

> After switching on the system, all outputs are OFF.
> These outputs, which have pre-defined functions according to the I/O parameter settings, will also be influenced by the **DigOutput** commands!

The actual output status remains as it is, even after program end or program abort.

Summary Set or reset digital outputs.

The output circuit and maximum load current can be taken from the hardware manual.

> Use the virtual digital output modules to take that proccess value to map it as source for any other modules. This allows to map the output to a bus module and use an I/O module which is connected over CAN or EtherCAT bus. By default the virtual digital output module(s) are mapped to the hardware outputs of a MACS controller.

Command Group Input/Output Commands

Syntax Samples

```
DigOutput(3, 1);          // switch input 3 to GND
DigOutput(6, 0);          // disable output 6
```

Program Sample See **File** → **Sample**: DigitalOutputs.mc

9.65 DigOutputByte

Summary Alteration of the condition of a digital output byte.

Syntax `DigOutputByte(n, value);`

Parameter

n	=	output byte (0, 1, 2, ...)
value	=	Value: 0 to 255

> Numbering of the bytes starts with 0; this is in contrast to the numbering of the individual inputs, which starts with 1.

Return Value -

Description With the **DigOutputByte** command the condition of the digital outputs can be changed byte-by-byte. The byte value transferred determines the condition of the individual outputs. The bit with the lowest value in the byte corresponds to the set condition of output 1.

> After switching on the system, all outputs are OFF. Outputs which have pre-defined functions according to the I/O parameter settings will also be influenced by the **DigOutputByte** command! The actual output status remains as it is even after program end or a program abort.

Output circuit and maximum load current see hardware manual.

> Use the virtual digital output modules to take that proccess value to map it as source for any other modules. This allows to map the output to a bus module and use an I/O module which is connected over CAN or EtherCAT bus. By default the virtual digital output module(s) are mapped to the hardware outputs of a MACS controller.

Command Group Input/Output Commands

Syntax Examples

```
DigOutputByte(0, 10);   // switch through outputs 2 and 4, disable others
DigOutputByte(0, 245);  // disable outputs 2 and 4, switch through others
DigOutputByte(0, 128);  // switch through output 8 only, disable others
```

Program Sample See **File** → **Sample**: DigitalOutputs.mc

9.66 DimArray

Summary Pass DIM Arrays to functions by referencing the array number of the DIM array rather than the array name.

Syntax `func(..., DimArray(arrnum),...);`

Parameter

arrnum	=	Array number of DIM array. DIM array numbers start at 0.

Return Value -

Description In certain cases, it is useful to be able to pass DIM Arrays to functions by referencing the array number of the DIM array

Summary Pass DIM Arrays to functions by referencing the array number of the DIM array rather than the array name.

rather than the array name. For example, an application program may contain multiple CAM arrays that are saved in different DIM arrays. An individual CAM array can then be chosen by index and passed to a function for processing. This is done using the **DimArray** keyword as is shown in the following example.

> "arraynum" must NEVER specify a number larger than the last DIM array declared in the application program.

> The **DimArray** keyword may *only* be used in the context of a function call and it may *only* be used for DIM arrays of type **long**.

Command Group Memory

Syntax Example

```
dim long ArrayA[50], ArrayB[50], ArrayC[50];

long ProcessArray(long arr[]);

long i;

for (i = 0; i < 3; i++)   // Process each DIM array.
    ProcessArray(DimArray(i));
```

9.67 DimArrayNum

Summary Get DIM Arrays number from the DIM array name.

Syntax `DimArrayNum(arrname);`

Parameter arrname = Name of DIM array.

Return Value DIM array number. DIM array numbers start at 0.

Description DIM Arrays are stored in the same consecutive order in memory as they are defined. In certain cases, it can useful to be able to get a sequence number of a certain DIM array only by knowing its name.

Command Group Memory

Syntax Example

```
dim long ArrayA[50], ArrayB[50], ArrayC[50];

long i;

i = DimArrayNum(ArrayC); // Value of i will be 2
```

9.68 ECatMasterCommand

Summary Controls the EtherCAT master.

Syntax `res = ECatMasterCommand(nodeid, command);`

Parameter nodeid = 0: All slaves
n: Slave number 'n' (starting at 1)
0x1000: The master itself

Slave Commands

command = 0x01: EC_STATE_INIT
0x02: EC_STATE_PRE_OP
0x03: EC_STATE_BOOT
0x04: EC_STATE_SAFE_OP
0x08: EC_STATE_OPERATIONAL
0x10: Recover slave after it was reconnected

Summary Controls the EtherCAT master.

>= 0x100
0xuuuuvv00: Start Sync0 Signal with cycletime of vv milliseconds, with shift of uuuu microseconds.

This command has to be called for all slaves that use DC. The first slave this command is applied to, will be the reference clock in the system. Typically, cycletime is 1ms, and shift is 0.

Master Commands

command = 0: Disable EtherCAT bus

1: Start and scan EtherCAT bus. Slaves will be in PREOP state after this command, so that PDOs can be set up on slaves.

2: Map Input and Output data (PDOs on slaves must be set up before calling this command)

3: Put all Slaves to OP State and start process data transfer

4: Put all Slaves to SAFEOP State

5: Clear error on slaves

6: Scan states of the slaves. Checks which slaves are disconnected. Can be called after a WC error, to acknowledge.

Return Value Error code:

0 = ok

-1: Link could not be established
-2: Slaves have not reached PREOP state
-3: Slaves could not be enumerated and configured
-4: Distributed Clocks could not be configured
-5: Input process data too big
-6: Output process data too big
-7: EtherCat master not running
-8: Slaves have not reached OP state
-9: Slaves have not reache SAFEOP state
-10: At least one slaves error could not be cleared
-100: Slave was not found
-200: Slave is a MACS but has not booted yet
-300: Slave is not the expected one

Description This command controls the EtherCAT master. Some command are for the master itself (see 0x1000 for the parameter nodeid). When the parameter nodeid contains a slave number the command is sent from the master to the slave.

See ApossC SDK for examples.

Typical startup sequence (simplified)

```
ECatMasterCommand(0x1000, 1);    // start Master
slaveCount = ECatMasterInfo(0x1000, 0);
//Here setup slave pdo mapping by SdoWrite commands
ECatMasterCommand(0x1000, 2); //  Map Input and Output buffers (go to safeop)
ECatMasterCommand(slaveNo, ( (cycleTime_ms<<8) + (offset_us<<16)));   //setup DC for all slaves supporting it
ECatMasterCommand(0x1000, 3); //  request & wait OP state for all slaves
```

Summary Controls the EtherCAT master.

Portability This feature can be controller dependent and is listed in the Controller Dependent Feature List.

Command Group Communication EtherCAT Master Commands

9.69 ECatMasterConfig

0 - Bus cycle time in microseconds.
1 - Target DIM array to which to write the incoming process data (1 = first array).
2 - Set startup delay time in microseconds (necessary for VACON drive, value = 100000).

Summary Configure an EtherCAT master.

Syntax
```
res = ECatMasterConfig(0x1000, select, value);
```

Parameter

select = Parameter to be set:

0: Bus cycle time in microseconds.
1: Target DIM array to which to write the incoming process data (1 = first array).
2: Set startup delay time in microseconds (necessary for VACON drive, value = 100000).

value = Parameter value to be set.

Return Value Error code:

0: Okay
-1: Selected value does not exist

Description This command will set the specified EtherCAT Master configuration parameter.

> The second (deprecated) form of the command can be used to assign a bus module to an EtherCAT slave (A), or to configure the master itself (B).
>
> By standard, the mapping parameters of bus modules apply to the very beginning of the whole IO Image. This makes it quite complicated to set up the mapping, as the offset for each slave must be handled very carefully.
>
> This command simplifies the setup, as it sets a process data offset value, so that the process data of the bus module starts at the process data section of the selected slave, not at the very beginning of the complete process data image. This makes it more comfortable to set up the bus modules mapping data.

Portability This feature can be controller dependent and is listed in the Controller Dependent Feature List.

Command Group Communication EtherCAT Master Commands

9.70 ECatMasterInfo

Summary This command can be used to read details about the connected slaves or the master itself.

Syntax
```
status = ECatMasterInfo(nodeid, select, value);
```

Parameter

nodeid = Slave number (starting at 1), 0x1000 = master.

select = Which value shall be read.

value = Variable of type long, which will contain the value of the requested information after execution of the command.

Return Value status: 0 = ok, negative values indicate an error.

Description List of readable slave information:

0	state	Slave state EC_STATE_INIT 0x01

Summary This command can be used to read details about the connected slaves or the master itself.

		EC_STATE_PRE_OP 0x02 EC_STATE_BOOT 0x03 EC_STATE_SAFE_OP 0x04 EC_STATE_OPERATIONAL 0x08 EC_STATE_ACK 0x10 EC_STATE_ERROR 0x10
1	Alstatuscode	AL status code
2	mbx_cnt	Counter value of mailbox link layer protocol (1..7).
3	eep_pdi	false = EEPROM to master, true = EEPROM to PDI
4	configadr	Configured address
5	aliasadr	Alias address
6	eep_man	Manufacturer from EEPROM
7	eep_id	ID from EEPROM
8	eep_rev	Revision from EEPROM
9	eep_8byte	EEPROM read length: false = 4 bytes per read, true = 8 bytes per read
10	blockLRW	Block use of LRW in ec_send_processdata()
11	mbx_wl	Length of write mailbox in bytes (0 if no mailbox)
12	mbx_wo	Mailbox write offset
13	mbx_rl	Length of read mailbox in bytes (0 if no mailbox)
14	mbx_ro	Mailbox read offset
15	mbx_proto	Supported mailbox protocols (0 if no mailbox)
		ECT_MBXPROT_AOE 0x0001 // ADS over ECAT
		ECT_MBXPROT_EOE 0x0002 // Ethernet over ECAT
		ECT_MBXPROT_COE 0x0004 // CANopen over ECAT
		ECT_MBXPROT_FOE 0x0008 // File over ECAT
		ECT_MBXPROT_SOE 0x0010 // Servo over ECAT
		ECT_MBXPROT_VOE 0x0020 // Vendor over ECAT
16	preconfigured	SM configuration given to Master (i.e. not read from EEPROM)
17	CoEdetails	CoE details
		ECT_COEDET_SDO 0x01
		ECT_COEDET_SDOINFO 0x02
		ECT_COEDET_PDOASSIGN 0x04
		ECT_COEDET_PDOCONFIG 0x08
		ECT_COEDET_UPLOAD 0x10
		ECT_COEDET_SDOCA 0x20

Summary This command can be used to read details about the connected slaves or the master itself.

18	FoEdetails	FoE details
19	EoEdetails	EoE details
20	SoEdetails	SoE details
21	Output.bits	Bit count
22	Output.Bytes	Byte count (bit count rounded up)
23	Output.Offset	Offset (in bytes) of buffer in user's IO buffer
24	Output.StartBit	Startbit in first byte (0 -> 7)
25	Input.Bits	Bit count
26	Input.Bytes	Byte count (bit count rounded up)
27	Input.Offset	Offset (in bytes) of buffer in user's IO buffer
28	Input.StartBit	Startbit in first byte (0 -> 7)
29	FMMUcnt	Number of used FMMU's
30	hasdc	Has DC capabillity?
99	ec_slavecount	Number of slaves found
1xy	x : Sync Manager number y = 0 : SMtype	ECT_MBXTYPE_UNKNOWN 0 ECT_MBXTYPE_WRITE 1 ECT_MBXTYPE_READ 2 ECT_MBXTYPE_OUTPUT 3 // PDO output ECT_MBXTYPE_INPUT 4 // PDO input
	y = 1 : SM.StartAddr y = 2 : SM.SMlength y = 3 : SM.SMflags	Start address (0 if unused) Length (in bytes)
1xxxx	Read slave EEPROM cell xxxx, 8bits	See description of EtherCAT ASIC memory
2xxxx	Read slave EEPROM cell xxxx, 16bits	See description of EtherCAT ASIC memory

List of readable master information:

0	ec_slavecount	Number of slaves found on bus
1	inputsLength	Number of input bytes in the IO Image
2	outputsLength	Number of output bytes in the IO Image
12	ec_systemTimeDiff	Offset time [ns] between Master and Reference clock
17	link state	0 = EtherCAT connection loss, 1 = EtherCAT connection established
22	SDO Abort Code	0x00000000: No error 0x05030000: Toggle bit not changed 0x05040000: SDO protocol timeout 0x05040001: Client/Server command specifier not valid or

Summary This command can be used to read details about the connected slaves or the master itself.

	unknown 0x05040005: Out of memory 0x06010000: Unsupported access to an object 0x06010001: Attempt to read to a write only object 0x06010002: Attempt to write to a read only object 0x06020000: The object does not exist in the object directory 0x06040041: The object can not be mapped into the PDO 0x06040042: The number and length of the objects to be mapped would exceed the PDO length 0x06040043: General parameter incompatibility reason 0x06040047: General internal incompatibility in the device 0x06060000: Access failed due to a hardware error 0x06070010: Data type does not match: length of service parameter does not match 0x06070012: Data type does not match: length of service parameter too high 0x06070013: Data type does not match: length of service parameter too low 0x06090011: Subindex does not exist 0x06090030: Value range of parameter exceeded (only for write access) 0x06090031: Value of parameter written too high 0x06090032: Value of parameter written too low 0x06090036: Maximum value is less than minimum value 0x08000000: General error 0x08000020: Data cannot be transferred or stored to the application 0x08000021: Data cannot be transferred or stored to the application because of local control 0x08000022: Data cannot be transferred or stored to the application because of the present device state 0x08000023: Object dictionary dynamic generation fails or no object dictionary is present

Portability This feature can be controller dependent and is listed in the Controller Dependent Feature List.

Command Group Communication EtherCAT Master Commands

9.71 ErrorAxis

Summary Determine the axis causing the last error.

Syntax
```
res = ErrorAxis();
```

Return Value Axis causing the last error. If 0xFF, last error is not related to any axis, as for example in case of F_UMIN, F_UMAX or F_TEMP.

Description This command will return the 0-based axis module number of the axis causing the last error.

Command Group System Commands

Syntax Example
```
module = ErrorAxis();
print("Error was caused by axis module: ",module);
```

9.72 ErrorClear

Summary Clear the last error.

Syntax `ErrorClear();`

Parameter -

Return Value -

Description Error alarms are not automatically erased after elimination of the cause of error, but must be re-set via an **ErrorClear** command. However, the cause of the error must be eliminated first; otherwise the same error alarm will occur again. If, in the meantime, another un-corrected error occurs, then only the first error will be canceled.

> The **ErrorClear** command should only be used in an ERROR interrupt routine.

> Only error will be cleared; motors are not automatically turned on again (see AxisControl).

Command Group System Commands

Program Sample See **File** → **Sample**: ISR_Error.mc

9.73 ErrorInfo

Summary Query additional information about the current error.

Syntax `res = ErrorInfo();`

Parameter -

Return Value Error info value

Description **ErrorInfo** is a system variable which is available in all programs and which indicates additional information about the current error.

Refer to the specific error article for the meaning of the value (see Error Messages).

Command Group System Commands

9.74 ErrorNo

Summary Query the actual error code.

Syntax `res = ErrorNo();`

Parameter -

Return Value Error number of last error.

Description **ErrorNo** contains the error code of the last error. This will be 0 if there is no active error. All error codes are explained in the chapter Error Messages.

Command Group System Commands

Program Sample See **File** → **Sample**: ISR_Error.mc

9.75 EthernetClose

Summary Closes the socket connection.

Syntax `EthernetClose(handle);`

Parameter handle = The handle returned by the EthernetOpenClient or EthernetOpenServer command.

Description This command is used to close the socket connection.

Command Group Ethernet Commands

9.76 EthernetGetConnectionStatus

Summary Returns the current ethernet connection status.

Syntax `status = EthernetGetConnectionStatus(handle);`

Parameter

handle = The handle returned by the EthernetOpenClient or EthernetOpenServer command.

Return Value Status of the ethernet connection.

SOCK_STATUS_INIT = 0,

SOCK_STATUS_WAITING = 1,

SOCK_STATUS_CONNECTING = 2,

SOCK_STATUS_READY = 3,

SOCK_STATUS_CLOSED = 4,

SOCK_STATUS_ERRORSENDING = -1,

SOCK_STATUS_ERROR = -2

Description This command can be used to check the status of the ethernet socket connection.

Command Group Ethernet Commands

9.77 EthernetOpenClient

Summary Opens a socket connection as client to a server

Syntax `handle = EthernetOpenClient(protocolType, ipAddress, portNumber);`

Parameter

protocolType = PROT_TCP or PROT_UDP

ipAddress = IP address of the server socket.

portNumber = Port of the server socket.

Return Value In case of success, a positvie value, which is the handle of the connection. This can be used in subsequent calls referencing current connection.

In case of error, one of the error return values, See list in Ethernet Commands.

Description This command can be used to open the connection to the server socket.

Command Group Ethernet Commands

Syntax Example

```
long socketHandle;
socketHandle = EthernetOpenClient(PROT_TCP, 192.168.1.22, 90);
if(socketHandle < 0) printf("There was an error: %ld \r\n", socketHandle);
else printf("Success. The handle is: %ld \r\n", socketHandle);
```

9.78 EthernetOpenServer

Summary Opens a socket server.

Syntax `handle = EthernetOpenServer(protocolType, portNumber);`

Parameter

protocolType = PROT_TCP or PROT_UDP

portNumber = Port which will be used for server socket.

Return Value In case of success, a positvie value, which is the handle of the connection. This can be used in subsequent calls referencing current connection.

In case of error, one of the error return values, See list in Ethernet Commands.

Summary Opens a socket server.

Description This command can be used to open the server socket and wait for connection from client.

Command Group Ethernet Commands

Syntax Example

```
long socketHandle;
socketHandle = EthernetOpenServer(PROT_TCP, 90);
if(socketHandle < 0) printf("There was an error: %ld \r\n", socketHandle);
else printf("Success. The handle is: %ld \r\n", socketHandle);
```

9.79 EthernetReceiveTelegram

Summary Receives a telegram.

Syntax `result = EthernetReceiveTelegram(handle, array);`

Parameter

handle	=	The handle returned by the EthernetOpenClient or EthernetOpenServer command.
array	=	Array which is used to receive the incoming user data, 1 received byte per array element. This must be a **long** or **wchar** array. Note that array indices may not be specified. Received data will be placed into the array, starting at the first element in the array.

Return Value The return value will be the amount of data received.

Description This command can be used to receive a telegram if there is one in the buffer.
The socket connection previously has to be setup with EthernetOpenClient or EthernetOpenServer. The connection status has to be SOCK_STATUS_READY = 3.

Maximum telegram length see Controller Dependent Feature List

Command Group Ethernet Commands

Syntax Example

```
long receiveData[50];
wchar charArray[50];
long retVal;
retVal = EthernetReceiveTelegram(handle, receiveData);
printf("first byte: %ld, second byte: %ld", receiveData[0], receiveData[1]);
retVal = EthernetReceiveTelegram(handle, charArray);
print(charArray); //print the whole array
```

9.80 EthernetSendTelegram

Summary Sends a telegram.

Syntax `result = EthernetSendTelegram(handle, array, length);`

Parameter

handle	=	The handle returned by the EthernetOpenClient or EthernetOpenServer command.
array	=	Array which contains the data to be sent, 1 byte per array element. This must be a **long** or **wchar** array. Note that array indices may not be specified; data will be sent, starting at the first element in the array.
length	=	Number of bytes to send. Maximum telegram length see Controller Dependent Feature List

Return Value One of the return values, See list in Ethernet Commands.

Summary Sends a telegram.

Description This command can be used to send a telegram over an open socket connection. The socket connection previously has to be setup with EthernetOpenClient or EthernetOpenServer. The connection status has to be SOCK_STATUS_READY = 3.

Command Group Ethernet Commands

Syntax Example

```
long sendData[50];
wchar charArray[50] = "hello world";
long retVal;
sendData[0] = 50;
sendData[1] = 100;
retVal = EthernetSendTelegram(handle, sendData, 2);
Delay(500);
retVal = EthernetSendTelegram(handle, charArray, arraylen(charArray));
```

9.81 Exit

Summary Premature program termination.

Syntax `Exit(retcod);`

Parameter retcod = Return code. This is currently discarded.

Description The **Exit** command ends a program where active positioning procedures are being carried out to the end. The **Exit** command is especially intended for use in an error handling routine, and permits an unplanned program termination in the case of an uncorrectable error occurrence.

After an abort with **Exit**, programs marked with **Autostart** will start up again automatically if PRGPAR = -1.

9.82 HallDetectStart

Summary Starts the detection of the hall sensor alignment.

Syntax `HallDetectStart(axis, parm);`

Parameter axis = Axis module number (see Command Reference).

parm = mode of alignment (only value 2 is used).

Return Value -

Description The purpose of this command is to find the hall sensor alignment for a BLDC motor. All motor phases are energized in a sequence and the hall signals are read to find the correct pattern. This command is helpful when the hall pattern of the motor is unknown. An encoder has to be connected and initialized as well.

Subsequently, a parameter can be set without running this alignment process. Note: The return value obtained by HallDetectStatus has to be decremented by one:

HWAMP_PARAM(axis, HWAMP_HALL_ALIGNMENT) = foundHallAllignment

> This command can only be performed on a MACS controller with integrated amplifiers. The axis must be setup as one of the following HWAMP_COMMTYPE:
>
> **HWAMP_COMMTYPE_BLDC**
>
> **HWAMP_COMMTYPE_BLDC_TWIN**
>
> **HWAMP_COMMTYPE_HALL_PMSM**
>
> **HWAMP_COMMTYPE_HALL_PMSM_TWIN**
>
> If a different HWAMP_COMMTYPE is assigned, error 101 will be raised.

Summary Starts the detection of the hall sensor alignment.

> ⚠ In order for this procedure to work, the motor must not be braked or be in a limit position. The rotor must be able to rotate freely for about half a revolution.
>
> Note also that a **HALL sensor** and an **encoder** has to be connected.

Command Group Amplifier Commands

Syntax Example

```
print("AxisNo ", AxisNo, " Start Hall Detection");
HallDetectStart(AxisNo, 2);
while((retVal = HallDetectStatus(AxisNo)) == 0)
{
    print("detecting... ", retVal);
    Delay(500);
}
if(retVal < 0)
    print("!!!! Hall detection FAILED !!!! retVal: ", retVal);
else
    print("Hall detection completed with ", retVal);
```

9.83 HallDetectStatus

Summary Retruns the Hall sensor detection status.

Syntax `res = HallDetectStatus(axis);`

Parameter axis = Axis module number (see Command Reference).

Return Value res = Current status

>=1: Process successful. Number represents the found hall pattern (see below).

1: HALL_ALIGNMENT_360120240 (120° CCW: H1 at 360°, H2 at 120°, H3 at 240°)

2: HALL_ALIGNMENT_060180300 (120° CCW: H1 at 060°, H2 at 180°, H3 at 300°)

3: HALL_ALIGNMENT_120240360 (120° CCW: H1 at 120°, H2 at 240°, H3 at 360°)

4: HALL_ALIGNMENT_180300060 (120° CCW: H1 at 180°, H2 at 300°, H3 at 060°)

5: HALL_ALIGNMENT_240360120 (120° CCW: H1 at 240°, H2 at 360°, H3 at 120°)

6: HALL_ALIGNMENT_300060180 (120° CCW: H1 at 300°, H2 at 060°, H3 at 180°)

6: HALL_ALIGNMENT_120DEGREE (120° standard: same as HALL_ALIGNMENT_300060180

7: HALL_ALIGNMENT_360240120 (120° CW: H1 at 360°, H2 at 240°, H3 at 120°)

8: HALL_ALIGNMENT_060300180 (120° CW: H1 at 060°, H2 at 300°, H3 at 180°)

9: HALL_ALIGNMENT_120360240 (120° CW: H1 at 120°, H2 at 360°, H3 at 240°)

10: HALL_ALIGNMENT_180060300 (120° CW: H1 at 180°, H2 at 060°, H3 at 300°)

11: HALL_ALIGNMENT_240120360 (120° CW: H1 at 240°, H2 at 120°, H3 at

Summary Retruns the Hall sensor detection status.

360°)

12: HALL_ALIGNMENT_300180060 (120° CW: H1 at 300°, H2 at 180°, H3 at 060°)

13: HALL_ALIGNMENT_360060120 (60° CCW: H1 at 360°, H2 at 060°, H3 at 120°)

14: HALL_ALIGNMENT_060120180 (60° CCW: H1 at 060°, H2 at 120°, H3 at 180°)

15: HALL_ALIGNMENT_120180240 (60° CCW: H1 at 120°, H2 at 180°, H3 at 240°)

16: HALL_ALIGNMENT_180240300 (60° CCW: H1 at 180°, H2 at 240°, H3 at 300°)

17: HALL_ALIGNMENT_240300360 (60° CCW: H1 at 240°, H2 at 300°, H3 at 360°)

18: HALL_ALIGNMENT_300360060 (60° CCW: H1 at 300°, H2 at 360°, H3 at 060°)

19: HALL_ALIGNMENT_360120060 (60° CW: H1 at 360°, H2 at 120°, H3 at 060°)

20: HALL_ALIGNMENT_060180120 (60° CW: H1 at 060°, H2 at 180°, H3 at 120°)

21: HALL_ALIGNMENT_120240180 (60° CW: H1 at 120°, H2 at 240°, H3 at 180°)

22: HALL_ALIGNMENT_180300240 (60° CW: H1 at 180°, H2 at 300°, H3 at 240°)

23: HALL_ALIGNMENT_240360300 (60° CW: H1 at 240°, H2 at 360°, H3 at 300°)

23: HALL_ALIGNMENT_060DEGREE (60° standard: same as HALL_ALIGNMENT_240360300)

24: HALL_ALIGNMENT_300060360 (60° CW: H1 at 300°, H2 at 060°, H3 at 360°)

0: Process is active

-1: Error, procedure aborted

-2: Error, unknown hall pattern. Possible reason for this error can be a disconnected or partly disconnected HALL sensor or encoder.

-3: Error, irregular motion

-4: Error, illegal state sequence

-5: Error, imprecise motion

-6: Error, illegal parameter

-10: Error, timeout

Description The purpose of this command is to query the current status of the hall detection routine. This command makes only sense after the HallDetectStart command.

Note: A valid return value (res >= 1) has to be decremented by 1 for assignment to HWAMP_HALL_ALIGNMENT.

Example:

```
HWAMP(AxisNo, HWAMP_HALL_ALIGNMENT) = res-1
```

Summary Retruns the Hall sensor detection status.
Better for readability. The return code of HallDetectStatus() was in this case res==3:

```
HWAMP(AxisNo, HWAMP_HALL_ALIGNMENT) = HALL_ALIGNMENT_120240360
```

Command Group Amplifier Commands

Syntax Example

```
print("AxisNo ", AxisNo, " Start Hall Detection");
HallDetectStart(AxisNo, 2);
while((retVal = HallDetectStatus(AxisNo)) == 0)
{
    print("detecting... ", retVal);
    Delay(500);
}
if(retVal < 0)
    print("!!!! Hall detection FAILED !!!! retVal: ", retVal);
else
    print("Hall detection completed with ", retVal);
```

9.84 InterruptDelete

Summary Delete an interrupt handler function.

Syntax `InterruptDelete(...);`

Parameter Parameters are specific to the interrupt being deleted.
See different interrupt that are available to setup:
ERROR
CANINPUT
DIGINPUT
SERIALCHAR
PARAM
PERIOD
Position (all position interrupts will be disabled together)
STATBIT

Return Value -

Description The InterruptDelete function allows an interrupt handler to be deleted.

See Interrupt Handling for a complete description of interrupt handling and the interrupt types available.

Command Group Interrupt Functions

9.85 InterruptDisable

Summary Locks the execution of an interrupt type or deletes all definitions valid for a type of interrupt completely (reset).

Syntax `InterruptDisable(inttyp);`

> This command cannot be called during interrupt procedures. (The system automatically switches back to enable after an interrupt.)

Parameter inttyp = Type of interrupt to disable:

ALL
CANINPUT
DIGINPUT
SERIALCHAR
PARAM
PERIOD

Summary Locks the execution of an interrupt type or deletes all definitions valid for a type of interrupt completely (reset).

Position (all position interrupts will be disabled together)
STATBIT

> Execution of the ERROR interrupt cannot be disabled. The error interrupt has the highest priority and it also interrupts other active interrupts.

> Disabling PERIOD interrupts will disable *both* PERIOD and TIME interrupts. These interrupts cannot be disabled independently.

Return Value -

Description **InterruptDisable** switches off all interrupts or explicitly specified interrupts, with the exception of ERROR interrupts. If **InterruptDisable** is used in the main program, it can prevent interrupts of the corresponding type.

This is particularly useful if a variable, which is set in an interrupt procedure, is used in the main program. To do this you should first switch off the corresponding (or all) interrupts in the main program using **InterruptDisable**, alter the variable, and then switch the corresponding (or all) interrupts back on with InterruptEnable.

> If an interrupt is disabled, then it still exists but is not processed anymore.

The detection is still running in the background and the interrupt is captured in the case of a *non* edge-sensitive or a message-oriented interrupt (PERIOD, Position, PARAM, etc.). If the interrupt is enabled again and there is a pending (non edge-sensitive) interrupt, then this interrupt is processed immediately.

In the case of edge-sensitive interrupts (e.g. DIGINPUT, STATBIT), all interrupts that occur while interrupts are disabled, are discarded. They will not be precessed even after interrupts are enabled again. Edge-sensitive interrupts which occur after interrupts are enabled again, will be processed as normal.

Exception: **InterruptDisable**(ALL)

In case of **InterruptDisable**(ALL) the request is stored (edge-sensitive interrupts too) and the interrupt are still executed after enabling (InterruptEnable(ALL) ! Note that the selective disabling of edge-sensitive interrupts (e.g. InterruptDisable(DIGINPUT)) - these will be ignored and not executed after enabling.

InterruptDisable(ALL) in combination with selective **InterruptDisable**(..)

An InterruptEnable(ALL) has no impact on simultaneous valid blockings defined by selective **InterruptDisable** commands (e.g. InterruptDisable(DIGINPUT)). Thus a selective blocking must also be cleared by the corresponding selective InterruptEnable!

Interrupt handling within an Interrupt routine

During execution of an interrupt routine a **InterruptDisable**(ALL) is automatically done internally. This blocks the execution of all other interrupts, but keeps upcoming interrupt requests in a queue. At the end of the 'current' interrupt routine an InterruptEnable(ALL) will be executed automatically. With the completion of the 'current' interrupt the next interrupt in the queue will be executed. Therefore the commands **InterruptDisable**(ALL) and InterruptEnable(ALL) within an interrupt is not necessary.

The selective blocking of single interrupts within an interrupt subroutine can be necessary, depending on the application. Think of the following example: If the execution of an interrupt should lock the request and execution of other interrupt types, a selective **InterruptDisable** (e.g. **InterruptDisable**(DIGINPUT)) can be done. In this case the selective interrupt blockage must be cleared (e.g. InterruptEnable(DIGINPUT)) by the application program later on again. Typically this is done at the end of the current interrupt subroutine and enables the execution of corresponding interrupt requests in future again. All edge triggered interrupts, which were received between the corresponding selective **InterruptDisable** and InterruptEnable, will be ignored and not executed any longer (nor later). All interrupts, which were received before the selective blocking (e.g. **InterruptDisable**(DIGINPUT)) or after the new selective release (e.g. InterruptEnable(DIGINPUT)) will be processed after the completion of the "first" interrupt.

Command Group Interrupt Functions

Syntax Examples

```
InterruptDisable(ALL);      // Disable all interrupts.
InterruptDisable(STATBIT);  // Disable status bit interrupt.
```

9.86 InterruptEnable

Summary Enables interrupts.

Syntax `InterruptEnable(inttyp);`

Parameter inttyp = Type of interrupt to enable:

ALL
CANINPUT
DIGINPUT
SERIALCHAR
PARAM
PERIOD
Position (all position interrupts will be disabled together)
STATBIT

> Enabling PERIOD interrupts will enable *both* PERIOD and TIME interrupts. These interrupts cannot be enabled independently.

Return Value -

Description **InterruptEnable** switches all interrupts, or explicitly specified interrupts, on again.

> Please see the InterruptDisable command for more details about disabling interrupts and how pending interrupts are handled after interrupts are enabled again.

Command Group Interrupt Functions

Syntax Examples

```
InterruptEnable(ALL);      // Switch on all interrupts
InterruptEnable(STATBIT);   // Switch on the interrupt for
                           // the status bit
```

9.87 InterruptSetup

ETHERNET STATBIT

Summary Set up an interrupt handler function.

Syntax `InterruptSetup(...);`

Parameter Parameters are specific to the interrupt being set up.
See different interrupt that are available to setup:
ERROR
CANINPUT
DIGINPUT
SERIALCHAR
PARAM
PERIOD
Position (all position interrupts will be disabled together)
STATBIT
ETHERNET

Return Value -

Description The **InterruptSetup** function allows a function to be set up as an interrupt handler.

See Interrupt Handling for a complete description of interrupt handling and the interrupt types available.

See Interrupt Routines for a description of how to write interrupt handler routines.

> Interrupt handlers should be set up at the beginning of the program so that they are valid for the entire program.

Command Interrupt Functions

Group

9.88 Ipos

Summary Query the last index or marker position of the slave.

Syntax `res = Ipos(axis);`

Parameter axis = Axis module number (see Command Reference).

Return Value Last slave position (index or marker) absolute to actual zero point.

Description The **Ipos** command returns the last index or marker position of the slave absolute to the current zero point.

The position input is made in user units uuand corresponds in the standard setting to the number of quad-counts:
Parameter POSFACT_Z / POSFACT_N = 1

> If a temporary zero point, set and activated via SetOrigin, exists, then the position is respective to this zero point.

The configuration of **Ipos**, that is whether the slave index- or marker position (= controlled drive) is returned, is done via the parameter in the latch module.

> Many MACS controller have hardware latch modules and incremental encoder interfaces for very accurate position latch information. It is also possible to use the latch functionality of a drive (if supported) and get the position over a bus (e.g. CAN or EtherCAT). Check the virtual latch module and bus modules for the correct settings. Also a software latching (accuracy of typically 1ms) is possible.

The position value in **Ipos** is accurate to ±1 qc. In opposite to the position information in Apos, which is just updated in a controller cycle of typically 1 ms, the actual position value is hardware stored in real time a buffer (in an internal processor register), when the configured signal is high. Then it will be copied in the system variable **Ipos**.

If simultaneously to the marker position, an interrupt is initiated (DIGINPUT) and **Ipos** is executed within this interrupt, then a delay of 2 milliseconds (Delay) should be executed *before* executing **Ipos**. This will ensure that the latched position value is already completely updated in the system, preventing an old value from being returned.

> The command **Ipos** returns only accurate position latching information if a incremental encoder and a built in hardware latch module is used.

See also sample.

Command Group Axis and Information Commands

Sample

```
// Definition of IPOS latching on positive edge at input 2

// Setup the the hardware latch module (virtual latch module uses default settings)
HWLATCH_PARAM(0, HWLATCH_PISRC_TRIGGER) = 0x140 + 1; //digital input 2
HWLATCH_PARAM(0, HWLATCH_SLOPE) = 1; //rising slope

// Setup a Interrupt on rising edge of digital input 2
InterruptSetup(DIGINPUT, SlaveInt, 2, RISING);
Cvel(0, 10);     // Start moving
AxisCvelStart(0);
while (1);       // Endless-Loop
...

void SlaveInt(void)
{
    long int_pos, triggered_pos;

    // Latching APOS for testing, how exact it would be ...
    int_pos = Apos(0);
    // Wait 2 ms to be sure that Ipos is correctly updated
    Delay(2);
```

Summary Query the last index or marker position of the slave.

```
    // Latching Ipos for a later handling etc.
    triggered_pos = Ipos(0);
    // ...
    // ...
    print("Interrupt position: ",int_pos);
    print("Triggered position: ",triggered_pos);
    return;
}
```

9.89 IposDiff

Summary Overflow handling of incremental encoders in applications.

Syntax `res = IposDiff(axis, oldpos);`

Parameter

axis = Axis module number (see Command Reference).

oldpos = Ipos value at a previous time.

Return Value Difference between Ipos and oldpos (res = Ipos - oldpos) in uu.

Description This command simplifies overflow handling of incremental encoders in applications. For example, if the user stores an actual position in the program and needs to calculate the difference at a later time, then the program must normally account for overflow of the position. Instead, this command may be used(see below).

Internally those routines look if the difference is bigger than POS_LIMIT (0x3FFFFFFF). If so then it is assumed that an overflow happened and it is handled correctly.

> This will not solve the problem of overflowing if the application uses user units.

Command Group Axis and Information Commands

Syntax Example Calculate the difference between Ipos and "oldpos" in user units, handling an overflow if necessary.

diff = Ipos(0) - oldpos;

```
oldpos = Ipos(0);
...
diff = IposDiff(0, oldpos);
```

9.90 JerkFinVel

Summary Calculate the final velocity for a jerk-limited stop with maximum acceleration/deceleration.

Syntax `res = JerkFinVel(axis);`

Parameter

axis = Axis module number (see Command Reference).

Return Value Final velocity in percent (or VELRES units)

Description The command calculates the final velocity which would be reached if the actual acceleration (or deceleration) stops under consideration of the given JERKMIN values. This also works if some movement other than a RAMPTYPE 2 movement is active. So you can calculate the final velocity if you want to leave your actual SyncPos, for example. The result is in percent (or VELRES units) so you can use it without conversion for a Vel or Cvel command.

> The result could also be negative (e.g. driving backwards, or driving near of the velocity zero with a high deceleration).

Command Group Axis and Information Commands

9.91 JerkStopDist

Summary Calculate the necessary distance for a jerk-limited stop with maximum deceleration.

Syntax `res = JerkStopDist(axis, dec);`

Parameter

axis	=	Axis module number (see Command Reference).
dec	=	Sets deceleration in % or VELRES units.

Return Value Distance in uu.

Description The command calculates the distance that the axis will need to stop if it allows a maximum deceleration of dec and uses the RAMPTYPE 2 with JERKMIN2 and JERKMIN4 (or JERKMIN). In the command, the deceleration dec is given in percent (or VELRES units) and the result is given in User-Units.

This command also works if you are not in a RAMPTYPE 2 movement. So you can calculate a RAMPTYPE 2 stop-ramp if you are in SyncPos, for example.

Command Group Axis and Information Commands

9.92 KinematicsCpoint

Summary Query the current kinematics position.

Syntax `KinematicsCpoint(hndl, pnt, type);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
pnt	=	**dpoint** variable into which the position will be returned.
type	=	Coordinate system in which the current position is to be returned. Possible values are: **PATHCOORD_MACHINE (0)** - Machine Coordinate System. **PATHCOORD_WORK (1)** - Work Coordinate System. **PATHCOORD_CYCLIC (2)** - Cyclic Coordinate System. This is only available when running in path synchronization mode. **PATHCOORD_PATH (3)** - Path Coordinate System. This is only available when running in path synchronization mode.

Return Value -

Description **KinematicsCpoint** will query the current "command" position when multiple-axis robot kinematics are being used. This corresponds to the Cpos command for non-kinematics positions. The position may be returned in any of the available coordinate systems (see Additional Coordinate Systems). Note that the Path and Cyclic coordinate systems are only available while path synchronization mode is active (i.e. after PathStart has been executed).

See Working with Kinematics for more information on robots.

Portability This feature can be controller dependent and is listed in the Controller Dependent Feature List.

Command Group Kinematics Commands

Syntax Example

```
long   hndl;
dpoint pnt;

hndl = KinematicsSetup(...);

KinematicsCpoint(hndl, pnt, PATHCOORD_CYCLIC);
```

9.93 KinematicsWorkToMc

Summary Set a Work-to-Machine transformation that will be used with the selected kinematics.

Syntax `KinematicsWorkToMc(hndl, trns);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.

Summary Set a Work-to-Machine transformation that will be used with the selected kinematics.

trns = **transform** variable containing the current transformation.

Return Value -

Description This command allows the Work-to-Machine transformation to be set when multiple-axis robot kinematics are being used. This transformation is the transformation that maps points *from* the Work Coordinate System and *into* the Machine Coordinate System.

See Additional Coordinate Systems for a description of the various coordinate systems and the relationship between them. See Multi-axis Kinematics for more information on robot kinematics.

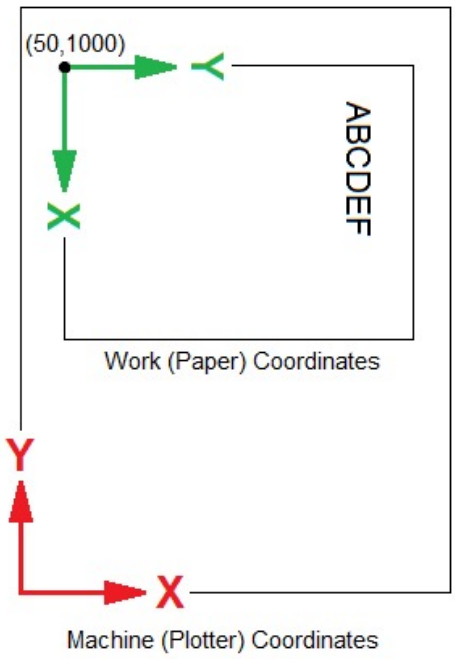
This transformation is "dynamic", meaning there is an expectation that it may change *while* a particular path is being followed. For example, an application may be drawing characters on an object while that object is moving along a conveyer belt. In this case, the application progarm should re-execute the command at appropriate time intervals in order to keep the Work-to-Machine transformation up-to-date.

> Note that the command is also extremely useful in non-dynamic situations where it provides the user with additional flexibility in specifying the transformations from the original path coordinates to the final motor positions. In this static case, the command would be executed once during some initialization procedure before starting to use path curves.

Static Example:

Consider a simple XY pen plotter that is set up to draw on a sheet of paper. For convenience in the application application, the Work Coordinate System will be defined to exactly match the paper as it would appear to a human reader. This means the PathMove and PathStart commands can work directly in paper coordinates and ignore all following machine-specific transformations. The origin will be at the bottom left corner of the paper and coordinates will be specified in millimeters.

The Machine Coordinate System will represent the drawing space of a plotter as it is defined by the kinematics being used by the plotter. For this example, the origin will be at the bottom left corner of the drawing space and coordinates will be specified in 1/10's of millimeters. The paper will be rotated and positioned on the plotter as shown here:



The Work-to-Machine transformation defining where the paper is to be positioned on the plotter, can now be created as follows:

1. Start with a coordinate system that matches the Work (Paper) Coordinate System. (x,y) points are measured in

Summary Set a Work-to-Machine transformation that will be used with the selected kinematics.

millimeters.

2. Rotate by -90 degrees (i.e. clockwise) so that it matches the orientation of the Work Coordinate System as it is placed in the Machine Coordinate System.
3. Scale by 10 (i.e. 1 Work Coordinate System unit equals 10 Machine Coordinate System units). This will convert the measurements from millimeters to 1/10's of millimeters. So (x,y) is now measured in 1/10's of millimeters. This matches the scale of the Machine Coordinate System.
4. Translate to (50,1000) where (50,1000) is specified in Machine Coordinate units (i.e. 1/10's millimeters). This will position the paper) so that its origin is at the correct position in the Machine Coordinate System (i.e. towards the top of the working area of the plotter).

Example 1:

The bottom left corner of the paper has coordinates (0,0) in the paper's coordinate system.

1. Rotate clockwise: (0,0) → (0,0)
2. Scale by 10: (0,0) → (10×0,10×0) → (0,0)
3. Translate by (50,1000): (0,0) → (0+50,0+1000) → (50,1000)
4. So the point (0,0) on the paper will map to (50,1000) on the plotter (as is shown in the diagram above).

Example 2:

Assume that the paper is 50 millimeters tall and 30 millimeters wide (i.e. a very small sheet of paper!). The top right corner of the paper has coordinates (30,50) in the paper's coordinate system.

1. Rotate clockwise: (30,50) → (50,-30)
2. Scale by 10: (50,-30) → (10×50,10×-30) → (500,-300)
3. Translate by (50,1000): (500,-300) → (500+50,-300+1000) → (550,700)
4. So the point (30,50) on the paper will map to (550,700) on the plotter.

Dynamic Example:

Consider the same example as above. However, now the paper in the plotter is slowly moving down *as the drawing is being done*. In order to keep the location of the Work (Paper) Coordinate System synchronized with the paper (so that the drawing does not smear as it is drawn), we also need to move the Work Coordinate System down. This can be done simply by appending a small translation onto the existing transformation. The frequency with which this must be done, will vary depending on the application.

Example:

Assume that the paper is moving down at a rate of 1 centimeter per second and that updating the transformation every 10 milliseconds will give us a smooth enough drawing. This means the paper will move down by 0.1 millimeters in 10 milliseconds. Since the Machine Coordinate System is measured in 1/10's millimeters, this means the paper moves down 1 machine unit in the Y coordinate in 10 milliseconds.

1. Set up the original transformation and start drawing.
2. Wait 10 milliseconds.
3. Translate the *current* transformation by (0,-1).
4. If the drawing is not yet finished, then go back to step 2.

Portability This feature can be controller dependent and is listed in the Controller Dependent Feature List.

Command Group Kinematics Commands

Syntax Example

```
transform PageToPlotter;

hndl = KinematicsSetup(...);

TransIdent(PageToPlotter);
TransRotateXY(PageToPlotter, rad(-90.0));
TransScale(PageToPlotter, 10);
TransTranslate(PageToPlotter, 50, 1000, 0);

KinematicsWorkToMc(hndl, PageToPlotter);
```

9.94 KinematicsSetup

Summary Select the multi-axis kinematics model to be used when working with robot control.
Enable/disable robot control.
Delete the kinematics model.

Syntax
```
hndl = KinematicsSetup("model", axisarray, parmarray);
KinematicsSetup(ENABLE, hndl);
KinematicsSetup(DISABLE, hndl);
KinematicsSetup(DELETE, hndl);
```

Parameter

hndl	=	Kinematics handle used to reference this group of axes.
"model"	=	Kinematics model to be used (quoted string).
axisarray	=	Ordered array of **long** values specifying the axes to be used. Axes must be specified using 0-based axis modules numbers.
parmarray	=	Array of **double** values specifying any parameters required by the kinematics model. Note that this array must be specified even if the selected kinematics model requires no parameters.

Return Value hndl = Kinematics handle

Description hndl = **KinematicsSetup**(...)

If the controller will be used to drive multiple-axis robot kinematics, then the **KinematicsSetup** command must be executed to select which kinematics model is to be used. It is also used to specify which axes will participate in the kinematics, along with various model-specific parameters. The command will return a "kinematics handle" which is then used to reference that group of axes when any other kinematics-related command is to be executed.

See Multi-axis Kinematics for more information on robot kinematics.

Each controller axis may participate in only one kinematics model at a time. However, the controller may drive more than one kinematics if it supports enough axes to do so. Once a group of axes has been assigned to a kinematics model, then that assignment is maintained for the duration of the execution of the application program or until the program explicitly deletes the group using the **KinematicsSetup**(DELETE) command. Axes cannot be removed from one kinematics and reassigned to a different kinematics without first deleting the group.

Each kinematics model requires a set of mandatory "kinematics axes"; one for each motor driven by the kinematics. These axes are used to position of the robot end effector in the real world. In some types of robots, these axes may correspond to linear positions of belts or lengths of hydrolic cylinders. In other robots, they may correspond to angular positions of arms. In addition to the mandatory kinematics axes, the **KinematicsSetup** command may define zero or more optional axes. The optional axes are used to define axes that do not directly influence the mandatory axes. For example, an optional axis could be used to define a rotation element attached to the robot end effector. If optional axes have been defined, then they will be driven synchronously with the mandatory axes. The "axisarray" parameter is used the specify the axis numbers of the axes that will be used with the kinematics. The order of this array is important. See Multi-axis Kinematics for the specific kinematics models that are available and how the mandatory axes must be defined for each model.

Generally, robot kinematics are driven using Aposs Path Curves. This allows two- and three-dimensional paths to be defined using "real world" coordinates, rather than using the more complicated "raw" coordinates of the individual axes. The controller handles all the conversion from world coordinates to axis coordinates internally without the user needing to be concerned with the complicated mathematics required for this conversion.

> Gearing factors must be correctly initialized *before* the **KinematicsSetup** command is executed (see How to Set Up Resolution and Gear Parameters). These factors are saved internally and will be used for the duration of the kinematics group's existence. If they must be changed, then the group must be deleted and recreated.

> After a kinematics group has been created, it is still possible to execute all normal commands. In particular, it is possible to use commands such as AxisPosAbsStart, AxisPosRelStart, etc., to initialize the position individual axes. In addition, the kinematics handle may now be used to initialize the robot setup (i.e. to set transformations, velocity limits, etc.), before the robot is used for the first time. The only commands that are *not* available, are the commands related to synchronized robot movement (e.g. PathMove, etc.).

KinematicsSetup(ENABLE)

Summary Select the multi-axis kinematics model to be used when working with robot control.
Enable/disable robot control.
Delete the kinematics model.

Once all axis initialization and robot setup has been completed, then the kinematics group may be enabled using the **KinematicsSetup**(ENABLE) command. This will cause all axes in the group to be switched into "kinematics mode". In this mode, the axes may only be moved synchronously as a group by using command such as PathMove and PathStart.

> ⚠ While an axis is in kinematics mode, all commands designed to move axes individually (e.g. AxisPosAbsStart, AxisPosRelStart, etc.) will be *disabled*. The reason for this is that, given the mechanical connections between motors in some type of kinematics, it is not always possible to move a single motor independently. With these types of kinematics, multiple motors must always be moved simulateously or the mechanics may be damaged.

KinematicsSetup(DISABLE)

The **KinematicsSetup**(DISABLE) command may be used to temporarily disable a kinematics group, removing the axes from kinematics mode. While the group is disabled, individual axes may again be moved independently by using commands such as AxisPosAbsStart. However, the path movement commands (e.g. PathMove, PathStart, etc.) will no longer be available.

The group may be re-enabled again by using the **KinematicsSetup**(ENABLE) command.

> ⚠ If the **KinematicsSetup**(DISABLE) command is execute *while the axes are moving*, then an implicit PathStop is executed first. Once the axes are stopped, then the kinematics will be disabled. Kinematics will *not* be disabled while the axes are moving.

> ⚠ Both the AxisControl(STOP) and AxisControl(OFF) commands may be executed while axes are moving in kinematics mode. If these commands are executed for *any* axis in kinematics mode, then *all* axes in the group will be stopped using an implicit PathStop command. Once stopped, kinematics will be disabled for the group and all axes will be removed from kinematics mode. In the case of AxisControl(OFF), the final step of tunrning the motors off will be done only after the kinematics has been disabled.

> ⚠ Controller errors will cause an implicit AxisControl(OFF) to be executed for all axes in all kinematics groups. This will be processed as described above.

KinematicsSetup(DELETE)

The **KinematicsSetup**(DELETE) command may be used to delete a kinematics group. This is normally unnecessary unless one or more of the axes are to be used in a different kinematics group. Once the group has been deleted, the kinematics handle for the group will become invalid.

Portability This feature can be controller dependent and is listed in the Controller Dependent Feature List.

Command Group Kinematics Commands

Syntax Example

```
// Set up the kinematics specification.

long   axsarr[3], hndl;
double prmarr[1];

axsarr[0] = 1;   // Axis module 1 will be the X axis.
axsarr[1] = 0;   // Axis module 0 will be the Y axis.
axsarr[2] = 2;   // Axis module 2 will be the Z axis.

prmarr[0] = 3;   // Use 3-dimensional kinematics.

// Simple Cartesian axis kinematics will be used.

hndl = KinematicsSetup("Cartesian", axsarr, prmarr);
```

9.95 LinkPdo

Summary Mapping of RxPDOs: Link the content of a RxPDO to elements of the internal Sysvar array. Each change of the RxPDO is copied automatically into the configured Sysvar elements afterwards.

Syntax `LinkPdo(no, len, indx, pdo);`

Parameter

no	=	rank order in the RxPDO, beginning with 1
len	=	number of the bits to be imported; requirement: length = a multiple of 8 (bit-by-bit), (e.g. 128 to copy 4 long values into the PDO, if PDO holds more than 8 bytes)
indx	=	index of the system variable (see Sysvar)
pdo	=	1 - n (number of supported PDOs see in Controller Dependent Feature List)

> The rank order number (no) must - beginning with 1 - increase.

Multiple PDOs

The command can be used for any PDO.

CANopen PDO size

A CANopen PDO is maximal 8 bytes long; it can therefore hold a maximum of 8 objects.

EtherCAT PDO Size

For EtherCAT, a PDO can be longer. See Controller Dependent Feature List

The PDO sizes can be different for Tx and RxPDOs.

> The summed up sizes of all EtherCAT PDOs must be an even amount of bytes! Otherwise, a COE emergency will be issued.

Return Value -

Description The command **LinkPdo** links the content of RxPDOs to elements of the Sysvar array. This is called PDO mapping. Each change of RxPDO data is copied automatically into the configured Sysvar elements.

The Sysvar array holds internal data and variables, as well as each SDO (according to the SDO object dictionary), which also means the first 250 elements of each application array are included. The RxPDO data can be directed to almost any variable, array or parameter by this. The Sysvar index of a SDO can be specified using the "xxx_INDEX" defines from the SysDef.mh include file. For example, the SDO index of actual position of axis module 2 is:

AXE_PROCESS_INDEX(2,REG_ACTPOS)

The RxPDO is also copied into the PDO array. The same data content can be accessed by reading the PDO array.

Automatic PDO activation

Just the PDO 1 (RxPDO = 0x200 + Node-ID / TxPDO = 0x180 + Node-ID) is enabled by default according to the CANopen specification, i.e. the “Valid” bit (0x1400 / Subindex 1) is set. If the mapping is configured for other PDOs using **LinkPdo** (or LinkSdo), then the “Valid” bits (0x1401-0x140n, Subindex 1) of these RxPDOs are also set automatically.

CANopen versus application mapping

If the Sysvar index refers to a SDO of the SDO object dictionary (i.e. Sysvar index starts with 0x01...), then a pure CAN mapping is internally executed. When a corresponding object is changed by a SDO command, the PDO is also immediately rewritten. The mapping for the corresponding CAN object can be read-out by a supervisor control unit.

If other Sysvar indices are used, an application mapping is carried out. This can be combined with the CAN Mapping, but does not conform to CANopen, because the correct map entries can not be read out by the CANopen mapping objects.

> The linking of internal system variables must be accomplished very carefully and should only be done from experienced users. Thorough knowledge about the usage and meaning of the internal system variable is necessary, not to cause an incorrect system behavior.

Summary Mapping of RxPDOs: Link the content of a RxPDO to elements of the internal Sysvar array. Each change of the RxPDO is copied automatically into the configured Sysvar elements afterwards.

Command Group Communication CANopen Commands

Syntax Example 1
```
// Link RxPDO 1 to user parameter 1 (= SDO 0x2201/01)
LinkPdo(1, 32, USER_PARAM_INDEX(1), 0);
```

Syntax Example 2
```
// Link 8 bits of RxPDO 1 to digital outputs 1 - 8 (= SDO 0x2202/10)
LinkPdo(1, 8, SYS_PROCESS_INDEX(SYS_OUTBYTE_0), 1);
// Link next 8 bits of RxPDO 1 to outputs 9 - 16 (= SDO 0x2202/11)
LinkPdo(2, 8, SYS_PROCESS_INDEX(SYS_OUTBYTE_1), 1);
```

Syntax Example 3
```
// Link 16 bits of RxPDO 1 into DS402 control word (= SDO 0x6040/0)
LinkPdo(1, 16, SDO_INTERNAL(0x6040, 0), 1);
```

9.96 LinkSdo

Summary Mapping of the TxPDOs: Link elements of the internal system variable pseudo array Sysvar to a TxPDO. Each change of the corresponding Sysvar element is forwarded to the TxPDO automatically afterwards.

Syntax
```
LinkSdo(indx, len, no, "text", pdo);
```

Parameter

indx	=	Index of the system variable Sysvar.
len	=	Length of the bits to be imported; requirement: length = a multiple of 8 (bit-by-bit) (e.g. 128 to copy 4 long values into the PDO, if PDO holds more than 8 bytes)
no	=	rank order in the TxPDO, beginning with 1
text	=	" " (string is not used internally; however, can be used as comment)
pdo	=	1 - n (number of supported PDOs see in Controller Dependent Feature List)
-pdo	=	dummy entry of the link object into the mapping list

> The rank order number (no) must - beginning with 1 - increase.

Multiple PDOs

The command can be used for any PDO.

CANopen PDO size

A CANopen PDO is maximal 8 bytes long; it can therefore hold a maximum of 8 objects.

EtherCAT PDO Size

For EtherCAT, a PDO can be longer. See Controller Dependent Feature List

The PDO sizes can be different for Tx and RxPDOs.

> The summed up sizes of all EtherCAT PDOs must be an even amount of bytes! Otherwise, a COE emergency will be issued.

Return Value -

Description The command **LinkSdo** links the content of one or more elements of the system variables pseudo array Sysvar to a TxPDO. This is called PDO mapping. Each change of a linked Sysvar element is forwarded automatically into the defined bytes of the TxPDO.

The system variables pseudo array Sysvar holds internal data and variables, as well as each SDO (according to the SDO object dictionary), which also means the first 250 elements of each application array are included. The content of almost any variable, array or parameter can be forwarded to TxPDOs by the **LinkSdo** mapping configuration. The Sysvar index of a SDO can be specified using the "xxx_INDEX" defines from the SysDef.mh include file. For example, the SDO index of actual position of axis module 2 is:

Summary Mapping of the TxPDOs: Link elements of the internal system variable pseudo array Sysvar to a TxPDO. Each change of the corresponding Sysvar element is forwarded to the TxPDO automatically afterwards.

AXE_PROCESS_INDEX(2,REG_ACTPOS)

The TxPDO is also copied into the PDO array. The same data content can be accessed by reading the PDO array.

> As standard, a changed PDO is automatically dispatched (asynchronous operating mode). If this is not desired, then, you can set the SDO-Index 0x1800 sub index 2 to another value (e.g. 254, instead of the standard 255). Thereby, dispatching no longer takes place automatically, but instead the PDO must be collected per remote frame.

Automatic PDO activation

Just the PDO 1 (RxPDO = 0x200 + Node-ID / TxPDO = 0x180 + Node-ID) is enabled by default according to the CANopen specification, i.e. the "Valid" bit (0x180n / Subindex 1) is set. If the mapping is configured for other PDOs using **LinkSdo** (or LinkPdo), then the "Valid" bits (0x1801-0x180n, Subindex 1) of these TxPDOs are also set automatically.

CANopen versus ApossIDE mapping

If the SYSVAR index refers to a SDO of the SDO object dictionary (i.e. Sysvar index starts with 0x01...), then a pure CAN mapping is internally executed. When a corresponding object is changed by a SDO command, the PDO is also immediately rewritten. The mapping for the corresponding CAN object can be read-out by a supervisor control unit.

If other Sysvar indices are used, an ApossIDE mapping is carried out. This can be combined with the CAN Mapping, but does not conform to CANopen, because the correct map entries can not be read out by the CANopen mapping objects.

> The linking of internal system variables must be done very carefully and should only be done by experienced users. Thorough knowledge of the usage and meaning of the internal system variables is necessary to prevent incorrect system behavior.

Transfer values from a master (MACS) to all slaves

Using a negative PDO number includes the LINK object only into the mapping list as a dummy entry and the object itself is handled in the low level EtherCAT driver. This is used especially for transferring encoder values from a “master MACS” to all slaves. This is mainly done to ensure synchronism between the process data cycle and the readout of the position.

Example:

```
// Map actual position into EtherCAT PDO1
LinkSdo(AXE_PROCESS_INDEX(0, REG_ACTPOS), 32, 1, "", -1);
    // PDO 1, offset 0 (in longs)
    // Negative number means synchronous mapping

// Map system time into the EtherCAT PDO
LinkSdo(SYS_PROCESS_INDEX(SYS_CLOCK), 32, 2, "", -1);
    // with an offset of 1 (long) in PDO 1
```

Command Group Communication CANopen Commands

Syntax Example 1
```
// Link current position error (= SDO 0x0x2500/6) to TxPDO 1
LinkSdo(AXE_PROCESS_INDEX(0, REG_TRACKERR), 32, 1, " ", 1);
```

Syntax Example 2
```
// Link digital inputs 1 - 8 (= SDO 0x2202/10) to TxPDO 1
LinkSdo(SYS_PROCESS_INDEX(SYS_INBYTE_0), 8, 1, " ", 1);
// Link digital inputs 9 - 16 (= SDO 0x2202/10) to next 8 bits of TxPDO 1
LinkSdo(SYS_PROCESS_INDEX(SYS_INBYTE_1), 8, 2, " ", 1);
```

Syntax Example 3
```
// Link DS402 status word (= SDO 0x6041/0) into TxPDO 1
LinkSdo(SDO_INTERNAL(0x6041, 0), 16, 1, " ", 1);
// Link DS402 "Mode of operation display" (= SDO 0x6061/0) into TxPDO 1
```

Summary Mapping of the TxPDOs: Link elements of the internal system variable pseudo array Sysvar to a TxPDO. Each change of the corresponding Sysvar element is forwarded to the TxPDO automatically afterwards.

```
LinkSdo(SDO_INTERNAL(0x6061, 0), 8, 2, " ", 1);
```

9.97 Mapos

Summary Query the current actual position of the master.

Syntax

```
res = Mapos(axis);
```

Parameter axis = Axis module number (see Command Reference).

Return Value Master position to absolute actual zero point in qc.

Description With the **Mapos** command it is possible to query the actual master position (absolute to the actual zero position).

Command Group Axis and Information Commands

9.98 MaposDiff

Summary Overflow handling of incremental encoders in applications.

Syntax

```
res = MaposDiff(axis, oldpos);
```

Parameter axis = Axis module number (see Command Reference).

oldpos = Mapos at a previous time.

Return Value Returns difference between Mapos and oldpos (res = Mapos - oldpos) in uu

Description This command simplifies overflow handling of incremental encoders in applications. For example, if the user stores an actual position in the program and needs to calculate the difference at a later time, then the program must normally account for overflow of the position. Instead, this command may be used (see below).

Internally those routines look if the difference is bigger than POS_LIMIT (0x3FFFFFFF). If so then it is assumed that an overflow happened and it is handled correctly.

> This will not solve the problem of overflowing if the application uses user units.

Command Group Axis and Information Commands

Syntax Example Calculate the difference between Mapos and "oldpos" in user units, handling an overflow if necessary.

diff = Mapos(0) - oldpos;

```
oldpos = Mapos(0);
...
diff = MaposDiff(0, oldpos);
```

9.99 Mavel

Summary Query the actual velocity of the master.

Syntax

```
res = Mavel(axis);
```

Parameter axis = Axis module number (see Command Reference).

Return Value Actual signed velocity of the master in qc/s.

Description This function returns the actual velocity of the master drive in qc/s, with qc referring to the master encoder.

The accuracy of the values depends on the duration of the measurement (averaging). The standard setting is 20 ms, but this can be changed by the user with the process data value REG_VELSMPT for each axis.

For example the setting:

Summary Query the actual velocity of the master.

```
AXE_PROCESS(axis,  REG_VELSMPT) = 100;
```

sets the duration of the measurement to 100 ms, so that you have a considerably better resolution of the speed with Avel and **Mavel**, however, in contrast, quick changes are reported with a delay of a maximum of 100 ms.

Command Group Axis and Information Commands

9.100 MemoryDump

Summary The **MemoryDump** function copies data from volatile memory of the controller to an SD card.

Syntax `MemoryDump(index, subindex, filename);`

Parameter

index	=	SDO index (see examples).
subindex	=	SDO subindex (see examples).
filename	=	Target file name (including folder if applicable). This may be either a literal string within double-quotes or an array with optional indices. Maximum length of filename is 32 characters for MACS5, 100 characters for MasterMACS.

Return Value -

Description The **MemoryDump** function copies data from volatile memory of the controller to an SD card or USB Memory Stick (controller dependent). Currently, the following data sources are supported:

- DIM arrays: index = 0x2100, subindex = arrayno
- Record Oscilloscope data
  Data in dynmen: index = 0x2214, subindex = 0xFFFF
  Data in array: index = 0x2214, subindex = array no
- Print Message Buffer: index = 0x2212, subindex = 1
- Ringbuffer Array with Log Data: index = 0x2212, subindex = 2

The function consists of two commands. A data transfer is initiated with the **MemoryDump** command. After that, the execution state can be polled with the MemoryDumpStatus command. A successive **MemoryDump** command will fail if the previous command has not yet completed.

> ⚠ The MemoryDump command is blocking on some controller types. This means the command will not return as long as the complete write process to the external storage takes.
>
> On some other controller types, the command just starts the process, and then immediately returns. The write process runs in the background.
>
> Refer to the Controller Dependent Feature List to see if a specific controller supports the non-blocking write.

For more information on using SD cards, see External Storage Device (microSD Card/ USB Memory Stick).

Command Group Memory Commands

Usage Examples **Print Message Buffer Example**

This example shows how to use the **MemoryDump** function to write the content of the Print Message Buffer into a file on the SD-Card. If the file already exists the additional data will be appended at the end of the file.

```
long main(void)
{
 // Be sure that there is at least one message in
 // the print buffer
 print("Disconnect your ApossIDE Interface now!");
 Delay(5000);
 print("Hello World!");
 // Initiate data transfer: For Print Message Buffer use
 // SDO index = 0x2212 and subindex = 1
 MemoryDump(0x2212, 1, "data/Print.txt");
```

Summary The **MemoryDump** function copies data from volatile memory of the controller to an SD card.

```
 // Poll the execution state
 while (MemoryDumpStatus() == 1)
 {
  // wait here while the task is processing
 }
 print("End: result = ", MemoryDumpStatus());
 return(0);
}
```

"Record" Oscilloscope Data Example

This example shows how to use the **MemoryDump** function to copy previously collected "Record" data into a file on the SD-Card. If the file already exists it will be replaced. If the filename is given a ".txt" file extension, then it may be opened with the Record Oscilloscope.

```
long main(void)
{
 long i=0;
 // Setup and start a internal recording
 // of two user parameters and dynamic memory
 RecordIndex(0x01220101, 0x01220102);
 RecordStart(0);
 // Write sine and cosine values to the user parameters
 i = 0;
 for(i=0;i<5000;i++)
 {
  Sysvar[0x01220101] = 1000.0 * sin((2.0 * PI * i) / 5000.0);
  Sysvar[0x01220102] = 1000.0 * cos((2.0 * PI * i) / 5000.0);
  Delay(1);
 }
 // Stop recording
 RecordStop(0, 0);
 // Initiate data transfer: For "Record" Data
 // in DYNMEM use
 // SDO index = 0x2214 and subindex = 0xFFFF
 MemoryDump(0x2214, 0xFFFF, "Oszi.txt");
 // Poll the execution state
 while (MemoryDumpStatus() == 1)
 {
  // Wait here while the task is processing
 }
 print("End: result = ", MemoryDumpStatus());
 return(0);
}
```

DIM Array Example

This example shows how to use the **MemoryDump** function to copy data from DIM Arrays to a file on the SD-Card. If the file already exists the additional data will be appended at the end of the file.

```
#define ARRAY_LEN 1000
// Create array
dim long TestArray[ARRAY_LEN];
long main(void)
{
 long i;
 // populate array with data
 for (i = 0; i < ARRAY_LEN; i++) TestArray[i] = i;
 // Copy Dimarray content to SD-Card: For Dimarrays use
```

Summary The **MemoryDump** function copies data from volatile memory of the controller to an SD card.

```
// SDO index = 0x2100 and subindex = 0...
MemoryDump(0x2100, 0, "Array.txt");
// Poll the execution state
while (MemoryDumpStatus() == 1)
{
 // Wait here while the task is processing
}
print("End: result = ", MemoryDumpStatus());
return(0);
}
```

Log Ringbuffer Example
See Logging Service article.

9.101 MemoryDumpStatus

Summary This command can be used to poll the status/result of a MemoryDump command

Syntax `res = MemoryDumpStatus();`

Parameter -

Return Value

0	The task has completed.
1	The task is processing.
2	Error: Invalid or unknown command parameter.
6	Error accessing the target file on the SD-Card or USB Stick.
≥ 240	Internal error.

Description Indicates the processing state of the last MemoryDump.

Command Group Memory Commands

Syntax Example See MemoryDump.

9.102 Mipos

Summary Query last index or marker position of the master.

Syntax `res = Mipos(axis);`

Parameter axis = Axis module number (see Command Reference).

Return Value Last index or marker position of the master absolute to actual zero point in qc.

Description The **Mipos** command returns the last index or marker position of the master absolute to the current zero point.

The configuration of **Mipos**, that is whether master-encoders index- or marker position (= controlled drive) is returned, is done with the parameter SYNCMTYPM.

> Many MACS controller have hardware latch modules and incremental encoder interfaces for very accurate position latch information. It is also possible to use the latch functionality of a drive (if supported) and get the position over a bus (e.g. CAN or EtherCAT). Check the virtual latch module and bus modules for the correct settings. Also a software latching (accuracy of typically 1ms) is possible.

The position value in **Mipos** is accurate to ±1 qc. In opposite to the position information in Mapos, which is just updated in a controller cycle of typically 1 ms, the actual position value is hardware stored in real time a buffer (in an internal processor register), when the configured signal is high. Then it will be copied in the system variable **Mipos**.

If an DIGINPUT interrupt is triggered and the interrupt routine executes **Mipos**, then a delay of 2 milliseconds should be executed before executing **Mipos** (see example below). This ensures that the latched position value has been completely updated in the system, preventing **Mipos** from returning an old value.

Summary Query last index or marker position of the master.

See also sample.

> The command **Mipos** returns only accurate position latching information if a incremental encoder and a built in hardware latch module is used.

Command Group Axis and Information Commands

Sample

```
// Define MIPOS latching on positive edge at input 1
// By default the first virtual latch module(0) is linked to the
// to the first hardware latch module(0). Masterlatch source is VIRTLATCH(0).
AXE_PARAM(0, PISRC_MVIRTLATCH) = 0;
// Moving slave axis is also master for this sample code
AXE_PARAM(0, PISRC_MVIRTCOUNT) = 0;


// Setting up the hardware latch (input number and edge)
HWLATCH_PARAM(0, HWLATCH_PISRC_TRIGGER) = 0x140; //digital input 1
HWLATCH_PARAM(0, HWLATCH_SLOPE) = 1; //rising slope

// Definition Interrupt-Handler
InterruptSetup(MasterInt, DIGINPUT, 1, RISING);

Cvel(0, 10);   // Start moving
AxisCvelStart(0);
while (1);     // Endless loop
...

void MasterInt(void)
{
    long int_mpos, triggered_mpos;
    // Latching Mapos for testing, how exact it would be ...
        int_mpos = Mapos(0);
    // Wait 2 ms, to be sure, that Mipos is correct updated
    Delay(2);
    // Latching IPOS for a later handling etc.
    triggered_mpos = Mipos(0);
    // ...
    // ...
    print("Interrupt master position: ",int_mpos);
    print("Triggered master position: ",triggered_mpos);
    return;
}
```

9.103 MiposDiff

Summary Overflow handling of incremental encoders in applications.

Syntax `res = MiposDiff(axis, oldpos);`

Parameter

axis = Axis module number (see Command Reference).

oldpos = Mipos at a previous time.

Return Value Difference between Mipos and oldpos (res = Mipos - oldpos) in uu

Description This command simplifies overflow handling of incremental encoders in applications. For example, if the user stores an actual position in the program and needs to calculate the difference at a later time, then the program must normally account for overflow of the position. Instead, this command may be used (see below).

Internally those routines look if the difference is bigger than POS_LIMIT (0x3FFFFFFF). If so then it is assumed that an

Summary Overflow handling of incremental encoders in applications.
overflow happened and it is handled correctly.

> This will not solve the problem of overflowing if the application uses user units.

Command Group Axis and Information Commands

Syntax Example Calculate the difference between Mipos and "oldpos" in user units, handling an overflow if necessary.

diff = Mipos(0) - oldpos;

```
oldpos = Mipos(0);
...
diff = MiposDiff(0, oldpos);
```

9.104 MotorAlignStart

Summary Starts rotor alignment procedure

Syntax `MotorAlignStart(axis, mode);`

Parameter

axis = Axis module number (see Command Reference).

mode = Which mode of alignment

1: EC-Motors: Forward and backwards alignment

4: EC-Motors: One direction alignment

3: For stepper motors: Forward and backwards alignment

Return Value Since this command sets some amplifier commands in the background, error 101 may be risen by the call of this function.

Description The purpose of this command is to find the rotor position of a brushless motor in relation to the encoder position feedback.
The command will start a program in the amplifier, which will apply a magnetic field to the rotor. The rotor will then align to this field. The field will then be varied a bit back and forth or only one direction to commit the measurement.

> This command can only be performed on a MACS controller with integrated amplifiers.
>
> The axis must be setup as one of the following HWAMP_COMMTYPE:
>
> **HWAMP_COMMTYPE_PMSM**
>
> **HWAMP_COMMTYPE_PMSM_TWIN**
>
> **HWAMP_COMMTYPE_STEP**
>
> **HWAMP_COMMTYPE_STEP_TWIN**
>
> If a different HWAMP_COMMTYPE is assigned, an error will be raised. Note that not all controller support these HWAMP_COMMTYPE.

> ⚠ In order for this procedure to work properly, the motor and encoder parameter has to be setup. This function use them to move the rotor in an appropriate manner.

> ⚠ In order for this procedure to work, the motor must not be braked or be in a limit position. The rotor must be able to rotate freely for about half electrical revolution (Example: Motor has 11 pole pairs. Alignment will move about 360 Degree / 11 / 2 = 17 Degree).

Command Group Amplifier Commands

Syntax Example

```
print("AxisNo ", AxisNo, " Start Elpos Detection");
MotorAlignStart(AxisNo, 1);
while((retVal = MotorAlignStatus(AxisNo)) == 0)
{
```

Summary Starts rotor alignment procedure

```
        print("Aligning... ", retVal);
        Delay(500);
    }
    if(retVal < 0)
        print("!!!! Rotor position detection FAILED !!!! retVal: ", retVal);
    else
        print("Alignment Completed with ", retVal);
```

9.105 MotorAlignStatus

Summary Returns rotor alignment procedure status

Syntax `res = MotorAlignStatus(axis);`

Parameter axis = Axis module number (see Command Reference).

Return Value res = Current status

2: Process successful. Rotational direction regular.

1: Process successful. Rotational direction inverse.

0: Process is active

-1: Error, procedure aborted

-2: Error, ELPOS reset failed

-3: Error, irregular motion. Check if encoder is connected and configured.

-4: Error, illegal state sequence

-5: Error, imprecise motion

-6: Error, illegal parameter

-10: Error, timeout

Since this command sets some amplifier commands in the background, error 101 may be risen by the call of this function.

Description The purpose of this command is to query the current status of the rotor alignment program. This command makes only sense after the MotorAlignStart command.

> ⚠ In order for this procedure to work, the motor must not be braked or be in a limit position. The rotor must be able to rotate freely for about half electrical revolution (Example: Motor has 11 pole pairs. Alignment will move about 360 Degree / 11 / 2 = 17 Degree).

Command Group Amplifier Commands

Syntax Example

```
print("AxisNo ", AxisNo, " Start Elpos Detection");
MotorAlignStart(AxisNo, 1);
while((retVal = MotorAlignStatus(AxisNo)) == 0)
{
    print("Aligning... ", retVal);
    Delay(500);
}
if(retVal < 0)
    print("!!!! Rotor position detection FAILED !!!! retVal: ", retVal);
else
    print("Alignment Completed with ", retVal);
```

9.106 PathAcc

Summary Set the maximum acceleration for path movements.

Syntax `PathAcc(hndl, acc);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
acc	=	**double** value specifying the maximum acceleration in units per second squared.

Return Value -

Description When a PathMove command is executed, or a PathStart command without the virtual master is executed, then movement along the line or curve is done using a Limited Jerk movement based on arc length along the line or curve. The **PathAcc** command is used to specify the maximum acceleration to be used.

> A simple formula to determine the maximum acceleration is:
>
> (maximum velocity) / (seconds to go from 0 to maximum velocity)
>
> The behaviour of the limited jerk movement will make the actual time required slightly longer. Note that the "maximum velocity" in the above formula, does not need to correspond to the velocity specified by the PathVel command.

> The way the **PathAcc** command specifies the maximum acceleration, is the *inverse* of the way that the Acc command specifies it.

> **PathMove**
>
> The PathMove command can specify a target position in either the Machine Coordinate System or the Work Coordinate System (see Basic Coordinate Systems). If the Machine Coordinate System is used, then the **PathAcc** value is interpreted as being in **Machine Coordinate System** units. If the Work Coordinate System is being used, then the **PathAcc** value is interpreted as being in **Work Coordinate System** units. Note that this means the **PathAcc** value must be respecified if PathMove is being used with different coordinate systems.
>
> **PathStart**
>
> When the PathStart command is being used, then the **PathAcc** value is *always* interpreted as being in **Work Coordinate System** units.

See Multi-axis Kinematics for more information on robot kinematics.

Command Group Kinematics Commands

Syntax Example

```
long hndl;

hndl = KinematicsSetup(...);

PathAcc(hndl, 0.4);
```

9.107 PathArrayIndex

Summary Determine the array index of various data blocks within a Path curve array.

Syntax

```
res = PathArrayIndex(PATH_INDEX_CURVE, arrayname, crvnum);
res = PathArrayIndex(PATH_INDEX_SEGMENT, arrayname, crvnum, segnum);
```

Parameter

keyword	=	this can only be the keyword `PATH_INDEX_CURVE` or `PATH_INDEX_SEGMENT`.
arrayname	=	Name of an existing array that contains a Path curve. Dimensions must not be specfied. The array must be a **long** array and it must be declared with only a single dimension. Multi-dimensional arrays are not supported.

Summary Determine the array index of various data blocks within a Path curve array.

crvnum	=	Curve number.
segnum	=	Segment number (only used when first parameter is `PATH_INDEX_SEGMENT`.

Return Value Start index of the requested data block within the array. When the function **PathArrayIndex** is used with the parameter `PATH_INDEX_CURVE` than the return value is the start index of the curve. If called with the parameter `PATH_INDEX_SEGMENT` than it is the start index of the segment. -1 is returned if the block cannot be found (e.g. the curve or point does not exist).

> ⚠ If the Path curve array is corrupt (i.e. it contains an invalid Path array structure), then the index returned by this function may or may not be correct. It is the user's resposibility to ensure that the array is valid.

Description Some types of applications may require access to the actual internal values of a Path curve array. For example, it may be necessary for the application to "look ahead" at upcoming curve segments. The **PathArrayIndex** function is designed to allow the application to easily locate the various data blocks within the Path array structure. The application may then use normal array access to read and modify the attributes of a Path curve.

This functions is intended to be used with the Path array index offsets that are defined in "SysDef.mh". See below for an example of how the function may be used.

See CAM Array Structure for a complete description of the structure of Path curve arrays. See "SysDef.mh" for a complete list of the #defines that may be used to access the array.

> 💡 The "SysDef.mh" file also contains the following #defines (macro) that allow easier access to the different parts of Path curve fixed point types:
>
> | | |
> |---|---|
> | `PathSegType` | Get the actual segment type of this segment. |
> | `PathSegCode` | Get the user code for this segment. |
> | `PathSegCorner` | The start of this segment is a "corner" point. |
> | `PathSegStop` | The start of this segment is a "stop" point. |

Command Group Synchronization Commands

Syntax Examples This example checks to see if the start of the *next* segment is a "stop" point.

```
dim long PthCrv[85];

long NextStop(long curcrv, long curseg)
{
    // Index of NEXT segment block.
    long seg = PathArrayIndex(PATH_INDEX_SEGMENT, PthCrv, curcrv, curseg+1);

    // Segment type of next segment.
    long type = PthCrv[seg+PTH_SEG_TYPE];

    return(PathSegStop(type) ? 1 : 0);
}
```

9.108 PathContinue

Summary Continue following a path that has been paused.

Syntax `PathContinue(hndl);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.

Return Value -

Summary Continue following a path that has been paused.

Description

> This command cannot be used with any motor that is *not* currently in kinematics mode (see KinematicsSetup).

If the PathStart command has been executed to follow a path and the path has been paused, then the **PathContinue** command will restart movement along the path. Paths may be paused for the following resons:

- Movement has arrived at a "Stop" point and the PathStart command requested stops at "Stop" points.
- The PathStop command was executed.
- Movement has arrived at the end of a non-cyclic path and there is no "next" path.

The **PathContinue** command will be ignored in the following cases:

- The PathStart command is stopped at the end of a non-cyclic path and there is no "next" path. The PathNext command must be used to set a new path before movement may be continued.
- The PathStart command is not paused. Applications must wait until movement has stopped before trying to continue it.
- Movement is not following a path curve. For example, a PathMove command may be active.
- The PathStart command is running using a vitual master. In this case, the **PathContinue** command should not be necessary. Stopping and starting is done by controlling the virtual master and is the responsibility of the application.

Command Group Kinematics Commands

Syntax Example

```
dim long ptharr[100];  // Downloaded from Path Editor.
transform PthToCyclic;

hndl = KinematicsSetup(...);

TransIdent(PthToCyclic);
TransTranslate(PthToCyclic, 500, 500, 0);

// Start following the path and stop at "Stop" points.

PathStart(hndl, ptharr, 0, PthToCyclic, 0, PATHSTART_STOP);

...

// Here we have stopped at a "Stop" point.

...

PathContinue(hndl);   // Restart movement.
```

9.109 PathDec

Summary Set the maximum deceleration for path movements.

Syntax `PathDec(hndl, dec);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
dec	=	**double** value specifying the maximum deceleration in units per second squared.

Return Value -

Description When a PathMove command is executed, or a PathStart command without the virtual master is executed, then movement along the line or curve is done using a Limited Jerk movement based on arc length along the line or curve. The **PathDec** command is used to specify the maximum deceleration to be used.

Summary Set the maximum deceleration for path movements.

A simple formula to determine the maximum deceleration is:

(maximum velocity) / (seconds to go from maximum velocity to 0)

The behaviour of the limited jerk movement will make the actual time required slightly longer. Note that the "maximum velocity" in the above formula, does not need to correspond to the velocity specified by the PathVel command.

The way the **PathDec** command specifies the maximum deceleration, is the *inverse* of the way that the Dec command specifies it.

PathMove

The PathMove command can specify a target position in either the Machine Coordinate System or the Work Coordinate System (see Basic Coordinate Systems). If the Machine Coordinate System is used, then the **PathDec** value is interpreted as being in **Machine Coordinate System** units. If the Work Coordinate System is being used, then the **PathDec** value is interpreted as being in **Work Coordinate System** units. Note that this means the **PathDec** value must be respecified if PathMove is being used with different coordinate systems.

PathStart

When the PathStart command is being used, then the **PathDec** value is *always* interpreted as being in **Work Coordinate System** units.

See Multi-axis Kinematics for more information on robot kinematics.

Command Group Kinematics Commands

Syntax Example

```
long hndl;

hndl = KinematicsSetup(...);

PathDec(hndl, 0.4);
```

9.110 PathJerk

Summary Set the maximum jerk for path movements.

Syntax

```
PathJerk(hndl, jrk);
PathJerk(hndl, jrk1, jrk2, jrk3, jrk4);
```

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
jrk	=	**double** value specifying the maximum jerk in units per second cubed.

Return Value -

Description When a PathMove command is executed or a PathStart command without the virtual master is executed, then movement along the line or curve is done using a Limited Jerk movement based on arc length along the line or curve. The **PathJerk** command is used to specify the maximum jerk to be used. If only one jerk value is specified, then all four jerk values are set to this same value. If four jerk values are specified, then each value may be set differently.

A simple formula to determine the maximum jerk is:

(maximum acceleration) / (seconds to go from 0 to maximum acceleration)

Note that the "maximum acceleration" in the above formula, does not need to correspond to the acceleration specified by the PathAcc command.

The way the **PathJerk** command specifies the maximum velocity, is the *inverse* of the way that the JERKMIN parameters specifies it.

PathMove

Summary Set the maximum jerk for path movements.

> The PathMove command can specify a target position in either the Machine Coordinate System or the Work Coordinate System (see Basic Coordinate Systems). If the Machine Coordinate System is used, then the **PathJerk** values are interpreted as being in **Machine Coordinate System** units. If the Work Coordinate System is being used, then the **PathJerk** values are interpreted as being in **Work Coordinate System** units. Note that this means the **PathJerk** values must be respecified if PathMove is being used with different coordinate systems.
>
> **PathStart**
>
> When the PathStart command is being used, then the **PathJerk** values are *always* interpreted as being in **Work Coordinate System** units.

See Multi-axis Kinematics for more information on robot kinematics.

Command Group Kinematics Commands

Syntax Example

```
long hndl;

hndl = KinematicsSetup(...);

PathJerk(hndl, 0.1);
```

9.111 PathMcToUu

Summary Convert machine coordinates to axis coordinates (in user units).

Syntax `PathMcToUu(hndl, pnt, array);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
pnt	=	**dpoint** variable containing the point to be converted.
array	=	Array of **double** values that will receive the converted coordinates. This array must be long enough to receive positions for all axes in the kinematics group defined by the KinematicsSetup command.

Return Value -

Description This command can be used convert machine coordinates directly into the corresponding axis coordinates (in user units) for the motor axes (see Basic Coordinate Systems).

See Working with Kinematics for more information on robots.

Command Group Kinematics Commands

Syntax Example

```
long   hndl;
dpoint pnt;
double AxisPos[3];

hndl = KinematicsSetup(...);

pnt.x = 100.0;
pnt.y = 200.0;
pnt.z = 300.0;

PathMcToUu(hndl, pnt, AxisPos);
```

9.112 PathMove

Summary Move to a specified point, using the selected kinematics.

Syntax `PathMove(hndl, trg, cs);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.

Summary Move to a specified point, using the selected kinematics.

trg = **dpoint** variable containing the target coordinate for each axis belonging to the kinematics.

cs = Coordinate system being used to specify the target point:

PATHCOORD_MACHINE (0) - The target coordinate is specified in the Machine Coordinate System.
PATHCOORD_WORK (1) - The target coordinate is specified in the Work Coordinate System.

Return Value -

Description

This command cannot be used with any motor that is *not* currently in kinematics mode (see KinematicsSetup).

If multiple-axis robot kinematics have been selected for a set of axes, then the **PathMove** command provides a simple way to move the end effector of the robot. The end effector will be moved to the target position and stopped.

Unlike the AxisPosAbsStart command which accepts raw axis positions, the **PathMove** command accepts positions specified in either the Machine or Work Coordinate System (see Basic Coordinate Systems). This allows application programs to work with a coordinate system that is much more convenient to use than raw axis positions.

If the robot is currently stopped when the command is executed, then the axes will be moved so the robot will move in a straight line from its current position to the target position. The robot will then stop. If the robot is moving when the command is executed, then the line will be adjusted to account for the existing movement at the start.

Movement along the line is done using a Limited Jerk movement. The velocity and acceleration behaviour is determined by the PathVel, PathAcc, PathDec, and PathJerk commands. These commands should be executed before the **PathMove** command is executed. However, unlike the Vel, Acc, and Dec commands, these commands do not need to be re-executed before each **PathMove** command.

If "cs" is 0 and the target is specified in Machine coordinates, then the PathVel, PathAcc, PathDec, and PathJerk values are interpreted as being in **Machine Coordinate System** units. If "cs" is 1 and the target is specified in Work coordinates, then the PathVel, PathAcc, PathDec, and PathJerk values are interpreted as being in **Work Coordinate System** units.

If "cs" is 0 and the target is specified in Machine coordinates, then dynamic transformations (see KinematicsWorkToMc) are not included in the movement. If the "cs" is 1 and the target is specified in Work coordinates, then support for continuously updated dynamic transformations is included in the movement.

If State Machines are being used, then the SM_PFG_PATHMOVEDONE event provides an easy mechanism for recognizing when the target position has been reached.

If State Machines are *not* being used, then application programs have the following options for recognizing when the **PathMove** is complete:

1. When the movement is complete, then the POSREACHED bit of the status flags SDO (0x2500/123) will be set on all axes belonging to the kinematics group. Note that the bit is set on all axes at the same time; it is only necessary to test one axis. Do *not* use the MOVING flag; this is *not* a reliable method of testing for the end a kinematics movement.
2. The STATBIT interrupt mechanism may be used, specifying the POSREACHED bit. Again, it is only necessary to test one axis. Do *not* use the MOVING bit; this is *not* a reliable method of testing for the end a kinematics movement.

At any time before the **PathMove** command has reached the target position, the movement may be stopped using the PathStop command. Note that if the command is already in the final deceleration stage towards the final target point, then PathStop will not be able to stop the movement any sooner.

When path movement is running, PathStop should be used in preference to other means of stopping movement (e.g. using the AxisControl command to turn off motors). This will give a much more controlled stop.

See Multi-axis Kinematics for more information on robot kinematics.

Command Group Kinematics Commands

Summary Move to a specified point, using the selected kinematics.

Syntax Example

```
long   hndl;
dpoint trg;

hndl = KinematicsSetup(...);

trg.x =  500.0;  // X axis
trg.y =  400.0;  // Y axis
trg.z = -100.0;  // Z axis

PathMove(hndl, trg, PATHCOORD_WORK);
```

9.113 PathNext

Summary Set the next multi-axis path curve to be used when the current path ends.

Syntax `PathNext(hndl, patharray, pathnum, trns);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
patharray	=	Path array.
pathnum	=	0-based path number of the path within the path array.
trns	=	**transform** variable containing the Path-to-Cyclic transformation to be used with this path.

Return Value -

Description

> This command cannot be used with any motor that is *not* currently in kinematics mode (see KinematicsSetup).

If multiple-axis robot kinematics have been selected for a set of axes and the PathStart command has been executed to follow a path, then the **PathNext** command will set the specified path as the *next* path to be followed once the current path has been completed. This allows multiple paths to be chained together.

> Usually, the Path-to-Cyclic transformation will be the same as the Path-to-Work transformation specified with the PathStart command.

See PathStart for details of the process of activating the next curve.

The **PathNext** command will always return immediately; it does not wait for the new path to actually be activated. Once **PathNext** returns, the user is free to modify or re-use the array without affecting the path that will be followed.

PathNext may be executed immediately following a PathStart command. It is not necessary to wait after executing PathStart before executing **PathNext**.

> If State Machines are being used, then various state machine events will be generated as the path is followed. This can be used, for example, to handle pen up/down requests while a path is being followed. See Profile Generator Events for more information.

Command Group Kinematics Commands

Syntax Example

```
dim long  ptharr[100];  // Downloaded from Path Editor.
transform PthToCyclic;

hndl = KinematicsSetup(...);

TranIdent(PthToCyclic);
TransTranslate(PthToCyclic, 500, 500, 0);

// Start following the first path in the path array.
```

Summary Set the next multi-axis path curve to be used when the current path ends.

```
PathStart(hndl, ptharr, 0, PthToCyclic, 0, PATHSTART_NORMAL);

// The next path with be path 1 (the second path in ptharr).

PathNext(hndl, ptharr, 1, PthToCyclic);
```

9.114 PathStart

Summary Start following a multi-axis path curve, based on the selected kinematics.

Syntax `PathStart(hndl, patharray, pathnum, trns, startpos, flags);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
patharray	=	Path array.
pathnum	=	0-based path number of the path within the path array.
trns	=	**transform** variable containing the Path-to-Work transformation to be used with this path.
startpos	=	Master position within the path curve of the point where synchronization is to start.
flags	=	Flags controlling the behaviour of the movement along the path. These flags may be or-ed together. **PATHSTART_NORMAL (0x00)** - Follow the path using a "normal" movement where movement along the path is controlled by the PathVel, PathAcc, PathDec, and PathJerk commands. Note that this flag value is 0x00 and therefore unnecessary; the value is provided in the hope that it can be used to improve program clarity. **PATHSTART_VIRTMAST (0x01)** - Follow the path using the virtual master of the first axis in the kinematics. **PATHSTART_STOP (0x02)** - Stop at each stop point specified in the path. This is ignored if a virtual master is being used. **PATHSTART_WAIT (0x04)** - Stop and wait at each stop point specified in the path. Note that it is not necessary to specify PATHSTART_STOP when using this flag. This is ignored if a virtual master is being used. **PATHSTART_IMMEDIATE (0x08)** - Start following the path immediately when the synchronization point is reached. Otherwise, stop (or stop and wait) when the synchronization point is reached. This is ignored if a virtual master is being used.

Return Value -

Description

> This command cannot be used with any motor that is *not* currently in kinematics mode (see KinematicsSetup). In addition, the motors must be stationary when the command is executed.

If multiple-axis robot kinematics have been selected for a set of axes, then the **PathStart** command does the following:

1. Extracts a copy of the selected path from the path array. Note that the path array is no longer needed after this. The user is free to modify or re-use the array without affecting the path that **PathStart** will follow.
2. Saves a copy of the Path-to-Work transformation. The transformation variable is no longer needed after this. The user is free to modify or re-use the variable without affecting the the running path. This transformation becomes the initial Cyclic-to-Work transformation (see below).
3. Creates a Path-to-Cyclic transformation and initializes it to the identity transformation. This transformation is used when movement goes beyond the end of the path (see below).
4. Creates a smooth "start" path that connects the current position, velocity, and acceleration of the motors with the starting point on the path where synchronization is to begin. This point is defined by the "startpos" parameter.

Summary Start following a multi-axis path curve, based on the selected kinematics.

5. Sets the value of the PFG_CWRAP parameter to 1 to indicate that the first path cycle has begun.
6. Begins to follow the "start" path. Note that if a virtual master is being used, then the controller does not autonomously move the axes along the path. Instead, movement of the axes will be driven by the virtual master axis associated with the *first* axis in the kinematics.

The PATHSTART command will always return immediately.

> After the **PathStart** command has been executed, the length of the "start" path can be queried using parameter PFG_CSTARTCLEN and the first axis specified with the KinematicsSetup command:
>
> ```
> hndl = KinematicsSetup(..., axsarr ...);
> PathStart(hndl, ...);
> length = AXE_PROCESS(axsarr[0],PFG_CSTARTCLEN);
> ```

Coordinate Matching

The fixed point coordinates in the path array are matched, in sequential order, to the coordinates in the Path Coordinate System. Note that the path coordinate configuration will mirror the coordinates of the Machine Coordinate System of the kinematics associated with the "hndl" parameter. See Multi-axis Kinematics for a description the appropriate Machine Coordinate System. The matching is done as follows:

1. Fixed points in the path array must always contain at least two coordinates. All Path Coordinate Systems will contain at least two spatial coordinates (X and Y). The first fixed point coordinate will be matched with the X coordinate of the Path Coordinate System and the second fixed point coordinate will be matched with the Y coordinate.
2. If the path array is defined to have 3 spatial coordinates and the Path Coordinate System is defined to be 3-dimensional, then the third fixed point coordinate is matched to the Z coordinate of the Path Coordinate System.
3. If the path array is defined to have 3 spatial coordinates and the Path Coordinate System is defined to be 2-dimensional, then the third fixed point coordinate (corresponding to the Z coordinate in the Path Editor) is *discarded*.
4. If the path array is defined to have 2 spatial coordinates and the Path Coordinate System is defined to be 3-dimensional, then the Z coordinate of the Path Coordinate System is set to 0.
5. All remaining fixed point coordinates will be matched 1-to-1 with the remaining coordinates in the Path Coordinate System. If there are more fixed point coordinates than Path Coordinate System coordinates, then the extra fixed point coordinates are *discarded*. If there are more Path Coordinate System coordinates than fixed point coordinates, then the extra Path Coordinate System coordinates are set to 0.

This matching scheme is intended to allow some flexibility when using path configurations that do not exactly match the selected kinematics. For example, it would allow a 2-dimensional character font path to be used with 3-dimensional kinematics. Otherwsie, it might be necessary to create both 2-dimensional *and* 3-dimensional versions of the same paths.

Normal Movement

If a normal movement is being executed (i.e. a virtual master is not being used), then movement along the path behaves as follows:

1. If no stops are expected, because
   - the PATHSTART_IMMEDIATE flag has been set and we are not stopping at the synchronization point
   - the path contains no stop points
   - PATHSTART_STOP and PATHSTART_WAIT flags have not been set
   - the path is specified as being cyclic

   then movement will increase to a constant velocity and continue at that velocity indefinitely (or until some other command interrupts the movement).
2. Otherwise, movement will slow down at stop at the first identified stopping point as follows:
   - the synchronization point if PATHSTART_IMMEDIATE has *not* been set and the synchronization point has not yet been passed
   - the next fixed point that has been flagged as being a stop point
   - the end of the path if the path is non-cyclic and there is no pending path

   Note that the first stopping point may be in the pending path if the PathNext command has been used.

Summary Start following a multi-axis path curve, based on the selected kinematics.

3. If movement stops at a stopping point and the PATHSTART_WAIT flag is not set, then movement will restart immediately. If the PATHSTART_WAIT flag is set, then movement will pause until the application executes the PathContinue command.
4. When a stopped movement is restarted, then it will stop again at the next stopping point (as identified using the rules above). If there is no next stopping point, then movement will increase to a constant velocity and continue at that velocity indefinitely.

Movement Using a Virtual Master

If a virtual master is being used to control movement along the path, then the PATHSTART_STOP and PATHSTART_WAIT flags are ignored. Stopping is expected to be controlled by the application's use of the virtual master.

If the virtual master axis causes the controller to move beyond the end of the path, then one of the following actions will be taken:

1. If a subsequent path has been specified using the PathNext command, then the new path will be automatically actived. Note that the new path is activated regardless of whether or not the current path is flagged as being cyclic. The Path-to-Cyclic transformation will be updated so that the starting point of the new path matches the end point of the old path and PFG_CWRAP will be incremented. The controller will then start following the new path. Note that the Path-to-Work transformation is now processed in two steps: the Path-to-Cyclic transformation followed by the Cyclic-to-Work transformation.
2. If the path has been flagged as a cyclic path and no subsequent path has been specified, then the current path will be repeated. The Path-to-Cyclic transformation will be automatically updated so that the start point of the next cycle matches the end point of the last cycle and PFG_CWRAP will be incremented.
3. If the path is not cyclic and no subsequent path has been specified, then all motors that belong to the kinematics will be stopped.

   > The motors will be stopped immediately *using the maximum possible torque*.

If the virtual master axis causes the controller to move before the start of the path, then one of the following actions will be taken:

1. If the path is cyclic, then the current path will be repeated. The Path-to-Cyclic transformation will be automatically updated so that the end point of the next cycle matches the start point of the last cycle and PFG_CWRAP will be decremented. Note that any subsequent path has been specified using the PathNext command is *ignored* when moving backwards along paths.
2. If the path is not cyclic, then all motors that belong to the kinematics will be stopped.

   > The motors will be stopped immediately *using the maximum possible torque*.

> If State Machines are being used, then various state machine events will be generated as the path is followed. These can be used, for example, to handle pen up/down requests while a path is being followed. See Profile Generator Events for more information.

At any time during movement along the path, the movement may be stopped using the PathStop command.

> When path movement is running, PathStop should be used in preference to other means of stopping movement (e.g. using the AxisControl command to turn off motors). This will give a much more controlled stop.

Command Group Kinematics Commands

Syntax Example

```
dim long  ptharr[100];  // Downloaded from Path Editor.
transform PthToWrk;

hndl = KinematicsSetup(...);

TransIdent(PthToWrk);
TransTranslate(PthToWrk, 500, 500, 0);

// Start following the first path in the path array.
```

Summary Start following a multi-axis path curve, based on the selected kinematics.
`PathStart(hndl, ptharr, 0, PthToWrk, 0, PATHSTART_STOP);`

9.115 PathStop

Summary Stop a running path movement (i.e. PathMove or PathStart).

Syntax `PathStop(hndl, position);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
position	=	Integer position along the curve at which to stop. Set this to -1 to stop as quickly as possible.

Return Value -

Description

> This command cannot be used with any motor that is *not* currently in kinematics mode (see KinematicsSetup).

If path movement is running, then the **PathStop** command will stop the movement. Note that movement will continue along the existing path until movement stops. The exact behaviour depends on how the original movement was started and the current state of the movement. The **PathStop** command will always return immediately.

> When path movement is running, **PathStop** should be used in preference to other means of stopping movement (e.g. using the AxisControl command to turn off motors). This will give a much more controlled stop.

PATHMOVE Movements

If a PathMove movement is running when **PathStop** is executed, then the "position" parameter is ignored. The movement will *always* be stopped as quickly as possible (i.e. as quickly as the original PathVel, PathAcc, etc., settings will allow it). Movement will continue in the straight line towards the target position while it is stopping. Note that if PathMove is *already* in the final deceleration phase towards the target position, then **PathStop** cannot stop any sooner and the target position will be reached.

A PathMove movement *cannot* be restarted with PathContinue. However, it can easily be restarted simply by executing another PathMove command.

PathStart Movements

If a PathStart movement is running when **PathStop** is executed, then the "position" parameter is used to specify exactly where on the curve to stop. If this is -1, then movement is stopped as quickly as possible. Otherwise, the parameter has the following meaning, depending on the current position along the path:

Current position is still within the start curve of the path:

The parameter is the arc length distance (in Path Coordinates) of the stop point *from the beginning of the start curve for the path*. This distance may be within the start curve or beyond the end of the start curve, extending into any number of cycles of the path curve itself. Note that the length of the start curve may be retrieved using parameter PFG_CSTARTCLEN (see PathStart).

Current position has reached or passed the synchronization point:

The parameter is the arc length distance (in Path Coordinates) of the stop point *from the beginning of the current active path curve* (i.e. from the first fixed point of the *current* cycle, regardless of how many cycles have already been completed). This distance may be beyond the end of the active curve and extend into any number of cycles of the path curve beyond the current cycle.

Please note the following important points:

1. Movement will continue to follow the current path until movement has stopped. If this causes the current path curve to cycle, then this is done in the normal way (including activating a pending curve if one has been specified).
2. If the controller has already passed the specified stopping position or it cannot stop movement without overshooting the specified stopping position, then the controller will stop as quickly as possible (i.e. as quickly as the original PathVel, PathAcc, etc., settings will allow it). The controller will *not* "back up" to get to the

Summary Stop a running path movement (i.e. PathMove or PathStart).

specified stopping position. This may also cause the current path to cycle if movement cannot be stopped before the end of the path.

3. If Stop points are enabled when movement was started, then all Stop points passed before the stopping point is reached, are handled in the normal way (i.e. as specified by the PathStart command). This allows, for example, sharp corners in the path to be negotiated.
4. Movement will always stop and wait at the requested stopping point, even if the PATHSTART_WAIT flag was *not* specified when the path was started.
5. Stopping positions greater than the length of the active curve are allowed and will cause movement to cycle onto subsequent paths in the normal way before stopping. Note that the normal behaviour for non-cyclic curves, is to stop at the end of the curve. For these curves, the command is basically ignored if a position beyond the end of the curve is specified; movement will stop at the end of the curve in any case.
6. Executing **PathStop** always has the effect of overwriting any previous **PathStop**.
7. If movement is already stopped and -1 is specified (or the position has already been passed), then the **PathStop** command is ignored. However, this will also clear any pending **PathStop** command. If the position has not yet been passed, then the specified position is set up as a pending stop position. If there is already a pending **PathStop**, then a new **PathStop** command will override the previous command.
8. Movement may be continued by using the PathContinue command.

> ⚠ **Using a Virtual Master**
>
> If movement was started with PathStart and a virtual master is being used, then all movement is expected to be controlled by the application's usage of the virtual master. In this case, the **PathStop** command is *discarded*; the application should simply stop the virtual master in whatever way is appropriate.
>
> This is necessary since the controller is unable to predict the required stopping behaviour. For example, the behaviour at stop points is unknown and the behaviour if an overshoot happens at the end of a non-cyclic curve is unknown.

Command Group Kinematics Commands

Syntax Example

```
dim long  ptharr[100];  // Downloaded from Path Editor.
transform PthToWrk;

hndl = KinematicsSetup(...);

TransIdent(PthToWrk);
TransTranslate(PthToWrk, 500, 500, 0);

// Start following the first path in the path array.

PathsStart(hndl, ptharr, 0, PthToWrk, 0, PATHSTART_STOP);

.....

PathStop(hndl, -1);    // Stop immediately.
```

9.116 PathUuToMc

Summary Convert axis coordinates (in user units) to machine coordinates.

Syntax `PathUuToMc(hndl, array, pnt);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
array	=	Array of **double** values containing the coordinates to be converted. This array must be long enough to contain positions for all axes in the kinematics group defined by the KinematicsSetup command.
pnt	=	**dpoint** variable that will receive the converted position.

Summary Convert axis coordinates (in user units) to machine coordinates.

Return Value -

Description This command can be used convert axis coordinates (in user units) for the motor axes directly into the corresponding machine coordinates (see Basic Coordinate Systems).

> If the current position is needed in machine coordinates, then Cpos commands may be executed, followed by the **PathUuToMc** command. However, the KinematicsCpoint command provides *much* more convenient method for doing this.

See Working with Kinematics for more information on robots.

Command Group Kinematics Commands

Syntax Example

```
long   hndl;
dpoint pnt;
double AxisPos[3];

hndl = KinematicsSetup(...);

AxisPos[0] = 10000.0;
AxisPos[1] = 20000.0;
AxisPos[2] = 30000.0;

PathUuToMc(hndl, AxisPos, pnt);
```

9.117 PathVel

Summary Set the maximum velocity for path movements.

Syntax `PathVel(hndl, vel);`

Parameter

hndl	=	Handle to the multi-axis kinematics. This is returned by the KinematicsSetup command.
vel	=	**double** value specifying the maximum velocity in units per second.

Return Value -

Description When a PathMove command is executed, or a PathStart command without the virtual master is executed, then movement along the line or curve is done using a Limited Jerk movement based on arc length along the line or curve. The **PathVel** command is used to specify the maximum velocity to be used.

> **PathMove**
>
> The PathMove command can specify a target position in either the Machine Coordinate System or the Work Coordinate System (see Basic Coordinate Systems). If the Machine Coordinate System is used, then the **PathVel** value is interpreted as being in **Machine Coordinate System** units. If the Work Coordinate System is being used, then the **PathVel** value is interpreted as being in **Work Coordinate System** units. Note that this means the **PathVel** value must be respecified if PathMove is being used with different coordinate systems.
>
> **PathStart**
>
> When the PathStart command is being used, then the **PathVel** value is *always* interpreted as being in **Work Coordinate System** units.

See Multi-axis Kinematics for more information on robot kinematics.

Command Group Kinematics Commands

Syntax Example

```
long hndl;

hndl = KinematicsSetup(...);

PathVel(hndl, 25.0);
```

9.118 Pdo

Summary Pseudo array for direct access to the CANopen PDOs.

Syntax Pdo[*n*]

Parameter *n* = 1001 for first 4 bytes of PDO1, 1002 for second 4 bytes of PDO1.

2001 for first 4 bytes of PDO2, 2002 for second 4 bytes of PDO2.

*m*001 for first 4 bytes of PDO*m*, m002 for second 4 bytes of PDOm.
Where *m* is the PDO number.

For backwards compatibility, the following are also supported:

n = 1 for first 4 bytes of PDO1, 2 for second 4 bytes of PDO1.

The maximal available PDOs is defined in the Controller Dependent Feature List

Return Value -

PDO Activation (enable / disable)

Just the PDO 1 (RxPDO = 0x200 + Node-ID / TxPDO = 0x180 + Node-ID) is enabled by default according to the CANopen specification. If other PDOs are required, these have to be enabled by direct setting the corresponding "Valid" bit (0x1400 - 0x1404 resp. 0x1800 - 0x1804, subindex 1) or by the configuration of mapped objects using LinkSdo or LinkPdo, which set the "Valid" bit of the corresponding PDO automatically.

CANopen PDO size

For CAN Bus, a PDO can be maximal 8 bytes long. It can therefore hold a maximum of 8 objects.

EtherCAT PDO Size

For EtherCAT, a PDO can be longer. See Controller Dependent Feature List

The PDO sizes can be different for Tx and RxPDOs.

> The summed up sizes of all EtherCAT PDOs must be an even amount of bytes! Otherwise, a COE emergency will be issued.

Description The **Pdo** pseudo array enables direct access to the CANopen data process objects.

During reading of a PDO (reaction to an incoming PDO) there are two possibilities:

1. After the command LinkPdo, the incoming telegram will always be diverted in the appropriate system variable. The "Valid" bit of the PDO is set automatically.
2. The program itself accesses and reads the PDO array.

In the case of outgoing PDO, you should differentiate between two processes:

1. After the command LinkSdo the PDO will be written in as soon as the variable changes. Cyclical update of the parameter every 10 ms. This default value can be changed via SDO entry 0x1800 - 0x1804 subindex 5 in accordance with the CAN Open specification. The “Valid” bit of the PDO is set automatically.
2. The command PDO writes directly in the outgoing PDO.

The first PDO array element (e.g. PDO[1001]) contains the data bytes 1 - 4 of the maximum possible 8 data bytes of the PDO in case of a CANopen PDO. The second PDO array element (e.g. PDO[1002]) contains the data bytes 5 - 8. More PDO array elements are just available for special systems or enhanced interfaces, e.g. the “serial” PDO 5.

Every PDO array element (e.g. PDO[1002, PDO[1002]) contains a 32-bit value. But the byte order within the 32 bit value must be attended when evaluating. The best way to check the corresponding location is with byte- (.ub) or word-wise (.uw) access to the PDO array element.

PDO[1001].ub1 -> Byte 1 of PDO
PDO[1001].ub2 -> Byte 2 of PDO
PDO[1001].ub3 -> Byte 3 of PDO
PDO[1001].ub4 -> Byte 4 of PDO

Summary Pseudo array for direct access to the CANopen PDOs.

PDO[1002].ub1 -> Byte 5 of PDO
PDO[1002].ub2 -> Byte 6 of PDO
PDO[1002].ub3 -> Byte 7 of PDO
PDO[1002].ub4 -> Byte 8 of PDO

The commands LinkPdo and LinkSdo link an internal system variable of the MACS control unit directly to a PDO and set the "Valid" bit of this PDO.

Even when mapping was configured (by LinkPdo or LinkSdo), you can still access the PDO by the PDO array elements. However, you must take into consideration that only bytes should be directly written into the PDO-array, which are not in use by the mapping configuration. Otherwise the integrity of data is not guaranteed!

As standard, a changed PDO content is automatically dispatched (asynchronous operating mode). If this is not desired, then the SDO Index 0x1800 - 0x180n subindex 2 can be set to another value (e.g. 254, instead of the standard 255). Thereby, active sending no longer takes place automatically, but the PDO must be collected per remote frame instead.

Of course, the PDO mapping can also be configured by a master device (e.g. PLC, PC) using the standard CANopen procedures and the mapping objects 0x1600 - 0x1604 resp. 0x1A00 - 0x1A04. In that case only SDOs can be mapped, but none of the MACS internal control data, which can just be accessed by internal SYSVAR numbers.

In the same manner, it is also possible to enable or disable PDOs 1-n by modifying the "Valid" bit of subindex 1 of the objects 0x1400 - 0x140n and 0x1800 - 0x180n.

Subindices 2-5 of 0x1800 can be used to define transmission type, inhibit time, and event time of the TxPDOs. The defined transmission type of the TxPDO is also used for the RxPDO by MACS control units.

RxPDOs do not use extra CAN memory, but TxPDOs need one object per TxPDO, if they are enabled, i.e. the "Valid" bit is set.

Command Group CANopen Slave, EtherCAT Master

Syntax Example

```
data = Pdo[1001];       // PDO1 data bytes 1-4
data = Pdo[1001].i1;    // Second bit of PDO1 data bytes 1-4
data = Pdo[1002].ub0;   // First byte of PDO1 data bytes 5-8
Pdo[1001] = data;       // Set bytes 1-4 of PDO1
Pdo[1001].i2 = 1;       // Set third bit of byte 1 of PDO1
```

Program Sample

```
if (Pdo[1001] == 256)   // Test data bytes 1-4
    if (Pdo[1002].i2)   // Second bit of data bytes 5-8 high
        {
        ...
        }
```

9.119 #pragma

#pragma statements are used to configure the compiler in various ways. They are executed by the compiler at compile time when the compiler encounters them in the source code. They may have an influence on the compiled binary code but they do not themselves become part of the binary code. #pragma statements are usually located towards the beginning of application programs.

The following #pragma statements are currently available:

#pragma once

If this statement is specified anywhere within an include file (see #include), then that include file will be included only once in an application program when it is first referenced by an #include statement. Subsequent references to the same file by other #include statements will be ignored. This is an easy and very efficient method of preventing multiple inclusions of the same include file (for example, by nested include files).

#pragma CAM_MAXFIXPOINT

When an application program will be using CAM curves (see Editing CAM Curves for more information), the controller must first preallocate internal memory before it can work with the curves. To be able to do this, the controller must know the maximum size of any CAM array that the application program will be using. This can be done either implicitly using the CamArrayLen function in the program or explicitly using this #pragma statement. The syntax is:

```
#pragma CAM_MAXFIXPOINT n
```

where "*n*" is either an integer constant or an expression that will evaluate to an integer constant. Note that this specifies the maximum number of fixed points in any CAM array and not the maximum length of the CAM array itself. This #pragma may be specified only once in an application program. An error will be generated if it is specified more than once. An error will also be generated if it specifies a maximum count that is smaller than a previously used CamArrayLen function.

The maximum fixed point count is determined as follows:

1. If there is a #pragma, then that is the fixed point count.
2. If there is no #pragma, then the count is the maximum of all CamArrayLen functions used.
3. If there is no #pragma and no CamArrayLen function was used, then the count is set to 100.

> Prior to firmware version 7.0.83, all CAM arrays were limited to a maximum of 100 fixed points. For firmware versions prior to 7.0.83, the #pragma statement may exist but it will be ignored.

#pragma DEPRECATEDWARNINGS

DEPRECATEDWARNINGS allows deprecated warning messages generated by the compiler to be suppressed. The syntax is:

```
#pragma DEPRECATEDWARNINGS value
```

where "*value*" is either `OFF`, `ON`, `0`, or `1`.

#pragma FILE_IDENT

FILE_IDENT enables "File Identification Support" for the application program. Note that this #pragma may appear in the main application source file or in *any* include file. Support may be enabled in *all* application programs by using **Settings** → **Compiler**. For a description of "File Identification Support", see **Development** → **Compare to Controller**.

#pragma PNT_LENGTH

If multi-dimensional kinematics (see Working with Kinematics) are being used with non-spatial coordinates, then this #pragma statement must be used to specify the length of **dpoint** variables. See Working with Points for more information. The syntax is:

```
#pragma PNT_LENGTH n
```

where "*n*" is either an integer constant or an expression that will evaluate to an integer constant. The valid range is 3-10. This specifies the total number of coordinates that will be stored in a **dpoint** variable. Note that this must include a Z coordinate even if 2-dimensional kinematics are being used. In the case where no #pragma statement is used, the number of coordinates is set to 3.

PNT_LENGTH may be set multiple times, however only the largest value is used. PNT_LENGTH may not be set after the first **dpoint** variable has been declared.

#pragma SmConfig

The `SmConfig` statement allows the user to configure various aspects of the behaviour and memory

usage of state machines. See Specifying the State Machine Configuration for a detailed description of the `SmConfig` statement.

9.120 print

Summary Information output.

Syntax
```
print();
print(value,...);
```

Parameters value = List of one or more values to be printed, separated by commas.

Return Value -

Description If the ApossIDE is open and the controller connection is active, then the **print** command can be used to display calculation results, variable values, text strings, etc., on the PC.

Any of the following values may be printed:

- **long** constants, variables, and expressions.
- **double** constants, variables, and expressions.
- **wchar** constants, variables, and expressions.
- Single elements of **long**, **double**, and **wchar** arrays.
- String literals (i.e. text strings enclosed in double quote characters).
- Text strings contained in **wchar** and **long** arrays.
- chr function.
- radixstr function.

The list of values will be printed *without* space characters separating them. If values are to be separated, then a string literal (e.g. " ") must be placed between each value in the list.

A line feed is automatically generated after the last value in the list has been printed. To print without a line feed, use printf.

> Maximum number of text strings is limited to 300 characters.

> The "print()" function can be used with an empty value list. This will generate a line feed only.

Single character values (i.e. **wchar** values and individual elements of "wchar" arrays) will be printed as **long** values and not as characters.

Integer values representing ascii characters (e.g. 59 for ';') can be printed using the chr function. This is particularly useful for non-printable characters such as tabs, line feeds, etc. See examples below.

Integer values are usually printed as decimal numbers (i.e. base 10). The radixstr function can be used to print these values using a different base (e.g. hex or binary). See examples below.

Text strings contained in arrays may be printed simply by specifying the name of the array. All characters will be printed up until the first NULL-terminator (i.e. 0) is found or the end of the array is found, whichever comes first. See examples below.

> ANSI text strings (i.e. 8-byte, packed, text strings) must be handled specially if they are to be printed (see ANSI Arrays).

> **long** arrays containing text strings may also be printed. The array is expected to conatin one character per array element.

Each **print** statement is limited to a maximum of 300 characters. Longer lines may be printed using printf.

> Various aspects of the printing functionality (e.g. target device, print enable/disable, etc.) can be controlled using the printcontrol command.

Command Group Debug

Syntax Examples
```
long  arr[10], ax = 0;
```

Summary Information output.

```
wchar dash[6] = "-----";

print("Axis information:");                    // Print title
print("Axis Module = ",ax);                    // Axis module
print("   Pos   = ",Apos(ax));                 // Axis position
print("   Flags = 0x",radixstr(Stat(ax), 16)); // Axis flags
print("   Array = ",arr[ax]);                  // Array value
print(dash);
```

Program Sample See **File** → **Sample**: Printing.mc

9.121 printcontrol

Summary Controls aspects of the printing functionality.

Syntax

```
printcontrol(DEV, dev);
printcontrol(RINGBUF, arrayName);
```

Parameter

dev = number for the print device
-1 = no output after that line
0 = standard output (print to ApossIDE print buffer)
2 = serial (see SerialSetup command)
3 = standard output, blocking (print to ApossIDE print buffer). Print commands will block when print buffer is full.
4 = serial NOWAIT

arrayName = Name of the array to be used as ring buffer for logging service

Return Value -

Description **DEV**

Stops (-1) or redirects the print information output.

printcontrol(DEV, -1); can be used to disable all prints in a program without commenting them out one by one.

> The instruction printcontrol(DEV, -1); is effective immediately and for all PRINT commands, that follow after that command.

The print device DEV_SER_NOWAIT = 4 is identical to the device DEV_SER = 2 but it does not wait if a transmission is still going on. In that case,the function simply returns without sending the data.
This should avoid execution problems with the release software. Whenever the serial line would be overloaded, the application does not wait or hang.

> The default setting is 0.

RINGBUF

This command can be used to configure an array as ringbuffer for log messages. See Logging Service for details.

Command Group Debug

Syntax Example

```
printcontrol(DEV, -1);
print("This print never appears on a console");

printcontrol(DEV, 0);
print("You have activated the print message again");
```

9.122 printf

Summary Formatted Information output.

Syntax `printf(format,...);`

Parameter

format = format string

arg = value arguments

Return Value -

Description This command prints a formatted string that may include optional format specifiers embedded within it. Each successive format specifier will be replaced by the value of the corresponding argument following the format string. See the description of the print command for the general behaviour of this command.

> Maximum number of parameters is limited dependent on the Controller type. There is also a maximum length of buffer per parameter, including the \0 character. Please check Controller Dependent Feature List for the limits.

Format specifiers are structured as follows:

%[flags][width][.precision][length]specifier

> Since the standard datatype in the controller is a "long", the length specifier 'l' should *always* be included for integer arguments (d, i, u, o, x, etc.). For floating point numbers, an 'L' should *always* be included. See examples below.

specifier	Output	Example
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
c	Character	a
s	String of characters. Max length 30 on a MACS5, 150 on a MasterMACS). This may be either a text string enclosed in double quotes or a STRING array variable followed by "[]" (see the examples below). See ANSI Arrays for how to handle ANSI characters strings.	"sample"
%	A % followed by another % character will write a single % to the stream.	%

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *.precision* and *modifiers* (in that order), which are optional and follow these specifications:

flags	description
-	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or -) even for

Summary Formatted Information output.

positive numbers. By default, only negative numbers are preceded with a - sign.

(space) If no sign is going to be written, a blank space is inserted before the value.

`#` Used with `o`, `x` or `X` specifiers the value is preceeded with `0`, `0x` or `0X` respectively for values different than zero.
Used with `a`, `A`, `e`, `E`, `f`, `F`, `g` or `G` it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.

`0` Left-pads the number with zeroes (`0`) instead of spaces when padding is specified (see *width* sub-specifier).

width	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width*format string, but as an additional integer value argument preceding the argument that must be formatted.

.precision	description
`.`*number*	For integer specifiers (`d`, `i`, `o`, `u`, `x`, `X`): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of `0` means that no character is written for the value `0`. For `a`, `A`, `e`, `E`, `f` and `F` specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For `g` and `G` specifiers: This is the maximum number of significant digits to be printed. For `s`: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, `0` is assumed.
`.*`	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that must be formatted.

The *length* sub-specifier modifies the length of the data type. An 'l' (lowercase L) specifier should be used for integer types since the controller uses "long" types for its integers. For floating point values, an 'L' should be used.

(Source: www.cplusplus.com, 15.4.2014)

Command Group Debug

Syntax Examples

```
wchar strarr[20] = "Some text";
wchar answer[2][10] = { "Yes", "No" };

printf("long number:  %ld", 123456789);
printf("hex number:   %lx", 15);
printf("float number: %Lf", 3.124);

printf("Text string: %s", strarr);
printf("The answer is %s.", answer[0]);
```

Program Sample See **File** → **Sample**: Printing.mc

9.123 RecordDest

Summary Specifies the memory location where recorded data is to be saved.

Syntax `RecordDest(arrayname);`

Parameter

arrayname	=	Name of the DIM array used for the recording or DYNMEM for recording data into free memory.

Return Value -

Description **RecordDest** defines where the recorded data is to be saved. The following options are available:

- A global DIM array.
- Dynamic (free) memory.

If no **RecordDest** command is executed before starting the recording, then DYNMEM is used as the default destination.

> If a DIM array is being used to save the recorded data, then it is important that the array size be defined large enough for the amount of data that is expected to be recorded. Please see "Record" Array Structure to determine the required array size. If the given memory does not have sufficient space for the requested number of measurements, then Error 71 or Error 94 will be generated during program execution.

> An advantage of using DYNMEM is that it eliminates the necessity of calculating the required array size.

A complete configuration of data recording requires these commands:

RecordIndex	Defines the variables that will be recorded.
RecordDest	Defines the destination array where the data will be recorded.
RecordTime	Defines the sample interval (i.e. the amount of time between samples).
RecordTimeBase	(Optional) Defines the time base (i.e. the cycle time).
RecordType	Defines the type of recording (cyclic or non-cyclic, etc.).
RecordStart	Defines the number of samples to be recorded and starts the actual recording.

Once the recording is complete, the data can be uploaded and viewed using **Tools** → **Oscilloscope (Record)**.

Command Group Data Recording Commands

Program Sample See **File** → **Sample**: Recording.mc

9.124 RecordIndex

Summary Specify system variables for data recording.

Syntax `RecordIndex(svi1, svi2, svi3,...);`

Parameter

svi	=	Sysvar indices of up to 20 system variables to be recorded.

Return Value -

Description **RecordIndex** configures the set of system variables that are to be captured during data recording. These are specified using Sysvar indices.

> The number of variables being recorded will have a significant affect on the size of the memory required for data storage. Please see "Record" Array Structure.

> Executing the **RecordIndex** command will cause all previous recordings (if any) in the destination array to be cleared the next time that recording is started.

Summary Specify system variables for data recording.
A complete configuration of data recording requires these commands:

RecordIndex	Defines the variables that will be recorded.
RecordDest	Defines the destination array where the data will be recorded.
RecordTime	Defines the sample interval (i.e. the amount of time between samples).
RecordTimeBase	(Optional) Defines the time base (i.e. the cycle time).
RecordType	Defines the type of recording (cyclic or non-cyclic, etc.).
RecordStart	Defines the number of samples to be recorded and starts the actual recording.

Once the recording is complete, the data can be uploaded and viewed using **Tools** → **Oscilloscope (Record)**.

> **Tools** → **Oscilloscope (Single Shot)** can be used to configure and start data recording online without the need for these commands within the application program code. However, the advantage of using these commands directly in the program code, is that the data recording may be started at exactly the desired line of code in the program.

Command Group Data Recording Commands

Syntax Example
```
RecordIndex(0x01250001, 0x01250004, 0x01220217);
// or
RecordIndex(AXE_PROCESS_INDEX(0,REG_ACTPOS));
```

Program Sample See **File** → **Sample**: Recording.mc

9.125 RecordStart

Summary Starts the data recording.

Syntax `RecordStart(n);`

Parameter

n = 0	=	Record the maximum number of samples that will fit into the defined memory section (array or DYNMEM).
n > 0	=	Number of samples to be recorded.

> If the given memory section (array or DYNMEM) does not have sufficient space for the defined number of samples (n > 0), then Error 71 or Error 94 will be triggered during program execution. Therefore it is recommended that **RecordStart**(0) be used. This will save the maximum amount of data and then stop without generating an error.

> If a recording was stopped (by RecordStop, or **Esc** is pressed, or in case of a single shot recording, or reaching the defined number of samples) and another RecordIndex and a following **RecordStart** is executed, then the new data recording fills up the configured memory (array or DYNMEM) right from the beginning again. This means, that all old data recordings in the same memory area (array or DYNMEM) are overwritten and lost.

If a next **RecordStart** is executed without a RecordIndex definition in between, then the new data is appended to the last data recording. The configured memory (array or DYNMEM) holds more than one recording then.

Return Value -

Description This command is used to start data recording according to the configuration made by the last RecordIndex, RecordDest, RecordTime, RecordType commands. The recorded data can be read out and visualized afterwards using **Tools** → **Oscilloscope (Record)**.

If not all parameters are completely specified before the **RecordStart** command is executed, the following default settings are in use for data recording:

- Default sampling period: 1 ms
- Default type of recording: one time

Summary Starts the data recording.

- Default memory section for data storage: DYNMEM
- Default system variables for recording: 0x01250001 (ACTPOS), 0x01250002 (COMPOS)

It is also possible to gather more than one recording in one array. For example, it is possible to start a recording of 1000 samples, then stop the recording and then start another 500 samples. If then the data are read out with the ApossIDE, two recordings will be seen, that you can look at. If a restart from scratch is desired, then a new RecordIndex command must be executed.

Command Group Data Recording Commands

Syntax Example

```
// Configure system objects for recording:
// Actual position, Command position, Tracking Error, system timer
RecordIndex(0x01250001, 0x01250002, 0x01250006, 0x01220217);
RecordStart(0);       // Start recording (until RecordStop or DYNMEM is filled up)

Vel(0, 50);                 // 50% of maximum velocity
AxisPosAbsStart(0, 10000);  // Start positioning
AxisWaitReached(0);         // Wait until position is reached
Vel(0, 100);                // Increase velocity to 100%
AxisPosAbsStart(0, 0);      // Use new velocity for moving back
AxisWaitReached(0);         // Wait until position is reached
Delay(200);                 // Wait 200ms
RecordStop(0, 0);            // Stop recording
```

Program Sample See **File** → **Sample**: Recording.mc

9.126 RecordStop

Summary Stops the data recording.

Syntax `RecordStop(method, param);`

Parameter

method = How stop should take place:
0 = Stop immediately

param = Parameter for stopping method. Currently, this must be set to 0.

Return Value -

Description This command stops the data recording in the configured way.

> Currently, the only valid command is: **RecordStop**(0,0);

> If a recording was stopped by **RecordStop** and another RecordIndex and a following RecordStart is executed, then the new data recording fills up the configured memory (array or DYNMEM) right from the beginning again. This means, that all old data recordings in the same memory area (array or DYNMEM) are overwritten and lost.

> For applications, which use cyclic recording, it is recommended to insert a **RecordStop** command at the end of the program or if the program quits in case of an error. Otherwise the recording still runs on even if the program is not executing anymore.

Command Group Data Recording Commands

Syntax Example

```
// Define an error handler
InterruptSetup(ERROR, ErrHandler);

// Define all necessary data recording settings:
RecordDest(DYNMEM);   // Use DYNMEM for data recording
RecordType(1);        // Configure cyclic recording
// Configure system variables used for data recording:
// ActPos, TrackError, Input 1-8, Output 1-8
RecordIndex(0x01250001, 0x01250006, 0x01220202, 0x0122020A);
```

Summary Stops the data recording.

```
RecordStart(0);          // Start recording (until RecordingStop)

// Program main loop
while (1)
    {
    // Execute the application code
    }

void ErrHandler(void)  // Error handler
{
    RecordStop(0,0);     // Stop recording
    Exit(1);           // Quit program
    return;            // End of error handler
}
```

Syntax Sample This example gathers data continuously until a special condition is reached and recording is stopped. If you then read out the data, you will get the last sampled data before the stop occurred. The amount of data is dependent on the available dynamic memory.

```
#define PFG_ACTPOS        0x01250001
#define PFG_COMPOS        0x01250002
#define PFG_VCMDSIGNED    0x012500B5

...

RecordIndex(PFG_ACTPOS, PFG_COMPOS, PFG_VCMDSIGNED);
RecordDest(DYNMEN);     // select dynamic memory for recording
RecordTime(5);     // select a sampling period of 5 ms
RecordType(1);     // select cyclic recording
RecordStart(0);       // start recording

...

if (whatever) RecordStop(0,0); // stop recording
```

This way of recording could also be used without changing your program at all by using the Single Shot Oscilloscope.

Program Sample See **File** → **Sample**: Recording.m

9.127 RecordTime

Summary Defines the sampling period for data recording.

Syntax `RecordTime(cycles);`

Parameter cycles = Sampling period in multiple of cycle times (default 1ms).

Return Value -

Description This command defines the sampling period for data recording, that means the time period with which the configured recording data (RecordIndex) is recorded.

The sampling period depends both on RecordTime and RecordTimeBase.
The principle is: sampling period = RecordTime x RecordTimeBase.

The default for RecordTimeBase is 1ms. That means, if RecordTime is set to a value of 2, the sampling period will be 2ms.

The minimum setting is 1.

If a RecordStart is executed after a **RecordTime** command, then the period given by the **RecordTime** command is valid. The last time setting before the RecordStart is the one which is always used. If no **RecordTime** is executed before the RecordStart, then the default value of 1 ms is in use.

Summary Defines the sampling period for data recording.
A complete configuration of data recording requires these commands:

RecordIndex	Defines the variables that will be recorded.
RecordDest	Defines the destination array where the data will be recorded.
RecordTime	Defines the sample interval (i.e. the amount of cycles between samples).
RecordTimeBase	(Optional) Defines the time base (i.e. the cycle time).
RecordType	Defines the type of recording (cyclic or non-cyclic, etc.).
RecordStart	Defines the number of samples to be recorded and starts the actual recording.

Once the recording is complete, the data can be uploaded and viewed using **Tools** → **Oscilloscope (Record)**.

Command Group Data Recording Commands

Program Sample See **File** → **Sample**: Recording.mc

9.128 RecordTimeBase

Summary Selects the time base to be used for the recording (optional)

Syntax `RecordTimeBase(loopSelector);`

Parameter loopSelector = One of the following defines:

RECORD_LOOP_PFG
RECORD_LOOP_POS
RECORD_LOOP_VEL
RECORD_LOOP_CUR

Return Value -

Description RecordTimeBase defines the calling loop in the firmware to be used for the oscilloscope recording.

Depending on the controller type, different loops are existing with different cycle times.

All controllers support RECORD_LOOP_PFG, which is the default and is usually 1 ms cycle time.

But some controllers also have additional loops like velocity controller loop and current controller loop. There it is possible to use these loops to have faster recording to e.g. record the motor current.

The resulting sample time of the oscilloscope is the period of RecordTimeBase x RecordTime.

A complete configuration of data recording requires these commands:

RecordIndex	Defines the variables that will be recorded.
RecordDest	Defines the destination array where the data will be recorded.
RecordTime	Defines the sample interval (i.e. the amount of cycles between samples).
RecordTimeBase	(Optional) Defines the time base (i.e. the cycle time).
RecordType	Defines the type of recording (cyclic or non-cyclic, etc.).
RecordStart	Defines the number of samples to be recorded and starts the actual recording.

Once the recording is complete, the data can be uploaded and viewed using **Tools** → **Oscilloscope (Record)**.

Portability This feature can be controller dependent and is listed in the Controller Dependent Feature List.

Command Group Data Recording Commands

Summary Selects the time base to be used for the recording (optional)

Program Sample See **File** → **Sample**: Recording.mc

9.129 RecordType

Summary Defines the type of data recording, i.e. one time recording or cyclic recording

Syntax `RecordType(type);`

Parameter type = 0 = one time recording (= default setting)
1 = cyclic recording

If no type is set before a RecordStart is executed, then the default (one time recording) is used.

> A cyclic recording even proceeds when a program is finished or stopped by an error. The recording just stops, if the command RecordStop is executed or an user break (= pressing **Esc**) is taking place.

> A cyclic recording always overwrites the given memory as soon as the memory limit or the defined number of samples is reached. The actually stored number of samples depends thereby on the size of the memory (RecordStart(0)) or the defined number of samples (RecordStart value > 0).

Return Value -

Description This command specifies, if a configured data recording is executed just once or runs endless in a cyclic mode in the background until a RecordStop is executed.

If cyclic recording is activated and RecordStart(0) is defined, the available memory space is filled, and is overwritten when full. If a defined number of samples is specified (i.e. RecordStart with value > 0) , just the corresponding memory section is used. If there are more samples than available space the oldest recording is overwritten.

A one time recording (**RecordType**(0)) stops, when:

- the defined number of samples is recorded (RecordStart value > 0)
- the specified memory (array or DYNMEM) is filled up (RecordStart(0))
- the command RecordStop is executed
- the program execution is stopped by the user by pressing the **Esc** key

A cyclic recording (**RecordType**(1)) stops, when:

- the recording is stopped by the command RecordStop
- the program execution is stopped by the user by pressing the **Esc** key

The cyclic recording can be used very well for debugging purposes. Especially in case of irregular and rare machine misbehavior. The usage of cyclic recording, DYNMEM and RecordStart(0) gives the power to track a maximum number of information for an endless period of time. When an error or interrupt occurs or a special event happens, the recording can be stopped by the RecordStop command. The recorded information can then be read out, evaluated and saved to disk using **Tools** → **Oscilloscope (Record)**.

Command Group Data Recording Commands

Syntax Example

```
// Define an error handler
InterruptSetup(ERROR, ErrHandler);

// Define all necessary data recording settings:
RecordDest(DYNMEM);   // Use DYNMEM for data recording
RecordType(1);        // Configure cyclic recording
// Configure system variables used for data recording:
// ActPos, TrackError, Input 1-8, Output 1-8
RecordIndex(0x01250001, 0x01250006, 0x01220202, 0x0122020A);
RecordStart(0);          // Start recording (until RecordingStop)

// Program main loop
while (1)
```

Summary Defines the type of data recording, i.e. one time recording or cyclic recording

```
    {
    // Execute the application code
    }

void ErrHandler(void)  // Error handler
{
    RecordStop(0,0);     // Stop recording
    Exit(1);           // Quit program
    return;            // End of error handler
}
```

Program Sample See **File** → **Sample**: Recording.mc

9.130 ResetOrigin

Summary Erase temporary zero point.

Syntax `ResetOrigin(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description A temporary zero point set previously with SetOrigin can be removed with the **ResetOrigin** command. After this, all absolute positioning commands (AxisPosAbsStart, AxisLinAbsStart, etc.) will again refer to the real zero point.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Axis and Information Commands

Syntax Example

```
ResetOrigin(0);        // Reset temp zero point Axis mod 0
ResetOrigin(0, 2);     // Reset temp zero point Axis mods 0 and 2
ResetOrigin(AXALL);    // Reset all temp zero points
```

Program Sample See **File** → **Sample**: AbsoluteRelative.mc

9.131 Save

Summary Save DIM arrays, axis parameters, global parameters, or user parameters in permanent memory.

Syntax `Save(part);`

Parameter part = ARRAYS - DIM arrays.
GLBPARS - Global parameters.
USRPARS - User parameters.

Return Value -

Description If the content of DIM arrays is modified at runtime or if axis, global, user parameters is modified at runtime, then it is possible to save the changes permanently in the flash memory of the controller's (i.e. the content is restored after a power off/on cycle).

> Saving ARRAYS only saves DIM arrays permenently. Global or local arrays cannot be saved.

> The flash memory can only handle execution of this command up to 10000 times.

Command Group Memory Commands

9.132 SdoRead

Summary Reads SDO of a connected CANopen device.
If id is > 1000000, an SDO of a connected EtherCAT slave will be read.

Syntax
```
res = SdoRead(id, index, subindex);
```

Parameter

id	=	CAN id (1...127, 100001...100127) offset of 100000 for nodes linked to the 2nd CAN bus (If no 2nd CAN bus is present, the 1st CAN bus is used instead) offset of 1000000 for connected EtherCAT slaves
-id	=	executes the command without waiting for the answer
index	=	index of object (0x0000...0xFFFF)
subindex	=	subindex (0x00 - 0xFF)

Return Value Value of the SDO with index and subindex.

Description This command allows reading SDO of a connected CANopen device.

After execution of the **SdoRead** command, the value of the slave's object is returned. If a CANopen error happens (e.g. timeout, unknown object) an error will be reported. The error can be handled by an ERROR interrupt routine if one has been defined.

It is possible to call **SdoRead** with negative CAN-Id numbers. Then the command is executed but it will not wait for the answer. However, **SdoRead** does not return a meaningful result in such a case. Therefore use SdoStatus to check the result of an active communication.

Remark: Timeout = 300 ms
The default timeout setting for **SdoRead**, SdoWrite, and SdoWriten commands is 300 ms. If a CANopen devices has a slower reaction time, it is possible to modify the timeout limit by the sysvar[0x01220240]:
sysvar[0x01220240] = 500 // SDO 0x2202/64 = 500 ms

> For reading SDO values on the controller itself (i.e. not a connected device), the Sysvar array should be used. For calculation of the corresponding Sysvar array index, the following formula can be used:
> Sysvar Array-Index = 0x01 << 24 + SDO-Index << 8 + SDO-Subindex
> Example: SDO 0x2202/25 can be accessed by Sysvar[0x01220219].

Command Group Communication CANopen Commands

Syntax Example 1 Read out the position data of a CANopen encoder (DS406) at node id 2.

```
encvalue = SdoRead(2, 0x6004, 0);
```

Syntax Example 2
```
// Syntax example for a MACS controller with 2 CAN buses:
// The CANopen slave devices is connected to the 2nd CAN bus,
// so add bus offset of 100000 to the node id.
encvalue = SdoRead(100002, 0x6004, 0);
```

9.133 SdoReadSeg

Summary Segmented read of SDOs (unpacked).

Syntax
```
res = SdoReadSeg(id, index, subindex, arrayname);
```

Parameter

id	=	CAN id number offset of 100000 for nodes linked to the 2nd CAN bus (If no 2nd CAN bus is present, the 1st CAN bus is used instead) offset of 1,000,000 for nodes on EtherCAT bus
-id	=	executes the command without waiting for the answer
index	=	SDO index (0x0000 ... 0xFFFF)
subindex	=	SDO sub index (0x00 ... 0xFF)
arrayname	=	name of any existing array (only DIM arrays are supported)

Return Value No. of readout bytes

Summary Segmented read of SDOs (unpacked).

In the array given as a parameter: 1 readout byte per 1 array element

Description **SdoReadSeg** allows segmented read of a SDO (if possible) and returns the readout bytes inside the array that was given as a parameter. This array contains one byte of the segmented data (character) in one array element. The array can be used to easily process a string or binary data later on.

The command can be used with waiting for result and producing an error if something goes wrong, or without waiting and not producing errors. In the second case, the result must be checked by the SdoStatus command. For results, see there.

> ⚠ Note: The MACS5 does not support this command on EtherCAT

Command Group Communication CANopen Commands

Example

```
// Array used to capture the data of the segmented SDO read
dim long sdodata[255];

// CAN node id of a Danfoss FC302 frequency converter,
// equipped with a MCA105 CANopen option.
long vltid = 3;

// Function to print out a string,
// which holds one character per array element.
void PrintString(long arr[], long len)
{
    wchar prtstr[256];
    long  i;

    for (i = 0; i < len; i++)
        prtstr[i] = (wchar) arr[i];

    print(prtstr);

    return;
}

long main(void)
{
    print();
    print("Segmented SDO unpacked read using SDOREADSEG and waiting");
    // Test: Read out a FC302 parameter value
    // FC302 parameter no.15-49 (= SW version of the control card)
    fc302parano = 1549;
    // F302 SDO index must be calculated by 0x2000 + par.no.
    fc302sdo = 0x2000 + fc302parano;
    // see Danfoss CAN manual
    sdosegcnt = SdoReadSeg(vltid, fc302sdo, 0, sdodata);

    print("FC302 Par.No.",fc302parano,": SW Version = ");
    PrintString(sdodata, sdosegcnt);
    print("No. of characters = ", sdosegcnt);

    return(0);
}
```

9.134 SdoReadSegp

Summary Segmented read of SDOs (packed).

Syntax `res = SdoReadSegp(id, index, subindex, arrayname);`

Parameter

id	=	CAN id number offset of 100000 for nodes linked to the 2nd CAN bus (If no 2nd CAN bus is present, the 1st CAN bus is used instead)
-id	=	executes the command without waiting for the answer
index	=	SDO index (0x0000 ... 0xFFFF)
subindex	=	SDO sub index (0x00 ... 0xFF)
arrayname	=	name of any existing array (only DIM arrays are supported)

Return Value No. of readout bytes

In the array given as a parameter: 4 readout byte per 1 array element

Description **SdoReadSegp** allows segmented read of a SDO (if possible) and returns the readout bytes inside the array that was given as a parameter. **SdoReadSegp** packs the bytes into the array elements, i.e. every array element contains four bytes. This is most efficient for compact handling of long strings or a bigger amount of binary data.

The command can be used with waiting for result and producing an error if something goes wrong, or without waiting and not producing errors. In the second case, the result must be checked by the SdoStatus command. For results, see there.

Extension starting with version 6.8.01: STRING arrays can be used in **SdoReadSegp** statements. This makes it possible to use print commands to print the strings later on.

```
res = SDOREADSEG 1 0x1008 0 stest
PRINT "SDOREADSEG delivers ",erg," string ",stest[]
```

It is even possible to read out an array from another controller using **SdoReadSegp** as follows:

```
// Testing SDOREADSEG and SDOREADSEGP
// It is necessary that the other controller has an array of
// maximum 100 LONG elements defined. These elements will
// be read out by the SDOREADSEG and SDOREADSEGP commands.
//

DIM test[100],utest[400]
DIM string stest[40]

long nid, res, arrno, size

stest[] = test[]

nid = 4    // node id of the other controller

res = sdoread nid 0x21FF 6
PRINT "number of arrays available " ,res
arrno = 0    // the first array is used (starts with zero)

SDOWRITE nid 0x21FF 1 arrno    // setting actual array number
size = SDOREAD nid 0x21FF 2    // size of actual array

PRINT "array ",arrno, " has size ",size

res = SDOREADSEG nid 0x21FF 3 utest
PRINT "SDOREADSEG delivers ",res

res = SDOREADSEGP nid 0x21FF 3 test
PRINT "SDOREADSEG delivers ",res
```

⚠ Note: The MACS5 does not support this command on EtherCAT

Summary Segmented read of SDOs (packed).

Command Group Communication CANopen Commands

Example

```
// Array used to capture the data of the segmented SDO read
DIM sdodata[255]

// CAN node id of a Danfoss FC302 frequency converter,
// equipped with a MCA105 CANopen option.
vltid = 3

// Function to print out a packed string,
// which holds in maximum 4 characters per array element.
long printpackedstring (long[] arr, long len)
{
    long ind,cc,rel
    ind = 1
    cc  = 1
    WHILE(cc <= len) DO
        ind = ((cc-1)  % 4) + 1
        rel = ((cc-1) mod 4) + 1
        PRINT chr(arr[ind].b rel);
        cc++
    ENDWHILE
    PRINT " "
}

PRINT ""
PRINT "Segmented SDO packed read using SDOREADSEGP without waiting"

// Test: Read out a FC302 parameter value
// FC302 parameter no.15-49 (= SW version of the control card)
fc302parano = 1549
// F302 SDO index must be calculated by 0x2000 + par.no.
fc302sdo = 0x2000 + fc302parano   // see Danfoss CAN manual
sdosegcnt = SDOREADSEGP -vltid, fc302sdo, 0, sdodata

res = 0
WHILE (res == 0) do
    res = sdostate vltid sdosegcnt
ENDWHILE

IF (res > 0) THEN
    PRINT "FC302-Par.No. ",fc302parano," = ";
    printpackedstring(sdodata, sdosegcnt)
    PRINT "No. of bytes of the parameter content: ",sdosegcnt
ELSE
    PRINT "Reading of  FC302 Par.No. ",fc302parano," failed - Error = ",res
ENDIF
```

9.135 SdoStatus

Summary Check the result of an active communication.

Syntax `res = SdoStatus(id, value);`

Parameter id = CAN id
offset of 100000 for nodes linked to the 2nd CAN bus
(If no 2nd CAN bus is present, the 1st CAN bus is used instead)

Summary Check the result of an active communication.

value = additional return value, whose meaning depends of the return value "res"

Return Value

0	=	id is busy, waiting for answer
1	=	answer arrived, result available in value
2	=	segmented read complete, data is in the array, number of bytes received is in value
-2	=	timeout occurred while waiting for answer value = (index << 8 + subindex)
-12	=	CAN-Error occurred (bus error) (will be reset internally)
-xx	=	internal errors in firmware
-33	=	id not in use (id is free for new communication)
-50	=	SDO was aborted by slave, SDO abort code is available in value
-51	=	array was too small for the segmented read (minimum size in value)
-52	=	toggle bit error in segmented transfer
-53	=	too much data came in (more than stated in the SdoRead)
-54	=	not enough data received when done was detected
-55	=	array write error in segmented read

Description It is possible to call SdoRead, SdoWrite, or SdoWriten with negative Can-Id numbers. In such a case, the command is executed but it will not wait for the answer. However, SdoRead does not return a meaningful result in such a case.

Therefore **SdoStatus** allows a check for the result of an active communication. **SdoStatus** returns 0 as long as communication is busy. It returns negative results in the case of an error (TIMEOUT, SDO-Abort, ..). If **SdoStatus** returns a positive result, the last SDO command was completed successful. If the last command was an SdoRead, then **SdoStatus** will return the result in the parameter value.

It is possible to start several transactions (up to 5) to different IDs in parallel. (Of course not with the same ID). For example, three SdoReads can be send out to IDs 1,2,3 and then poll for the results.

Command Group Communication CANopen Commands

Syntax Example

```
id    = 1
value = 0
res   = SdoRead(-id, idx, subidx);

while (res == 0) res = SdoState(id, value);

if (res > 0)
    {
    // use value as result of the SDOREAD
    }
else
    {
    // handle the error case
    }
```

9.136 SdoWrite

Summary Sets an SDO of a connected CANopen device and uses the data exchange without defined number of valid data bytes (= "Data set size is not indicated").

If id is > 1,000,000, an SDO of a connected EtherCAT slave will be set

Syntax

```
SdoWrite(id, index, subindex, value);
```

Parameter

id = CAN id (1...127, 100001...100127)
offset of 100000 for nodes linked to the 2nd CAN bus

Summary Sets an SDO of a connected CANopen device and uses the data exchange without defined number of valid data bytes (= "Data set size is not indicated").

If id is > 1,000,000, an SDO of a connected EtherCAT slave will be set

(If no 2nd CAN bus is present, the 1st CAN bus is used instead)
offset of 1000000 + node id (starting at 1) will write an SDO on a connected EtherCAT slave.

-id	=	executes the command without waiting for the answer
index	=	index of object (0x0000...0xFFFF)
subindex	=	subindex (0x00 ... 0xFF)
value	=	parameter value

Return Value -

Description This command writes a SDO of a connected CANopen device. Data download is done by using the mechanism "Data set size is not indicated" as described by the CANopen specification DS301 V4.02. If a CANopen slave does not support this type of data exchange, there is also the command SdoWriten available, which allows to define the number of valid data bytes (= "Data set size is indicated").

After execution of the **SdoWrite** command, the passed value (= parameter "val") is written by the CANopen device into its corresponding object (= parameters "index", "subidx"). If a CANopen error happens (e.g. timeout, unknown object) an error will be reported. The error can be handled by an ERROR interrupt routine if one has been defined.

It is possible to call **SdoWrite** with negative CAN-ID numbers. Then the command is executed but it will not wait for the answer. However, **SdoWrite** does not return a meaningful result in such a case. Therefore use SdoStatus. to check the result of an active communication.

Remark: Timeout = 300ms
The default timeout setting for SdoRead, **SdoWrite**, and SdoWriten commands is 300ms. If a CANopen devices has a slower reaction time, it is possible to modify the timeout limit by the Sysvar[0x01220240]:
Sysvar[0x01220240] = 500 // SDO 0x2202/64 = 500 ms.

> For writing SDO values on the controller itself (i.e. not a connected device), the Sysvar array should be used. For calculation of the corresponding Sysvar array index, the following formula can be used:
> Sysvar Array-Index = 0x01 << 24 + SDO-Index << 8 + SDO-Subindex
> Example: SDO 0x2202/64 can be written by Sysvar[0x01220240].

Command Group Communication CANopen Commands

Syntax Example 1
```
// Set CANopen Guarding objects of a CANopen slave (node id: 1)
// Guard time: 0x100C (UNSIGNED16) = 20 ms
SdoWrite(1, 0x100C, 0, 20);
// Life time: 0x100D (UNSIGNED8) = 5
SdoWrite(1, 0x100D, 0, 5);
```

Syntax Example 2 Syntax example for a MACS control unit with 2 CAN buses. The CANopen slave devices are connected to the 2nd CAN bus, the so-called slave bus (=> bus offset of node id = 100000).

```
// Set CANopen Guarding objects of a CANopen-Slave (node id: 1)
// Guard time: 0x100C (UNSIGNED16) = 20 ms
SdoWrite(100001, 0x100C, 0, 20);    // bus offset = 100000
// Life time: 0x100D (UNSIGNED8) = 5
SdoWrite(100001, 0x100D, 0, 5);    // bus offset = 100000
```

9.137 SdoWriten

Summary Set an SDO and specifies the number of valid data bytes (= "Data set size is indicated").

Syntax
```
SdoWriten(id, index, subindex, byteno, value);
```

Parameter

id	=	CAN id (1...127, 100001...100127) Offset of 100000 for nodes linked to the 2nd CAN bus

Summary Set an SDO and specifies the number of valid data bytes (= "Data set size is indicated").
(If no 2nd CAN bus is present, the 1st CAN bus is used instead)

-id	=	executes the command without waiting for the answer
index	=	SDO index which should be written
subindex	=	SDO subindex which should be written
byteno	=	number of valid data byes that should be written (1..4)
value	=	value which should be written

Return Value -

Description This command writes a SDO of a connected CAN-open device. Data download is done by specifying the number of valid data bytes (= "Data set size is indicated" as described by the CANopen specification DS301 V4.02).

After execution of the **SdoWriten** command, the passed value (= parameter "value") is written by the CANopen device into its corresponding object (= parameters "index", "subidx") taking the defined number of valid data bytes (= parameter "byetno") into account. If a CANopen error happens (e.g. timeout, unknown object) an error will be reported. The error can be handled by an ERROR interrupt routine if one has been defined.

Remark: Timeout = 300 ms
The default timeout setting for SdoRead, SdoWrite, and **SdoWriten** commands is 300ms. If a CANopen devices has a slower reaction time, it is possible to modify the timeout limit by the Sysvar[0x01220240]:
Sysvar[0x01220240 = 500 // SDO 0x2202/64 = 500 ms

Command Group Communication CANopen Commands

Syntax Example 1
```
// Set CANopen Guarding objects of a CANopen slave (node id: 1)
// Guard time: 0x100C (UNSIGNED16) = 20 ms
SdoWriten(1, 0x100C, 0, 2, 20);
// Life time: 0x100D (UNSIGNED8) = 5
SdoWriten(1, 0x100D, 0, 1, 5);
```

Syntax Example 2
```
// Syntax example for a MACS control unit with 2 CAN buses:
// The CANopen slave devices are connected to the 2nd CAN bus,
// => bus offset of the node id = 100000
// Set CANopen Guarding objects of a CANopen-Slave (node id: 1)
// Guard time: 0x100C (UNSIGNED16) = 20 ms
SdoWriten(100001, 0x100C, 0, 2, 20);   // bus offset = 100000
// Life time: 0x100D (UNSIGNED8) = 5
SdoWriten(100001, 0x100D, 0, 1, 5);    // bus offset = 100000
```

9.138 SdoWriteSeg

Summary Segmented write of SDOs (only for EtherCAT master).

Syntax `res = SdoWriteSeg(id, index, subindex, arrayname);`

Parameter

id	=	EtherCAT slave number, with offset of 1,000,000, e.g. 1,000,001 = first slave
index	=	SDO index (0x0000 ... 0xFFFF)
subindex	=	SDO subindex (0x00 ... 0xFF)
arrayname	=	name of an existing array (DIM, global, local)

Return Value Number of bytes written.

Description **SdoWriteSeg** allows data from an array to be written by segmented write to an SDO on the slave.

> ⚠ Note: The MACS5 does not support this command on EtherCAT

Portability This command is only available on MasterMACS controllers.

Command Group Communication CANopen Commands

9.139 SerialSetup

Summary Initialize the RS232 or RS485 interface handler.

Syntax `SerialSetup(interface, baudrate, parity, termMode, endOption);`

Parameter **interface**

Selects the serial interface (see hardware manual for available interfaces)

INTF_RS232_1: RS232

INTF_RS485_1: RS485 (MACS5: X4, pins 5 & 6. Encoder 4 cannot be used then)

baudrate

Sets the baudrate of the serial interface (9600...115200)

parity

Sets the parity mode

PAR_NONE: No parity

PAR_EVEN: Even parity

PAR_ODD: Odd parity

termMode

Sets the termination mode. This is the way in which the end of a serial telegram is recognized. This is only applicable to the SerialReadComm command.

TERM_CHAR: The end of a telegram is detected by a specific character (ASCII code).

TERM_TIMEOUT: The end of a telegram is detected by timeout (serial line idle)

termOption

Additional info for the termination mode.
In case termMode is TERM_CHAR, this value is the termination character (ASCII code).
In case termMode is TERM_TIMEOUT, this value is the termination timeout in milliseconds.

Return Value

Description

This functions sets up the serial interface of the controller. This is necessary before any of the commands SerialReadComm, SerialWriteComm, SerialReadChar or print functions to the serial line can be used.

Interrupts on full telegram reception

Non-state machine programs: The CANINPUT interrupt can be used.
State machine programs: The SERIALRECV event can be used.

Interrupt on single character reception
Non-state machine programs: The SERIALCHAR interrupt can be used.
Non-state machine programs: The SERIALCHAR event can be used.

Samples See Samples in ApossIDE: File->Sample->Communication
ISR_Dcp.mc (for telegram-oriented communication)
ISR_Keypressed.mc (for character-oriented communication)

Command Group Communication Serial Commands

9.140 SerialReadChar

Summary Character input on serial interface.

Syntax `res = SerialReadChar(p);`

Summary Character input on serial interface.

Parameter p = The maximum waiting time:
p = 0 wait for character
p > 0 wait of max. p milliseconds
p < 0 no wait for character

Return Value ASCII-Code for the received character (-1 if no character available)

Description A character can be read in via the **SerialReadChar** command through the serial interface (RS232 or RS485).

The parameter given with the **SerialReadChar** command determines if a character will be accepted unconditionally, with a certain time delay or not at all.

For each successful **SerialReadChar** command, one character will be read in. For inputting character chains, the **SerialReadChar** command (p ≠ 0) must be repeated in a loop as often as necessary, until no further characters are available.

The **SerialReadChar** and print commands are especially suitable for a data exchange with terminals, and any computer system, as well as PLC systems.

The pin assignment of the communication interface, as well as the data exchange format can be taken from the Hardware manual.

See Parity baudrate and interface parameters for information on how to set up the RS232 port.

Command Group Communication Serial Commands

Syntax Examples

```
character = SerialReadChar(0);    // Wait until character is typed in
character = SerialReadChar(5000); // Wait max. 5 seconds to input
character = SerialReadChar(-1);   // Do not wait for input
```

Program Sample See **File** → **Sample**: ISR_Keypressed.mc

9.141 SerialReadComm

Summary Check for an incoming telegram and return its user data.

Syntax `res = SerialReadComm(arr);`

Parameter arr = Array which is used to receive the incoming user data, 1 received unicode character per array element. This must be a **long** or **wchar** array. Note that array indices may not be specified; received data will be placed into the array, starting at the first element in the array.

Return Value

>0	OK	Telegram has arrived with `res` unicode characters of user data.
0	ACTIVE	No telegram has arrived.
-1	TIMEOUT	
-2	BADFRAME	The received frame is not valid.
-4	OVERFLOW	Incoming user data is longer than the internal receive array (255 characters).
-5	NOT_SUPPORTED	This command is not supported on this controller.
-6	NOT_INITIALIZED	SerialSetup has to be called first.
<-100	USER_ARRAY_OVERFLOW	Incoming user data is longer than the 'arr' array. The number of received characters is 101-res.

Description This command checks for an incoming telegram and will return its user data if a telegram has arrived.

Command Group Communication Serial Commands

9.142 SerialWriteComm

Summary Sends a full telegram over the serial interface

Syntax `res = SerialWriteComm(length, arr);`

Parameter

length	=	Number of characters to send. If set to 0, then no telegram will be sent. However, this can be used to query the communication state.
arr	=	Array which contains the data to be sent, 1 character per array element. This must be a **long** or **wchar** array. Note that array indices may not be specified; data will be sent, starting at the first element in the array.

Return Value

1	SENT	Telegram will be sent and previous telegram has been sent successfully.
0	OK	Telegram will be sent.
-1	TIMEOUT	The previous transmission has timed out. The new telegram will be sent.
-2		`length` is longer than the maximum allowed telegram length.
-3	BUSY	There is still another ongoing transmission that has not timed out. You must retry later.
-5	NOT_SUPPORTED	This command is not supported on this controller.
-6	NOT_INITIALIZED	SerialSetup has to be called first.
<-100	ARRLEN	The array is shorter than 'length'

Description The data which in the array will be sent as it is (with no telegram control bytes).

> ⚠ Before this command can be used, the command SerialSetup has to be called.

Command Group Communication Serial Commands

9.143 SetMasterOrigin

Summary Set any position as the zero point for the master.

Syntax `SetMasterOrigin(axis, value,...);`

Parameter

axis	=	Axis module number (see Command Reference).
value	=	Absolute position.

Return Value -

Description With the **SetMasterOrigin** command you can set any position as the new zero point for the master.

> The command **SetMasterOrigin** cancels the command DefMasterOrigin.

> Thus, to alter the zero point for the master again, you must reset it with **SetMasterOrigin** or DefMasterOrigin. ResetOrigin does not have any effect on the zero point for the master.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Axis and Information Commands

9.144 SetOrigin

Summary Set absolute position as temporary zero point.

Syntax `SetOrigin(axis, pos,...);`

Parameter

axis	=	Axis module number (see Command Reference).

Summary Set absolute position as temporary zero point.

pos = Absolute position in relation to the real zero point.

Return Value -

Description Any absolute position can temporarily be set as a new reference point for absolute positioning commands (AxisPosAbsStart, AxisLinAbsStart) by use of the **SetOrigin** command. This position is called temporary zero point.

In combination with the command CurvePos, one can fix in this way that the current slave position matches the corresponding value of the curve.

> It is possible to carry out several **SetOrigin** commands without carrying out a previous ResetOrigin. The absolute position value always refers to the real zero point. The last carried out **SetOrigin** command therefore determines the position of the temporary zero point in relation to the real zero point.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Axis and Information Commands

Syntax Example

```
SetOrigin(0, 50000); // axis mod 0: set temp zero point to 50000
SetOrigin(0, 10000, 1, 30000);  // set axis mods 0, 1
SetOrigin(AXALL, 10000);    // set temporary zero point all axes
```

Program Sample See **File** → **Sample**: AbsoluteRelative.mc

9.145 SmClear

Summary Clear the event queue.

Syntax `SmClear(machine, sig-expr);`

Description See Clearing Pending Events for detailed information.

9.146 SmConfig

Summary Define the state machine configuration parameters.

Syntax

```
#pragma SmConfig { runflags, poolsize, timercnt, subsize,
                   prmsize, possize }
```

Description See Specifying the State Machine Configuration for detailed information.

9.147 SmEvent

Summary Define an event and its parameters.

Syntax `SmEvent signal { parameter1, parameter2, ... }`

Description See Defining Signals and Event Types for detailed information.

9.148 SmInput

Summary Request an event when a digital input changes.

Syntax `SmInput(input-expr, test-expr, machine, sig-expr);`

Description See Detecting Digital Input Changes for detailed information.

9.149 SmMachine

Summary Define a state machine variable.

Syntax
```
SmMachine machine;
SmMachine machine { id, initfunc, topstate, queuesize,
                    datasize }
```

Description See Defining State Machine Variables for detailed information.

9.150 SmParam

Summary Request an event when a parameter value changes.

Syntax
```
SmParam(param-expr, mask-expr, test-expr, machine, sig-expr);
```

Description See Detecting Changes to Parameters for detailed information.

9.151 SmPeriod

Summary Start a periodic timer.

Syntax
```
SmPeriod(timeout, machine, sig-expr);
```

Description See State Machine Timers for detailed information.

9.152 SmPosition

Summary Request an event when the motor passes a specified position.

Syntax
```
SmPosition(sign postype, axis, position, machine, sig-expr);
SmPosition(sign postype, axis, position, machine, sig-expr,
           ONCE);

SmPosition(sign postype, axis, position, SETOUT, output);
SmPosition(sign postype, axis, position, SETOUT, output, ONCE);

SmPosition(sign postype, axis, position, SETOUT, output, input);
SmPosition(sign postype, axis, position, SETOUT, output, input,
           ONCE);
```

Description See Detecting Motor Positions for detailed information.

9.153 SmPost

Summary Create an event and post it to the end of the event queue.

Syntax
```
SmPost(machine, sig-expr, param-expr, param-expr,...);
```

Description See Posting Events to State Machines for detailed information.

9.154 SmPublish

Summary Create an event and publish for all state machines.

Syntax
```
SmPublish(sig-expr, param-expr, param-expr,...);
```

Description See Subscribing to Events and Publishing Events for detailed information.

9.155 SmRun

Summary Start state machines running.

Syntax
```
res = SmRun(machine1, machine2,...);
```

Summary Start state machines running.

Description See Starting and Stopping State Machines for detailed information.

9.156 SmState

Summary Define a state machine state, including its event handlers and substates.

Syntax `SmState state { ... }`

Description See Defining the State Hierarchy for detailed information.

9.157 SmStop

Summary Stop running state machines.

Syntax `SmStop(expr);`

Description See Starting and Stopping State Machines for detailed information.

9.158 SmSubscribe

Summary Subscribe to receive published events.

Syntax `SmSubscribe(machine, sig-expr, param-expr, param-expr,...);`

Description See Subscribing to Events and Publishing Events for detailed information.

9.159 SmSystem

Summary Request an event when a system event occurs.

Syntax
```
SmSystem(keyword, param-expr, param-expr, ..., machine,
         sig-expr);
```

Description See Detecting General Operation Events for detailed information.

9.160 SmTimer

Summary Start a "once only" timer.

Syntax `SmTimer(timeout, machine, sig-expr);`

Description See State Machine Timers for detailed information.

9.161 SmUnsubscribe

Summary Request to stop receiving published events.

Syntax `SmUnsubscribe(machine, sig-expr);`

Description See Subscribing to Events and Publishing Events for detailed information.

9.162 SmUrgent

Summary Create an event and post it to the head of the event queue.

Syntax `SmUrgent(machine, sig-expr, param-expr, param-expr,...);`

Description See Posting Events to State Machines for detailed information.

9.163 sprint

Summary Information output to a string array.

Syntax `sprint(stringarray, value,...);`

Parameter

stringarray	=	target string array
value	=	List of one or more values to be printed, separated by commas.
...	=	additional arguments

Return Value -

Description This command prints to a target string array instead of to the ApossIDE output window. See print for a detailed description of the behaviour of this command.

> Maximum number of parameters is limited dependent on the Controller type. There is also a maximum length of buffer per parameter, including the \0 character. Please check Controller Dependent Feature List for the limits.

Command Group Debug

Syntax Examples

```
wchar filename[50];


//This will print the text "log number: 5" to the array.
sprint(filename, "log number: ", 5);
```

9.164 sprintf

Summary Formatted information output to a string array.

Syntax `sprintf(stringarray, format,...);`

Parameter

stringarray	=	target string array
format	=	format string
arg	=	additional arguments

Return Value -

Description This command prints to a target string array instead of to the ApossIDE output window. See printf for a detailed description of the behaviour of this command.

If the format string includes format specifiers (i.e. sequences beginning with %), then additional arguments *must* follow the format string. These will be formatted and inserted in the output string, replacing their respective specifiers.

> Maximum number of parameters is limited dependent on the Controller type. There is also a maximum length of buffer per parameter, including the \0 character. Please check Controller Dependent Feature List for the limits.

Command Group Debug

Syntax Examples

```
wchar filename[50];

sprintf(filename, "log number %ld", 5);
```

9.165 SyncCam

Summary Enter CAM mode synchronization and wait in an "idle" state.

Syntax `SyncCam(axis, num,...);`

Parameter

axis = Axis module number (see Command Reference).

num = Number of curves to be processed:

> 0 - Once started (with SyncCamStart), the drive will remain synchronized with the curve until *num* curves have been processed. After *num* curves have been

Summary Enter CAM mode synchronization and wait in an "idle" state.

processed, curve synchronization will be exited "normally" (see below) and the drive will return to the synchronization "idle" state. While these curves are being processed, any new curve set with SyncSetCurve will *not* be activated; the original curve will continue to be used until all *num* curves have been processed. This option should not be used with non-cyclic curves (types CU_GRAD and CU_GRAD_INTRPT).

0 - Once started (with SyncCamStart), the drive will remain synchronized with the curve until synchronization is explicitly stopped using the SyncCamStop command.

-1 - Enter CAM-Mode without resetting the markers. Once started (with SyncCamStart), the drive will remain synchronized with the curve until synchronization is explicitly stopped using the SyncCamStop command.

< -1 - Enter CAM-Mode without resetting the markers. The number of curves to be processed is set to "-(*num*+1)" and then the curves are processed as if *num* > 0. For example, *num* = -2 will not reset markers and will process one single curve (i.e. -(-2+1) = 1). This option should not be used with non-cyclic curves (types CU_GRAD and CU_GRAD_INTRPT).

Return Value -

Description

This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

The **SyncCam** command starts CAM-Mode (CAM control) and enters an "idle" state. Note that a curve must be set (using SyncSetCurve) before **SyncCam** may be executed. From this moment, master positions are tracked and corresponding slave positions are calculated (i.e. parameters such as PFG_CMASTERCPOS and PFG_CCURVEPOS will return meaningful values).

The starting behaviour may be affected by the SYNCMSTART parameter. For example, when SYNCMSTART = 2000, the curve positions of the master are only counted after the next master marker.

When the synchronization "idle" state is entered, the controller will determine the current drive velocity and then attempt to maintain that velocity as a constant velocity. This is done so that applications have more flexibility in the timing of when synchronization mode is entered (i.e. it is not necessary for the drive to be stopped). However, this does imply that the drive should be in a "stable" state (i.e. either stopped or moving at constant velocity) when synchronization mode is entered. Constant velocity is maintained until the SyncCamStart command is executed to initiate synchronization with the actual curve.

If the drive is not stopped or moving at constant velocity when the "idle" state is entered, then *maximum possible torque* will be used to force the drive to a constant velocity. It is recommended that applications have the drive in a stable state before entering synchronization mode.

If curve synchronization is being exited normally (i.e. *num* = 0 or -1) and start stop point pair 2 does *not* exist, then curve synchronization will be exited after curve *num* has been completed. In this case, the drive velocity at the end of the curve will be maintained as the "constant velocity" when the "idle" state is re-entered.

If curve synchronization is being exited normally and start stop point pair 2 *does* exist, then this pair will be used to stop the drive *during* the processing of curve number *num*. This means that the last curve processing cycle will not follow the complete CAM curve cycle. The drive will come to a stop at position 0 (in slave units).

Start stop point pairs cannot be used with curves of type CU_GRAD_INTRPT. See Immediate Curves.

When non-cyclic curves (type CU_GRAD or CU_GRAD_INTPRT) are used, default start stop point pair 2 is not used.

Note that at any time, regardless of the value of *num*, curve synchronization may be explicitly stopped using the SyncCamStop command. Note that synchronization will also be explicitly stopped if another mode is started with commands such as AxisControl(STOP), AxisCvelStart, AxisPosAbsStart, etc.

While curve synchronization is active, SDO PFG_CWRAP (0x2500/130) can be used to track the curve number that

Summary Enter CAM mode synchronization and wait in an "idle" state.

> is currently being processed.

Command Group Synchronization Commands

Sample For an example of **SyncCam** usage, see Example 1: Basic Synchronization.

9.166 SyncCamMarkerMaster

Summary Synchronization in CAM mode with master marker correction.

Syntax `SyncCamMarkerMaster(axis, num,...);`

Parameter

axis = Axis module number (see Command Reference).

num = Number of curves to be processed:

0 = The drive remains in CAM-Mode until another mode is started with commands such as AxisControl(STOP), AxisCvelStart, AxisPosAbsStart, etc.

Return Value -

Description

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Like SyncCam, the command **SyncCamMarkerMaster** brings about synchronization in CAM-Mode, but beyond that it also performs a marker correction (only if the master moves forward).

In order to save the distance between sensor and processing point, the parameter SYNCMPULSM is used. It allows the correction of the marker position without changing the curve. Also, larger sensor distances than the actual curve length are possible. In this case, a FIFO is used for the marker correction (see example).

> SyncCam does not start the slave drive nor does it interrupt on-going motions (e.g. Cvel), only SyncCamStart does.

> The drive remains in CAM-Mode until *num* curves have been processed successfully.

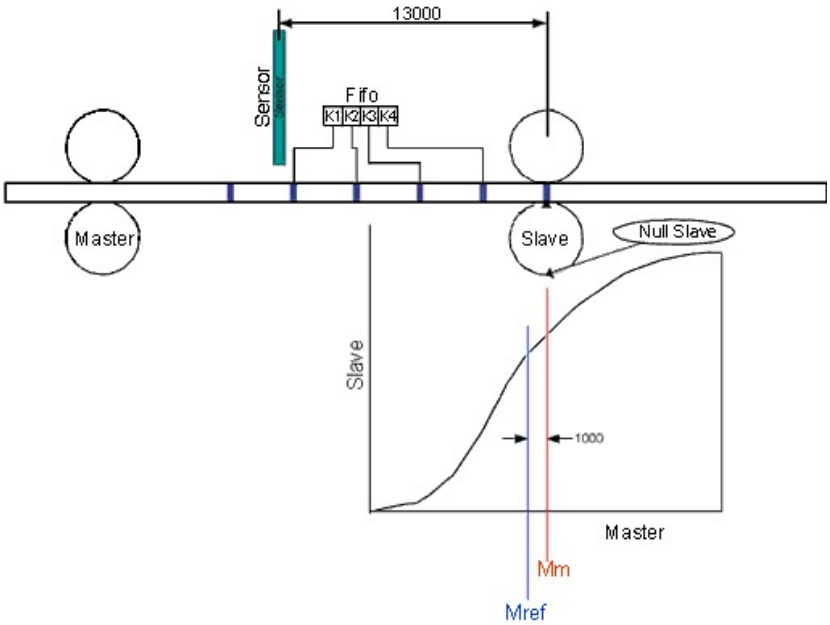
If the synchronization (after *num* curves) is being closed normally, the start stop point pair 2 will be used - if no SyncCamStop with a corresponding point pair is defined - in order to stop the drive. It will then come to a stop at the position 'slavepos' (see parameters).

Command Group Synchronization Commands

Sample If for example curve length is 3000 and distance of sensor to working point is 13000, we will have a FIFO with 4 register and an offset of 1000 which must be concerned.

See the following diagram:

Summary Synchronization in CAM mode with master marker correction.



9.167 SyncCamMarkerSlave

Summary Synchronization in CAM mode with slave marker correction.

Syntax `SyncCamMarkerSlave(axis, num,...);`

Parameter

axis = Axis module number (see Command Reference).

num = Number of curves to be processed:

0 = The drive remains in CAM-Mode until another mode is started with commands such as AxisControl, AxisCvelStart, AxisPosAbsStart, etc.

Return Value -

Description

This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Like SyncCam, the command **SyncCamMarkerSlave** brings about synchronization in CAM-Mode, but beyond that it also performs a marker correction of the slave. Here, the slave position is corrected, not the curve position.

In contrast to SyncCamMarkerMaster, no FIFO is created.

SyncCam does not start the slave drive nor does it interrupt on-going motions (e.g. Cvel), only SyncCamStart does.

The drive remains in CAM-Mode until *num* curves have been processed successfully.

If the synchronization (after *num* curves) is being closed normally, the start stop point pair 2 will be used - if no SyncCamStop with a corresponding point pair is defined - in order to stop the drive. It will then come to a stop at the position 'slavepos' (see parameters).

Command Group Synchronization Commands

9.168 SyncCamStart

Summary Start slave for synchronization in CAM mode.

Syntax `SyncCamStart(axis, pnum,...);`

Parameter axis = Axis module number (see Command Reference).

Summary Start slave for synchronization in CAM mode.

pnum = Start stop points number:

> 0 = Engaging begins when the corresponding point A is reached, provided the master moves in positive direction; the engage curve is finished at point B. If point A and B are identical, the slave will be engaged with the set maximum velocity (i.e. without curve) as soon as the master has reached this point.

= 0 = The slave will be engaged immediately with the set maximum velocity. It does not matter in what direction the master moves or whether it moves at all.

< 0 = Again, the corresponding point pair is used. However, engaging begins at point B and is finished at point A (i.e. in negative direction).

Return Value -

Description The command starts the movement of the slave. With 'pnum', the point pair is selected that determines in which master position the synchronization begins and where it should be finished.

When moving forward, the synchronization begins at point A and is finished up to point B. When moving backward, it begins at point B and is finished up to point A.

Command Group Synchronization Commands

9.169 SyncCamStop

Summary Stop slave after the CAM synchronization.

Syntax `SyncCamStop(axis, pnum, slavepos);`

Parameter axis = Axis module number (see Command Reference).

pnum = Start stop points number:

> 0 = Engaging begins when the corresponding point A is reached, provided the master moves in positive direction; the engage curve is finished at point B. If point A and B are identical, the slave will be engaged with the set maximum velocity (i.e. without curve) as soon as the master has reached this point.

= 0 = The slave will be engaged immediately with the set maximum velocity. It does not matter in what direction the master moves or whether it moves at all.

< 0 = Again, the corresponding point pair is used. However, engaging begins at point B and is finished at point A (i.e. in negative direction).

slavepos = Position where the slave is supposed to stand after disengaging.

Description This command stops synchronization without leaving SyncCam mode. The slave will be disengaged according to the point pair defined in 'pnum'. Only then will the slave actually be stopped. When the stop point has been reached, the slave must be at 'slavepos'.

> When moving forward, disengaging begins at point A and is finished at point B; vice versa when moving backward.

> If the program is closed without **SyncCamStop** command, disengaging occurs by default with the second point pair and a stop occurs at the **Slave Stop Position** defined in the curve data.

Return Value -

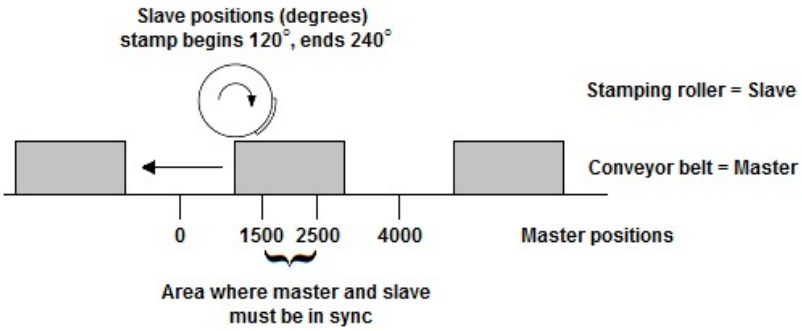
Command Group Synchronization Commands

Syntax Example

```
SyncSetCurve(0, curve);
SyncCam(0, 0);          // Synchronize in CAM-Mode
SyncCamStart(0, 1);     // Start slave with start point pair 1
SyncCamStop(0, 2, 0);   // Stop slave with stop point pair 2 at
                        // the slave position 0 or 3600
```

Summary Stop slave after the CAM synchronization.

Sample



9.170 SyncDefOrigin

Summary Defines master-slave relationship to be used when the next synchronization becomes active.

Syntax `SyncDefOrigin(mcpos, axis, spos,...);`

Parameter Position Synchronization (SyncPos and SyncMarker)

mcpos = Master reference position in qc.

axis = Axis module number (see Command Reference).

spos = Slave reference position.

Curve Synchronization (SyncCam, SyncCamMarkerMaster, and SyncCamMarkerSlave)

mcpos = Master curve position in mu.

axis = Axis module number (see Command Reference).

spos = Slave curve offset in uu.

Return Value -

Description

This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Position Synchronization (SyncPos and SyncMarker)

If position synchronization is the next synchronization mode to become active, then this command defines how much distance ahead or behind the slave should be in relation to the master position. It defines the relation between master and slave for the next SyncPos or SyncMarker command. It sets the internal slave command position Mpcmd to the slave value.

The master value is used for an internal SyncShiftOrigin. For that reason, a SyncShiftOrigin will be overwritten by this command. Both actions are done at the moment, when the SYNC command is activated. So it is guaranteed, that master and slave will be synchronized at the above master-slave position.

Curve Synchronization (SyncCam, SyncCamMarkerMaster, and SyncCamMarkerSlave)

If curve synchronization is the next synchronization mode to become active, then this defines the start values for the next CAM curve to be activated.

The start values will be defined as follows:

mcpos is the master curve position that will correspond to the actual master position (Mapos). This is the distance from the beginning of the curve and is specified in master units (mu). Negative values are allowed and will be interpretted as measuring backwards from the end of the curve rather than forwards from the beginning of the curve. If the value is larger than the master cycle length of the curve, then full curve cycles are removed from the value until the value is within the range of the curve.

spos has two different meanings, based on a comparison its magnitude (absolute value) with the magnitude of the slave curve length:

Summary Defines master-slave relationship to be used when the next synchronization becomes active.

|*spos*| ≤ |SlaveCurveLength|	=	*spos* defines the offset (in uu) from the origin of the slave curve coordinates (i.e. slave curve position 0) to the actual slave position.
|*spos*| > |SlaveCurveLength|	=	The actual slave position will be set to correspond to the slave curve position that corresponds to the master position *mcpos*. *spos* itself does not have any meaning; it only needs to be greater than slave curve length.

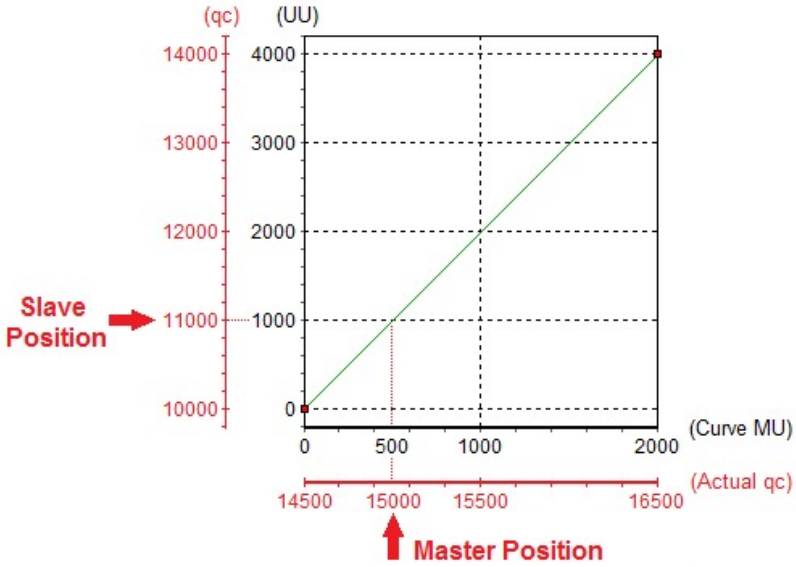
Example 1:

Assume that the curve is a straight line going from (0,0) to (2000,4000). Also assume that the slave is at an actual position of 11000 qc and that the master has an actual position of 15000 qc. To simplify the example, assume that the gear factors are 1 : 1 (so 1 mu = 1 qc and 1 uu = 1 qc). This is shown in the diagram below.

Now, the following command is executed:

```
SyncDefOrigin(500, 0, 100000);
```

The master curve position (*mcpos* = 500 mu) and located on the curve. This location is then set to correspond to the actual master position of 15000 qc. This fixes the horizontal position of the curve. Since the slave curve position (*spos* = 100000 uu) is larger than the slave curve length of 4000 uu, the actual slave position of 11000 qc will be used as the slave curve position. The slave curve position corresponding to 500 mu is then calculated using the curve. From the diagram below, this can be seen to be 1000 uu. This location is then set to the actual slave position of 11000 qc. This fixes the vertical position of the curve. Now, the actual master and slave positions correspond to a point lying on the curve and when the master starts moving, the slave will follow the curve.



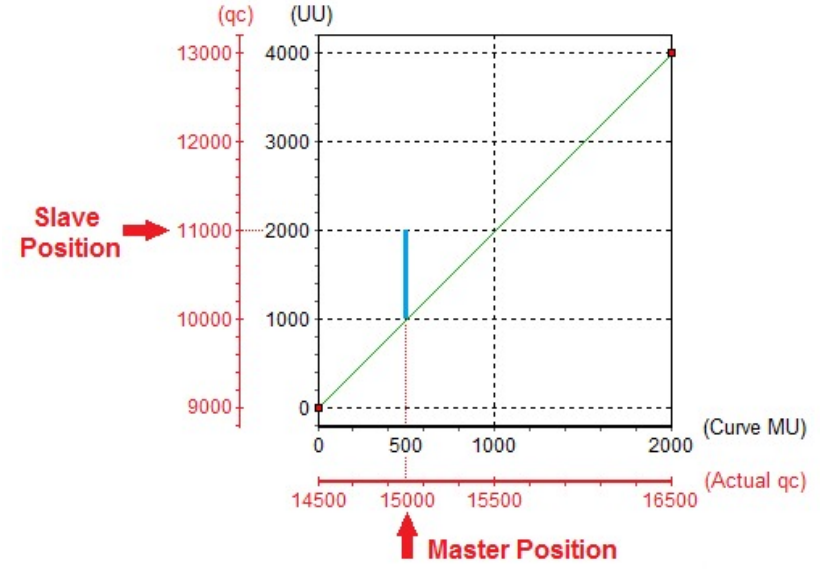
Example 2:

Using the same curve and settings as before, the following command is executed:

```
SyncDefOrigin(500, 0, 2000);
```

The master curve position (*mcpos* = 500 mu) and located on the curve. This location is then set to correspond to the actual master position of 15000 qc. This fixes the horizontal position of the curve. Since the slave curve position (*spos* = 2000 uu) is smaller than the slave curve length of 4000 uu, *spos* will represent the offset from the origin of the slave curve coordinates (i.e. slave curve position 0) to the actual slave position. *spos* will be converted from uu to qc (2000 uu = 2000 qc because the gear factors are 1 : 1) and then used to calculate the origin of the slave coordinates. The actual slave position (of 11000 qc) is 2000 qc ahead of the slave coordinate origin. This puts the slave origin at 9000 qc and fixes the vertical position of the curve.

Summary Defines master-slave relationship to be used when the next synchronization becomes active.



> Note that the actual master and slave positions do *not* correspond to a point lying on the curve. The slave is ahead of the curve. The blue line shows how far the slave is currently off the curve. If this curve is started, the slave would try to correct the position and move toward the curve.

SyncDefOrigin in conjunction with CU_GRAD curves (type 3)

In case of a CU_GRAD curve, **SyncDefOrigin** can be used to define the absolute end position of the curve in qc. In that case, the curve is started immediately and is calculated in such a manner that the end positions will be reached at the end of the curve.

> To avoid degenerate polynomials, the distances must be in a correct relation. If you start with velocity zero and want to end up with a velocity of 1 (slave has same velocity as master), then the master distance should be less than two times the slave distance. Otherwise, the polynomial will have extremes within the interval. This is more difficult to predict in other cases (start velocity not 0 or end velocity not 1). Therefore, you can check the PG_FLAG_CURVE_ERR to see if the last SyncSetCurve produced a curve with extremes (see AxisStatus). Then you can read out the SYSVAR PFG_LASTERROR (see SDO dictionary) to decide what it was.

Command Group Synchronization Commands

Syntax Example

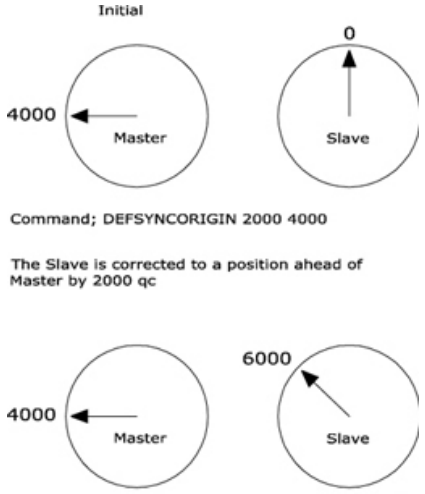
```
SyncDefOrigin(1000, 0, 2000);
SyncDefOrigin(1000, 0, 2000, 1, 4000, 2, 8000);
SyncDefOrigin(1000, AXALL, 2000);
```

Sample Position Synchronization

Here when the master is in 2000 qc the slave should be in 4000 qc, i.e., slave should be ahead of master by 2000 qc.

Also when the master is in 3000 qc the slave should be in 5000 qc.

Summary Defines master-slave relationship to be used when the next synchronization becomes active.



9.171 SyncError

Summary Queries actual synchronization error of the slave.

Syntax `res = SyncError(axis);`

Parameter axis = Axis module number (see Command Reference).

Return Value Actual synchronization error of the slave in uu and

1. as an absolute value when the value of the accuracy window is defined with a plus sign in the parameter SYNCACCURACY;
2. with polarity sign when in SYNCACCURACY the value of the window is defined with a minus sign.

Description **SyncError** returns the actual synchronization error in User Units UU. This is the distance between the actual master position (converted with drive factor and offset) and the actual position of the slave.

If the parameter SYNCACCURACY is defined by a minus sign, you can also determine whether the synchronization is running ahead (negative result) or running behind (positive result).

Command Group Synchronization Commands

9.172 SyncMarker

Summary Angle/position synchronization with the master with marker correction

Syntax `SyncMarker(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

The **SyncMarker** command functions just like the SyncPos command by making an angle/position synchronization with the master, but also make a marker correction. Thus, during the starting of synchronization the program is synchronized to the next marker calculated. In this manner it is possible to compensate for differing running behaviors, for example slippage.

If RAMPTYPE ≥ 2, then synchronization is started with limited jerk. This only concerns the start; reaching the master velocity still looks like a trapezoid. This helps smoothen the start procedure in the case of heavy loads or fragile mechanics.

Once synchronization has been completed, deviation is determined at every marker (or every nth marker if the number

Summary Angle/position synchronization with the master with marker correction

of markers is not identical for the master and slave). This is input into the control as the new offset and the program immediately attempts to compensate for this. However, in doing so the values set for velocity, Vel, and acceleration, Acc or Dec are not exceeded.

AxisPosAbsStartSyncPos**SyncMarker** does not contain the command AxisControl(ON) which turns on the motor control. When using **SyncMarker** an explicit execution of AxisControl(ON) is necessary after a previous use of AxisControl(OFF).

> In addition to the parameters used by SyncPos, SYNCREADY and SYNCFAULT are also of significance.

> Since the following parameters could lead to overdefinition, it is important to ensure that these values are logical, match one another, and are consistent with the information on the gear factors.
>
> SYNCMARKM and SYNCMARKS
> SYNCMPULSM and SYNCMPULSS
> SYNCMTYPM and SYNCMTYPS

> **SyncMarker** should only be called up once since the synchronizing continues until the next motion or stop command. All additional **SyncMarker** commands cause the synchronization to start over again from the beginning and this is not normally intended, as you reset the actual SyncError.

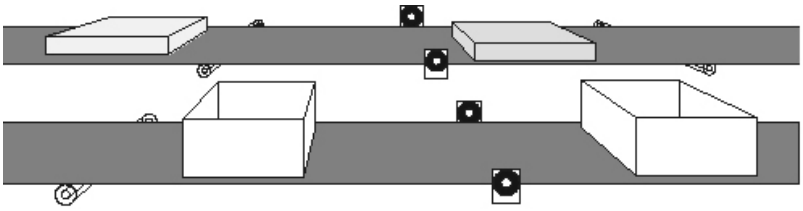
> When defined in SYNCMSTART, the system waits for the first evaluation of the marker pulses on starting **SyncMarker** and only then the offset SYNCPOSOFFS is applied.

Marker signal

The marker can be the zero pulse from the encoder or an external 24 volt signal
(I5 = master; I6 = slave).

Command Group Synchronization Commands

Example



Even when both belts are running synchronously the lids may not be aligned with the boxes at the right time. With **SyncMarker** the difference between master and slave is detected by means of the external markers and the possible position deviation is corrected.

9.173 SyncMarkerRestart

Summary Resets a marker or resets marker handling.

Syntax

```
SyncMarkerRestart(axis, restarttype,...);
```

Parameter

axis	=	Axis module number (see Command Reference).
restarttype	=	0, 1, or 2.

Return Value -

Description This command is used for restarting the marker handling. It resets some internal flags according to the value of restarttype:

restarttype = 0

This command does a marker reset which means

- Clears the flags (PG_FLAG_SYNCMMERR, PG_FLAG_SYNCSMERR)
- Resets the SyncWindow, the correction values, the gear correction, the faked counts
- Resets illegal counts, filtered marker distances, deviation values and all marker filters.

Summary Resets a marker or resets marker handling.

- Sets two Flags (SYN_ACCEPTNXTM, SYN_ACCEPTNXTS) in PFG_SYNCSTART (114).

This will lead to the following result.

- Next master and next slave marker are accepted in all cases.
- Correction is calculated corresponding to the actual value of **SyncMarkerRestart**
- The resulting correction will be handled as a start correction
- The n:m relation of the markers should be kept (if possible).
- Marker Windows are reset corresponding to the parameter values
- All filters restart with default values.

This command also starts marker handling if not already active.

This command can be used to start marker handling outside of SyncMarker.

It also can be used to restart a running SyncMarker without losing n:m relation.

restarttype = 1

A real Reset of the marker handling will be done. That means that as well as the things described in Marker Restart, we also reset the following values

Markercounts - that means a n:m relation will be lost

SyncMarkerRestart is reevaluated. This will possibly result in a new start sequence if SyncMarker is active.

Axis process data PFG_SYNCSTART (114) is set to start condition. That means all flags are cleared (SYN_START, SYN_NOSLAVE, SYN_NOMASTER, SYN_NOMVEL, SYN_ACCEPTNXTM, SYN_ACCEPTNXTS).

This feature was created to allow a complete marker restart outside of SyncMarker. It does exactly the same as if you do a SyncMarker command. So the next markers will be accepted and there will be no faking if no markers are seen. That is the main difference to the normal restart function.

restarttype = 2

(Not for use with CAM control and marker correction)

The parameter 2 for **SyncMarkerRestart** allows to reset the marker distance and marker window size immediately without waiting for next SyncMarker or SyncSetCurve.

> Parameter 2 cannot run for CAM control with marker correction (i.e. for SyncCamMarkerMaster or SyncCamMarkerSlave modes of operation. CAM control uses a different marker handling, which does not result in the defined and expected behavior of parameter value 2.

Command Group Synchronization Commands

9.174 SyncPos

Summary Angle/position synchronization with the master.

Syntax

```
SyncPos(axis,...);
```

Parameter axis = Axis module number (see Command Reference).

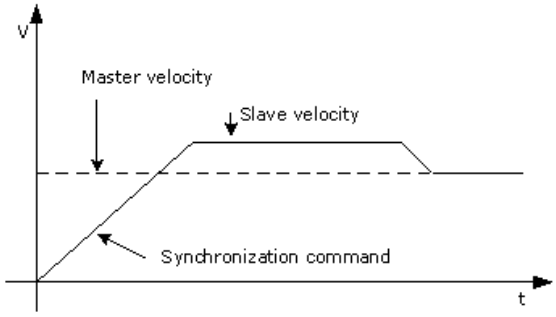
Return Value -

Description

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

The **SyncPos** command completes an angle-/position synchronization with the master. In doing so the position according to the gear factors to the master is kept synchronous, that means after an external disturbance the program subsequently tries to recover the corresponding stretch.

Summary Angle/position synchronization with the master.



Here the values set for velocity Vel, and acceleration Acc or Dec are not exceeded. However, In case Bit 1 (= +2) of SYNCTYPE is set, the slave should follow the master's angle/position without taking Vel, Acc and Dec limitations into account (see SYNCTYPE).

The following parameters affect the behavior:

gear factors: SYNCFACTM, SYNCFACTS
position offset: SYNCPOSOFFS
accuracy for flag: SYNCACCURACY

During synchronization the program proceeds as follows:

When the **SyncPos** command is started, the actual master position is determined and is retained. From the master velocity, and in consideration of the acceleration allowed, the necessary slave velocity is calculated in order to reach the master position. The slave is accelerated so long until the calculated position has been reached or until it is close enough to the reference position to reach it.

If RAMPTYPE ≥ 2, then synchronization is started with limited jerk. This only concerns the start; reaching the master velocity still looks like a trapezoid. This helps smoothen the start procedure in the case of heavy loads or fragile mechanics.

AxisPosAbsStart**SyncPos** does not contain the command AxisControl(ON) which turns on the motor control. When using **SyncPos** an explicit execution of AxisControl(ON) is necessary after a previous use of AxisControl(OFF).

> As soon as the deviation between the position of the slave and master is less than SYNCACCURACY, the ACCURACY flag is set.

In doing so the slave takes its own acceleration time into consideration and will start moving if necessary before the correct position has been reached if the master already has a higher velocity.

Instead of using this catch-up procedure it is also possible to move the slave with Cvel to approximately the same velocity as the master and then trigger **SyncPos**.

A change in the SYNCPOSOFFS during the synchronization leads to a new synchronization procedure with ramps (see above).

> **SyncPos** should only be called up once since the synchronizing continues until the next motion or stop command. All additional **SyncPos** commands cause the synchronization to start over again from the beginning and this is not normally intended, as you reset the actual SyncError.

Command Group Synchronization Commands

Syntax Example

```
SyncPos(0);     // Normal synchronization of the position.
Cvel(0, 50);    // Achieve velocity before synchronization.
AxisCvelStart(0);
Delay(500);
SyncPos(0);
```

Program Sample See **File** → **Sample**: PositionSynchronization.mc

What to do if... the synchronization drifts slowly?

Check whether **SyncPos** has been started more than once and eliminate the repeated commands.

9.175 SyncSetCurve

Summary Set a CAM curve.

Syntax `SyncSetCurve(axis, array);`

Parameter

axis	=	Axis module number (see Command Reference).
array	=	Name of the array or curve.

Return Value -

Description This command defines the curve which will be used the next time that curve synchronization begins a new curve (see SyncCam and SyncCamStart). This command must be executed before you may use the CurvePos, SyncCam, SyncCamMarkerMaster, SyncCamMarkerSlave, SyncCamStart, or SyncCamStop commands since these commands require a curve to be defined.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

See CAM Curves for an overview of curves and curve synchronization.

If curve synchronization is already active:

If **SyncSetCurve** is used while curve synchronization is active (i.e. SyncCam has been executed), then an active curve already exists and is being used. In this case, the specified curve will be preselected as *next* curve to be used when curve synchonization requires a new curve (see Dynamic CAM Curves). The existing active curve being used by curve synchronization is not affected in any way.

> If curve synchronization is active but a curve has not yet been started (i.e. SyncCamStart has not been executed), then a new CAM curve may be set if necessary by executing the **SyncSetCurve** command and then re-executing the SyncCam command to restart curve synchronization. This will cause the new curve to be activated.

If curve synchronization is not active:

If curve synchronization is not active but some other non-curve tracking *is* active (see below), then the new curve cannot be activated immediately. **SyncSetCurve** will simply preselect the curve as the *next* curve to be used when curve synchonization requires a new curve. This will happen, and the curve will be activated, when the next SyncCam, SyncCamMarkerMaster, or SyncCamMarkerSlave command is used.

If curve synchronization is not active and tracking is either not active or already tracking an old curve, then the new curve can be activated immediately.

Activating a curve results in the following actions:

- The preselected curve will become the active curve and will be used once curve synchronization becomes active. Any previously existing active curve is discarded.
- The curve master and slave positions will be initialized based on the current actual master and slave positions. This fixes the relative location of the curve with respect to the actual master and slave positions. For a description of exactly how this initialization is done, please see CAM Curve Scaling and Positioning.
- Tracking will be enabled for the curve. This means that PFG_CMASTERCPOS and PFG_CCURVEPOS will start being updated as the actual master position changes.

> Once a curve has been set, then SDOs PFG_CALCMCURVPOS and PFG_CALCCURVEPOS may be used to evaluate master and slave positions along the curve. Note that these values will be derived from the *active* curve, rather than from any pending preselected curve. See Dynamic CAM Curves for a description of active and preselected curves.

> Once curve tracking is enabled, the CurvePos command may be used to read the current position of the slave. In addition, SDOs PFG_CMASTERCPOS and PFG_CCURVEPOS will allow master and slave curve positions to be read.
>
> In general, tracking will continue even when curve synchronization is not active (i.e. before curve synchronization has been started or after curve synchronization has been exited altogether). This is done in order to allow applications to determine when and how synchronization can be smoothly re-enabled if necessary. Note that curve tracking is terminated if SyncPos or SyncMarker synchronization mode is started. Tracking is only possible in one synchronization mode at a time.

Summary Set a CAM curve.

Command Group Synchronization Commands

9.176 SyncShiftOrigin

Summary Relative shifting of the origin of synchronization.

Syntax `SyncShiftOrigin(axis, mvalue,...);`

Parameter

axis = Axis module number (see Command Reference).

mvalue = Relative offset in relation to the Master in qc.

Value range:
-MLONG / SYNCFACTS ... MLONG / SYNCFACTS

Return Value -

Description The command shifts the origin of synchronization in relation to the master. While SYNCPOSOFFS sets the ***position offset*** absolutely, **SyncShiftOrigin** always relates to the last one and shifts the offset position relatively. If you must shift the offset position continually, you can prevent too large numbers or an overflow in this way.

> Valid for position synchronization SyncPos and position synchronization with marker correction SyncMarker.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

Command Group Synchronization Commands

9.177 SyncStatus

Summary Query synchronization status.

Syntax `res = SyncStatus(axis);`

Parameter

axis = Axis module number (see Command Reference).

Return Value Synchronization status with the following meaning:

Value	Bit	
1	0	SYNCREADY
2	1	SYNCFAULT
4	2	SYNCACCURACY
8	3	SYNCMMHIT
16	4	SYNCSMHIT
32	5	SYNCMMERR
64	6	SYNCSMERR

Description The following flags are defined and can be queried with **SyncStatus**: READY, FAULT, ACCURACY and MHIT and MERR for both the master and the slave.

SYNCACCURACY

The controller checks whether SYNCERR < SYNCACCURACY is true every ms. If this is true, then SYNCACCURACY is set, otherwise the flag is deleted. This check is made for both SyncPos and SyncMarker.

This flag is not used with SyncVel.

When executing a SyncPos or SyncMarker command the flag is deleted.

SYNCFAULT / SYNCREADY

For every SyncPos or SyncMarker) command these flags are deleted. Subsequently the program checks whether SYNCACCURACY is set or not at every marker pulse of the slave (SyncPos) or when a marker pulse of the master and a marker pulse of the slave exist (SyncMarker).

If it is set the ready counter is increased and the fault counter is set to 0, otherwise the fault counter is increased and the ready counter set to 0.

Summary Query synchronization status.

If the ready counter is greater than the value determined by the parameter SYNCREADY, then the flag SYNCREADY is set. Otherwise it is deleted.

If the fault counter is greater that the value determined by the parameter SYNCFAULT then the flag SYNCFAULT is set. Otherwise it is deleted.

SYNCMMHIT/ SYNCSMHIT

SYNCMMHIT and SYNCSMHIT are set, if the master marker or the slave marker is occurred. These flags are deleted for every SyncMarker command. Subsequently the flag SYNCMMHIT is set after the first occurrence of a master marker pulse or after the n-th marker pulse (parameter SYNCMARKM).

The same is true for SYNCSMHIT with the slave.

> This flag is no longer deleted unless SyncMarker is started again or explicitly deleted with SyncStatusClear.

SYNCMMERR / SYNCSMERR

If in the ***Master Marker Tolerance Window*** SYNCMWINM or SYNCMWINS a tolerance range is defined, then SYNCMMERR or SYNCSMERR are set as soon as the maximum distance allowed has been achieved and no marker was identified.

Example:
Distance between two master markers SYNCMPULSM = 30000
Master Marker Tolerance Window SYNCMWINM = 1000
The flag is set at 31000 if no marker is identified.

These flags are deleted for every SyncMarker command.

If the ***Master Marker Tolerance Window*** is 0 and thus no tolerance range is defined, the program checks every marker pulse (or after every n-th pulse) whether the distance between the two last markers registered is less than 1.8 times the value defined by the parameter SYNCMPULSM. If not the corresponding flag is set.

The same applies analogously for SYNCSMERR in the slave.

> These flags are automatically reset: during the next successful marker correction and in the event of a new start of SyncMarker or through the command SyncStatusClear.

Command Group Synchronization Commands

Syntax Example

```
if (SyncStatus(0) & 0x04)
    DigOutput(1, 1);   // If ACCURACY then set output
```

9.178 SyncStatusClear

Summary Resetting of the flags MERR and MHIT.

Syntax

```
SyncStatusClear(axis, value,...);
```

Parameter

axis = Axis module number (see Command Reference).

value = 8 = SYNCMMHIT
16 = SYNCSMHIT
32 = SYNCMMERR
64 = SYNCSMERR

Return Value -

Description The **SyncStatusClear** command should only be used in a function for dealing with errors. See ERROR.

The corresponding bits can be reset in SyncStatus with **SyncStatusClear** *value* thus resetting the error flags MERR and the HIT flags MHIT. None of the other flags can be altered.

Command Group Synchronization Commands

9.179 SyncVel

Summary Velocity synchronization with the master.

Syntax `SyncVel(axis,...);`

Parameter axis = Axis module number (see Command Reference).

Return Value -

Description

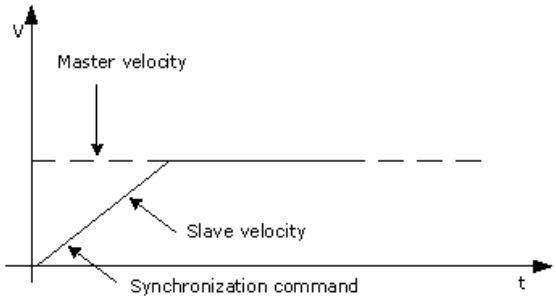
> **SyncVel** should only be called up once since the synchronizing continues until the next motion or stop command. All additional **SyncVel** commands cause the synchronization to start over again from the beginning and this is not normally intended, as you reset the actual synchronization error.

> Track and synchronization error is not monitored in **SyncVel** mode.

> This command cannot be used with any motor that is currently in kinematics mode (see KinematicsSetup).

AxisCvelStart**SyncVel** does not contain the command AxisControl(ON) which turns on the motor control. When using **SyncVel** an explicit execution of AxisControl(ON) is necessary after a previous use of AxisControl(OFF).

With **SyncVel** velocity synchronization with the master is completed, for example after an external disturbance. In doing so only the velocity is taken into consideration and the controller does not attempt to recover the position.



For synchronization and during the synchronization process neither the pre-set velocity, Vel, nor the pre-set acceleration, Acc or Dec, are exceeded. However, In case Bit 1 (= +2) of SYNCTYPE is set, the slave should follow the master's velocity without taking Vel, Acc and Dec limitations into account (see SYNCTYPE).

The parameters of the gear factors used for synchronization are: SYNCFACTM, SYNCFACTS.

The velocity controller for SYNCV functions according to the following formula:

1 = (FFVEL standardized)* (Setpoint velocity CV)

2 = (KFFACC / KFFDEC standardized)* (Setpoint acceleration CA)

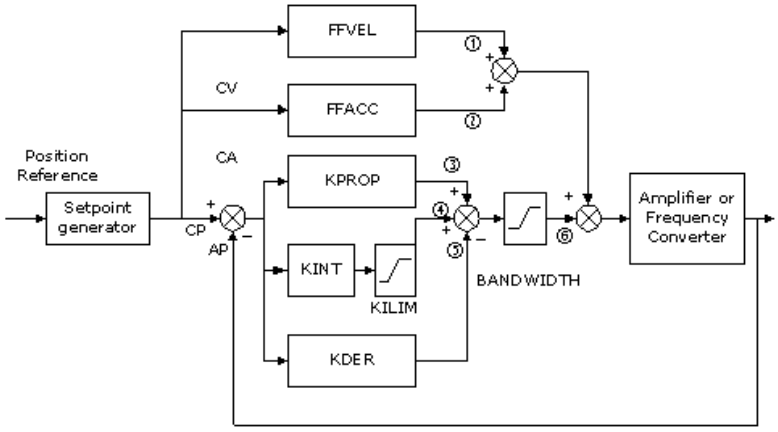
3 = KPROP * (Velocity deviation CV - AV)

4 = KINT * (Sum of all previous velocity deviations) (limited by KILIM standardized)

5 = KDER * (Difference of velocity deviation)

6 = 3 + 4 + 5 (limited by BANDWIDTH)

Summary Velocity synchronization with the master.



AV is the actual velocity; it is calculated as difference of the actual velocity minus last velocity

AP is the actual position (calculated from encoder feedback) in qc.

CP is the current position setpoint in qc.

CV is the velocity setpoint in qc/ms;
it is calculated as difference of the actual position setpoint minus the last position setpoint.

CA is the acceleration setpoint

Velocity deviation = calculated by CV-AV.

Difference of the velocity deviation = velocity deviation - old velocity deviation

KILIM standardized

FFVEL standardized

KFFACC / KFFDEC standardized

Furthermore, the speeds are not simply determined on the basis of the difference between the current position minus the last position (master/slave), rather the values are filtered according to the settings in SYNCVFTIME. This means the filter for the slave is determined by the maximum speed; in other words:
MAXVEL * 5 corresponds to the encoder resolution for the filter table, where MAXVEL is the speed in qc/ms. (The formula is a result of the assumption that the filter table for the encoder resolution was made with a maximum speed of 3000 RPM.)

During the transition from the speed controller to the position controller this is done as smoothly as possible. In addition the new set position is defined in such a manner that the following is true:

command_pos = actual_pos + error

old_error, Cvel, Avel are maintained. SUMERR is also accepted unchanged.

Command Group Synchronization Commands

9.180 Sysvar

Summary System variable (Pseudo array) reads system values.

Syntax `Sysvar[n]`

Parameter n = SDO index (see below).

Description Using the **Sysvar** pseudo array, it is possible to read system and process data. This index can also be used to link the system variable using LinkPdo or LinkSdo or specify recording data with RecordIndex.

The SDO index is expected to be constructed as follows:

Summary System variable (Pseudo array) reads system values.

0x01nnnnss = Can Index nnnn, Subindex ss

See SDO Dictionary.

> The values of the system variables are internal, hardware-dependent values which may change.

> ⚠ If parameters are modified by an application program with **Sysvar**, then it is necessary to re-execute the Vel, Acc, and Dec commands in order to activate the new settings.

Command Group System Information

9.181 Time

Summary Query the system time.

Syntax `res = Time();`

Parameter -

Return Value Elapsed system time since power on.

Description This value is an internal counter that increments in very close to 1 ms intervals, although the exact interval depends on the type of controller. The **Time** command is useful for calculating the execution time of a command sequence or device cycle time. TimeHw is a similar command that provides a finer resolution.

> ⚠ The system time starts at 0 when the controller is powered on. It will count up to 2147483647 (0x7FFFFFFF) and then "wrap around" to -2147483648 (0x80000000) before starting to count up again. This is important if the user program is calculating time differences because the system time may "wrap" between the start and end times. The first "wrap" following power on occurs after about 24 days; subsequent "wraps" occur approximately every 49 days.

Command Group System Commands

Program Sample See **File** → **Sample**: BasicPositionControl.mc

9.182 TimeDiffus

Summary Calculate an elapsed time.

Syntax `res = TimeDiffus(expr);`

Parameter expr = Previous time.

Return Value Elapsed time between the current time and the previous time.

Description This returns the time difference in microseconds between the current time and the time given by "expr", where "expr" is a value returned by the TimeHw command. Note that "expr" is hardware-dependent and will not necessarily be in microseconds.

Command Group System Commands

9.183 TimeHw

Summary Query the system timer.

Syntax `res = TimeHw();`

Parameter -

Return Value System time in ticks.

Description This value is an internal counter that increments in very close to 1 microsecond intervals, although the exact interval depends on the type of controller. The **TimeHw** command is useful for calculating the execution time of a command sequence or device cycle time. Time is a similar command that provides a millisecond resolution and can therefore be

Summary Query the system timer.

used to measure much longer time intervals.

The unit is hardware dependent. See Controller Dependent Feature List.

To determine a time difference in microseconds, use the function TimeDiffus.

> ⚠ The system time starts at 0 when the controller is powered on. It will count up to 2147483647 (0x7FFFFFFF) and then "wrap around" to -2147483648 (0x80000000) before starting to count up again. This is important if the user program is calculating time differences (without using TimeDiffus) because the system time may "wrap" between the start and end times. The first "wrap" following power on occurs after about 35 minutes; subsequent "wraps" occur approximately every 71 minutes.

Command Group System Commands

9.184 TrackError

Summary Query the the actual tracking error of an axis.

Syntax `res = TrackError(axis);`

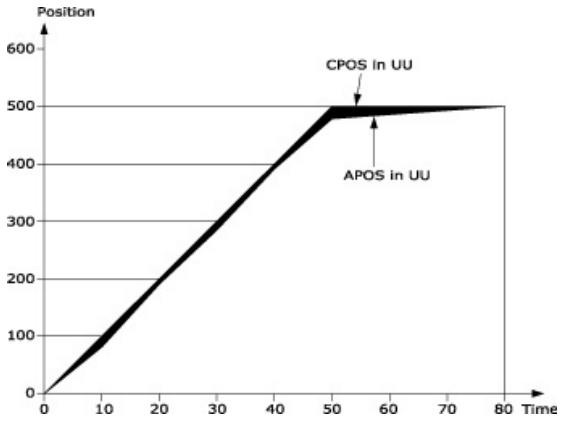
Parameter axis = Axis module number (see Command Reference).

Return Value Actual tracking error of the axis in uu.

Description This queries the difference between Cpos and Apos (i.e. the difference between where the axis *should* be and where the axis actually *is*.

The actual position will often lag behind the command position. This is normal and it can happen for a variety of reasons. For example, the motor may not be physically able to match the acceleration requested by the application program (i.e. the motor cannot "keep up" and its position will lag behind where it should be). The controller automatically tries to correct tracking errors.

The dark shaded region between the Cpos and the Apos in the diagram below, shows what the tracking error might look like.



Command Group Axis and Information Commands

Sample

```
AxisPosAbsStart(0, 500);
while (1)
    {
    print("Command Position ", Cpos(0));
    print("Actual Position  ", Apos(0));
    print("Error ", TrackErr(0));
    Delay(10);
    }
```

Both of the above code fragments will produce output similar to the following:

```
Command Position 100
```

Summary Query the the actual tracking error of an axis.

```
Actual Position  98
Error 2
Command Position 200
Actual Position  199
Error 1
Command Position 300
Actual Position  297
Error 3
Command Position 500
Actual Position  500
Error 0
...
```

9.185 TransIdent

Summary Set a transformation to the identity transformation (i.e. no transformation).

Syntax `TransIdent(trns);`

Parameter trns = **transform** variable to be set.

Return Value -

Description The **TransIdent** command will set the specified transformation variable to the identity transformation, clearing any previous transformation contained in the variable.

See Transformations for a more detailed description of transformations.

Command Group Kinematics Commands

Syntax Example "PathToPaper" will be used to transform path curve points to paper coordinates.

```
transform PathToPaper;

TransIdent(PathToPaper);                  // Initialize.
TransScale(PathToPaper, 2.0);             // Double the size.
TransRotateXY(PathToPaper, rad(45.0));    // Rotate 45 degrees.
TransTranslate(PathToPaper, 500, 500, 0); // Move to (500,500).
```

9.186 TransInverse

Summary Transform a **dpoint** variable by the inverse of a transformation.

Syntax `TransInverse(trns, pnt);`

Parameter trns = **transform** variable to be used.

pnt = **dpoint** variable to be transformed.

Return Value -

Description The **TransInverse** command will transform the specified point by the *inverse* of the transformation.

Command Group Kinematics Commands

Syntax Example

```
transform PathToPaper;
dpoint    pnt;

TransInverse(PathToPaper, pnt);
```

9.187 TransPoint

Summary Transform a **dpoint** variable by a transformation.

Syntax `TransPoint(trns, pnt);`

Parameter

trns = **transform** variable to be used.

pnt = **dpoint** variable to be transformed.

Return Value -

Description The **TransPoint** command will transform the specified point by the transformation.

Command Group Kinematics Commands

Syntax Example

```
transform PathToPaper;
dpoint    pnt;

TransPoint(PathToPaper, pnt);
```

9.188 TransReflect

Summary Reflect a transformation across the YZ plane (i.e. X = 0).

Syntax `TransReflect(trns);`

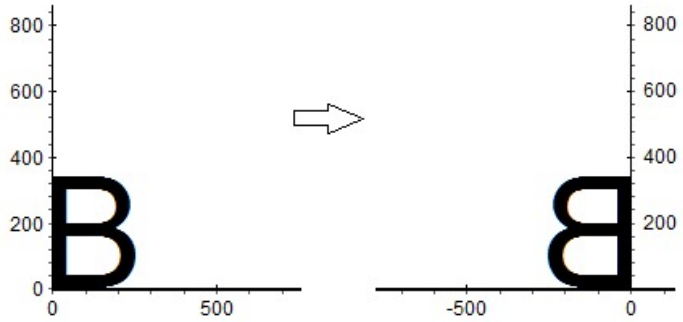
Parameter

trns = **transform** variable to be reflected.

Return Value -

Description The **TransReflect** command will reflect the transformation contained in the specified transformation variable across the YZ plane. This effectively reverses the sign of all X coordinates.

This diagram shows the effect of a reflection:



See Transformations for a more detailed description of transformations.

Command Group Kinematics Commands

Syntax Example "PathToPaper" will be used to transform path curve points to paper coordinates.

```
transform PathToPaper;

TransIdent(PathToPaper);                // Initialize.
TransReflect(PathToPaper);              // Reflect right-to-left.
TransTranslate(PathToPaper, 500, 500, 0);  // Move to (500,500).
```

9.189 TransRotateXY

Summary Rotate a transformation in the XY plane (i.e. about the Z axis).

Syntax `TransRotateXY(trns, angle);`

Parameter

trns = **transform** variable to be rotated.

angle = Floating point rotation angle in radians.

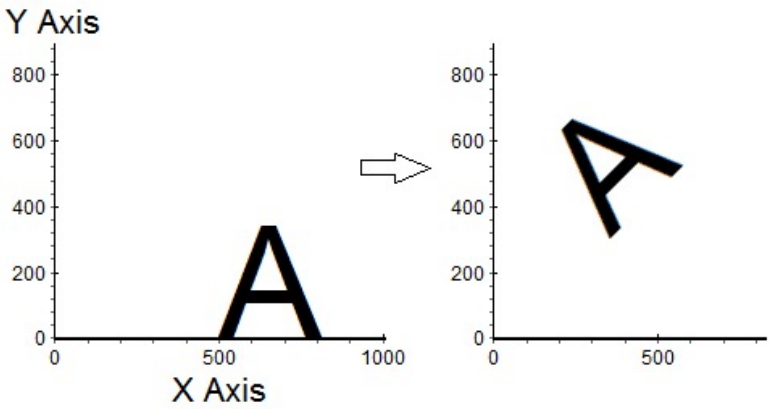
Summary Rotate a transformation in the XY plane (i.e. about the Z axis).

Return Value -

Description The **TransRotateXY** command will rotate the transformation contained in the specified transformation variable in the XY plane (i.e. about the Z axis) by the specified angle. A positive angle corresponds to a counter clockwise rotation when looking at the XY plane.

The TransRotateYZ and TransRotateZX commands will do similar rotations in the other axes.

This diagram shows the effect of a rotation by 45 degrees (π/4 radians):



See Transformations for a more detailed description of transformations.

Command Group Kinematics Commands

Syntax Example "PathToPaper" will be used to transform path curve points to paper coordinates.

```
transform PathToPaper;

TransIdent(PathToPaper);                  // Initialize.
TransScale(PathToPaper, 2.0);             // Double the size.
TransRotateXY(PathToPaper, rad(45.0));    // Rotate 45 degrees.
TransTranslate(PathToPaper, 500, 500, 0); // Move to (500,500).
```

9.190 TransRotateYZ

Summary Rotate a transformation in the YZ plane (i.e. about the X axis).

Syntax `TransRotateYZ(trns, angle);`

Parameter

trns = **transform** variable to be rotated.

angle = Floating point rotation angle in radians.

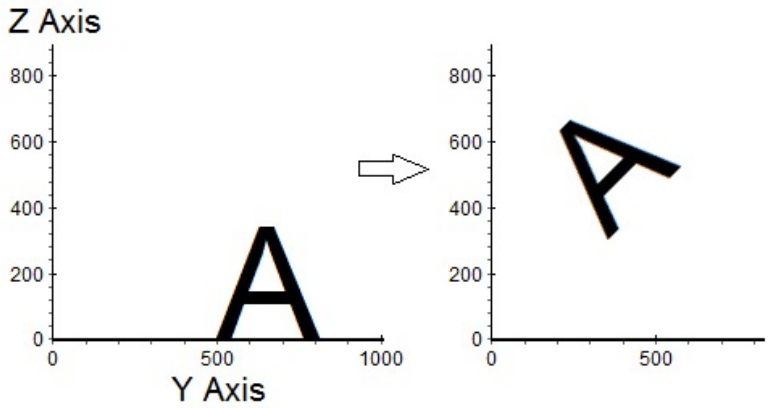
Return Value -

Description The **TransRotateYZ** command will rotate the transformation contained in the specified transformation variable in the YZ plane (i.e. about the X axis) by the specified angle. A positive angle corresponds to a counter clockwise rotation when looking at the YZ plane.

The TransRotateXY and TransRotateZX commands will do similar rotations in the other axes.

This diagram shows the effect of a rotation by 45 degrees (π/4 radians):

Summary Rotate a transformation in the YZ plane (i.e. about the X axis).



See Transformations for a more detailed description of transformations.

Command Group Kinematics Commands

Syntax Example "PathToPaper" will be used to transform path curve points to paper coordinates.

```
transform PathToPaper;

TransIdent(PathToPaper);                    // Initialize.
TransScale(PathToPaper, 2.0);               // Double the size.
TransRotateYZ(PathToPaper, rad(45.0));      // Rotate 45 degrees.
TransTranslate(PathToPaper, 500, 500, 0);   // Move to (500,500).
```

9.191 TransRotateZX

Summary Rotate a transformation in the ZX plane (i.e. about the Y axis).

Syntax `TransRotateZX(trns, angle);`

Parameter

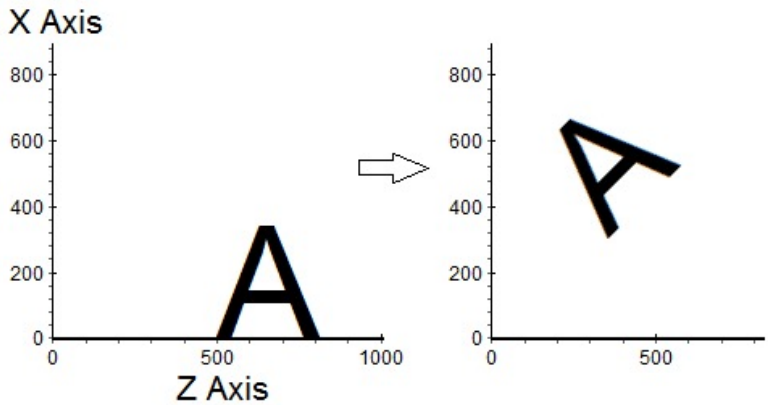
trns	=	**transform** variable to be rotated.
angle	=	Floating point rotation angle in radians.

Return Value -

Description The **TransRotateZX** command will rotate the transformation contained in the specified transformation variable in the ZX plane (i.e. about the Y axis) by the specified angle. A positive angle corresponds to a counter clockwise rotation when looking at the ZX plane.

The TransRotateXY and TransRotateYZ commands will do similar rotations in the other axes.

This diagram shows the effect of a rotation by 45 degrees (π/4 radians):



See Transformations for a more detailed description of transformations.

Summary Rotate a transformation in the ZX plane (i.e. about the Y axis).

Command Group Kinematics Commands

Syntax Example "PathToPaper" will be used to transform path curve points to paper coordinates.

```
transform PathToPaper;

TransIdent(PathToPaper);                  // Initialize.
TransScale(PathToPaper, 2.0);             // Double the size.
TransRotateZX(PathToPaper, rad(45.0));    // Rotate 45 degrees.
TransTranslate(PathToPaper, 500, 500, 0); // Move to (500,500).
```

9.192 TransScale

Summary Scale a transformation.

Syntax `TransScale(trns, factor);`

Parameter

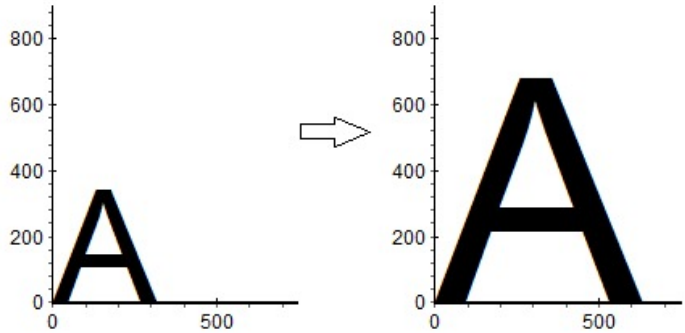
trns	=	**transform** variable to be scaled.
factor	=	Floating point scale factor (≠0).

Return Value -

Description The **TransScale** command will scale the transformation contained in the specified transformation variable by the specified scale factor. So for example, a scale factor of 2 will double the size a path curve if the transformation is used with the PathStart command.

The scale factor must not be 0 (or close to 0). If it is, then the command will be ignored. Note that negative scale factors are allowed.

This diagram shows the effect of a scale factor of 2:



See Transformations for a more detailed description of transformations.

Command Group Kinematics Commands

Syntax Example "PathToPaper" will be used to transform path curve points to paper coordinates.

```
transform PathToPaper;

TransIdent(PathToPaper);                  // Initialize.
TransScale(PathToPaper, 2.0);             // Double the size.
TransRotateXY(PathToPaper, rad(45.0));    // Rotate 45 degrees.
TransTranslate(PathToPaper, 500, 500, 0); // Move to (500,500).
```

9.193 TransStretch

Summary Stretch a transformation independently in each dimension.

Syntax

```
TransStretch(trns, xfact, yfact);
TransStretch(trns, xfact, yfact, zfact);
```

Summary Stretch a transformation independently in each dimension.

Parameter

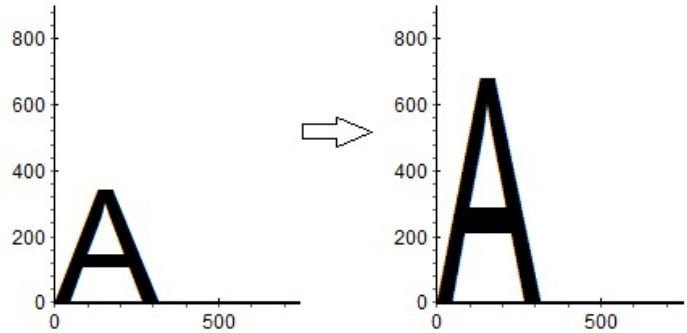
trns	=	**transform** variable to be stretched.
xfact	=	Floating point scale factor for the X dimension (≠0).
yfact	=	Floating point scale factor for the Y dimension (≠0).
zfact	=	Floating point scale factor for the Z dimension (≠0). This parameter is optional and will default to 1.

Return Value -

Description The Transformations command will stretch the transformation contained in the specified transformation variable. Each dimension will be stretched independently. Note that negative factors are allowed and will have the same effect as reflecting the transformation in the corresponding plane. So for example, an X factor of 1 and a Y factor of 2 will double the height a path curve if the transformation is used with the PathStart command.

No factor may be 0 (or close to 0). If any factor is 0, then the command will be ignored.

This diagram shows the effect of an X factor of 1 and a Y factor of 2:



See Transformations for a more detailed description of transformations.

Command Group Kinematics Commands

Syntax Example "PathToPaper" will be used to transform path curve points to paper coordinates.

```
transform PathToPaper;

TransIdent(PathToPaper);                  // Initialize.
TransStretch(PathToPaper, 1.0, 2.0);      // Double the height.
TransRotateXY(PathToPaper, rad(45.0));    // Rotate 45 degrees.
TransTranslate(PathToPaper, 500, 500, 0); // Move to (500,500).
```

9.194 TransTranslate

Summary Translate a transformation.

Syntax

```
TransTranslate(trns, dx, dy);
TransTranslate(trns, dx, dy, dz);
```

Parameter

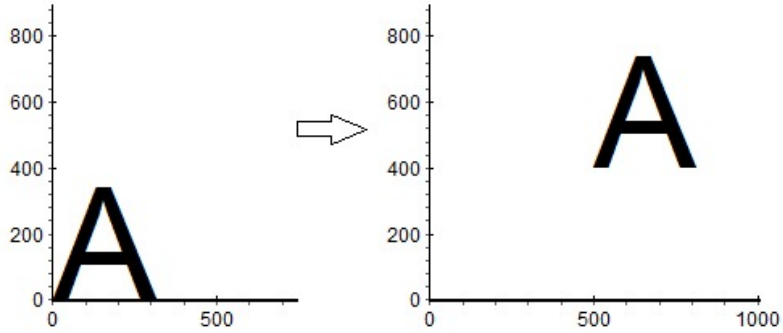
trns	=	**transform** variable to be translated.
dx	=	Floating point distance to be translated in the X coordinate.
dy	=	Floating point distance to be translated in the Y coordinate.
dz	=	Floating point distance to be translated in the Z coordinate. This parameter is optional and will default to 0.

Return Value -

Description The **TransTranslate** command will translate the transformation contained in the specified transformation variable by the specified amount. So for example, this will move a path curve if the transformation is used with the PathStart command.

Summary Translate a transformation.

This diagram shows the effect of a translation by 500,400,0:



See Transformations for a more detailed description of transformations.

Command Group Kinematics Commands

Syntax Example "PathToPaper" will be used to transform path curve points to paper coordinates.

```
transform PathToPaper;

TransIdent(PathToPaper);                  // Initialize.
TransScale(PathToPaper, 2.0);             // Double the size.
TransRotateXY(PathToPaper, rad(45.0));    // Rotate 45 degrees.
TransTranslate(PathToPaper, 500, 500, 0); // Move to (500,500).
```

9.195 Tvel

Summary Set the target velocity for the AxisPosAbsStartTvel command.

Syntax `Tvel(axis, vel,...);`

Parameter

axis	=	Axis module number (see Command Reference).
vel	=	Scaled velocity value.

Return Value -

Description The **Tvel** command sets the target velocity for the next AxisPosAbsStartTvel command. This is the end velocity required when the target position is finally reached; it is not the velocity that will be used during the move.

The value is specified as a scaled velocity value (see Vel for more infomation about scaled velocity values):

Target Velocity [RPM] = vel * ***Maximum Velocity*** (VELMAX)
Velocity Resolution (VELRES)

The target velocity remains valid until a new velocity is set using another **Tvel** command. The target velocity will default to 0 if it has not been specified before AxisPosAbsStartTvel is executed.

> If any parameter affecting the maximum velocity is changed (e.g. VELMAX, VELRES, gearing factors, etc.), then the **Tvel** command must be re-executed. Parameter changes are *not* automatically incorporated into existing target velocities.

Command Group Axis and Information Commands

Syntax Example

```
Tvel(0, 50);                    // Set the final target velocity to 50.
AxisPosAbsStartTvel(0, 50000);  // Move axis module 0 to position 50000.
```

9.196 UserStatus

Summary Set the user status.

Syntax `UserStatus(value);`

Parameter value = value to be set

Return Value -

Description Sets the user status and controls the yellow LED(s) on the controller.

```
UserStatus(1); // enable first status LED on the controller
UserStatus(2); // enable second status LED on the controller
```

As a standard function, the 'ST' LED on the controller will be enabled as soon as a program starts and disabled as soon as it quits.

The user status can also be read with SDO 0x2206 / 0, byte 3.

> If the LED is enabled by the Autostart program, it will be overwritten by the **UserStatus** command, e.g. if UserStatus(20) is executed, the led will also be disabled because bit 0 is set to 0.

Command Group System Commands

9.197 Vel

Summary Set velocity for relative and absolute motion.

Syntax `Vel(axis, vel,...);`

Parameter

axis	=	Axis module number (see Command Reference).
vel	=	Scaled velocity value.

Return Value -

Description The velocity for the next absolute and relative positioning procedures and the maximum allowed velocity for synchronizing procedures are determined with the **Vel** command.

Command Velocity [RPM] = vel * ***Maximum Velocity*** (VELMAX)
Velocity Resolution (VELRES)

The value remains valid until a new velocity is set via another **Vel** command. The new velocity value will be set in reference to the parameters ***Maximum Velocity*** VELMAX and ***Velocity Resolution*** VELRES. If the velocity value equals the maximum speed scaling, then the RPM value set in ***Maximum Velocity*** will be used.

NOTE: Slave velocity in synchronizing mode is also limited by the **Vel** command.

> If no velocity has been set prior to a positioning or synchronizing command, then the value of parameter DFLTVEL will be used.

If the velocity needs to be altered during positioning, it is possible when the **Vel** command is followed by another AxisPosAbsStart command targeting to the desired position.

The maximum speed allowed can be changed at any time with the command **Vel**, if a SyncVel, SyncPos, or SyncMarker follows the **Vel** command.

Default The default value is 100.

Command Group Axis and Information Commands

Syntax Examples

```
Vel(0, 100);        // Velocity: Axis mod 0: 100
Vel(0, 50, 1, 10);  // Velocity: Axis mod 0: 50, Axis mod 1: 10
Vel(AXALL, 30);     // Velocity: all Axes: 30
```

Program Sample See **File** → **Sample**: BasicPositionControl.m

9.198 WdActivate

Summary Activate a watchdog.

Syntax `WdActivate(time);`

Parameter time = Trigger time in milliseconds.

Return Value -

Description Activate a watchdog with a trigger time of "time" milliseconds. The minimum time for "time" is 120ms, the maximum is 5000ms.

After activation, the watchdog must be reset using the WdReset command before *each* cycle of the trigger time has expired. If the WdReset is not executed before "time" milliseconds has expired, then the controller will reboot.

When the controller is rebooted by the watchdog, Error 18 will be active after startup.

For more details on the use of the watchdog, please see Watchdog.

Portability This feature can be controller dependent and is listed in the Controller Dependent Feature List.

Command Group System Commands

9.199 WdReset

Summary Reset the watchdog.

Syntax `WdReset();`

Parameter -

Return Value -

Description Reset the watchdog. Note that the watchdog will need to be reset before the *each* trigger timeout cycle expires. Resetting the watchdog does not deactivate the watchdog; it only resets the timer. The watchdog is activated using the WdActivate command.

For more details on the use of the watchdog, please see Watchdog.

Portability This feature can be controller dependent and is listed in the Controller Dependent Feature List.

Command Group System Commands

10 Parameter Reference

The controller supports a significant number of operational "parameters". These can be set by the user to control all aspects of the operation of the controller. For example, maximum velocities, maximum accelerations, maximum allowable tracking errors, etc., can be specified. See **Parameter Reference** for a complete list of the available parameters.

The controller supports a "temporary" set of all parameters in RAM (a "permanent" copy of all parameters in flash memory is not supported). The parameters located in RAM will keep their values only during a *single* execution of an application program. At power-on and *each* time a program starts, the parameters are restored with default values. This has the advantage that application programs always start with the same known parameter values. While user application programs are executing the parameters in the controller can be changed at any time by the application program or through SDO access.

Parameter values may be changed by the user either with **Controller** → **Parameters** → **Global / Axis** or with the Module Editor or Array Editor. They may also be changed by an application program using the Sysvar array. However, parameter values that are modified by a user application program are updated *only* in RAM and must be explicitly copied to an array and saved in flash memory (using the Save command) if that is the user's intent. Otherwise, after a controller reset or the next time a program is started, the modified temporary parameter settings are discarded and the settings will be restored to the default parameter values. Parameters in arrays and user parameters can be reset with help of the ApossIDE. **Controller** → **Reset** → **Parameter**.

The ApossIDE can be used to back up the user parameter and array values to a file on the PC hard drive. This is done using **Controller** → **Parameters** → **Save to file**. These values can then be restored by the user at a later time with **Controller** → **Parameters** → **Restore from file**. As a result, **Restore from file** cannot generally be used to initialize parameter values for application programs. To handle the issue of initializing temporary parameters, users should design their programs either to use #config statements or to use "Parameter Include Files" (see Initializing Temporary Parameters).

See System Architecture for more information.

> ⚠ The EPROM used for flash memory has limited lifetime. However, it can be reprogrammed approximately 10000 times.

> ⚠ Some parameters specify values limited to MLONG. This is derived from a signed 31-bit integer (*not* a 32-bit integer) and has a value of 1,073,741,823 or 0x3FFFFFFF.

> ⚠ Some parameters specify a very wide limit range. This is due to large differences in motor performance and the wide variety of possible applications. Neither the ApossIDE nor the controller firmware are able to consistently verify that values are within range. It is the responsibility of the programmer and user to ensure that parameter values are within the performance ranges of the drive and system. Out-of-range values may result in incorrect execution and possible damage to equipment.

The following groups of permanent parameters may be changed by the user:

User Parameters	Parameters intended to be used as application-specific program initialization or configuration values.
Global Parameters	Parameters affecting the controller as a whole.
Axis Parameters	Parameters affecting each single axis (i.e. each motor).

All other parameters supported by the controller exist *only* as temporary parameters and are therefore *always* initialized to default values when the controller is powered on and when application programs are started. See Initializing Temporary Parameters for the recommended method of handling parameter initialization within

application programs.

> ⚠ For MACS5 and newer controllers (firmware version 7.0.00 and later), permanent axis parameters are *no longer* supported. Please see Initializing Temporary Parameters for the recommended method of handling axis parameter initialization in these controllers.

10.1 Initializing Temporary Parameters

Each time an application program is started, all temporary parameter values in RAM are initialized with default values saved in firmware memory. Application programs will then always access the temporary parameters in RAM. This has the advantage that programs always start with the same known parameter values in RAM.

If programs need to have parameter values initialized to values *other* than those specified by firmware, then the recommended method of doing this is with a "Parameter Include File". This is a file containing a program fragment that will explicitly initialize the parameters that must be initialized to non-default values. The program should execute this fragment of code as part of the program's initialization.

The ApossIDE provides several ways to automatically generate a Parameter Include File (e.g. using **Controller** → **Parameters** → **Save to file**).

Parameter Include Files

The "Parameter Include File" will have a format similar to the following example:

```
long SetConfigParams()
{

    // Axis 0 Temporary Params

    AXE_PARAM(0,POSFACT_Z) = 1;
    AXE_PARAM(0,POSFACT_N) = 1;
    AXE_PARAM(0,DFLTVEL) = 100;
    AXE_PARAM(0,DFLTACC) = 100;
    AXE_PARAM(0,PISRC_MVIRTCOUNT) = 32;
    AXE_PARAM(0,PISRC_MVIRTLATCH) = 32;

    // Axis 1 Temporary Params

    AXE_PARAM(1,DFLTVEL) = 100;
    AXE_PARAM(1,DFLTACC) = 100;
    AXE_PARAM(1,PISRC_SVIRTCOUNT) = 1;
    AXE_PARAM(1,PISRC_MVIRTCOUNT) = 33;
    AXE_PARAM(1,PISRC_SVIRTLATCH) = 1;
    AXE_PARAM(1,PISRC_MVIRTLATCH) = 33;

    ....


    return(0);
}
```

This file is intended to be a starting point for modifications. Users may edit this file in any way. For example, if there are unused axes in the application, then the code may be deleted for these axes. Similarly, unused parameters may be deleted or more complicated custom initializations may be done or additional comments

added.

Please note the following:

1. Parameters are initialized by calling the “SetConfigParams” function. There is nothing special about this function. It may be called from anywhere within a user program where parameter initialization must be done.
2. The default “SetConfigParams” function always returns a 0 return code. The calling program can ignore this return code. Note that the user can modify this behaviour in any way that is appropriate for the application.
3. Parameter Include Files normally do *not* include global parameters, user parameters, or DIM arrays since user changes to these can be saved permanently in the controller's flash memory. However, users are free to manually add these parameters if that is appropriate for their particular program.

The include file is expected to be integerated into the user’s main program in some way similar to the following example:

```
#include <SysDef.mh>
#include "SetConfigParams.mh"

////////////////////////////////////////////////////////////
long main(void)
{
        // Initialize all temporary parameters
        SetConfigParam();

        // The rest of the mainline goes here.

    return(0);
}
```

10.2 User Parameters

There are 100 user parameters available to application programs in the controller. Since user parameter values are permanent and persist across programs starts, they are intended to be used as application-specific program initialization or configuration values. They may be used on an “as-needed” basis and their meanings and values are entirely user-defined.

Users may view and modify user parameters using the Module Editor or Array Editor. Parameter values modified by the user with the ApossIDE are automatically updated in both RAM and flash memory so they will become permanent.

Application programs can read and write user parameters by using the predefined array Sysvar[0x01220100+1...100]. If the "SysDef.mh" file (see SysDef Defines) has been included, then statements of the form "USER_PARAM(1...100)" may be used.

If the controller is part of network, then it is possible for a PLC, PC, or any other control unit to access to the user parameters by using SDO 0x2201 (see User Parameters). These changes are temporary only. Application programs can use the Save command to save user parameters permanently in flash memory.

10.3 Global Parameters

Global parameters affect the controller as a whole. For example, they define the communication attributes of the

controller and how the controller behaves when it powers up. Note that individual parameters may or may not be available depending on the type and version of controller.

Users may view and modify global parameters using **Controller** → **Parameters** → **Global / Axis** or the Module Editor or Array Editor. Parameter values modified by the user with the ApossIDE are automatically updated in both RAM and flash memory so they will become permanent.

Application programs can read and write global parameters by using the predefined array Sysvar. If the controller is part of network, then it is possible for a PLC, PC, or any other control unit to access to the global parameters by using SDO 0x2200 (see Global Parameters). These changes are temporary only. Application programs can use the Save command to save user parameters permanently in flash memory.

The following global parameters are available:

Parameter Name	Parameter Description
CANBAUD	CAN-Baud rate.
CANNR	CAN node ID.
CANSYNCTIMER	Defines the cycle time for sending SYNC telegrams on the CAN bus.
ESCCONDGLB	Set all outputs to 0 or 1 in case of abortion
I_ERRCLR	Clear error.
IPADDRESSMODE	Mode of Ethernet addressing
IPDEFAULTGW	Default gateway for Ethernet interface
IPMASK	Subnet mask for Ethernet interface
IPSUBNET	Network number or IP address.
O_ERROR	Output for error.
PRGPAR	Activated program number.
PRGSTARTCOND	Influences the behavior when a program is started.

10.4 Axis Parameters

Axis parameters affect the axis being driven by the controller. For example, they define maximum speeds and accelerations, acceleration profiles, encoder tick rates and synchronization characteristics, etc. Note that individual parameters may or may not be available depending on the type and version of controller.

Users may view and modify axis parameters using the Module Editor or Array Editor.

Application programs can read and write axis parameters by using the predefined array Sysvar. If the "SysDef.mh" file (see SysDef Defines) has been included, then statements of the form "AXE_PARAM(..)" may be used. Ifthe controller is part of network, then it is possible for a PLC, PC, or any other control unit to access to the axis parameters by using SDO 0x2300 (see Axis Parameters). These changes are temporary only. All axis parameter that differ from default value must set in the application program.

> There are more parameters for specific virtual function modules. Please check SDO Dictionary for more information. (see SDO Dictionary)

The following axis parameters are available.

Parameter Name	Parameter Description	Topic
ACCMAXQC	Maximum acceleration.	-
BANDWIDTH	PID Bandwidth.	Position Control
DFLTACC	Default acceleration.	Basic Settings
DFLTVEL	Default velocity.	Basic Settings
ENDSWMOD	Behavior at end limit switch.	Predefined I/Os
ERRCOND	Behavior after error.	Predefined I/Os
ESCCOND	Condition on program termination.	Predefined I/Os
FEEDDIST	User unit factor to get CANopen compatible user units.	-
FEEDREV	User unit factor to be CANopen compatible.	-
FFVEL	Velocity feed forward.	Position Control
HOME_FORCE	Forced movement to home position.	Homing
HOME_OFFSET	Zero point offset from Home Position.	Homing
HOME_RAMP	Ramp for home motion.	Homing
HOME_VEL	Velocity of home motion.	Homing
HOME_TYPE	Behavior during home motion.	Homing
HOMEZEROVEL	Scaled velocity for moving into the encoder index as part of the HOME command or CANopen homing procedures.	-
I_NEGLIMITSW	Negative limit switch.	Predefined I/Os
I_POSLIMITSW	Positive limit switch.	Predefined I/Os
I_REFSWITCH	Input for reference switch.	Predefined I/Os
I2TFTIME	Filter time in ms to build up the I^2 filtered value.	Amplifier
I2TLIMIT	Limitation of the I2TVALUE.	Amplifier
JERKMIN	Minimum time required before reaching the maximum acceleration.	Basic Settings
JERKMIN2	Minimum time required to ramp the acceleration down from maximum acceleration to 0.	Basic Settings
JERKMIN3	Minimum time required to ramp the deceleration up from 0 to maximum deceleration.	Basic Settings
JERKMIN4	Minimum time required to ramp the deceleration down from maximum deceleration to 0.	Basic Settings
KDER	Derivative factor for PID control.	Position Control
KFFACC	Acceleration feed forward.	Position Control

Parameter Name	Parameter Description	Topic
KFFDEC	Deceleration feed forward.	Position Control
KILIM	Limit value for integral sum for PID control.	Position Control
KILIMTIME	Time (ms) which is used to increase or decrease the integral limit of the position control loop up to KILIM.	Position Control
KINT	Integral value for PID control.	Position Control
KPROP	Proportional factor for PID control.	Position Control
NEGLIMIT	Negative software end limit.	Predefined I/Os
O_AXMOVE	Output for motion command active.	Predefined I/Os
O_BRAKE	Output for mechanical brake.	Predefined I/Os
PISRC_MVIRTCOUNT	Linked virtual counter module number for master information.	-
PISRC_MVIRTLATCH	Linked virtual latch module number for master information.	-
PISRC_SVIRTCOUNT	Linked virtual counter module number for slave information.	-
PISRC_SVIRTLATCH	Linked virtual latch module number for slave information.	-
POSDRCT	Rotational direction.	Encoder Data
POSENCQC	User unit factor to be CANopen compatible.	-
POSENCREV	User unit factor to be CANopen compatible.	-
POSERR	Maximum tolerated position error.	Position Control
POSERRTIME	Time frame for triggering position error state.	Position Control
POSFACT_N	Denominator user factor.	Encoder Data
POSFACT_Z	Numerator user factor.	Encoder Data
POSLIMIT	Positive software end limit.	Predefined I/Os
PROFTIME	Scan time for profile generator.	Position Control
RAMPMIN	Maximum acceleration.	Basic Settings
RAMPTYPE	Ramp type.	Basic Settings
REGWMAX	Size of the control window (activation).	Position Control
REGWMIN	Size of the control window (deactivation).	Position Control
SWNEGLIMACT	Negative software end limit active.	Predefined I/Os
SWPOSLIMACT	Positive software end limit active.	Predefined I/Os
SYNCACCURACY	Size of the precision window for position synchronization.	Synchronization

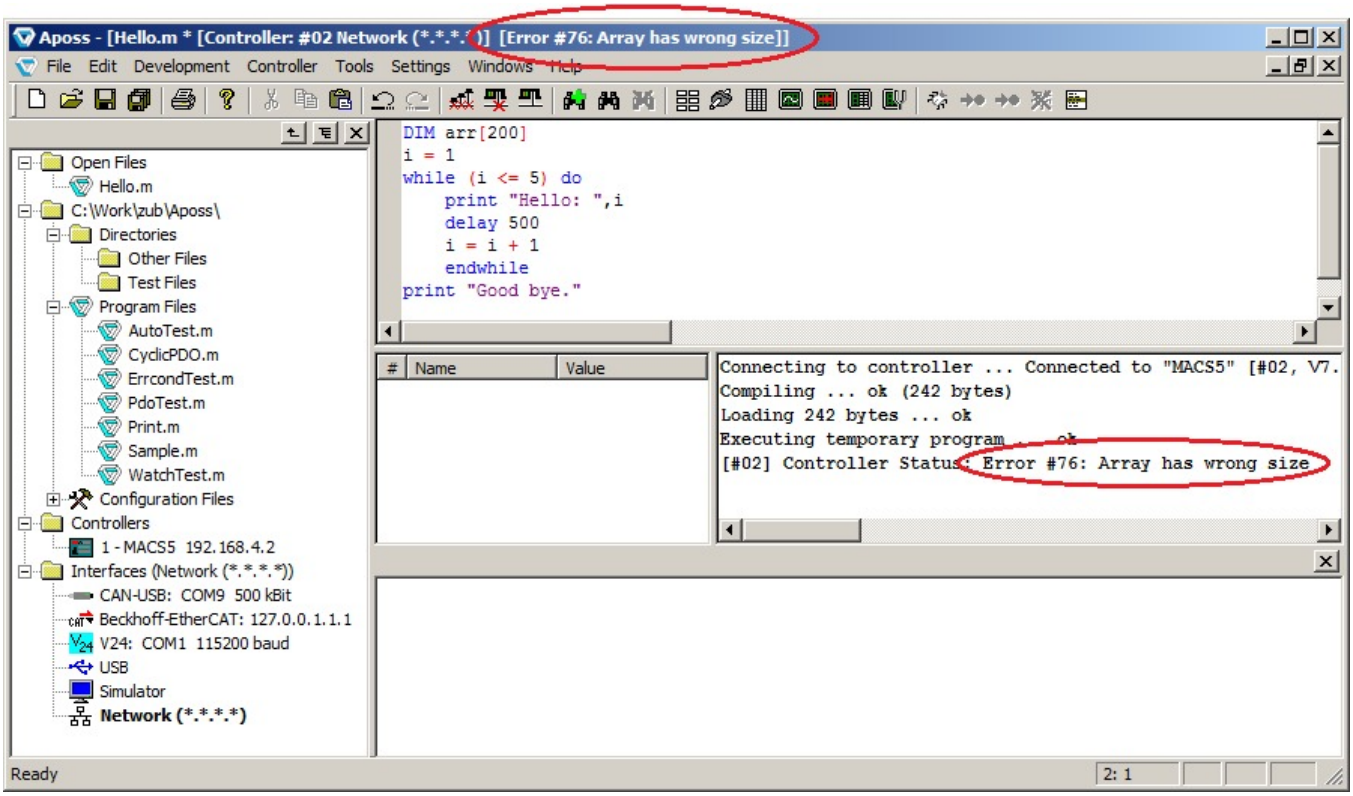
Parameter Name	Parameter Description	Topic
SYNCFACTM	Synchronization factor master (M:S).	Synchronization
SYNCFACTS	Synchronization factor slave (M:S).	Synchronization
SYNCFAULT	Marker number for fault.	Synchronization
SYNCMARKM	Marker number for master.	Sync Marker
SYNCMARKS	Marker number for slave.	Sync Marker
SYNCMFPAR	Marker filter configuration.	Synchronization
SYNCMFTIME	Filter Time for Marker Filter.	Synchronization
SYNCMMAXCORR	Maximum Marker Correction.	Synchronization
SYNCMPULSM	Master Marker Distance.	Sync Marker
SYNCMPULSS	Slave Marker Distance.	Sync Marker
SYNCMSTART	Start behavior for marker synchronization.	Synchronization
SYNCMWINM	Master Marker Tolerance Window.	Sync Marker
SYNCMWINS	Slave Marker Tolerance Window.	Sync Marker
SYNCOFFTIME	Offset Filter Time.	Synchronization
SYNCPOSOFFS	Position offset for positioning synchronization.	Synchronization
SYNCREADY	Marker number for ready.	Synchronization
SYNCSFTIME	Filter time [ms] for slave marker correction.	Synchronization
SYNCTYPE	Type of synchronization.	Synchronization
SYNCVELREL	Relative Slave Velocity Limit.	Synchronization
SYNCVFLIMIT	Sync error window [qc] for automatic deactivation of SYNCVFTIME.	Synchronization
SYNCVFTIME	Velocity filter.	Synchronization
TESTTIM	Time in target window.	Predefined I/Os
TESTVAL	Target window limit value.	Predefined I/Os
TESTWIN	Size of target window.	Predefined I/Os
TIMER	Sampling time for PID control and for sending output value to the drive.	Position Control
VELMAX	Maximum velocity.	Basic Settings
VELMAXQC	Maximum velocity (read only).	-
VELRES	Velocity resolution.	Basic Settings

Syntax for Axis Parameter Access

```
// Write to Axis Parameter
AXE_PARAM(0, POSFACT_Z) = 1024;
AXE_PARAM(0, POSFACT_N) = 1000;


// Read from Axis Parameter
temp_z = AXE_PARAM(0, POSFACT_Z);
temp_n = AXE_PARAM(0, POSFACT_N);
```

11 Error Reference

The error number and the meaning is shown in the ApossIDE window title and in the communication window:



The last 50 errors since power up are stored internally. This information is not stored in flash. It can be read out by the ApossIDE (see Error History) or by SDO access. For every error, not only time and axis are stored, but also additional error information. This information is error specific.

11.1 Error Messages

All of the #define keywords listed below, can be found in the SysDef.mh include file and may be used by developers in application programs if this file is included.

Error #	ApossIDE error message	#define
2	There are no more CAN objects available (CANINI).	F_CANMEM
3	Axis not in system.	F_NOAXES
5	Error not cleared.	F_ERR
6	Failed to move to HOME position.	F_NOMNU
7	Home velocity 0.	F_MNU
8	Position error.	F_POSERR
9	Index pulse (encoder) not found.	F_NIO
10	Unknown command.	F_UNBEK

11	Software end limit activated.	F_GRENZE
12	Illegal parameter number.	F_PARNUM
14	Too many nested loops.	F_NOCHZ
15	INLONG command got an illegal string.	F_NUMFORMAT
16	Parameters in memory are corrupted.	F_CRCPAR
17	Programs in memory are corrupted.	F_CRCPRG
18	Reset by CPU.	F_WDT
19	User abort.	F_ABBRUCH
20	FC communication error.	F_VLTCOM
21	Number of SDO channels exceeded.	F_SDOCHN
22	Featuren protection error.	F_FEATUREPROT
23	Communication with ARM lost.	F_ARMCOM
25	Limit switch activated.	F_ENDSCHALT
26	SDO access error in SYSVAR or LINK command.	F_ILLGLSDO
27	Trial to execute a cleared or empty program.	F_NOPROG
39	Wrong or no FPGA firmware loaded.	F_FPGA
40	Amplifier error.	F_AMP
42	EtherCAT Master Error.	F_ECAT_MASTER
49	Too many interrupt functions.	F_NOINTLEFT
50	No amplifier power.	F_NO_24VOLT (deprecated)
50	Minimum voltage of the amplifier has been undershot.	F_UMIN
51	Too many nested function/ interrups calls.	F_STACK
52	Too many return() commands.	F_RETURN
56	A floating point function was called with an invalid argument.	F_MATHERR
57	State machine error.	F_STATEMACHINE
60	Interrupt happened, but interrupt address is no longer valid.	F_INTPTR
62	Error in verifying.	F_MEEP
63	Path control only: Path error.	F_PATHERR
64	Path control only: Time out in V24 path control.	F_TIMEOUT
65	Path control only.	F_PATHINT

70	Too many DIM arrays defined.	F_DIM
71	Invalid array index.	F_ARRBDS
73	Array number does not exist.	F_LTARRAY
74	Array is empty.	F_NOARRAY
75	No more memory space for the new array defined by DIM.	F_NOSPACE
76	Array size does not correspond to the size of the existing array.	F_ARRSIZERR
77	Maximum temperature of the amplifier has been exceeded.	F_TEMP
78	Maximum voltage of the amplifier has been exceeded.	F_UMAX
79	Timeout while waiting for index.	F_TNDX
83	Internal command error (illegal parameter or format or range)	F_CMDERR
84	Too many TIME or PERIOD interrupts.	F_TIM
87	Not enough memory for variables.	F_NOVARMEM
88	CAN guarding error.	F_CANGUARD
89	CAN send or receive error.	F_CANIO
90	Memory locked.	F_MEMLOCK
91	Illegal curve array in SETCURVE.	F_CURARR
92	Encoder error.	F_ENCERR
93	Stack overflow: Too many local variables or nested function calls.	F_DYNSTACK
94	Out of dynamic memory.	F_DYNMEM
95	Too many test indices in data logging command.	F_OPALINDX
96	Code is too old for the current firmware.	F_ILLGLCODE
97	Internal overcurrent detection of power stage.	F_IMAX
98	Wrong direction after limit switch tripped and error reset.	F_LIMIT_VIOLATION
99	I^2T Limit exceeded.	F_I2TLIMIT
100	Communication with amplifier interrupted.	F_AMPCOM
101	Illegal access to amplifier parameter.	F_AMPPARAM
102	Command is not supported	F_NOTSUPPORTED
103	MemoryDump error.	F_MEMDUMP

104	Divide by 0.	F_DIVIDEBY0
105	EtherCAT Slave Error.	F_ECAT_SLAVE
106	Profile Generator Error.	F_PROFGEN
107	DIM array is too short.	F_DIMTOOSHORT
108	Internal firmware error.	F_INTERNALERROR
109	Option boot failure.	F_OPTIONBOOT
110	Wrong command usage.	F_COMMAND_USAGE
111	Illegal write access to process data.	F_PROCESSDATAUSAGE
112	The size of string argument too long.	F_ARGTOOLONG
113	Authentication failed.	F_AUTHFAILED
114	Amplifier not ready.	F_AMP_NOTREADY
115	Logic Supply Voltage too low.	F_LOGIC_UMIN
116	ZFC file update failed.	F_ZFCUPDATE
117	Hardware Failure.	F_HWFAIL
118	Amplifier's initialization order violated.	F_AMP_INIT_ORDER
119	Hall sensor problem.	F_HALL
120	Hall angle detection error.	F_HALL_ANGLE
121	STO error.	F_STO
122	Logic Supply Voltage too high.	F_LOGIC_UMAX
123	PROFINET error.	F_PROFINET
xx	Internal error ##.	

12 Technical Reference

12.1 Special Features, product specific

12.1.1 External Storage Device (microSD Card/ USB Memory Stick)

Some types of zub Controller Types provide a microSD card or USB slot. The Controller Dependent Feature List shows which functionalities are available for the respective controller.

This section is organised into three different areas which are listed below.

- Writing Process (microSD Card/ USB Memory Stick)
- Reading Process (microSD Card/ USB Memory Stick)
- Other Functionalities (microSD Card/ USB Memory Stick)

> MACS Controller expects an SD formatted with a FAT16 or FAT32 file system. All files relevant for the update must be located in the root directory. File names must not be longer than an eight character file name and a three character file extension. Use the unmount function before removing the SD Card physically from your PC or Card reader.

> ⚠ An external storage device must never be removed from the controller during the process.

12.1.1.1 Writing Process (microSD Card/ USB Memory Stick)

The MACS controller has several functions for writing data to an External Storage Device (microSD Card/ USB Memory Stick).

- Logging Service
- Memory Dump Process

12.1.1.1.1 Logging Service

The Memory Dump Process commands can be used to log user messages to an external storage device.

The principle of this function is, that the sprintf command can be used to print to a global application array. For this purpose, a user-selected array can be marked as log buffer using the printcontrol command.
This will configure the array to work as a ring buffer. The user will then print multiple log messages to the array using the sprintf command. The data of the array is only in volatile memory, so it would be lost if the controller is switched off.
So the array from time to time has to be "dumped" to the external storage device.
This can be done using the Memory Dump Process commands. When an existing file is used, the content will be appended to the existing file.
On controllers, where these commands are blocking, the dump action has to be started on occasions, where the controller is idling. See comments on Memory Dump Process page.

Example application

PrintToUSBStick.mc

```
// Create array
wchar dim printArray[100000];  //Space for 100000 chars

long main(void)
{
    //Mark printArray as ring buffer
    printcontrol(RINGBUF, printArray);

    //Now print to the printArray
    sprintf(printArray, "A log line with a number: %ld", 500);
    sprintf(printArray, "A log line with a number and a line feed: %ld\r\n", 500);
    sprintf(printArray, "A simple text log line with line feed.\r\n");


    // Copy content of the buffer to the external storage device
    MemoryDump(0x2212, 2, "Log.txt");
    // Poll the execution state
    while (MemoryDumpState() == 1)
    {
        // Wait here while the task is processing
    }
    print("End: result = ", MemoryDumpStatus());
}
```

12.1.1.1.2 Memory Dump Process

The memory dump commands can be used to move data from the volatile memory in the controller to non-volatile memory plugged into the controller.
This can be a microSD card or an USB memory stick (controller dependent).

The memory dump provides different modes to write oscilloscope, print, array or log data to the external memory. The data will be stored in a file.

The process is typically started with the MemoryDump command, and afterwards polled for completion with the MemoryDumpStatus command.
Depending on the controller type, the MemoryDump command will be blocking or non-blocking. If the command is non-blocking, this means that the user application will be blocked for a potentially long time. This can be up to 1-2 seconds, depending on the memory type and file system format. Check the Controller Dependent Feature List to see if a specific controller type supports the non-blocking write feature.

12.1.1.2 Reading Process (microSD Card/ USB Memory Stick)

If an external memory device (USB memory stick or microSD card) is used, certain commands can be executed with a Command File when the controller is started. It uses the same syntax and command names as the ApossIDE Batch Mode . The command list is slightly less comprehensive. A Report File is available as return and debug value. If this already exists, the batch file is no longer executed in the next reboot cycles.

The following functions are available when reading external files.

- User SDO Download
- Array SDO Download
- System Update Without ApossIDE

12.1.1.2.1 Command File

The batch file containing the various commands must have the name **BATCH.bat.**

Currently the following subset of commands is supported.

Command List

Command	Description
`A [program number]`	Set the autostart flag at given program number in volatile memory
`A 255`	Clear the autostart flag in volatile memory
`B`	Stops the running program (dummy command)
`E`	Erase all application programs
`F [source file]`	Update DSP, FPGA, AMP firmware or Bootloader
`F [1.src file] [2 src file]`	Update DSP firmware (1st source file -> MACS5v1, 2nd source file -> MACS5v2)
`D [source file]`	Download application program or Array and User-SDO data to volatile memory
`R [prg name] [prg number]`	Write program name and number to volatile memory
`M`	Copy data from volatile memory (D, R and A commands) to persistent memory
`X [program number]`	Start execution of a program (dummy command)
`r`	Reset the device
`~ Report Off`	Disable writing to the report file (by default enabled)
`~ Report On`	Enable writing to the report file (by default enabled)

Parameter Restrictions

[source file]	File must be located in the root directory of the SD Card. File

	name length is limited (“8.3” - eight character file name and three character extension).
[program name]	Name must be an alphanumeric string of maximum eight characters length.
[program number]	Number in the range of 1 - 90.

Example - Program download and start

Command File - BATCH.bat

```
E // Erase all application programs currently saved on the device
D PGM_Test.m0 // Copy the application from file PGM_Test.m0 on the SD-Card to volatile memory
R TestProg 1 // Write the name Test.Prog and program number 1 for the downloaded application
A 1 // Set the autostart flag for application number 1
M // Copy the data of the previous D, R, A commands from volatile to persistent memory
X 1 // Start the execution of application number 1
```

Report File - REPORT.txt

```
#C 0x0000 0000093004 $E
#C 0x0000 0000093004 $D PGM_Test.m0
#C 0x0000 0000093004 $R TestProg 1
#C 0x0000 0000093004 $A 1
#C 0x0000 0000093004 $M
#C 0x0000 0000093004 $X 1
```

12.1.1.2.2 Report File

The Report file "REPORT.txt" lists the update commands that have been executed. A record consists of the error code, the device serial number, and the command string. By using the Command ***~ Report Off***, the generation of the report file can be switched off. In any case, the datas are (not persistently) available in the SDO 0x2220 - SD-Card Update .

An operation that is listed in the command file will not be executed if there already is an equivalent record (identical device id and command string) in the report file. This prevents the updater from repeatedly executing a command after each power up. To force a second execution of a command, the corresponding record, or the report file as a whole, must be deleted. Since the records are device specific, one can use a single SD Card to update multiple devices without editing the report file.

Example Syntax

#C 0x0000 0000093004 $F FPGA_xxx.m0

Error Codes

The first number of the record indicates whether the command has been executed successfully or the reason why it has failed. The table below lists the error codes:

0x0000	command completed without error
0x0001	unknown/unsupported command

0x0002	invalid command parameter
0x0003	failed to access source file
0x0004	wrong source file structure
0x0005	wrong source file type (ex: FPGA source file for a program download)
0x0006	failed to access target file
0x0007	Accessing invalid element (ex: array index, user SDO No. is to large)
0x__F_	internal error

12.1.1.2.3 System Update Without ApossIDE

The microSD Card/ USB Stick Update offers the possibility to update DSP, FPGA, AMP, and BOOT firmware, as well as ApossIDE application programs, without the need to have a connection to an ApossIDE environment running on a PC.

The microSD Card/ USB Stick Update is a software routine that is called during device start up. It scans the microSD Card/ USB Stick for a Command File (BATCH.bat) located in the root directory. The command file llists a set of update commands that shall be executed after device reset. If there is no SD Card in the slot or no command file is present, the update is skipped and the device enters regular operation immediately. If a command file is present, then the updater goes through the list of commands. Before executing any of the commands, it will check whether the command has already been executed during a previous device reset. In this case, there would be an equivalent record (identical device id and command string) in the Report File (REPORT.txt). This file is also located in the root directory of the microSD Card/ USB Stick . If there is no equivalent record or no log file at all, the command is executed and a new log file entry is written. Commands that have been executed during previous runs are skipped. If there are no commands left to execute, then the updater terminates and the device enters regular operation.

> The update time depends on the control system and can take up to 5 minutes. The LEDs of the control unit show the status of the process. More information can be found in the Hardware Reference Manual.

Example - DSP and FPGA firmware Update MACS5

Command File - BATCH.bat

```
F DSPv1_xx.m0 DSPv2_xx.m0
F FPGA_xxx.m0
```

Report File - REPORT.txt

```
#C 0x0000 0000093004 $E
#C 0x0000 0000093004 $D PGM_Test.m0
#C 0x0000 0000093004 $R TestProg 1
#C 0x0000 0000093004 $A 1
#C 0x0000 0000093004 $M
#C 0x0000 0000093004 $X 1
```

12.1.1.2.4 User SDO Download

During the start process, a Batch File and source file can be used to describe the user parameters that are available on the MACS controllers.

Example

Command File - BATCH.bat

```
~ Report Off // Disable report writing, Batch file is repeated every restart
D usersdo.txt // Copy User-SDO data from file usersdo.txt on the SD-Card to volatile memory
M // Copy array and User-SDO data from volatile to persistent memory
```

Source File - usersdo.txt

```
sdobegin 0x2201 15 22 // Write single User-SDO SubIndex=15, Value=22
sdobegin 0x2201 0 0 // Write User-SDO's, 0 -> elementwise "sdoval..." follows
sdoval 1 1001  // SubIndex=1 Value=1001
sdoval 2 2002
sdoval 3 3003
sdoval 4 4004
sdoval 5 5005
sdoval 10 -10000
sdoval 11 11000
sdoval 12 -12000
sdoval 13 22222
sdoval 14 0x9A71 // Hex values allowed
sdoval 15 3
sdoval 100 0 // Index=100, highest subindex allowed
sdoend
```

12.1.1.2.5 Array SDO Download

During the start process, a Batch File and source file can be used to write on DIM Arrays on the MACS controllers. There are two different methods for writing the array.

With the "complete array" method the array data must be present in ascending order, 10 values per line. The number of values must be indicated on the first line. If the array does not yet exist it will be created. Be sure that newly created arrays do not colide with existing and that they are
created in ascending order. You can use this to update existing arrays if the array number and the requested size is equal.

The "element by element" method uses an index-value-pair notation. It does not expect the complete data nor any specific order. This type of access is limited to the first 250 elements. There is an offset of 4 between subindex and array element (element 1 subindex 5, element 2
subindex 6, etc). The write access will fail if it does not match with the array structure present on the device (array does not exist, index to high).

Example

Command File - BATCH.bat

```
D arrays.txt // Copy array data from file arrays.txt on the SD-Card to volatile memory
D array.txt // Copy array data from file array.txt on the SD-Card to volatile memory
M #1 // Copy array and User-SDO data from volatile to persistent memory
```

Report File - REPORT.txt

```
#C 0x0000 0000093004 $D arrays.txt
#C 0x0000 0000093004 $D array.txt
#C 0x0000 0000093004 $M
```

Source File - arrays.txt

```
sdobegin 0x2100 0 44 // Aposs Array 0, write/create 44 elements
1 2 3 4 5 6 7 8 9 10 // 10 values per row
0 0 0 0 0 0 0 0 0 20
0 0 0 0 0 0 0 0 0 30
0 0 0 0 0 0 0 0 0 40
0 0 0 9
sdoend

sdobegin 0x2100 1 10 // Aposs Array 1, write/create 10 elements
1 2 3 4 5 6 7 8 9 10 // 10 values per row
sdoend

sdobegin 0x2100 2 12 // Aposs Array 2, write/create 12 elements
1 2 3 4 5 6 7 8 9 10 // 10 values per row
12 13
sdoend
```

Source File - array.txt

```
sdobegin 0x2102 0 0 // Aposs Array 2, 0 -> elementwise “sdoval...” follows
sdoval 5 11 // write Element 1 (+4 Offset)
sdoval 6 22 // write Element 2 (+4 Offset)
sdoval 10 666 // write Element 6 (+4 Offset)
sdoval 254 434 // write Element 250 (+4 Offset), highest element allowed
sdoend
```

12.1.1.3 Other Functionalities (microSD Card/ USB Memory Stick)

This section contains special features that can be used in combination with an External Storage Device (microSD Card/ USB Memory Stick).

- Providing website access to the controller via Webserver
- Access with FTP

12.1.1.3.1 Access with FTP

To access the FTP server of the controller, open the following URL in Windows Explorer: **ftp://<ip-address>/**, e.g. **ftp://172.16.1.33/**

There is also a command in the ApossIDE Menu to directly open the Windows Explorer with this URL, see Controller Menu.

Passive FTP is not (yet) supported. Currently, only the Windows FTP client (GUI and command line) and FileZilla have been tested.

12.1.1.3.2 Webserver

The webserver can access files from either the internal flash or the SD card. To access the files in the internal flash, open the URL **http://<controller-ip>/** where **<controller-ip>** is the IP address of the controller. To access the

files on the SD card, open the URL **http://<controller-ip>/sdcard/**. If no file name is specified in the URL, then the file 'index.htm' will be opened.

zub provides an example website that users are free to customize for their own use. The source code for the example is located in the ApossIDE installation directory in the subfolder /Development/Webserver/website. Note that Java script and HTML skills are necessary to adapt the website. The Java script provides functions to read and write SDOs from the controller.

How to load the internal flash with a website

To load a website into internal flash, the html files must first be converted into an .m0 file. This .m0 file must then be downloaded to the controller using the ApossIDE firmware download function.

Follow these steps to build the .m0 file for the example zub website:

1. Create a working copy of the entire "Developer\Webserver" directory.

   ⚠ Do not directly use the files in the ApossIDE install directory (even if using a user account with administrator permissions) since this may result in problems caused by the use of Windows "VirtualStore" in Windows operating systems newer than Windows XP.

2. Execute the "buildInternalFlashFileSystem.bat" script that is in the working copy of the Webserver directory. This will build the final "webserverFileSystem.m0" file from the contents of the "website" subdirectory.
3. To build a compressed version, execute the "buildInternalFlashFileSystemWithCompression.bat" script. This script will compress the website into the subdirectory "website_compressed" and then build the "webserverFileSystem.m0" file. Note that to execute the script, Java must be installed and present in the Windows system PATH environment variable.

After the .m0 file has been created, follow these steps to download it to the controller:

1. Start the ApossIDE and close all document windows (i.e. edit windows, oscilloscope windows, etc.).
2. Execute the **Download** → **Firmware** command to open the Firmware Download Wizard dialog.
3. Follow the directions in the dialog to download the file.

   ⚠ Depending on the controller type, the size of the internal flash file system is very limited. The controller will display an error message if the file is too big.

4. The Firmware Download Wizard will automatically restart the controller after the file has been downloaded. This restart is necessary before the new website can be displayed.

This feature can be controller dependent and is listed in the Controller Dependent Feature List

How to load the SD card with a website

The website files on the SD card must be located in the /html/ subfolder. To access the files on the SD card, open the URL **http://<controller-ip>/sdcard/**.

12.1.2 Powerdown Save Functions

This function can be used to save specific data when the power is lost.

This function is deactivated by default, it must be activated in the application program after every power-up of the controller.

The function will be started as soon as the voltage drops below a trigger voltage (SDO 0x2219 / 129). As soon as the function is executed, the controller will not react to serial communication anymore, nor will it execute the application program. So the trigger voltage should be set up low enough to not trigger during normal operation.

The function can be initialized by writing a 1 to SDO 0x2219 / 128. This will erase previous power-down data and prepare the flash for a fast saving at power-down. It will also setup indices 1 - M_AX*2 to save the positions of the virtual input counters 1 - M_AX*2 by default. This means, the controller will save these positions by default into the first indices (depending on how many axis there are).

When enabled, it is possible to save more data, which can be configured by the user by SDO 0x2219 / 128.

Each subindex maps a long value to be saved to flash at power-down. Also the ones that are defined by default can be overwritten if desired. The mapping is done by "Sysvar" values, so that practically every data can be saved by the user.

When the setup is finished, the power-down save function must be enabled by writing a 2 to SDO 0x2219 / 128.

Ask zub for an example program.

See SDO 0x2219 Description.

> This feature can be controller dependent and is listed in the Controller Dependent Feature List

> The power-down save function *is not enabled by default* (see below).

Example of configuring entries (SDO 0x2219 / 1-127)

```
Sysvar[0x01221900 + 7] = 0x01220100 + 1;
```

This will map user parameter 1 to index 7 of the power-down save.

After a power-down saving has taken place, the data can be read out after the next startup. First, the "data valid" flag must be read, which also reads the whole data set from the flash. This is done by reading SDO 0x2219 / 254. When a 1 is read, the data set is valid, otherwise there is no valid power-down dataset.

Then the real data can be read out by SDOs 0x2219 /1-127.

Example of reading out data

```
print("position 1: ", Sysvar[0x01221900 + 1]);
```

This will read the first saved value, which by default is the first position.

Trigger voltage

The trigger voltage can be set in mV by SDO 0x2219 / 129.

Setting the voltage closer to the normal supply level gives more life time on power-down, but increases the risk to trigger the function at normal operation. Setting the trigger voltage too low will eventually leave too little time to save the data, which will result in failure of the function. It is strongly recommended to leave the value at default.

12.1.3 Webserver

The webserver can access files from either the internal flash or the SD card. To access the files in the internal flash, open the URL **http://<controller-ip>/** where **<controller-ip>** is the IP address of the controller. To access the files on the SD card, open the URL **http://<controller-ip>/sdcard/**. If no file name is specified in the URL, then the file 'index.htm' will be opened.

zub provides an example website that users are free to customize for their own use. The source code for the example is located in the ApossIDE installation directory in the subfolder /Development/Webserver/website. Note that Java script and HTML skills are necessary to adapt the website. The Java script provides functions to read and

write SDOs from the controller.

How to load the internal flash with a website

To load a website into internal flash, the html files must first be converted into an .m0 file. This .m0 file must then be downloaded to the controller using the ApossIDE firmware download function.

Follow these steps to build the .m0 file for the example zub website:

1. Create a working copy of the entire "Developer\Webserver" directory.

   ⚠ Do not directly use the files in the ApossIDE install directory (even if using a user account with administrator permissions) since this may result in problems caused by the use of Windows "VirtualStore" in Windows operating systems newer than Windows XP.

2. Execute the "buildInternalFlashFileSystem.bat" script that is in the working copy of the Webserver directory. This will build the final "webserverFileSystem.m0" file from the contents of the "website" subdirectory.
3. To build a compressed version, execute the "buildInternalFlashFileSystemWithCompression.bat" script. This script will compress the website into the subdirectory "website_compressed" and then build the "webserverFileSystem.m0" file. Note that to execute the script, Java must be installed and present in the Windows system PATH environment variable.

After the .m0 file has been created, follow these steps to download it to the controller:

1. Start the ApossIDE and close all document windows (i.e. edit windows, oscilloscope windows, etc.).
2. Execute the **Download** → **Firmware** command to open the Firmware Download Wizard dialog.
3. Follow the directions in the dialog to download the file.

   ⚠ Depending on the controller type, the size of the internal flash file system is very limited. The controller will display an error message if the file is too big.

4. The Firmware Download Wizard will automatically restart the controller after the file has been downloaded. This restart is necessary before the new website can be displayed.

This feature can be controller dependent and is listed in the Controller Dependent Feature List

How to load the SD card with a website

The website files on the SD card must be located in the /html/ subfolder. To access the files on the SD card, open the URL **http://<controller-ip>/sdcard/**.

12.1.4 Real Time Clock

The real time clock can be set with the ApossIDE using **Controller** → **Diagnostics**.

To use the real time in an application, the SDOs SYS_RTC_DATE and SYS_RTC_TIME can be used.

This feature can be controller dependent and is listed in the Controller Dependent Feature List

12.1.5 Emergency Reset

There are cases where an application running on the controller can block connections, making it impossible for the ApossIDE to regain control over the device. The Emergency Reset procedure provides a way to recover the controller.

⚠ This procedure will completely reset the controller. All saved programs and arrays will be deleted and all

parameters will be reset to default values.

Using the CAN DIP switch, proceed as follows:

1. Turn the controller OFF.
2. Enable the highest available CAN Address DIP switch (typically switch 7 or 8).
3. Watch the ST and the ER LEDs.
   After some seconds, the ST LED (yellow) will be permanently ON and the ER LED (red) will blink.
   During this state, which lasts for only 2-3 seconds, disable DIP switch 7/8 again.
   The blinking will stop immediately and the controller will be reset.

This feature can be controller dependent and is listed in the Controller Dependent Feature List

12.1.6 FoE Firmware Update

The following steps must be executed

- Put the controller in BOOT mode
- Download the .m0 firmware file with TwinCAT (or other master software)
  The password and the filename don't matter
- Put the controller to INIT mode
  This will reset the controller and the F/W will be updated

This feature can be controller dependent and is listed in the Controller Dependent Feature List

12.1.7 Ethernet protocol

Port Number

The port number can be read out specifically for the controller in the Controller Types chapter.

Protocol Specification

The protocol specification can be requested from zub.

SDO Channel

Please note that there are three SDO channels. Each communication channel must use a different SDO channel. Otherwise, the communication will fail. For example, if a computer is connected to the controller using SDO channel 2 and a PLC wants to talk to the Ethernet interface, then it must use SDO channel 1 or 3.

Force DHCP

If the IP address of the controller is not known, a DHCP connection can be forced with the help of the DIP switch.

Using the CAN DIP switch, proceed as follows:

1. Turn the controller OFF.
2. Enable all DIP switch 1-8 to ON.
3. Turn the controller ON.
4. After a controller-specific startup time, the connection via the DHCP-assigned IP address will be possible.

All this features can be controller dependent and is listed in the Controller Dependent Feature List

12.1.8 Watchdog

Some controller types provide a watchdog feature. The watchdog is a feature that can be used as a "method of last resort" to help applications recover from unexpected catastrophic failures such as system crashes and hangs. When activated, the watchdog will continuously monitor program execution. If execution stops for any reason, then the watchdog will trigger a reboot of the controller and a restart of the user application program.

> ⚠ In order for the watchdog to function properly and to restart the user application program, the controller must have the Autostart feature correctly configured. Otherwise, the controller will reboot but the application program will *not* be restarted.

> ⚠ The watchdog cannot handle situations involving persistent problems. For example, a persistent external event that the application cannot handle and cannot clear, or an implementation error resulting in a endless loop, will cause the watchdog to endlessly reboot the controller. Each time the controller is rebooted and the application autostarted, the application will reactivate the watchdog which will then trigger another reboot. This generally indicates a design flaw in the application and it would need to be addressed by the application developer.

A watchdog is activated with the WdActivate command, specifying a user-defined trigger time. After being activated, the WdReset command *must* be used to reset the watchdog at least once within *every* cycle of the trigger time. If the trigger time expires without the watchdog being reset, then the watchdog will trigger a reboot of the controller.

When the controller is rebooted by the watchdog, Error 18 (F_WDT) will be active after startup. Applications can use this to determine whether or not they have been started as a result of a watchdog reboot. If so, then they should take whatever action is necesssary to deal with the problem. Typically, this would include logging the event (e.g. to a External Storage Device (microSD Card/ USB Memory Stick)) or somehow reporting the event to an external device.

Example:

```
#include <SysDef.mh>

long main(void)
{
    long i;

    // Test if this is a restart after a watchdog timout.

    if (ErrorNo() == F_WDT) {
       print("Restart after a watchdog time.");
       ErrorClear();
    }

    // Activate watchdog with 300ms period.

    WdActivate(300);

    // The following loop resets the watchdog every 200ms.
    // This prevents a reboot of the controller.

    for (i = 0; i < 100; i++) {
        Delay(200);
        WdReset();
    }

    // The following loop does NOT reset the watchdog
    // within the 300ms trigger time. This will result
    // in a reboot of the controller.
```

```
        for (i = 0; i < 100; i++) {
           Delay(400);
           WdReset();
        }

        return(0);
    }
```

This example illustrates two important points:

1. Even a correctly implemented application (i.e. one that does not contain an endless loop) may trigger the watchdog if the developer chooses a watchdog timout that is too short for the longest execution path within the application.
2. If set as the autostart program, this program will result in endless reboots of the controller. The program itself has a flaw and no amount of rebooting will ever avoid the flaw.

This feature can be controller dependent and is listed in the Controller Dependent Feature List

12.1.9 FTDI Driver

The older generation of MACS controllers use a driver from FTDI for the USB connection. This includes the following controller groups:

- MACS4
- MACS5
- MiniMACS
- MasterMACS
- USBMiniCounter

Default Installation

By default, the installation of the ApossIDE uses the re certified driver version FTD2XX_2.08.14. This driver version is compatible **with all MACS firmware** and can be used with Windows 7/10/11. This version is not compatible with certain Windwos features (see below). However, it is recommended by default.

Windows Defender (Credential Guard/ Memory Integrity)

From **Windows 10 onwards**, Microsoft offers enhanced protection features which can be activated. These features, such as Windows Defender Credential Guard or Core Isolation with memory integrity, require a driver to have higher standards. Specifically, all drivers must support **HVCI** (Hypervisor Protected Code Integrity).

If one of these Windows Defender features is activated, **a newer driver version** (FTD2XX_2.12.36.4) must be loaded. This has also been re-certified by Zub machine control AG and is HVCI compatible. After installing this driver, **a minimum firmware version** of the MACS controller of >=7.5.22 or >=9.4.00 is required.

Driver Files

All drivers are delivered with the installation of the ApossIDE and are available in the directory of the installation path in the Drivers folder (C:*installationpath*\Drivers).

Delete specific FTDI Product ID drivers

1. Unplug all Zub controls

2. Delete driver
3. Run the programme "CDMUninstallerGUI.exe" as administrator (C:*installationpath*\Drivers)
4. Add the devices that use the FTDI driver from Zub
    1. MACS4-DC3: VID_0403 PID_6001
    2. MACS5/ MiniMACS/ MasterMACS/ USBMiniCounter: VID_0403 PID_B0A8
    3. USB CAN: VID_0403 PID_ECD9
5. Confirm the button "RemoveDevices".

12.2 CAM Array Structure

If a controller uses CAM Profiles, then the definition of each profile is stored in a DIM array. Normally, the CAM Editor can be used to handle all aspects of the creation of these arrays. However, in certain types of user applications, a running application program may need to modify CAM profiles on-the-fly. This section provides the internal array structure information that will be required for developers to implement such application programs. See Curve Array Diagram for a graphical illustration of the array structure of CAM profiles.

> The "sysdef.mi" include file contains #define statements for several macros that can make access to the CAM array easier for application programs. See Modifying the CAM Curve Array.

Different versions of CAM Profile array structures are supported. The specific version that must be used will depend on the firmware version and type of controller:

Version	Firmware	Description
100		Obsolete. Use version 101 instead.
101	< 6.4.44	Curve interpolation points are required.
102	>= 6.4.44	"On-the-fly" curve interpolation is supported.
103	>= 6.4.44	Required if the CAM Profile is a component of a multi-dimensional path curve.

One of the major differences between version 101 and 102 profiles is that controllers supporting version 102 profiles are capable of doing “on-the-fly” curve interpolation. As a result, the profile contains no curve interpolation information. Older controllers that support only version 101 profiles require equally spaced curve interpolation points to be pre-calculated and included in the profile. This interpolation is done by the CAM Editor and is based on the number of interpolation points specified in the curve information section of the profile.

The CAM Profile array is divided into the following sections. The sections may or may not exist in the array, depending on the version of the array (i.e. 100, 101, etc.) and the type of curve in the array. If the sections exist, then they must be present in the array in the order listed below (i.e. Header first, followed by Curve Information, etc.).

Section	Description
Header	This section contains general information. It must always exist.
Curve Information	This section contains basic information about the curve. It must always exist.
Start/Stop Point	Points used for synchronization engaging and

Section	Description
	disengaging. This section will exist only if start/stop paths are used.
Fixed Point	Fixed points defining the curve. This section must always exist.
Interpolation Point	Interpolation points for old controllers. This exists only for version 100 and 101 arrays.
Start/stop Point Index	Interpolation point indices for start/stop paths. This exists only for version 100 and 101 arrays.
Start Stop Velocities	Velocities for start/stop paths. This exists only for version 100 and 101 arrays.
Startpath Points	Reserved working memory. This exists only for version 100 and 101 arrays.
Stoppath Points	Reserved working memory. This exists only for version 100 and 101 arrays.
Gradient Data	Start and end gradient specifications. This exists only for version 102 and later arrays. It may be omitted for curve types other than 3-8 (see below).
Path Information	This section contains basic information about the path to which this curve belongs. This exists only for version 103 and later arrays if the curve belongs to a path.

Header section

The header section must always be the first section in the array. There are two variations of this section: One for version 101 and earlier arrays and one for version 102 and later arrays.

Version 101 and Earlier

Index	Name	Description
1	Identification	The first value must be 999000001. This is a special number used to identify the array as a CAM Profile array.
2	Version	CAM Profile version (100 or 101).
3	CurveType	Type of curve: 0 - Start and end velocities averaged. 1 - End velocity set to start velocity (Hauser compatible).
4-7	CurveName[4]	Curve name (up to 16 characters).
8	IndexCIF	Index to Curve Information section.
9	IndexSTP	Index to Start/Stop Point section. If there are no start/stop pairs, then this value should be set to the same value as “IndexFIP”.

Index	Name	Description
10	IndexFIP	Index to Fixed Point section.
11	IndexINP	Index to Interpolation Point section.
12	IndexSTPInd	Index to StartStop Interpolation Indices.
13	IndexSTPVel	Index to StartStop Velocities.
14	IndexSTIP	Index to Startpath interpolation points.
15	IndexSTPIP	Index to Stoppath interpolation points.

Version 102 and Later

Index	Name	Description
1	Identification	The first value must be 999000001. This is a special number used to identify the array as a CAM Profile array.
2	Version	CAM Profile version (102, 103,...).
3	CurveType	The low-order byte defines the type of curve: 0 - Start and end velocities averaged. 1 - End velocity set to start velocity (Hauser compatible). 2 - Start and end velocities set to 0 (5th order). 3 - Start/end velocities set by gradient data. 4 - Start/end velocities set by gradient data (immediate). 5 - Start/end velocities set by gradient data. 6 - Start velocity and end acceleration set by gradient data. 7 - Start acceleration and end velocity set by gradient data. 8 - Start/end accelerations set by gradient data. 9 - The curve is cyclic. The high-order 3 bytes define various flags: 0x80000000 - All curve segments are to be tangents.
4-7	CurveName[4]	Curve name (up to 16 characters).
8	IndexCIF	Index to Curve Information section.
9	IndexSTP	Index to Start/Stop Point section. If there are no start/stop pairs, then this value should be set to the same value as “IndexFIP”.
10	IndexFIP	Index to Fixed Point section.
11	-	This is not used and should be set to a value 1 larger than the array size.
12	IndexPath	Index to Path Information. For curves that do not

Index	Name	Description
		belong to a path curve (and for version 102 arrays), this is not used and should be set to a value 1 larger than the array size.
13	-	This is not used and should be set to a value 1 larger than the array size.
14	-	This is not used and should be set to a value 1 larger than the array size.
15	-	This is not used and should be set to a value 1 larger than the array size.
16	IndexGrad	Index to gradient data. If the curve does not require start and end gradient values, then this should be set to a value 1 larger than the array size.

For firmware versions earlier than 7.1.00: Only curve types 0-4 are supported. Curve type flags (the high-order 3 bytes) are not supported and must be set to 0.

Curve Information section

This section of the array contains the basic information about the type of curve.

Index	Name	Description
1	MasterCycleLen	Length of master curve (in master units).
2	SlaveCycleLen	Maximum slave travel distance (in slave units).
3	FixPointNo	Number of fixed points (maximum 100).
4	PathSize	For version 103 and later, this is the number of values in the Path Information section (or 0 if there is no Path Information section). For version 102, this is not used and should be set to 0. For version 101 and earlier, this is the number of interpolation points (including the first and last, which correspond to the same location).
5	InterpolType	Interpolation type. This is reserved for the use of the controller and should be set to 0.
6	SlaveStopPosition	Slave stop point (in slave units). This is the position where the slave is stopped when synchronization is topped.
7	CorrectionStartPoint	Correction start point (in master units). This is only valid for marker synchronization.
8	CorrectionStopPoint	Correction stop point (in master units). This is only valid for marker synchronization.
9	MaximumCorrection	Maximum correction allowed in one cycle (in slave units). This is only valid for marker synchronization.

Index	Name	Description
10	MaxStartStopLen	Maximum length of start/stop path (number of interpolation points). For version 102 and later, this is not used and should be set to 0.
11	StartStopNo	Number of start/stop point pairs (*not* points!).
12	MMaxCycles	Maximum number of cycles per minute (application information).
13	MMarkerPos	Master Marker Position (in master units). This is only valid for marker correction. This value is omitted in version 100 profiles.
14	SMarkerPos	Slave Marker Position (in slave units). This is only valid for marker correction. This value is omitted in version 100 profiles.
15	GradDataSize	Number of values in the Gradient Data section (or 0 if there is no Gradient Data section). This value is omitted for version 101 and earlier profiles.

Start/Stop Point section

This section contains the start/stop points used for engaging and disengaging the slave during synchronization. The section will be empty (i.e. omitted) if the application does not use start/stop points. These points always come in pairs. For engagement, the pair consists of the master position where the ramp-up to synchronization should start and the master position where synchronization should be complete. For disengagement, the pair consists of the master position where the disengagement should start and the master position where the disengagement should be complete. These points are used by the SyncCamStart and SyncCamStop commands.

Index	Name	Description
1	STPoint.a	Start point (in master units) for forward movement. Stop point for backward movement.
2	STPoint.b	Stop point (in master units) for forward movement. Start point for backward movement.

For version 101 and earlier profiles: These points must lie on interpolation points. Either the position of the start/stop points or the resolution of the interpolation must be adjusted to accommodate this. For example, if we have a rotating master which makes one 360 degree revolution per cycle, we choose a cycle length of 3600 MU (i.e. 1 MU = 1/10 degree), and we choose the number of interpolation points as 1200, then the interpolation resolution will be 3 MU = 3/10 degree and the start/stop points must be placed on a multiple of 3/10 degree.

Fixed Point section

This section contains the fixed points which are the basis for the interpolation calculation. Each fixed point will always consist of the following triple and points must be ordered by increasing master coordinate.

Index	Name	Description
1	FixPoint.master	Master coordinate of fixed point (in master units). The first fixpoint will have a coordinate of 0; the last point will have a coordinate of "MasterCycleLen".
2	FixPoint.slave	Slave coordinate of fixed point (in slave units).
3	FixPoint.type	Type of fixed point. The low-order byte defines the type of the *next* curve segment: 1 - Curve point. 2 - Tangent point. 3 - Tangent point (obsolete). 4 - Tangent point. 5 - Trapezoid point. 6 - 3rd order point. The high-order 3 bytes define a user-specified operation code. The curve processing functions in the controller ignore this code.

For version 101 and earlier profiles: These points must lie on interpolation points (see above). Only point types 1 and 2 are supported. For tangent segments, the fixed points at *both* ends the segment must have a point type of 2.

For firmware versions earlier than 6.6.40: Only point types 1 and 2 are supported. For tangent segments, the fixed points at *both* ends the segment must have a point type of 2.

For firmware versions 6.6.40 and later: Point types 2 and 3 are not supported. Tangent segments should be specified using point type 4 (only for the fixed point at the start of the tangent segment).

For firmware versions earlier than 7.1.00: Operation codes in the high-order 3 bytes are not supported. These bytes must be set to 0.

Interpolation Point section

This section contains a list of values, one per interpolation point. Each value will be the slave coordinate of the curve at that interpolation point.

For version 102 and later profiles: This section is not used and should not exist in the array.

Index	Name	Description
1	IntPoint	Slave coordinate at interpolation point (in slave units).

Start/stop Point Index section

This section contains the indices within the Interpolation Point section (i.e. *not* within the profile array) to the start and end points of each start/stop path. There will be one pair for each start/stop path.

For version 102 and later profiles: This section is not used and should not exist in the array.

Index	Name	Description
1	STPoint.a-index	Index into the Interpolation Point section corresponding to the path start point.
2	STPoint.b-index	Index into the Interpolation Point section

Index	Name	Description
		corresponding to the path stop point.

Start Stop Velocities section

This section contains the velocity multiplied by 100000 (in slave units per master unit) at the start point and end point of each start/stop path. There will be one pair for each start/stop path.

For version 102 and later profiles: This section is not used and should not exist in the array.

Index	Name	Description
1	STPoint.a-veloc	Velocity (slave/master units * 100000) at start point.
2	STPoint.b-veloc	Velocity (slave/master units * 100000) at stop point.

Startpath Points section

This section provides reserved working memory for the controller when it is executing SYNCCSTART or SYNCCSTOP commands. This section must be "MaxStartStopLen" values long and all values should be set to 0.

For version 102 and later profiles: This section is not used and should not exist in the array.

Stoppath Points section

This section provides reserved working memory for the controller when it is executing SYNCCSTART or SYNCCSTOP commands. This section must be "MaxStartStopLen" values long and all values should be set to 0.

For version 102 and later profiles: This section is not used and should not exist in the array.

Gradient Data section

This section contains the initialization values for the start and end of the curve for curve types 3-8 (defined start and end gradients).

For version 101 and earlier profiles: This section is not used and should not exist in the array.

For version 102 and later profiles: This section is required for curve types 3-8. It is optional for other curve types.

Index	Name	Description
1	StartVelocityNum	Numerator of curve start velocity (spline and Poly5 curves).
2	StartVelocityDen	Denominator of curve start velocity (spline and Poly5 curves).
3	StartAccelerationNum	Numerator of curve start acceleration (Poly5 curves).
4	StartAccelerationDen	Denominator of curve start acceleration (Poly5 curves).
5	StartJerkNum	Numerator of curve start jerk. Not currently used. Set to 0.
6	StartJerkDen	Denominator of curve start jerk. Not currently used. Set to 1.
7	StopVelocityNum	Numerator of curve stop velocity (spline and Poly5 curves).

Index	Name	Description
8	StopVelocityDen	Denominator of curve stop velocity (spline and Poly5 curves).
9	StopAccelerationNum	Numerator of curve stop acceleration (Poly5 curves).
10	StopAccelerationDen	Denominator of curve stop acceleration (Poly5 curves).
11	StopJerkNum	Numerator of curve stop jerk. Not currently used. Set to 0.
12	StopJerkDen	Denominator of curve stop jerk. Not currently used. Set to 1.

Path Information section

This section contains basic information about the path curve to which this curve belongs.

For version 102 and earlier profiles: This section is not used and should not exist in the array.

For version 103 and later profiles: This section is required for curves that belong to a path curve. Otherwise, it should not exist in the array.

Index	Name	Description
1	PathId	ID number of the path to which this curve belongs.
2	AxisCnt	Total number of axes (dimensions) in the path curve.
3	Spatial	Number of spatial axes in the path curve.
4	Sequence	Axis sequence for this curve: 0 - X 1 - Y 2 - Z 0xFFFF - Arc length.
5	Flags	Miscellaneous flags: 0x0004 - Path is a tangent segment path. 0x0008 - Path is cyclic.
6	SpeedNum	Path speed numerator.
7	SpeedDen	Path speed denominator.
8	MaxDev	Maximum allowed deviation.
9-12	PathName	Path name (up to 16 characters).

12.2.1 CAM Array Diagram

CurveArray (Long orientiert)

1	1	Identification (999.000.001)	
2	2	Version (101, 102, or 103)	
3	3	CurveType	Defines endpoint handling
4	4	CurveName1 (4char)	
5	5	CurveName2 (4char)	
6	6	CurveName3 (4char)	
7	7	CurveName4 (4char)	
8	8	IndexCIF - CurveInformation -	Default = 17
9	9	Index STP - Start-Stop Points	Default = 31 (32 for Version 102)
10	10	Index FIP - Fixpoint Part	Default STP+STPno*2
11	11		
12	12	Index Path Information	Default ExtraInfo+12
13	13		
14	14		
15	15		
16	16	Extra Info Index	**Always directly after fix point area**
17	1	MasterCycleLen (MU)	Length of Curve in Master Units
18	2	SlaveCycleLen (UU) -	Max. Slave travel dist in UU
19	3	FixPoint number	Number of fix points (minimum 2)
20	4	PathInfoLength	Size of path info part in bytes
21	5	Interpolation Type	(0 = open, 1 = periodic, 2+3 spec)
22	6	SlaveStopPosition	position slave has to be after stop
23	7	CorrectionStartPoint	Pos. where correction may start
24	8	CorrectionStopPoint	Pos. where correction has to stop
25	9	Maximum Correction	maxmal allowed correction
26	10		
27	11	StartStopNo	Number of start stop point pairs
28	12	MMaxCycles	Max cycles per minute (info only)
29	13	MMarkerPos	Master Marker Pos. in curve
30	14	SMarkerPos	Slave Marker Pos in curve (cmd)
31	15	ExtraDataSize	Size of extra Data (Version>=102)
32	1	STPoint_1a	Start Point Pair 1 - Point A
		STPoint_1b	Start Point Pair 1 - Point B
		.....	
		FixPoint_1.master	Master Coordinate
		FixPoint_1.slave	Slave Coordinate
		FixPoint_1.type	Type (C = curve, T = Tangent)
		.....	
		FixPoint_n.master	Master Coordinate
		FixPoint_n.slave	Slave Coordinate
		FixPoint_n.type	Type (C = curve, T = Tangent)
		StartVelocityNum	
		StartVelocityDen	
		StartAccelerationNum	
		StartAccelerationDen	
		StartJerkNum	(currently, not used)
		StartJerkDen	(currently, not used)
		EndVelocityNum	
		EndVelocityDen	
		EndAccelerationNum	
		EndAccelerationDen	
		EndJerkNum	(currently, not used)

103 = Path info added

Defines the handling of start / end velocities and/or accelerations of the curve. See the Aposs help for a list of the currently supported values.

Used since 103 – for path info

Since Version 102, used if CurveType (see 3) is >= 3. In this case, the extra info contains start and stop gradients. **This section must always immediately follow the fix points (firmware restriction).**

internally overwritten by FixPoint_n.master - FixPoint_1.master

internally overwritten by (FixPoint_n.slave- FixPoint_1.slave)

maximum is 100 at the moment

Length of path information added since version >= 103

Interpolation Types are interpreted as follows:
0 = open (only relevant for CAM - editor)
1 = periodic (only relevant for CAM - editor)
2 = Labeling - old version with precalculation
3 = Labeling - actual development version

wird im Moment nicht übergeben !! Stop ohne CURVESTOP wird wohl nicht funktionieren

next 3 only relevant for marker correction

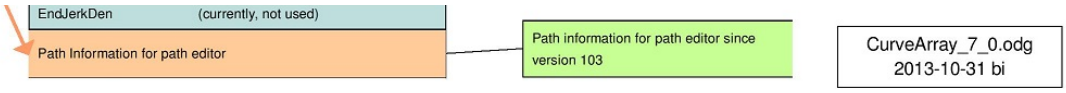
next 2 only relevant for marker correction

At the moment length is 12

values are given in MU. Point A is where the stopping begins and B is where it has to be finished. If driving backward, it is vice versa.

Fixpoints are give in MasterUnits (master coordinate) or UserUnits (slave coordinate). The type can be either
C = CurvePoint
T = TangentPoint
More then two curvepoints are interpolated by cubic spline functions. That means, that a 3rd order Polynom connects two curvepoints in that way, that the position, velocity and acceleration are identical in intermediate Curvepoints.
If there is only one CurvePoint followed by an TangentPoint ore a TangentPoint followed by only one CurvePoint (endpoint) or two tangents following each other, then we use 5th order polynoms to connect these points.

This block defines velocities and accelerations (as numerators and denominators) for the curve start and end points. One or more of these values are used if the specified CurveType requires user-specified start/end gradients and the gradient is not predetermined in some other way (e.g. by the curve starting or ending with a tangent).



12.3 "Record" Array Structure

"Record" arrays are designed to be used to assist in the debugging of applications. They allow user-specified operational data to be recorded in real time while the application is running. This data can then be analyzed at a later time, for example after an error has occured.

When recording is active, "samples" are recorded at regular intervals and each sample consists of a set of one or more system variables. Samples are saved in the array sequentially. However, the array is designed to support cyclic recording if the array becomes full. It is also possible to have multiple recordings in a single "Record" array as long as all recordings refer to the same set of system variables. Note that cyclic recording is typically incompatible with multiple recordings, since one cyclic recording will consume the entire available array leaving no more room for any subsequent recordings.

See RecordIndex for information about how to configure "Record" arrays in an application program. See **Tools** → **Oscilloscope (Record)** for information about how to upload and view the recorded data.

"Record" arrays can be stored either in global DIM arrays or in dynamic (free) memory. The required size of the array is determined partly by the firmware version of the controller but primarily by the number of values to be stored per sample and the total number of samples that will be stored. The structure and size of "Record" arrays is shown below. Note that all values stored in the "Record" array will have a length of 4 bytes.

In general, "Record" arrays are always structured as follows:

Header
Recording 1
Recording 2
Recording ...

The number of recordings is determined when the "next" recording has a sample count of 0 or when there is not enough room in the array for another recording.

Version 0 and Version 1 (see in APOSS Help Legacy)

Version 2 Array Format (controllers with firmware version ≥ 7.2.07)

Header Format:

Index	Content	Value	Description
1	Version	999009002	Version identification of the data structure.
2	Reserved	0	Reserved.
3	Interval	ticks	Sample interval in system clock ticks (approx. 1 ms/tick).
4	Time base	µs	Length of sample interval tick (in microseconds).
5	Axis count	k	Number of axes.
5+1	VELRES	...	VELRES value for first axis.
...	...	...	...

5+k	VELRES	...	VELRES value for k'th axis.
5+k+1	SCALESHIFT	...	SCALESHIFT value for first axis.
...	...	...	...
5+2k	SCALESHIFT	...	SCALESHIFT value for k'th axis.
5+2k+1	Date	...	Date from real time clock (SDO 0x2202/118) at start of recording.
5+2k+2	Time	...	Time from real time clock (SDO 0x2202/119) at start of recording.
5+2k+3	System clock	ticks	System clock time (SDO 0x2202/23) at start of recording.
5+2k+4	Variable count	m	Number of variables defined.
5+2k+4+1	Variable 1	index	SYSVAR index of first variable.
5+2k+4+2	Variable 2	index	SYSVAR index of second variable.
...	...	...	...
5+2k+4+m	Variable m	index	SYSVAR index of m'th variable.

Recording Format:

Index	Content	Value	Description
5+2k+4+m+1	Sample count	n	Number of samples in recording.
5+2k+4+m+2	First sample	0 to n-1	Sample number of oldest sample in recording (0 if non-cyclic).
5+2k+4+m+3	Stop time	ticks	System clock time when recording was stopped.
5+2k+4+m+3+1	Value	...	Sample 1, Variable 1.
5+2k+4+m+3+2	Value	...	Sample 1, Variable 2.
...	...	...	...
...	Value	...	Sample 1, Variable m.
...	Value	...	Sample 2, Variable 1.
...	Value	...	Sample 2, Variable 2.
...	...	...	...
5+2k+4+m+3+nm	Value	...	Sample n, Variable m.

12.4 Illustrations

12.4.1 BinFileMap

Bin File Map (Compiler >= 6.5.36)

Offset (Bytes) CodeVersion >= 5

Offset (Bytes) CodeVersion < 5

0	0	COD_VERSION (0)
2	1	CodeVersion
4	2	COD_PRGHDR (119 / 0x77) - (0x77 0x00) (CodeVersion >= 5)
6	3	Size of Header information in Bytes
8	5	MaxVars (number of Variables defined in Aposs + 20)
10	7	Stacksize (size of stack defined in Aposs)
12	9	Unique 20 byte hex number (RIPEMD-160)
32	29	First command of the application program
34	30	..... more code .....
...	...	.....
n	n	COD_ENDE (127 / 0x7F) - (0x00 0x7F) (CodeVersion >= 5)
0	0	Filename[0] - 1st character of file name (s = strlen)
1	1	Filename[1] - 2nd character of filename
..	..	more characters
s-1	s-1	Filename[s-1] - last character of fname
s	s	Filename[s] - terminating \0 character
?	?	Extra \0 if length of filename was odd
	e-28	BigEndian (word - 2 bytes) - either 1 or 0 (true or false)
	e-26	DoubleSize (word - 2 bytes) Size of a double (number of longs)
	e-24	FirmwareVersion (long - 4 bytes)
	e-20	Minfw (Minimum Firmware Version) (long 4 - bytes)
	e-16	Cmpvers (Compiler Version) (long 4 - bytes)
	e-12	filtime.dwHighDateTime (long 4 - bytes)
	e-8	filtime.dwLowDateTime (long 4 - bytes)
	e-4	InfoSize (in bytes) - l(long 4 bytes -- PC notation, LSB first)
	e	0x55FF44EE - marks end of information section (LSB first)

New information has to be added here.

CodeVersion is as follows
2 - firmware < 6.04.00 exesize had to be smaller than 65000
3 - firmware < 6.06.00
4 - new fp commands and optimizing commands
5 - firmware < 6.06.00 - commands and parameters are always 2 bytes , all jumps are 4 bytes.
6 - firmware >= 6.06.00 - firmware supports new posints and posintio (axes specific) OnStatbit does support axes now.
7 - firmware >= 6.07.00 - firmware needs numbers in cpu spedivic notation (see BigEndian)

This section is only present (for compatibility with old versions) if firmware >= 4.0.30. Then MaxVars is calculated as 92 + 20, where 92 is the number of variables in old controllers and 20 is the number of temporary registers. This number was fix in old controllers. And those controllers do not understand the PRGHDR command.

Size of program header in bytes including command and sizeinfo.
27 - for CodeVersion < 5
28 - for CodeVerision >= 5 (all commands are now 2 bytes)

Unique number calculated by RIPEMD-160. To calculate the total binary part from 0 to n is used inclusive, whith this array filled with 0.

Since version 6.06.00 (Codeversion >= 5) all commands and parameters have at least 2 bytes. Therefore the end of a program looks like 0x7f 0x00 now.

The complete filename is appended. The size of this block is always even. There may be two \0 at the end if necessary.

a number like 60650 means 6.06.50

minimum firmware version required for binary program

Compiler version used to produce binary code (i.e. 60520 means 6.05.20)

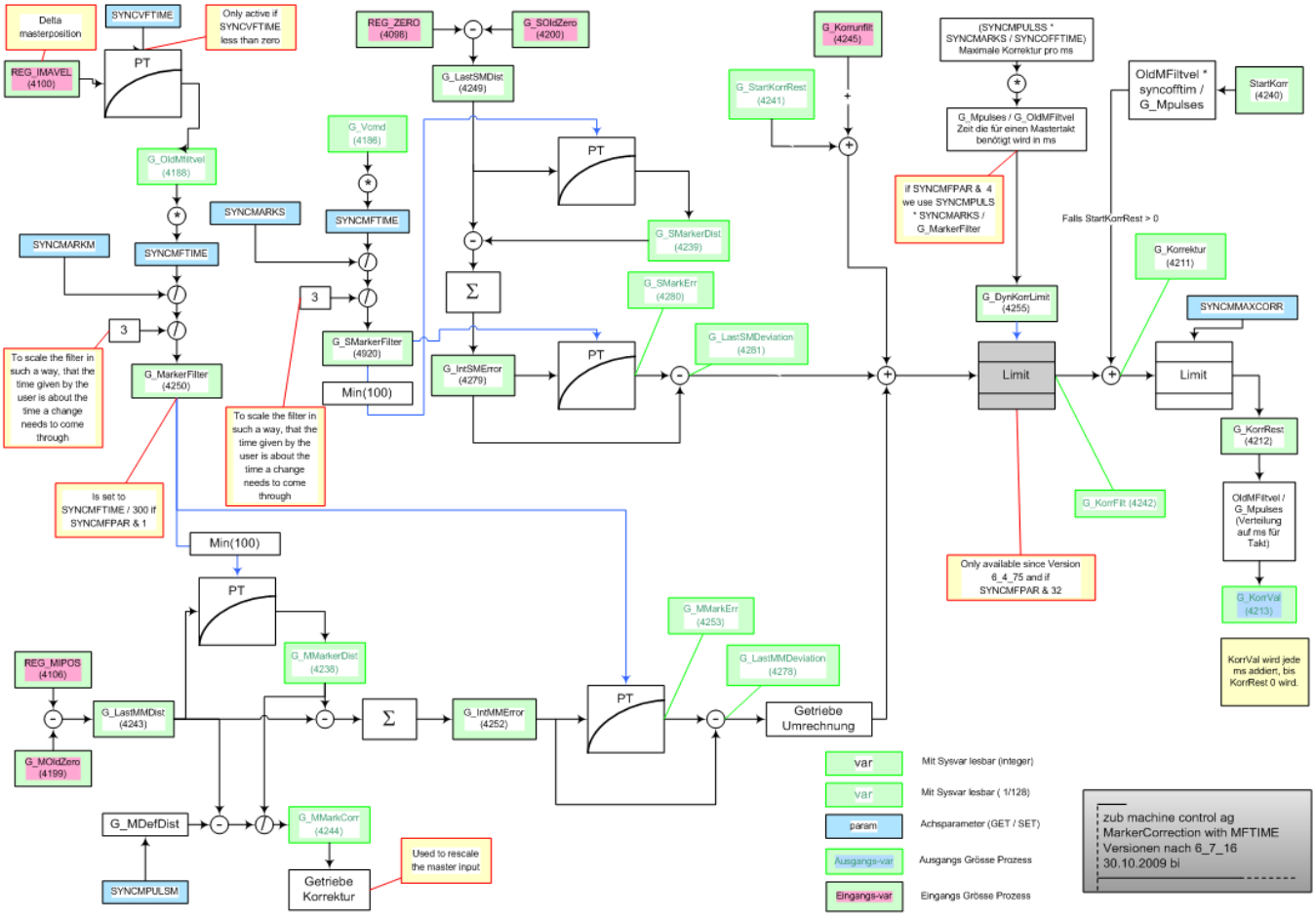
"File time" is the number of 100-nanosecond intervals that have elapsed since 12:00AM, January 1, 1601 (UTC). This can be converted back to a "nice" local time using the functions FileTimeToSystemTime() and SystemTimeToTzSpecificLocalTime().

Size of info section - at the moment this is (s+1+24 or s+2+24)
That means, address of InfoSize - InfoSize should lead to start of string.

All longs in the info section are in PC notation which means LSB first.
This section has to be read from the end backward.

All longs in other sections are MSB first, for Codeversion < 7 which means firmware < 6.7.0
For firmware >= 6.7.01 Codeversion 7 is used and the Integers are ordered corresponding to the CPU.
That means MSB first for Motorola and LSB first for DSP.

12.4.2 Marker Correction



12.5 SysDef Defines

"SysDef.mh" Include File

```
#pragma once

//==========================================================================================
// Error Codes
//==========================================================================================

#include "ErrorCodes.mh"

//==========================================================================================
// SDO Dictionary
//==========================================================================================

#include "SdoDictionary.mh"
```

```
//==========================================================================================
// Version
//==========================================================================================

#define PRODUCED_WITH_APOSS_VERSION    700102

//==========================================================================================
// AxisStatus bits
//==========================================================================================

#define AXSTAT_INDEXMASTER      0x00000001      // Index has occurred on master encoder.
#define AXSTAT_INDEXSLAVE       0x00000002      // Index has occurred on slave encoder.
#define AXSTAT_POSCTRLOFF       0x00000004      // Position control switched off (= MOTOR OFF state).
#define AXSTAT_VOLTAGERANGE     0x00000008      // Supply voltage of power stage is out of range.
#define AXSTAT_STARTACTIVE      0x00000020      // Start switch active.
#define AXSTAT_REFACTIVE        0x00000040      // Reference switch active.
#define AXSTAT_LIMITACTIVE      0x00000080      // Limit switch active.
#define AXSTAT_I2TTHRESHOLD     0x00000100      // I2T treshold or overcurrent error at amplifier.
#define AXSTAT_OVERVOLTAGE      0x00000200      // Overvoltage at the amplifier.
#define AXSTAT_TEMPERATURE      0x00000400      // Overtemperature threshold.
#define AXSTAT_POSREACHED       0x00040000      // Position reached.
#define AXSTAT_FLOATING         0x00800000      // Position control switched off (identical to AXSTAT_POSCTRLOFF).
#define AXSTAT_MOVING           0x01000000      // Axis is moving.
#define AXSTAT_OVERFLOWS        0x02000000      // Overflow slave encoder.
#define AXSTAT_OVERFLOWM        0x04000000      // Overflow master encoder.
#define AXSTAT_POSLIMERR        0x10000000      // Positive software position limit reached.
#define AXSTAT_NEGLIMERR        0x20000000      // Negative software position limit reached.
#define AXSTAT_CURVE_ERR        0x40000000      // Error during execution of SyncCam.
#define AXSTAT_OVERSHOOT        0x80000000      // AxisPosAbsStartTvel command will overshoot the target.

//==========================================================================================
// STATBIT interrupt bit numbers
//==========================================================================================

#define STATBIT_BUSY        0       // Internal: Profile generator busy.
#define STATBIT_COMMANDERR  1       // Internal: Profile generator error.
#define STATBIT_POSREACHED  2       // Position reached.
#define STATBIT_INDEX_HIT   3       // Slave Index observed.
#define STATBIT_INDEXM_HIT  4       // Master Index observed.
#define STATBIT_BRKPT_RCHD  6       // Internal breakpoint reached.
#define STATBIT_FLOATING    7       // Pos. control switched off (= MOTOR OFF state).
#define STATBIT_MOVING      8       // Axis is moving.
#define STATBIT_OVERFLOWS   9       // Overflow of slave position.
#define STATBIT_OVERFLOWM   10      // Overflow of master position.
#define STATBIT_POSFLOAT    11      // Pos. within REGWMAX / REGWMIN.
#define STATBIT_POSLIMERR   12      // Positive pos. sw limit (-> POSLIMIT).
#define STATBIT_NEGLIMERR   13      // Negative pos. sw limit (-> NEGLIMIT).
#define STATBIT_CURVE_ERR   14      // CAM sync. error.
#define STATBIT_OVERSHOOT   15      // AxisPosAbsStartTvel will overshoot or not reach target.
#define STATBIT_SYNCREADY   24      // Sync. ready.
#define STATBIT_SYNCFAULT   25      // Sync. failed.
#define STATBIT_SYNCACCUR   26      // Sync. accuracy reached.
#define STATBIT_SYNCMMHIT   27      // Master marker hit.
#define STATBIT_SYNCSMHIT   28      // Slave marker hit.
#define STATBIT_SYNCMMERR   29      // Master marker error.
#define STATBIT_SYNCSMERR   30      // Slave marker error.

//==========================================================================================
// CAM and PATH Arrays
//==========================================================================================
```

```
// Array index offsets into CAM array header block:

#define CAM_HDR_TYPE        2       // Curve type.
#define CAM_HDR_NAME        3       // Name (16 characters).

// Array index offsets into CAM array info block:

#define CAM_INF_MCLEN       0       // Master cycle len.
#define CAM_INF_SCLEN       1       // Slave cycle len.
#define CAM_INF_FIXNO       2       // Number of fixpoints.
#define CAM_INF_PATHNO      3       // Number of path information values.
#define CAM_INF_SSTOP       5       // Slave stop position.
#define CAM_INF_CORRSTART   6       // Correction start point.
#define CAM_INF_CORRSTOP    7       // Correction stop point.
#define CAM_INF_CORRMAX     8       // Maximum correction allowed.
#define CAM_INF_STARTSTOPNO 10      // Number of startstop pairs.
#define CAM_INF_MMAXCYCLES  11      // Maximum cycles per minute (Appl. Info).
#define CAM_INF_MMARKERPOS  12      // Master marker position.
#define CAM_INF_SMARKERPOS  13      // Slave marker position.
#define CAM_INF_GRADNO      14      // Number of gradient data values (specified end gradients).

// Array index offsets into CAM array gradient block:

#define CAM_GRD_SVELNUM     0       // Start velocity numerator.
#define CAM_GRD_SVELDEN     1       // Start velocity denominator.
#define CAM_GRD_SACCNUM     2       // Start acceleration numerator.
#define CAM_GRD_SACCDEN     3       // Start acceleration denominator.
#define CAM_GRD_SJRKNUM     4       // Start jerk numerator.
#define CAM_GRD_SJTKDEN     5       // Start jerk denominator.
#define CAM_GRD_EVELNUM     6       // End velocity numerator.
#define CAM_GRD_EVELDEN     7       // End velocity denominator.
#define CAM_GRD_EACCNUM     8       // End acceleration numerator.
#define CAM_GRD_EACCDEN     9       // End acceleration denominator.
#define CAM_GRD_EJRKNUM     10      // End jerk numerator.
#define CAM_GRD_EJTKDEN     11      // End jerk denominator.

// Array index offsets into CAM array curve segment block:

#define CAM_SEG_MASTER      0       // Master position.
#define CAM_SEG_SLAVE       1       // Slave position.
#define CAM_SEG_TYPE        2       // Segment type.

// Array index offsets into CAM array start/stop pair block:

#define CAM_SSP_A           0       // First position.
#define CAM_SSP_B           1       // Second position.

// Curve Types (and associated flags):

#define CAM_TYPE_ZUB          0             // Average estimated endpoint velocities.
#define CAM_TYPE_HAUSER       1             // Use estimated start velocity as end velocity.
#define CAM_TYPE_ZERO         2             // Start and end velocities set to 0.
#define CAM_TYPE_GRAD         3             // Start and end velocities specified and possibly updated by
controller.
#define CAM_TYPE_GRAD_INTRPT  4             // Same as 3 but curve starts immediately (i.e. not waiting for the
last to finish).
#define CAM_TYPE_VEL_VEL      5             // Start and end velocities specified.
#define CAM_TYPE_VEL_ACC      6             // Start velocity and end acceleration specified.
#define CAM_TYPE_ACC_VEL      7             // Start acceleration and end velocity specified.
```

```
#define CAM_TYPE_ACC_ACC      8             // Start and end accelerations specified.
#define CAM_TYPE_CYCLIC       9             // Cyclic curve.

#define CAM_TYPE_TANGENT      0x80000000    // All curve segments are to be tangents.

// Curve segment Types (and associated flags):

#define SEG_TYPE_CPOINT       1             // Curve point (next segment is 3'rd or 5'th order polynomial).
#define SEG_TYPE_TPOINT       4             // Next segment is a tangent segment.
#define SEG_TYPE_ZPOINT       5             // Next segment is a trapezoidal segment.
#define SEG_TYPE_3POINT       6             // Next segment is a 3'rd order polynomial.

#define SEG_TYPE_CORNER       0x00000080    // This fixed point is a "corner" point.
#define SEG_TYPE_STOP         0x00000040    // This fixed point is a "stop" point.

// Array index offsets into Path array header block:

#define PTH_HDR_FLAGS         2             // Path flags.
#define PTH_HDR_NUMBER        3             // Path number.
#define PTH_HDR_NAME          4             // Name (16 characters).
#define PTH_HDR_AXISCNT       8             // Number of axes.
#define PTH_HDR_SPATIAL       9             // Number of spatial axes.
#define PTH_HDR_SPEEDN        10            // Speed numerator (uu/sec).
#define PTH_HDR_SPEEDD        11            // Speed denominator (uu/sec).
#define PTH_HDR_MAXDEV        12            // Maximum deviation.
#define PTH_HDR_PATHCNT       14            // Number of paths in array.

// Array index offsets into Path curve info block:

#define PTH_INF_FIXCNT        1             // Number of fixed points.
#define PTH_INF_FLAGS         2             // Path curve flags.
#define PTH_INF_NAME          3             // Name (16 characters).
#define PTH_INF_DRCT          7             // Directions for each axis (4 per axis).

// Array index offsets into Path curve direction block:

#define PTH_DIR_STARTNUM      0             // Start velocity numerator.
#define PTH_DIR_STARTDEN      1             // Start velocity denominator.
#define PTH_DIR_STOPNUM       2             // Stop velocity numerator.
#define PTH_DIR_STOPDEN       3             // Stop velocity denominator.

// Array index offsets into Path curve point block:

#define PTH_SEG_DIST          0             // Approximate arc length to start of segment.
#define PTH_SEG_TYPE          1             // Segment type of this segment.
#define PTH_SEG_COORD         2             // Fixed point coordinate of start of segment (1 per axis).

#define PathSegType(type)   ((type) & 0x0000003F)           // Actual type of this segment.
#define PathSegCode(type)   (((type) >> 8) & 0x00FFFFFF)    // User code associated with this segment.
#define PathSegCorner(type) (((type) & 0x00000080) != 0)    // The start of the segment is a "corner" point.
#define PathSegStop(type)   (((type) & 0x00000040) != 0)    // The start of the segment is a "stop" point.

// Path curve flags:

#define PATH_FLAG_TANGENT     0x0004        // Path is a tangent segment path.
#define PATH_FLAG_CYCLIC      0x0008        // Path is a cyclic path.

//=============================================================================================================
// Kinematics Defines
//=============================================================================================================
```

```
// Coordinate Systems:

#define PATHCOORD_MACHINE       0       // Machine Coordinate System.
#define PATHCOORD_WORK          1       // Work Coordinate System.
#define PATHCOORD_CYCLIC        2       // Cyclic Coordinate System.
#define PATHCOORD_PATH          3       // Path Coordinate System.

// PATHSTART behaviour flags:

#define PATHSTART_NORMAL        0x00    // Follow the path using a normal movement.
#define PATHSTART_VIRTMAST      0x01    // Follow the path using the virtual master of the first axis.
#define PATHSTART_STOP          0x02    // Stop at each stop point.
#define PATHSTART_WAIT          0x04    // Stop and wait at each stop point.
#define PATHSTART_IMMEDIATE     0x08    // Start the path curve immediately when the sync point is reached.

//=================================================================================================================
// #config Statement Defines
//=================================================================================================================

#define CONFIGFLG_DIM           0x01    // Load DIM arrays.
#define CONFIGFLG_USER          0x02    // Load User parameters.
#define CONFIGFLG_AXIS          0x04    // Load Axis parameters.
#define CONFIGFLG_MODULE        0x08    // Load Module parameters.

//=================================================================================================================
// State Machine Defines
//=================================================================================================================

#define SM_MACH_DATA            SM_MACH_DATA_1      // First subindex of private data

// Bit numbers for axis status events:

#define SM_STAT_BUSY             0      // Internal: Profile generator busy
#define SM_STAT_COMMANDERR       1      // Internal: Profile generator error
#define SM_STAT_POSREACHED       2      // Position reached
#define SM_STAT_INDEX_HIT        3      // Index observed
#define SM_STAT_BRKPT_RCHD       6      // Internal Breakpoint reached
#define SM_STAT_FLOATING         7      // Pos. control switched off (= MOTOR OFF state)
#define SM_STAT_MOVING           8      // Axis is moving
#define SM_STAT_OVERFLOWS        9      // Overflow of slave position
#define SM_STAT_OVERFLOWM       10      // Overflow of master position
#define SM_STAT_POSFLOAT        11      // Pos. within REGWMAX / REGWMIN
#define SM_STAT_POSLIMERR       12      // Positive pos. sw limit (-> POSLIMIT)
#define SM_STAT_NEGLIMERR       13      // Negative pos. sw limit (-> NEGLIMIT)
#define SM_STAT_CURVE_ERR       14      // CAM sync. error
#define SM_STAT_OVERSHOOT       15      // AxisPosAbsStartTvel will overshoot or not reach target
#define SM_STAT_SYNCREADY       24      // Sync. ready
#define SM_STAT_SYNCFAULT       25      // Sync. failed
#define SM_STAT_SYNCACCUR       26      // Sync. accuracy reached
#define SM_STAT_SYNCMMHIT       27      // Master marker hit
#define SM_STAT_SYNCSMHIT       28      // Slave marker hit
#define SM_STAT_SYNCMMERR       29      // Master marker error
#define SM_STAT_SYNCSMERR       30      // Slave marker error

// State machine runtime flags:

#define SM_RUN_DELAY              0x0001  // Aposs commands that "delay" are allowed.
#define SM_RUN_INTERRUPT          0x0002  // Interrupts are allowed.
#define SM_RUN_DELDISABLE         0x0004  // Disable error when unknown SmParam/SmInput/SmSystem record is
```

```
deleted.
#define SM_RUN_SIGINIT             0x0008  // Always post SIG_INIT to target state.
#define SM_RUN_NO_DUPLICATE_EVENTS 0x0010  // Ignore any events that are already in the pool

// Test types for the SmParam command:

#define SM_PARAM_CHANGE         0       // Parameter value changed.
#define SM_PARAM_EQUAL          1       // Parameter value changed to equal the mask value.
#define SM_PARAM_NOTEQUAL       2       // Parameter value changed to a value not equal to the mask value.
#define SM_PARAM_MASK           3       // Any bit corresponding to a mask bit has changed.
#define SM_PARAM_RISING         4       // The bit specified by the mask value has changed to 1.
#define SM_PARAM_FALLING        5       // The bit specified by the mask value has changed to 0.
#define SM_PARAM_GT             6       // Parameter value changed to a value greater than the mask value.
#define SM_PARAM_GE             7       // Parameter value changed to a value greater than or equal to the mask
value.
#define SM_PARAM_LT             8       // Parameter value changed to a value less than the mask value.
#define SM_PARAM_LE             9       // Parameter value changed to a value less than or equal to the mask value.

// Test types for the SmInput command:

#define SM_INPUT_CHANGE         0       // Input changed.
#define SM_INPUT_RISING         1       // Input has changed to 1.
#define SM_INPUT_FALLING        2       // Input has changed to 0.

// Test types for the SmSystem PFGEVENT command:

#define SM_PFG_PATHMOVEDONE     1       // The PathMove is complete.
#define SM_PFG_PATHSTARTDONE    2       // The start curve is complete.
#define SM_PFG_PATHFIXPNT       3       // Arrived at a fixed point.
#define SM_PFG_PATHSTOP         4       // Path movement has stopped.
#define SM_PFG_PATHCONSTVEL     5       // Path movement has reached a constant velocity.
#define SM_PFG_PATHCYCLE        6       // The current path curve is complete and the next path has been started.
#define SM_PFG_PATHTERMINATE    7       // Kinematics mode terminated.

// Event value flags for SM_PFG_PATHMOVEDONE and SM_PFG_PATHSTOP events:

#define SM_PATHSTOP_USER        0x01    // PathStop was executed.
#define SM_PATHSTOP_MOVE        0x02    // PathMove command reached its target position.
#define SM_PATHSTOP_SYNC        0x04    // Path movement stopped at the synchronization point.
#define SM_PATHSTOP_FIX         0x08    // Path movement stopped at a fixed point.
#define SM_PATHSTOP_END         0x10    // Path movement stopped at the end of the current active path curve.
```

"SdoDictionary.mh" Include File

```
#pragma once
#define sdodictionary_h     // For backwards compatibility in case user programs use this #define.

//=============================================================================================================
// SDO Defines
//=============================================================================================================

#define SDOSUBCLASS(index) (index & 0xFFC0)    // important for firmware !!!!

#define PISRC_NOTDEFINED                    0x0000

// General definition for SDO Read and Write
#define SDO_INTERNAL(ind,subindex)          Sysvar[(0x01000000 | (((long) (ind  ))<<8)) | ((long) (subindex))]
#define SDO_DIMARRAY(arrno,index)           Sysvar[(0x01210005 | (((long) (arrno))<<8)) + ((long) (index)))]

//=============================================================================================================
```

```
// 0x2100 APOSS_ARRAY - Aposs Array
//=================================================================================================
#define SDOINDEX_APOSS_ARRAY                        0x2100
#define APOSS_ARRAY_INDEX(modno,parno)              ((0x01000000 | (SDOINDEX_APOSS_ARRAY<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define APOSS_ARRAY_SRCINDEX(modno,parno)           ((SDOINDEX_APOSS_ARRAY<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define APOSS_ARRAY(modno,parno)                    Sysvar[APOSS_ARRAY_INDEX((modno),(parno))]
#define APOSS_ARRAY_MAX                             255

//=================================================================================================
// 0x2200 GLB_PARAM - Global Parameters
//=================================================================================================
#define SDOINDEX_GLB_PARAM                          0x2200
#define GLB_PARAM_INDEX(parno)                      ((0x01000000 | (SDOINDEX_GLB_PARAM<<8)) | ((long) (parno)))
#define GLB_PARAM_SRCINDEX(parno)                   ((SDOINDEX_GLB_PARAM<<8) | ((long) (parno)))
#define GLB_PARAM(parno)                            Sysvar[GLB_PARAM_INDEX(parno)]
#define GLB_PARAM_MAX                               102

#define CANNR                           1           // CAN node ID
#define CANBAUD                         2           // CAN baud rate
#define PRGPAR                          3           // Activated program number
#define I_ERRCLR                        8           // Input to clear error
#define O_ERROR                         9           // Output set when error occurs
#define RSBAUDRATE                      13          // RS485 baud rate
#define CANSYNCTIMER                    15          // Cycle time for sending SYNC telegrams on the CAN bus
#define CANGUARDTIMER                   16          // Period of guarding
#define IPADDRESSMODE                   20          // IP address mode
#define IPSUBNET                        21          // IP subnet
#define PRGSTARTCOND                    22          // Program start condition
#define ESCCONDGLB                      23          // ESC Behaviour global
#define IPMASK                          24          // IP mask
#define IPDEFAULTGW                     25          // IP default gateway
#define CANNRFORCE                      26          // Node Id DIP Switch overwrite

    // Values for ESCCONDGLB:
    #define GLB_PARAM_ESCCONDGLB_NOTHING   0        // The actual values of the outputs are not changed.
    #define GLB_PARAM_ESCCONDGLB_OUT_OFF   1        // Set all outputs to 0
    #define GLB_PARAM_ESCCONDGLB_OUT_ON    2        // Set all outputs to 1

//=================================================================================================
// 0x2201 USER_PARAM - User Parameters
//=================================================================================================
#define SDOINDEX_USER_PARAM                         0x2201
#define USER_PARAM_INDEX(parno)                     ((0x01000000 | (SDOINDEX_USER_PARAM<<8)) | ((long) (parno)))
#define USER_PARAM_SRCINDEX(parno)                  ((SDOINDEX_USER_PARAM<<8) | ((long) (parno)))
#define USER_PARAM(parno)                           Sysvar[USER_PARAM_INDEX(parno)]
#define USER_PARAM_MAX                              101

//=================================================================================================
// 0x2202 SYS_PROCESS - System Process Data
//=================================================================================================
#define SDOINDEX_SYS_PROCESS                        0x2202
#define SYS_PROCESS_INDEX(parno)                    ((0x01000000 | (SDOINDEX_SYS_PROCESS<<8)) | ((long) (parno)))
#define SYS_PROCESS_SRCINDEX(parno)                 ((SDOINDEX_SYS_PROCESS<<8) | ((long) (parno)))
#define SYS_PROCESS(parno)                          Sysvar[SYS_PROCESS_INDEX(parno)]
#define SYS_PROCESS_MAX                             143

#define SYS_INBYTE_0                    2           // Digital input state (Byte 0 = Input 1-8)
#define SYS_INBYTE_1                    3           // Digital input state (Byte 1 = Input 9-16)
```

```
#define SYS_INBYTE_2                4           // Digital input state (Byte 2 = Input 17-24)
#define SYS_INBYTE_3                5           // Digital input state (Byte 3 = Input 25-32)
#define SYS_INBYTE_4                6           // Digital input state (Byte 4 = Input 33-40)
#define SYS_INBYTE_5                7           // Digital input state (Byte 5 = Input 40-48)
#define SYS_INBYTE_6                8           // Digital input state (Byte 6 = Input 49-56)
#define SYS_INBYTE_7                9           // Digital input state (Byte 7 = Input 57-64)
#define SYS_OUTBYTE_0               10          // Digital output state (Byte 0 = Output 1-8)
#define SYS_OUTBYTE_1               11          // Digital output state (Byte 1 = Output 9-16)
#define SYS_OUTBYTE_2               12          // Digital output state (Byte 2 = Output 17-24)
#define SYS_OUTBYTE_3               13          // Digital output state (Byte 3 = Output 25-32)
#define SYS_OUTBYTE_4               14          // Digital output state (Byte 4 = Output 33-40)
#define SYS_OUTBYTE_5               15          // Digital output state (Byte 5= Output 41-48)
#define SYS_OUTBYTE_6               16          // Digital output state (Byte 6 = Output 49-56)
#define SYS_OUTBYTE_7               17          // Digital output state (Byte 7 = Output 57-64)
#define SYS_AXSTATUS                18          //
#define SYS_INAD_1                  19          // Analog input value (inputs 1-4)
#define SYS_INAD_2                  20          // Analog input value (inputs 1-4)
#define SYS_INAD_3                  21          // Analog input value (inputs 1-4)
#define SYS_INAD_4                  22          // Analog input value (inputs 1-4)
#define SYS_CLOCK                   23          // Internal millisecond counter
#define SYS_USRSTAT                 24          // User status
#define SYS_MAXAX                   25          // Maximum number of axes
#define SYS_CAN_NODESTATE           26          // CAN NMT status
#define SYS_TIMEINT_LENGTH          27          // Duration of the last ms-interrupt
#define SYS_MAX_LOOPTIME            28          // Maximum loop time
#define SYS_AMP0_CURRENT            29          // Up-to-date motor current
#define SYS_NC_0                    30          //
#define SYS_AMP0_VOLTAGE            31          // Up-to-date motor voltage
#define SYS_AMP0_STATUS             32          // VLT status
#define SYS_AMP0_SPEED              33          // VLT output frequency
#define SYS_CANDIP                  34          // Dip-Switch value (CAN node ID)
#define SYS_CANOM_SDOABORT          35          // Last CANopen SDO abort code
#define SYS_HWANIN_UNFILT           36          // Unfiltered value of analog input
#define SYS_CANERR_RX_0             37          // CAN error counter for RX Master-Bus
#define SYS_CANERR_RX_1             38          // CAN error counter for RX Slave-Bus
#define SYS_CANERR_TX_0             39          // CAN error counter for TX Master-Bus
#define SYS_CANERR_TX_1             40          // CAN error counter for TX Slave-Bus
#define SYS_MAXTIMEINT_LENGTH       41          // Maximum duration of the ms-interrupt
#define SYS_HWCNTINC_0_VALUE        42          // Internal encoder counter (encoders 1-9)
#define SYS_HWCNTINC_1_VALUE        43          // Internal encoder counter (encoders 1-9)
#define SYS_HWCNTINC_2_VALUE        44          // Internal encoder counter (encoders 1-9)
#define SYS_HWCNTINC_3_VALUE        45          // Internal encoder counter (encoders 1-9)
#define SYS_HWCNTINC_4_VALUE        46          // Internal encoder counter (encoders 1-9)
#define SYS_HWCNTINC_5_VALUE        47          // Internal encoder counter (encoders 1-9)
#define SYS_HWCNTINC_6_VALUE        48          // Internal encoder counter (encoders 1-9)
#define SYS_HWCNTINC_7_VALUE        49          // Internal encoder counter (encoders 1-9)
#define SYS_HWCNTINC_8_VALUE        50          // Internal encoder counter (encoders 1-9)
#define SYS_STACKSIZE               51          // Stack size of running Aposs program (or last program to
run)
#define SYS_HWLATCH_0_VALUE         52          // Internal encoder zero pulse position (encoders 1-9)
#define SYS_HWLATCH_1_VALUE         53          // Internal encoder zero pulse position (encoders 1-9)
#define SYS_HWLATCH_2_VALUE         54          // Internal encoder zero pulse position (encoders 1-9)
#define SYS_HWLATCH_3_VALUE         55          // Internal encoder zero pulse position (encoders 1-9)
#define SYS_HWLATCH_4_VALUE         56          // Internal encoder zero pulse position (encoders 1-9)
#define SYS_HWLATCH_5_VALUE         57          // Internal encoder zero pulse position (encoders 1-9)
#define SYS_HWLATCH_6_VALUE         58          // Internal encoder zero pulse position (encoders 1-9)
#define SYS_HWLATCH_7_VALUE         59          // Internal encoder zero pulse position (encoders 1-9)
#define SYS_HWLATCH_8_VALUE         60          // Internal encoder zero pulse position (encoders 1-9)
#define SYS_EEPROM_SIZE             61          // Size of Flash (or EEprom)
#define SYS_US_TIMER                62          // Internal microsecond time base
```

```
#define SYS_CANOS_TIM_LASTTXPDO        63          // Period of time since the last TxPDO was transmitted
#define SYS_CANOM_SDO_TIMEOUT          64          // SDO timeout
#define SYS_ECAT_NODESTATE             65          // Node state of the EtherCAT slave
#define SYS_ECAT_PDOLENGTH_IN          66          // Mailbox length of the EtherCAT PDO which was received via
the bus
#define SYS_ECAT_PDOLENGTH_OUT         67          // Mailbox length of the EtherCAT PDO which was put on the bus
for transmission
#define SYS_CANOM_GUARDERR_ID          68          // Node ID which caused the last error
#define SYS_CANOM_MASTERSTATE          69          // State of CAN master functionality
#define SYS_UART_BYTECOUNT             70          //
#define SYS_HWCNTINC_0_STATUS          72          // Internal encoder status (encoders 1-9)
#define SYS_HWCNTINC_1_STATUS          73          // Internal encoder status (encoders 1-9)
#define SYS_HWCNTINC_2_STATUS          74          // Internal encoder status (encoders 1-9)
#define SYS_HWCNTINC_3_STATUS          75          // Internal encoder status (encoders 1-9)
#define SYS_HWCNTINC_4_STATUS          76          // Internal encoder status (encoders 1-9)
#define SYS_HWCNTINC_5_STATUS          77          // Internal encoder status (encoders 1-9)
#define SYS_HWCNTINC_6_STATUS          78          // Internal encoder status (encoders 1-9)
#define SYS_HWCNTINC_7_STATUS          79          // Internal encoder status (encoders 1-9)
#define SYS_HWCNTINC_8_STATUS          80          // Internal encoder status (encoders 1-9)
#define SYS_HWLATCH_COUNT_0            93          // Encoder index counter (encoders 1-9)
#define SYS_HWLATCH_COUNT_1            94          // Encoder index counter (encoders 1-9)
#define SYS_HWLATCH_COUNT_2            95          // Encoder index counter (encoders 1-9)
#define SYS_HWLATCH_COUNT_3            96          // Encoder index counter (encoders 1-9)
#define SYS_HWLATCH_COUNT_4            97          // Encoder index counter (encoders 1-9)
#define SYS_HWLATCH_COUNT_5            98          // Encoder index counter (encoders 1-9)
#define SYS_HWLATCH_COUNT_6            99          // Encoder index counter (encoders 1-9)
#define SYS_HWLATCH_COUNT_7            100         // Encoder index counter (encoders 1-9)
#define SYS_HWLATCH_COUNT_8            101         // Encoder index counter (encoders 1-9)
#define SYS_TIMEINT_DIFF               104         // Time between the last two millisecond interrupts
#define SYS_LED_0_CONTROL              110         // Control LED state
#define SYS_LED_1_CONTROL              111         // Control LED state
#define SYS_LED_2_CONTROL              112         // Control LED state
#define SYS_LED_3_CONTROL              113         // Control LED state
#define SYS_LED_4_CONTROL              114         // Control LED state
#define SYS_LED_5_CONTROL              115         // Control LED state
#define SYS_INAD_5                     116         // Analog input value (input 5)
#define SYS_INAD_6                     117         // Analog input value (input 6)
#define SYS_TIME_DAY                   118         // Day value of the time stamp structure
#define SYS_TIME_TOD                   119         // Time of day (from beginning of day)
#define SYS_RTC_DATE                   120         // Real time date in format yyyymmdd (decimal). E.g. 20160209
= Sept 2nd 2016
#define SYS_RTC_TIME                   121         // Real time in format hhmmss
#define SYS_TIMER_GETDAY               122         // Returns the day information (days since 1984-1-1), at which
the controller started up
#define SYS_TIMER_GETTIMEOFDAY         123         // Returns the timeclk information (seconds since start of
this day), at which the controller started up
#define SYS_GETMAXINHIBITTIME          124         // Returns the maximum time for which the millisecond timer
was inhibited
#define SYS_PRETIMEINT_LENGTH          125         // Returns the length of the last pre-ms-interrupt in
microseconds
#define SYS_MAXPRETIMEINT_LENGTH       126         // Returns the max length of the last pre-ms-interrupt in
microseconds. Clear by reading.
#define SYS_PRETIMEINT_DIFF            127         // Returns the difference between this and the last pre-ms-
interrupt in microseconds
#define SYS_MAXPRETIMEINT_DIFF         128         // Returns the max difference between this and the last pre-
ms-interrupt in microseconds
#define SYS_MAXTIMEINT_DIFF            129         // Returns the max difference between this and the last ms-
interrupt in microseconds
#define SYS_CAN_NR_OF_SYNC             130         // Returns the number of received SYNC telegrams
#define SYS_CANOM_SEND_NMT             131         // Activates the sending of NMT start telegrams
```

```
#define SYS_STACKUSAGE                  132         // Current stack usage of running Aposs program (or last
program to run)
#define SYS_STACKMAX                    133         // Maximum stack usage of running Aposs program (or last
program to run)
#define SYS_BL_STATUS                   134         // Bootloader update status
#define SYS_TEMPERATURE                 135         // Temperature of the PCB in 1/100th Degrees Celsius
#define SYS_FWSTACKMAX                  136         // Returns the maximum of the used firmware stack in bytes
#define SYS_CURINT_LENGTH               137         // Returns the duration of the last current control interrupt
in microseconds
#define SYS_MAXCURINT_LENGTH            138         // Returns the max duration of the last current control
interrupt in microseconds. Clear by reading.
#define SYS_CURINT_DIFF                 139         // Returns the difference between this and the last current
control interrupt in microseconds
#define SYS_MAXCURINT_DIFF              140         // Returns the maximum difference between this and the last
current control interrupt in microseconds
#define SYS_CPU_TEMPERATURE             141         // Temperature of the CPU in 1/100th Degrees Celsius
#define SYS_CAN_SYNC_OFFSET             142         // Returns a debug value for the CAN Sync function

//==========================================================================================================
// 0x2209 SYS_INFO - System Information
//==========================================================================================================
#define SDOINDEX_SYS_INFO                           0x2209
#define SYS_INFO_INDEX(parno)                       ((0x01000000 | (SDOINDEX_SYS_INFO<<8)) | ((long) (parno)))
#define SYS_INFO_SRCINDEX(parno)                    ((SDOINDEX_SYS_INFO<<8) | ((long) (parno)))
#define SYS_INFO(parno)                             Sysvar[SYS_INFO_INDEX(parno)]
#define SYS_INFO_MAX                                79

#define SYS_HARDWARE_ID                 1           // Hardware identification
#define SYS_FW_CPU                      2           // Software version
#define SYS_CUSTOMER_ID                 3           // Customer identification code
#define SYS_MAX_AXES                    5           // Number of axes available
#define SYS_FPGA_SW_VERSION             6           // FPGA software version
#define SYS_BOOTLOADER_SW_VERSION       7           // Bootloader software version
#define SYS_APP_VERSION_INFO1           8           // Application extra version 1
#define SYS_APP_VERSION_INFO2           9           // Application extra version 2
#define SYS_NUMBER_OF_INTEGRATED_AMPLIFIERS 10          // Number of integrated amplifiers
#define SYS_INTERNAL_FLASH_MEMORY       11          // Device type of the internal flash memory
#define SYS_FW_AMPLIFIER_VERSION        12          // Amplifier Version
#define SYS_FW_COPROCESSOR_VERSION      13          // Coprocessor software version
#define SYS_IP_ADDRESS_CONTROLLER       14          // IP address of controller
#define SYS_ANALOG_OPTION_TYPE          15          // Analog option type
#define SYS_SD_CARD_STATUS              16          // SD Card status
#define SYS_MAX_VIRTMAST                20          // Number of virtual masters
#define SYS_MAX_VIRTCOUNTIN             21          // Number of virtual counter inputs
#define SYS_MAX_VIRTLATCH               22          // Number of virtual inputs
#define SYS_MAX_VIRTAMP                 23          // Number of virtual amplifiers
#define SYS_MAX_VIRTDIGIN               24          // Number of virtual digital inputs
#define SYS_MAX_VIRTDIGOUT              25          // Number of virtual digital outputs
#define SYS_MAX_VIRTANIN                26          // Number of virtual analog inputs
#define SYS_MAX_VIRTANOUT               27          // Number of virtual analog outputs
#define SYS_MAX_DIM_ARRAYS              28          // Maximum allowed number of DIM arrays
#define SYS_MAX_VIRTCUSTOMMOD           29          // Maximum allowed number of virtual customer modules
#define SYS_MAX_VIRTMATH                30          // Maximum allowed number of virtual math modules
#define SYS_ZB_AMP_NO                   40          // Number of H/W amplifiers
#define SYS_MAX_ENCODER                 41          // Number of H/W encoders
#define SYS_MAX_CNTINC                  42          // Number of H/W incremental counters
#define SYS_MAX_CNTABS                  43          // Number of H/W absolute counters
#define SYS_MAX_CNTUNI                  44          // Number of H/W universal counters
#define SYS_MAX_COMPARATORS             45          // Number of H/W comparators
#define SYS_MAX_LATCH                   46          // Number of H/W latches
```

```
#define SYS_MAX_SIGGEN                47          // Number of H/W signal generators
#define SYS_MAX_DDS                   48          // Number of DDS modules
#define SYS_MAX_PULSGEN               49          // Number of pulse generators
#define SYS_MAX_DIGIN                 50          // Number of digital input modules
#define SYS_MAX_DIGOUT                51          // Number of digital output modules
#define SYS_MAX_ANIN                  52          // Number of analog input modules
#define SYS_MAX_ANOUT                 53          // Number of analog output modules
#define SYS_MAX_BRENCREF              54          // Number of bridges Enc-Ref
#define SYS_MAX_BRREFENC              55          // Number of bridges Ref-Enc
#define SYS_MAX_BUSINFO               56          // Number of businfos
#define SYS_MAX_REFBUS                57          // Number of Refbus
#define SYS_MAX_HALL                  58          // Number of Hall input modules
#define SYS_MAX_PWMGEN                59          // Number of PWM Generator modules
#define SYS_MAX_BUSMOD                60          // Number of bus modules
#define SYS_MAX_SHIFTREG              61          // Number of shift registers
#define SYS_CONTROLLER_FEATURES       62          // Delivers information about active features on this
controller
#define SYS_MAX_PDO_SDO_SIZE          63          // Returns the max size of a "simple serial" telegram on this
controller
#define SYS_MAX_XTLG_SIZE             64          // Returns the max size for X-Telegrams
#define SYS_BUILD_NUMBER              65          // Build number
#define SYS_MAX_CANBUS                66          // Number of physical CAN busses
#define SYS_MAX_MACHINES              67          // Number of Machine Modules
#define SYS_COMMOPT_PROTTYPE          68          // Protocol type running on bus option
#define SYS_HW_IDENTIFICATION         69          // Returns the hw identification of the PCB
#define SYS_HW_INDEX                  70          // Returns the HW index of the PCB
#define SYS_PLATFORM                  71          // Returns the platform number
#define SYS_RELEASE_TYPE              72          // Returns the release type
#define SYS_TG_VERSION                73          // Returns the version of the trajectory generator library
#define SYS_COMMOPT_INFO              74          // Info about bus option
#define SYS_UNIQUE_ID1                75          // SYS_UNIQUE_ID1 and SYS_UNIQUE_ID2 are together the unique
ID for the controller
#define SYS_UNIQUE_ID2                76          // SYS_UNIQUE_ID1 and SYS_UNIQUE_ID2 are together the unique
ID for the controller
#define SYS_MAC0_LOW                  77          // SYS_MAC0_LOW and SYS_MAC0_HIGH together build the first MAC
address of the device
#define SYS_MAC0_HIGH                 78          // SYS_MAC0_LOW and SYS_MAC0_HIGH together build the first MAC
address of the device

    // Values for SYS_HW_IDENTIFICATION:
    #define SYS_HW_IDENTIFICATION_PC_ENC_OEM1 1        // PC Encoder Inrterface OEM1
    #define SYS_HW_IDENTIFICATION_PC_ENC_OEM2 2        // PC Encoder Inrterface OEM1
    #define SYS_HW_IDENTIFICATION_MICROMACS6 0        // MicroMACS6
    #define SYS_HW_IDENTIFICATION_MICROMACS6_MODULE 1        // MicroMACS6 Module

//=============================================================================================================
// 0x2216 FPGA_REG - Read or write FPGA register
//=============================================================================================================
#define SDOINDEX_FPGA_REG                          0x2216
#define FPGA_REG_INDEX(parno)                      ((0x01000000 | (SDOINDEX_FPGA_REG<<8)) | ((long) (parno)))
#define FPGA_REG_SRCINDEX(parno)                   ((SDOINDEX_FPGA_REG<<8) | ((long) (parno)))
#define FPGA_REG(parno)                            Sysvar[FPGA_REG_INDEX(parno)]
#define FPGA_REG_MAX                               256

#define FPGAREG_FPGAFWVERS            1            //
#define FPGAREG_CLEAROVFL             2            //
#define FPGAREG_INCENC1POS            3            //
#define FPGAREG_INCENC2POS            4            //
#define FPGAREG_INCENC3POS            5            //
#define FPGAREG_INCENC4POS            6            //
```

```
#define FPGAREG_INCENC5POS              7           //
#define FPGAREG_INCENC6POS              8           //
#define FPGAREG_POSLATCH1               9           //
#define FPGAREG_POSLATCH2               10          //
#define FPGAREG_POSLATCH3               11          //
#define FPGAREG_POSLATCH4               12          //
#define FPGAREG_POSLATCH5               13          //
#define FPGAREG_POSLATCH6               14          //
#define FPGAREG_POSLATCH_FLAG           15          //
#define FPGAREG_ENC1QUADCNT             16          //
#define FPGAREG_ENC2QUADCNT             17          //
#define FPGAREG_ENC3QUADCNT             18          //
#define FPGAREG_ENC4QUADCNT             19          //
#define FPGAREG_ENC5QUADCNT             20          //
#define FPGAREG_ENC6QUADCNT             21          //
#define FPGAREG_U32CNT10                22          //
#define FPGAREG_U32CNT11                23          //
#define FPGAREG_U32CNT20                24          //
#define FPGAREG_U32CNT21                25          //
#define FPGAREG_U32CNT30                26          //
#define FPGAREG_U32CNT31                27          //
#define FPGAREG_U32CNT40                28          //
#define FPGAREG_U32CNT41                29          //
#define FPGAREG_SSI1POS0                30          //
#define FPGAREG_SSI1POS1                31          //
#define FPGAREG_SSI2POS0                32          //
#define FPGAREG_SSI2POS1                33          //
#define FPGAREG_SSI3POS0                34          //
#define FPGAREG_SSI3POS1                35          //
#define FPGAREG_SSIFLAG                 36          //
#define FPGAREG_DIN                     37          //
#define FPGAREG_DOUT                    38          //
#define FPGAREG_REFVEC0                 39          //
#define FPGAREG_REFVEC1                 40          //
#define FPGAREG_REFVEC2                 41          //
#define FPGAREG_REFVEC3                 42          //

//==========================================================================================
// 0x2217 FPGA_PARAM - Write custom specific parameters
//==========================================================================================
#define SDOINDEX_FPGA_PARAM                         0x2217
#define FPGA_PARAM_INDEX(parno)                     ((0x01000000 | (SDOINDEX_FPGA_PARAM<<8)) | ((long) (parno)))
#define FPGA_PARAM_SRCINDEX(parno)                  ((SDOINDEX_FPGA_PARAM<<8) | ((long) (parno)))
#define FPGA_PARAM(parno)                           Sysvar[FPGA_PARAM_INDEX(parno)]
#define FPGA_PARAM_MAX                              256

#define FPGAPAR_REF_ENC_BRIDGE1_CNFGREG 2            //
#define FPGAPAR_REF_ENC_BRIDGE2_CNFGREG 3            //
#define FPGAPAR_ENC_REF_BRIDGE_CNFGREG 4            //
#define FPGAPAR_ENC1_3_MODEREG          5           //
#define FPGAPAR_ABR_QDZ_POLARITY_REG    6           //
#define FPGAPAR_DGX1_3_GLITCHFLTREG     7           //
#define FPGAPAR_DGX4_6_GLITCHFLTREG     8           //
#define FPGAPAR_ROT_SENSE_REG           9           //
#define FPGAPAR_POSCNT_1_CNFGREG        10          //
#define FPGAPAR_POSCNT_2_CNFGREG        11          //
#define FPGAPAR_POSCNT_3_CNFGREG        12          //
#define FPGAPAR_POSCNT_4_CNFGREG        13          //
#define FPGAPAR_POSCNT_5_CNFGREG        14          //
#define FPGAPAR_POSCNT_6_CNFGREG        15          //
```

```
#define FPGAPAR_Reserve15              16          //
#define FPGAPAR_POSLATCH_1_CNFGREG     17          //
#define FPGAPAR_POSLATCH_2_CNFGREG     18          //
#define FPGAPAR_POSLATCH_3_CNFGREG     19          //
#define FPGAPAR_POSLATCH_4_CNFGREG     20          //
#define FPGAPAR_POSLATCH_5_CNFGREG     21          //
#define FPGAPAR_POSLATCH_6_CNFGREG     22          //
#define FPGAPAR_Reserve22              23          //
#define FPGAPAR_Reserve23              24          //
#define FPGAPAR_Reserve24              25          //
#define FPGAPAR_VM1_CNFGREG            26          //
#define FPGAPAR_VM1_FREQ0REG           27          //
#define FPGAPAR_VM1_FREQ1REG           28          //
#define FPGAPAR_VM1_ZPULSE_RATIO       29          //
#define FPGAPAR_Reserve29              30          //
#define FPGAPAR_VM2_CNFGREG            31          //
#define FPGAPAR_VM2_FREQ0REG           32          //
#define FPGAPAR_VM2_FREQ1REG           33          //
#define FPGAPAR_VM2_ZPULSE_RATIO       34          //
#define FPGAPAR_Reserve34              35          //
#define FPGAPAR_VM3_CNFGREG            36          //
#define FPGAPAR_VM3_FREQ0REG           37          //
#define FPGAPAR_VM3_FREQ1REG           38          //
#define FPGAPAR_VM3_ZPULSE_RATIO       39          //
#define FPGAPAR_Reserve39              40          //
#define FPGAPAR_U32CNT_MODEREG         41          //
#define FPGAPAR_U32CNT1_CNFGREG        42          //
#define FPGAPAR_U32CNT2_CNFGREG        43          //
#define FPGAPAR_U32CNT3_CNFGREG        44          //
#define FPGAPAR_U32CNT4_CNFGREG        45          //
#define FPGAPAR_Reserve45              46          //
#define FPGAPAR_Reserve46              47          //
#define FPGAPAR_Reserve47              48          //
#define FPGAPAR_Reserve48              49          //
#define FPGAPAR_CMP1_CNFG_REG          50          //
#define FPGAPAR_CMP1_UPPERLEV_LOWWORD  51          //
#define FPGAPAR_CMP1_UPPERLEV_HIGHWORD 52          //
#define FPGAPAR_CMP1_LOWERLEV_LOWWORD  53          //
#define FPGAPAR_CMP1_LOWERLEV_HIGHWORD 54          //
#define FPGAPAR_Reserve54              55          //
#define FPGAPAR_CMP2_CNFG_REG          56          //
#define FPGAPAR_CMP2_UPPERLEV_LOWWORD  57          //
#define FPGAPAR_CMP2_UPPERLEV_HIGHWORD 58          //
#define FPGAPAR_CMP2_LOWERLEV_LOWWORD  59          //
#define FPGAPAR_CMP2_LOWERLEV_HIGHWORD 60          //
#define FPGAPAR_Reserve60              61          //
#define FPGAPAR_CMP3_CNFG_REG          62          //
#define FPGAPAR_CMP3_UPPERLEV_LOWWORD  63          //
#define FPGAPAR_CMP3_UPPERLEV_HIGHWORD 64          //
#define FPGAPAR_CMP3_LOWERLEV_LOWWORD  65          //
#define FPGAPAR_CMP3_LOWERLEV_HIGHWORD 66          //
#define FPGAPAR_Reserve66              67          //
#define FPGAPAR_CMP4_CNFG_REG          68          //
#define FPGAPAR_CMP4_UPPERLEV_LOWWORD  69          //
#define FPGAPAR_CMP4_UPPERLEV_HIGHWORD 70          //
#define FPGAPAR_CMP4_LOWERLEV_LOWWORD  71          //
#define FPGAPAR_CMP4_LOWERLEV_HIGHWORD 72          //
#define FPGAPAR_Reserve72              73          //
#define FPGAPAR_SSI1CLKREG             74          //
#define FPGAPAR_SSI1DELMODEREG         75          //
```

```
#define FPGAPAR_SSI2CLKREG              76          //
#define FPGAPAR_SSI2DELMODEREG          77          //
#define FPGAPAR_SSI3CLKREG              78          //
#define FPGAPAR_SSI3DELMODEREG          79          //
#define FPGAPAR_Reserve79               80          //
#define FPGAPAR_DDS1INPUTSELREG         81          //
#define FPGAPAR_DDS1ZREG0               82          //
#define FPGAPAR_DDS1ZREG1               83          //
#define FPGAPAR_DDS1NREG0               84          //
#define FPGAPAR_DDS1NREG1               85          //
#define FPGAPAR_DDS1_Z_PULSE_RATIO      86          //
#define FPGAPAR_DDS2INPUTSELREG         87          //
#define FPGAPAR_DDS2ZREG0               88          //
#define FPGAPAR_DDS2ZREG1               89          //
#define FPGAPAR_DDS2NREG0               90          //
#define FPGAPAR_DDS2NREG1               91          //
#define FPGAPAR_DDS2_Z_PULSE_RATIO      92          //
#define FPGAPAR_Reserve92               93          //
#define FPGAPAR_PULSE_CNFGREG           94          //
#define FPGAPAR_PULSE_DELAYREG          95          //
#define FPGAPAR_PULSE_LENGTHREG         96          //
#define FPGAPAR_Reserve96               97          //
#define FPGAPAR_DIN_POLREG              98          //
#define FPGAPAR_DIN1_4_FILTCNFG         99          //
#define FPGAPAR_DIN5_8_FILTCNFG         100         //
#define FPGAPAR_DIN9_12_FILTCNFG        101         //
#define FPGAPAR_DIN13_15_FILTCNFG       102         //
#define FPGAPAR_DOUT9_16CNFGREG         103         //
#define FPGAPAR_DOUT1_2CNFGREG          104         //
#define FPGAPAR_DOUT3_4CNFGREG          105         //
#define FPGAPAR_RS232_CNFGREG           106         //
#define FPGAPAR_ENC4PORTMODEREG         107         //
#define FPGAPAR_ENC5PORTMODEREG         108         //
#define FPGAPAR_ENC6PORTMODEREG         109         //
#define FPGAPAR_Reserve109              110         //
#define FPGAPAR_Reserve110              111         //
#define FPGAPAR_Reserve111              112         //
#define FPGAPAR_REFVEC_0_15_F2DIRQMASK 113          //
#define FPGAPAR_REFVEC16_31_F2DIRQMASK 114          //
#define FPGAPAR_REFVEC32_47_F2DIRQMASK 115          //
#define FPGAPAR_REFVEC48_63_F2DIRQMASK 116          //
#define FPGAPAR_REFVEC_0_15_SOCIRQMASK 117          //
#define FPGAPAR_REFVEC16_31_SOCIRQMASK 118          //
#define FPGAPAR_REFVEC32_47_SOCIRQMASK 119          //
#define FPGAPAR_REFVEC48_63_SOCIRQMASK 120          //
#define FPGAPAR_REFVEC_0_15_MODEREG    121          //
#define FPGAPAR_REFVEC16_31_MODEREG    122          //
#define FPGAPAR_REFVEC32_47_MODEREG    123          //
#define FPGAPAR_REFVEC48_63_MODEREG    124          //
#define FPGAPAR_REFVEC_0_15_SLOPEREG   125          //
#define FPGAPAR_REFVEC16_31_SLOPEREG   126          //
#define FPGAPAR_REFVEC32_47_SLOPEREG   127          //
#define FPGAPAR_REFVEC48_63_SLOPEREG   128          //

//=============================================================================================
// 0x2218 FPGA_USRREG - Address for the page number
//=============================================================================================
#define SDOINDEX_FPGA_USRREG                        0x2218
#define FPGA_USRREG_INDEX(parno)                    ((0x01000000 | (SDOINDEX_FPGA_USRREG<<8)) | ((long) (parno)))
#define FPGA_USRREG_SRCINDEX(parno)                 ((SDOINDEX_FPGA_USRREG<<8) | ((long) (parno)))
```

```
#define FPGA_USRREG(parno)                          Sysvar[FPGA_USRREG_INDEX(parno)]
#define FPGA_USRREG_MAX                             2

//=============================================================================================================
// 0x2220 SDUPDATE - SD-Card Update Info Object
//=============================================================================================================
#define SDOINDEX_SDUPDATE                           0x2220
#define SDUPDATE_INDEX(parno)                       ((0x01000000 | (SDOINDEX_SDUPDATE<<8)) | ((long) (parno)))
#define SDUPDATE_SRCINDEX(parno)                    ((SDOINDEX_SDUPDATE<<8) | ((long) (parno)))
#define SDUPDATE(parno)                             Sysvar[SDUPDATE_INDEX(parno)]
#define SDUPDATE_MAX                                21

#define SDUPDATE_COMMAND_1              1           // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_2              2           // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_3              3           // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_4              4           // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_5              5           // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_6              6           // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_7              7           // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_8              8           // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_9              9           // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_10             10          // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_11             11          // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_12             12          // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_13             13          // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_14             14          // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_15             15          // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_COMMAND_16             16          // Detailed information about executed (completed or failed)
update commands
#define SDUPDATE_STATE_FLAGS            17          // SD-Card Update state flags
#define SDUPDATE_COMMANDS_SKIPPED       18          // Number of SD-Card Update commands skipped due to existing
report file entry
#define SDUPDATE_COMMANDS_COMPLETED     19          // Number of SD-Card Update commands executed successfully
#define SDUPDATE_COMMANDS_FAILED        20          // Number of SD-Card Update commands failed

//=============================================================================================================
// 0x2300 AXE_PARAM - Axis Parameters
//=============================================================================================================
#define SDOINDEX_AXE_PARAM                          0x2300
#define AXE_PARAM_INDEX(modno,parno)                ((0x01000000 | (SDOINDEX_AXE_PARAM<<8)) | (((long) (modno))<<8)
| ((long) (parno)))
#define AXE_PARAM_SRCINDEX(modno,parno)             ((SDOINDEX_AXE_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define AXE_PARAM(modno,parno)                      Sysvar[AXE_PARAM_INDEX((modno),(parno))]
#define AXE_PARAM_MAX                               141
```

```
#define VELMAX                  2           // Rated speed of drive
#define HOME_FORCE              4           // Forced movement to home position
#define NEGLIMIT                5           // Negative software limit position
#define POSLIMIT                6           // Positive software limit position
#define SYNCMMAXCORR            7           // Limits the maximum correction done by marker correction
#define HOME_VEL                8           // Velocity for movement to home position
#define TESTWIN                 9           // Size of the target window
#define VELMAXQC                10          // Maximum velocity
#define ACCMAXQC                11          // Maximum acceleration
#define KPROP                   12          // Proportional value for PID position control loop
#define KDER                    13          // Derivative Factor for PID position control loop
#define KINT                    14          // Integral value for PID position control loop
#define TIMER                   15          // Parameter for determining the sampling time of the position
control algorithm
#define POSERR                  16          // Maximum tolerated position error
#define SYNCOFFTIME             17          // Compensation velocity of an offset
#define SYNCMFPAR               18          // Marker filter configuration
#define SYNCMFTIME              19          // Filter time for marker correction
#define SWNEGLIMACT             20          // Negative software end limit active
#define SWPOSLIMACT             21          // Positive software end limit active
#define KILIM                   22          // Limit value for the integral sum of the PID position
control loop
#define VELRES                  23          // Velocity resolution
#define POSFACT_Z               24          // User factor numerator
#define TESTTIM                 25          // Measuring time in target window
#define TESTVAL                 26          // Limit value for readings in target window
#define POSFACT_N               27          // User factor denominator
#define POSDRCT                 29          // Rotational direction
#define PROFTIME                30          // Scan time for profile generator
#define MENCODER                31          // Resolution of master encoder
#define RAMPMIN                 32          // Maximum acceleration
#define RAMPTYPE                33          // Ramp type
#define DFLTVEL                 34          // Default velocity
#define DFLTACC                 35          // Default acceleration
#define BANDWIDTH               36          // Bandwidth within which the PID filter is active. 1000
equals to 100% velocity setpoint.
#define FFVEL                   37          // Velocity Feed forward
#define REGWMAX                 39          // Size of the control window (activation)
#define REGWMIN                 40          // Size of the control window (deactivation)
#define HOME_TYPE               41          // Behavior during movement to home
#define HOME_RAMP               42          // Ramp for home motion
#define HOME_OFFSET             43          // Zero point offset for machine zero point or index
#define ERRCOND                 44          // Behavior in event of an error
#define I_REFSWITCH             46          // Input reference switch
#define I_POSLIMITSW            47          // Positive limit switch
#define I_NEGLIMITSW            48          // Negative limit switch
#define O_BRAKE                 49          // Output for brake
#define SYNCFACTM               50          // Synchronization factor master (M:S) or in CAM mode
conversion qc in MU units
#define SYNCFACTS               51          // Synchronization factor slave (M:S) or in CAM mode
conversion qc in MU units
#define SYNCTYPE                52          // Type of synchronization
#define SYNCMARKM               53          // Number of master markers
#define SYNCMARKS               54          // Number of slave markers
#define SYNCPOSOFFS             55          // Position offset for position synchronization
#define SYNCACCURACY            56          // Size of the accuracy window for position synchronization
#define SYNCREADY               57          // Number of markers for READY
#define SYNCFAULT               58          // Number of markers for fault
#define SYNCMPULSM              59          // Master marker distance
```

```
#define SYNCMPULSS                    60          // Slave marker distance
#define SYNCMSTART                    63          // Start behavior for synchronization with marker correction
#define O_AXMOVE                      65          // Output for motion command active
#define SYNCVFTIME                    66          // Velocity filter
#define SYNCVELREL                    67          // Tolerated deviance of the slave drive from the master
velocity
#define SYNCMWINM                     69          // Tolerance window for marker monitoring
#define SYNCMWINS                     70          // Tolerance window for slave marker monitoring
#define ESCCOND                       71          // Condition on program termination
#define JERKMIN                       99          // Minimum time required before reaching the maximum
acceleration
#define JERKMIN2                      101         // Minimum time required to ramp the acceleration down from
maximum acceleration to 0
#define JERKMIN3                      102         // Minimum time required to ramp the deceleration up from 0 to
maximum deceleration
#define JERKMIN4                      103         // Minimum time required to ramp the deceleration down from
maximum deceleration to 0
#define KILIMTIME                     106         // Time used to increase or decrease the integral limit
#define POSERRTIME                    112         // Minimum time before position error produced
#define FEEDDIST                      113         // Feed distance
#define FEEDREV                       114         // Feed revolutions
#define POSENCQC                      115         // Encoder resolution quadcount
#define POSENCREV                     116         // Encoder resolution revolution count
#define HOMEZEROVEL                   117         // Scaled velocity for moving into encoder index as part of
the homing procedure
#define PISRC_SVIRTCOUNT              131         // Linked virtual counter module number for slave information
#define PISRC_MVIRTCOUNT              132         // Linked virtual counter module number for master information
#define PISRC_SVIRTLATCH              133         // Linked virtual latch module number for slave information
#define PISRC_MVIRTLATCH              134         // Linked virtual latch module number for master information
#define KFFACC                        135         // Acceleration Feed Forward
#define KFFDEC                        136         // Deceleration Feed Forward
#define I_POSLIMITSWACT               137         // Enables the monitoring of the positive limit switch
#define I_NEGLIMITSWACT               138         // Enables the monitoring of the negative limit switch
#define HOME_CURTHRESHOLD             139         // Current Threshold for homing mode
#define PIDOPTIONS                    140         // Position PID Special options

    // Values for RAMPTYPE:
    #define RAMPTYPE_TRAPEZ               0
    #define RAMPTYPE_JERKLIMITED          2

    // Values for PIDOPTIONS:
    #define PID_OPTINTSUMCLEAR_ARRIVAL     0x0001   // Integral sum is cleared on target arrival
    #define PID_OPTINTSUMCLEAR_ZEROTRACKERR 0x0002   // Integral sum is cleared once after arrival, when trackerr
is 0

//=================================================================================================================
// 0x2500 AXE_PROCESS - Axis Process Data
//=================================================================================================================
#define SDOINDEX_AXE_PROCESS                        0x2500
#define AXE_PROCESS_INDEX(modno,parno)              ((0x01000000 | (SDOINDEX_AXE_PROCESS<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define AXE_PROCESS_SRCINDEX(modno,parno)           ((SDOINDEX_AXE_PROCESS<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define AXE_PROCESS(modno,parno)                    Sysvar[AXE_PROCESS_INDEX((modno),(parno))]
#define AXE_PROCESS_MAX                             236

#define REG_ACTPOS                    1           // Actual position value
#define REG_COMPOS                    2           // Set-point position value
#define REG_ZERO                      3           // Last detected index position
#define REG_AVEL                      4           // Actual velocity
```

```
#define REG_MAVEL                    5          // Actual velocity of the master
#define REG_TRACKERR                 6          // Actual position tracking error
#define REG_VELSMPT                  9          // Number of samples used for calculation of speed values
#define REG_MAPOS                    10         // Actual master position value
#define REG_MIPOS                    11         // Last detected master index position
#define REG_IAVEL                    12         // Internal actual velocity
#define REG_IMAVEL                   13         // Internal actual master velocity
#define REG_REFERENCE                16         // Actual reference value generated by position control loop
#define REG_TIMER                    18         // Sample period used by the PID position control loop
#define REG_PROFTIMER                19         // Number of PID controller steps in each Profile Generator
step.
#define REG_REVERSE                  20         // Negative command values blocked
#define REG_REVDOUT                  21         // Analog command output
#define REG_SOVERFLOW                22         // Slave Position Overflow
#define REG_MOVERFLOW                23         // Master Position Overflow
#define REG_USERREFVEL               27         // User velocity reference value for AxisControl(AxisNo,
USERREFVEL) mode
#define REG_VAPOS                    28         //
#define REG_NORMTRACKERR             29         // Size of position tracking error in relation to the command
velocity
#define REG_SENCINDXF                30         //
#define REG_MENCINDXF                31         //
#define REG_CNTRLSOURCE              32         //
#define REG_ENCAVEL                  33         // Summarized velocity of master encoder
#define REG_ENCLEAP                  34         //
#define REG_MENCLEAP                 35         //
#define REG_CNTRLWORD                36         //
#define REG_PIDFLAG                  37         //
#define REG_SM_COMMAND               38         //
#define REG_SM_STATE                 39         //
#define REG_USERREFPOS               40         // User reference position value for AxisControl(USERREFPOS)
command
#define REG_SVIRTCOUNT               41         // Linked virtual counter number for slave information
#define REG_MVIRTCOUNT               42         // Linked virtual counter number for master information
#define REG_HOMINGSTATE              43         // State of the homing procedure. Result is the same as the
function AxisHomeStatus returns.
#define REG_SVIRTLATCH               44         // Linked virtual latch module number for slave information
#define REG_MVIRTLATCH               45         // Linked virtual latch module number for master information
#define REG_USERREFCUR               46         // Motor current reference value in AxisControl(USERREFCUR)
mode
#define PID_ACTINTSUM                49         // Total sum of position tracking errors
#define PID_INTLIMIT                 50         // Limitation of the summarized position tracking error
#define PID_100PERCENT               51         // PID position control loop output corresponding to a
reference value of 100%
#define PID_VELRESVAL                52         // Default scaling factor internal high-resolution velocity or
position information
#define PID_PIDPART                  53         // Part of the reference value due to the PID position control
loop
#define PID_FFPART                   54         // Part of the reference value due to the FFVEL feed forward
control
#define PID_FFACCPART                55         //
#define PID_ACT_INTLIMIT             56         // Actual integration limit
#define PID_FF_FACTOR                57         // Internal representation of the FFVEL factor used in PID
calculation
#define PID_FF_CMDVEL                58         // Internal representation of the velocity used for FFVEL
calculation
#define PID_MODE                     59         // Internal PID Mode
#define PFG_DIR                      82         // Direction of command velocity
#define PFG_AACC                     83         // Maximum allowed acceleration
#define PFG_ADEC                     84         // Maximum allowed deceleration
```

```
#define PFG_VFIN                    85          // Scaled target velocity
#define PFG_AACT                    86          // Command acceleration
#define PFG_AACC_DELTA              87          // Scaled acceleration deviation
#define PFG_ADELTCALC               88          // Scaled acceleration deviation
#define PFG_PATHCALC                89          // Length of the brake ramp
#define PFG_PCMD                    90          // Command position
#define PFG_VCMD                    91          // Command velocity
#define PFG_PCORR                   92          // Scaled correction value for the brake ramp
#define PFG_OLDMFILTVEL             93          // Scaled and filtered master velocity
#define PFG_OLDSFILTVEL             94          // Scaled and filtered slave velocity
#define PFG_MFILTERR                95          // Inaccuracy due to filtering of the master position value
#define PFG_MDIR                    96          // Master direction for synchronization
#define PFG_MACTDIR                 97          // Actual master direction
#define PFG_SACTDIR                 98          // Actual slave direction
#define PFG_MPCMD                   99          // Command position calculated by the synchronization
algorithm
#define PFG_MPCMDOLD                100         // Last synchronization command position
#define PFG_REVERSING               101         // Reversing active
#define PFG_MMARKCNT                102         // Required master marker pulse count
#define PFG_SMARKCNT                103         // Required slave marker pulse count
#define PFG_MOLDZERO                104         // Master position of the last detected master marker
#define PFG_SOLDZERO                105         // Slave position of the last detected slave marker
#define PFG_MZEROCMD                106         // Master marker positionused for marker correction
#define PFG_SZEROCMD                107         // Master slave position used for marker correction
#define PFG_MHIT                    108         // Master marker detection
#define PFG_SHIT                    109         // Slave marker detection
#define PFG_READYCNT                110         // Synchronous marker check count
#define PFG_FAULTCNT                111         // Asynchronous marker check count
#define PFG_MPCMDERR                112         // Current synchronization position error
#define PFG_MARKERDIFF              113         // Slave/master marker difference count
#define PFG_SYNCSTART               114         // Status of marker synchronization start-up
#define PFG_OLDSYPOFFS              115         // Old value of the parameter SYNCPOSOFFS
#define PFG_KORREKTUR               116         // Value of the last marker correction [qc (slave)]
#define PFG_KORRREST                117         // Remainder of the marker correction
#define PFG_KORRVAL                 118         // Value of the marker correction
#define PFG_MARKERFAKED             119         // Virtual marker created
#define PFG_INTOFFSET               120         // Current offset
#define PFG_SCALESHIFT              121         // Scaling factor for internal 64-bit values that are returned
in 32-bit variables
#define PFG_AKTSTATE                122         // Current state of profile generator state machine
#define PFG_FLAGS                   123         // Profile generator flags
#define PFG_STOPLEN                 124         // Brake ramp length for stopping using deceleration setting
#define PFG_CINDEX                  125         // Current curve interpolation index
#define PFG_CVINDEX                 126         // Start/stop curve data index
#define PFG_CMAXINDEX               127         // Maximum curve interpolation index
#define PFG_SYNCSTOPLEN             128         // Brake ramp length for stopping using acceleration setting
#define PFG_CWRAP                   130         // Current value of the curve counter
#define PFG_CSSTART                 131         // Offset used for the start of current slave curve
#define PFG_CCOUNTER                132         // Number of curves to be processed
#define PFG_CCURVEPOS               133         // Current curve slave position
#define PFG_CSLAVECPOSQ             134         // Current curve position of the slave related to CSSTART
#define PFG_CMASTERCPOS             135         // Current curve master position
#define PFG_GETCMDVEL               136         // Scaled set point velocity
#define PFG_MMFAKEDCNT              137         // Consecutive virtual generated master markers
#define PFG_SMFAKEDCNT              138         // Consecutive virtual generated slave markers
#define PFG_MMILLEGALCNT            139         // Number of non-fitting master markers
#define PFG_SMILLEGALCNT            140         // Number of non-fitting slave markers
#define PFG_SYNCVELDIFF             141         // Difference between the slave and master velocity
#define PFG_SYNCPATHLEN             142         // Path length of velocity difference compensation
#define PFG_MMARKERDIST             143         // Filtered distance between two master markers
```

```
#define PFG_SMARKERDIST          144     // Filtered distance between two slave markers
#define PFG_STARTKORR            145     // First correction value after the start of a SYNCM
#define PFG_STARTKORRREST        146     // Remainder of the start-up correction value
#define PFG_KORRFILT             147     // Filtered correction value
#define PFG_LASTMMDIST           148     // Last measured distance between two master markers
#define PFG_MMARKCORR            149     // Calculated gear ratio correction value
#define PFG_KORRUNFILT           150     // Last unfiltered correction value
#define PFG_MDISTMARK            151     // Distance between current master position and last master
marker
#define PFG_SDISTMARK            152     // Distance between current slave position and last slave
marker
#define PFG_STARTKORRVAL         153     // Start-up correction value at each marker correction
#define PFG_LASTSMDIST           154     // Last measured distance between two slave markers
#define PFG_MARKERFILTER         155     // Number of markers defining the filtering period
#define PFG_KORRTAU              156     // Tau used by PT marker filter
#define PFG_INTMMERROR           157     // Sum of master marker distance errors
#define PFG_MMARKERR             158     // Filtered sum master marker errors
#define PFG_ORGREALPOS           159     // "Real" origin position after a SETORIGIN command
#define PFG_DYNKORRLIMIT         160     // Calculated limit of marker correction
#define PFG_MMARKFIFOVALID       161     // "Valid" flag for actual master marker FIFO read address
#define PFG_MMARKFIFOREAD        162     // Read the master marker FIFO
#define PFG_LASTERROR            163     // Additional information about the last PFG error
#define PFG_STOUCHPOS            164     // Not used
#define PFG_MTOUCHPOS            165     //
#define PFG_MCENDPOS             166     // Master position where the last curve ended
#define PFG_SMARKFIFOVALID       167     // "Valid" flag for actual slave marker FIFO read address
#define PFG_SMARKFIFOREAD        168     // Read the slave marker FIFO
#define PFG_CMARKPOS             173     // Reserved
#define PFG_MMARKFIFOTEST        174     // Read the master marker FIFO
#define PFG_SMARKFIFOTEST        175     // Read the slave marker FIFO
#define PFG_RESTOFFSDBL          176     // Reserved
#define PFG_OFFTOTALDIST         177     // Reserved
#define PFG_OFFDISTDONE          178     // Reserved
#define PFG_LASTMOVSYNC          179     // Reserved
#define PFG_JSTATE               180     // State of the Limited Jerk Movement state machine
#define PFG_VCMDSIGNED           181     // Scaled SIGNED set point velocity
#define PFG_JERKSTOPPATH         182     // Length of brake ramp if stopped
#define PFG_LASTMMDEVIATION      183     // Last master marker deviation
#define PFG_INTSMERROR           184     // Sum of slave marker distance errors
#define PFG_SMARKERR             185     // Filtered sum of slave marker errors
#define PFG_LASTSMDEVIATION      186     // Last slave marker deviation
#define PFG_MMDEVPOSDIFF         187     // Distance between last two master markers
#define PFG_SMDEVPOSDIFF         188     // Distance between last two slave markers
#define PFG_JERKFINVEL           189     // Velocity reached if the current acceleration is reduced to
0
#define PFG_JERKSTOPDIST         190     // Length of brake ramp if stopped with specified deceleration
#define PFG_LASTREALMZERO        191     // Position value of the last "real" master marker
#define PFG_LASTREALSZERO        192     // Position value of the last "real" slave marker
#define PFG_CPOLYMAXVEL          193     // Maximum velocity of a curve polynomial
#define PFG_CPOLYMINVEL          194     // Minimum velocity of a curve polynomial
#define PFG_SMARKERFILTER        195     // Slave filter value
#define PFG_SYNCDIFFPOSL         196     // Calculated position difference
#define PFG_CALCMCURVPOS         197     // Calculate the curve master position corresponding to a
slave position
#define PFG_RESTOFFSET           198     // Offset which must still be handled
#define PFG_OFFSETVALUE          199     // Actual offset step value used to distribute offsets or
corrections
#define PFG_MASTERTARGET         200     // Corrected slave position
#define PFG_REVERSEOFFSET        201     // Compensation offset for blocked reversal
#define PFG_MSTARTCURVESTATE     202     // State of the SYNCMSTART = 7 curve handling
```

```
#define PFG_MSTARTCURVESTARTMPCMD      203          // Master position where curve was started
#define PFG_MSTARTCURVEMPOSSL          204          // Master curvepos of start curve
#define PFG_MSTARTCURVEMENDPOSSL       205          // Master curve endpos of start curve
#define PFG_MSTARTCURVECPOSSL          206          // Slave curve position of start curve
#define PFG_MSTARTCURVEDELTAS          207          // Calculated slave distance
#define PFG_MSTARTCURVESMARKTODO       208          // Actual distance of the slave position to the next slave
marker
#define PFG_MSTARTCURVEDELTASREST      209          // Remainder of MSTARTCURVEDELTAS divided by slave marker
distance
#define PFG_MSTARTCURVEERR             210          // Difference between MSTARTCURVEDELTASREST and the rest of
the master distance
#define PFG_MSTARTMARKERTODO           211          // Master markers necessary
#define PFG_SYNCAACT                   212          // Actual acceleration value used for synchronization
#define PFG_ACTSYNCVFTIME              213          // Actual syncvftime active
#define PFG_FILTERCOUNT                214          // Internal relative count since PFG_MFILTERR was reset last
time
#define PFG_MASTERVELSQSL              215          // Master velocity in slave qc per sample time
#define PFG_COMMANDPOS                 216          // SollPos returned to sw_reg
#define PFG_OLDCOMMANDPOS              217          // OldSollPos returned to sw_reg
#define PFG_CMASTERCREST               218          // Remaining master distance
#define PFG_CSLAVECREST                219          // Remaining slave distance
#define PFG_SYNCMNEGBUFFER             220          // Internal buffer for suppressed negative master movement in
synchronisation
#define PFG_CMASTERCLEN                221          // Master curve length of the actual curve running
#define PFG_CALCCURVEPOS               222          // Calculate the slave position corresponding to a curve
master position
#define PFG_VCMDSIGNED_USERSCALED      223          // Used for virtual master working with VCMD
#define PFG_PROFTIMER                  224          // PROFTIMER value from the axis. Used for internal
calculation of PFG_VCMDSIGNED_USERSCALED
#define PFG_CCALCULATE                 225          //
#define PFG_OFFSET_TIME                226          //
#define PFG_OFFSET_VFIN                227          //
#define PFG_OFFSET_ACC                 228          //
#define PFG_OFFSET_CORR                229          //
#define PFG_FINPOS                     230          // Target position of the profile generator
#define PFG_COMPOSDIFF                 231          // Difference of Command position and last command position
#define PFG_CSTARTCLEN                 232          // Start curve length for the active PATHSTART command
#define PFG_ACMD                       233          // Signed command acceleration
#define PFG_USERCMDPOS                 234          // User command position
#define PFG_RUNTIME_ERR                235          // Returns last active runtime error

    // Values for PFG_AKTSTATE:
    #define PGS_POSCTRL                    0        // Position control, no calculations
    #define PGS_TPZPREDEC                  1        // Deceleration before start of acceleration
    #define PGS_TPZACC                     2        // Trapezoidal acceleration
    #define PGS_TPZDEC                     3        // Trapezoidal deceleration
    #define PGS_TPZVEL                     4        // Trapezoidal constant velocity
    #define PGS_CVEL                       5        // Continuous velocity
    #define PGS_CACC                       6        // Continuous acceleration
    #define PGS_CPREDEC                    7        // Continuous deceleration before acceleration
    #define PGS_CDEC                       8        // Continuous deceleration
    #define PGS_VSYNC                      12       // Velocity synchronization
    #define PGS_PSYNC                      13       // Position synchronization
    #define PGS_PSYNCM                     14       // Position synchronization with marker
    #define PGS_FLOAT                      15       // Floating, no regulation
    #define PGS_POSFLOAT                   16       // Not used anymore
    #define PGS_CSYIDLE                    17       // Idle while in curve sync mode
    #define PGS_CSYSTARTING                18       // Start procedure for curve sync
    #define PGS_CSYSTOPPING                19       // Stop procedure for curve sync
    #define PGS_CSYRUNNING                 20       // Curve synchronization is running
```

```
    #define PGS_LMTJERK                     21      // Running in limited jerk mode (for detail look at
PFG_G_JSTATE)
    #define PGS_FLOATUSER                   22      // Floating, command position is taken from user
    #define PGS_PATHMOVE                    23      // Path synchronization: Following PATHMOVE curve
    #define PGS_PATHSTART                   24      // Path synchronization: Following start path
    #define PGS_PATHSYNC                    25      // Path synchronization: Following main path
    #define PGS_PATHWAIT                    26      // Path synchronization: Stopped and waiting to continue
    #define PGS_PATHIDLE                    27      // Path synchronization: Stopped and idle

//==========================================================================================================
// 0x2600 MACHINE_PARAM -
//==========================================================================================================
#define SDOINDEX_MACHINE_PARAM                      0x2600
#define MACHINE_PARAM_INDEX(modno,parno)            ((0x01000000 | (SDOINDEX_MACHINE_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define MACHINE_PARAM_SRCINDEX(modno,parno)         ((SDOINDEX_MACHINE_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define MACHINE_PARAM(modno,parno)                  Sysvar[MACHINE_PARAM_INDEX((modno),(parno))]
#define MACHINE_PARAM_MAX                           31

#define MACHINE_PARAM_MAX_SUBINDEX     0            //
#define MACHINE_MODE                   1            //
#define MACHINE_PISRC_VIRTCOUNT        10           //
#define MACHINE_PISRC_VIRTLATCH        11           //
#define KIN_SYNC_TYPE                  21           // The type of the external movement the robot movement is
synchronized to. 1: translational movement 2: rotational movement
#define KIN_SYNC_STARTFLAG             22           // The KIN_SYNC_STARTFLAG tells the TG that the next path that
is executed should be synchronized.
#define KIN_SYNC_STOPFLAG              23           // The KIN_SYNC_STOPFLAG tells the TG that the synchronization
should stop with the stop of the next path.
#define KIN_SYNC_VECX                  24           // X component of the translation or rotation vector
#define KIN_SYNC_VECY                  25           // Y component of the translation or rotation vector
#define KIN_SYNC_VECZ                  26           // Z component of the translation or rotation vector
#define KIN_SYNC_ROTCX                 27           // X component of the rotation center
#define KIN_SYNC_ROTCY                 28           // Y component of the rotation center
#define KIN_SYNC_ROTCZ                 29           // Z component of the rotation center
#define KIN_SYNC_FPSCALE               30           // The quadcounts of the input movments are divided by this
factor to allow for floating point scaling when converting to machine units.

    // Values for MACHINE_MODE:
    #define MACHINE_MODE_DISABLED           0
    #define MACHINE_MODE_ENABLED            1

//==========================================================================================================
// 0x2640 MACHINE_PROCESS -
//==========================================================================================================
#define SDOINDEX_MACHINE_PROCESS                    0x2640
#define MACHINE_PROCESS_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_MACHINE_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define MACHINE_PROCESS_SRCINDEX(modno,parno)       ((SDOINDEX_MACHINE_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define MACHINE_PROCESS(modno,parno)                Sysvar[MACHINE_PROCESS_INDEX((modno),(parno))]
#define MACHINE_PROCESS_MAX                         11

#define MACHINE_PROCESS_MAX_SUBINDEX   0            //
#define MACHINE_ERROR_FLAG             1            //
#define KIN_SYNC_ACTIVE                3            // Indicates if the synchronization to an external movement is
currently active
#define MACHINE_ACTPOS                 4            //
#define MACHINE_ACTVEL                 5            //
```

```
#define MACHINE_CPOS_MACHCOORD_X       6            //
#define MACHINE_CPOS_MACHCOORD_Y       7            //
#define MACHINE_CPOS_MACHCOORD_Z       8            //
#define MACHINE_ACT_PATH_DISTANCE      9            // Actual path distance (absolute)
#define MACHINE_ACT_PATH_VELOCITY      10           // Actual path velocity (absolute)

//==========================================================================================
// 0x2800 VIRTMAST_PARAM - Virtual Master Parameters
//==========================================================================================
#define SDOINDEX_VIRTMAST_PARAM                     0x2800
#define VIRTMAST_PARAM_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_VIRTMAST_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTMAST_PARAM_SRCINDEX(modno,parno)        ((SDOINDEX_VIRTMAST_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define VIRTMAST_PARAM(modno,parno)                 Sysvar[VIRTMAST_PARAM_INDEX((modno),(parno))]
#define VIRTMAST_PARAM_MAX                          36

#define VIRTMAST_MODE                  1            // Enable / Disable / mode
#define VIRTMAST_PISRC_CMDWORD         10           // Link to the command word of an axis
#define VIRTMAST_PISRC_CMDVEL          11           // Index and subindex of the command velocity
#define VIRTMAST_VEL                   30           // Target velocity
#define VIRTMAST_ACC                   31           // Acceleration to reach target velocity
#define VIRTMAST_DEC                   32           // Deceleration to reach target velocity
#define VIRTMAST_UUFACT_UNITNO         34           // Conversion factor uu to 1/1000 Hz, number of user units.
#define VIRTMAST_UUFACT_INCNO          35           // Conversion factor uu to 1/1000 Hz, number of increments.

    // Values for VIRTMAST_MODE:
    #define VIRTMAST_MODE_DISABLED         0        // Disabled, output velocity is zero.
    #define VIRTMAST_MODE_VELOCITY         1        // Velocity mode, PO_VIRTMAST_VEL is taken directly from
VIRTMAST_PISRC_CMDVEL, and scaled using VIRTMAST_UUFACT_INCNO and VIRTMAST_UUFACT_UNITNO.
    #define VIRTMAST_MODE_PROFILE          3        // Velocity profile mode, PO_VIRTMAST_VEL is generated using
VIRTMAST_VEL/ACC/DEC.

//==========================================================================================
// 0x2840 VIRTCOUNTIN_PARAM - Virtual Counter Inputs parameters
//==========================================================================================
#define SDOINDEX_VIRTCOUNTIN_PARAM                  0x2840
#define VIRTCOUNTIN_PARAM_INDEX(modno,parno)        ((0x01000000 | (SDOINDEX_VIRTCOUNTIN_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTCOUNTIN_PARAM_SRCINDEX(modno,parno)     ((SDOINDEX_VIRTCOUNTIN_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTCOUNTIN_PARAM(modno,parno)              Sysvar[VIRTCOUNTIN_PARAM_INDEX((modno),(parno))]
#define VIRTCOUNTIN_PARAM_MAX                       37

#define VIRTCNTIN_MODE                 1            // Counter mode
#define VIRTCNTIN_SIMULATION_SPEED     4            // defines the speed when counter is in simulation mode
#define VIRTCNTIN_INHIBIT              5            // Module inhibit
#define VIRTCNTIN_PISRC_COUNTER        10           // Index/subindex of counter source
#define VIRTCNTIN_MAXSTEP              30           // Maximum stepsize allowed
#define VIRTCNTIN_OVFL_VALUE           32           // Value (Range) where the input will overflow
#define VIRTCNTIN_OVFL_FORCE           33           // Value where counter should produce an overflow
#define VIRTCNTIN_UUFACT_UNITNO        34           // Conversion Factor - number of user units
#define VIRTCNTIN_UUFACT_INCNO         35           // Conversion Factor - number of increments
#define VIRTCNTIN_MOVING_AVERAGE_N     36           //

    // Values for VIRTCNTIN_MODE:
    #define VIRTCNTIN_MODE_ABSOLUTE        0        // Source is a position value and difference to last value is
added (incremental encoders)
    #define VIRTCNTIN_MODE_DELTA           1        // Source is a velocity value and it is added as is
    #define VIRTCNTIN_MODE_SIMULATION_SLAVE 2        // Source is taken as it is - e.g. COMPOS from an axis
```

```
    #define VIRTCNTIN_MODE_ABSOLUTE_DIRECT 3        // Source is absolute and is directly taken as output position
(absolute encoders)
    #define VIRTCNTIN_MODE_ABSOLUTE_16BIT  4        // Source is 16bit position counter, difference to last value
is added to current position
    #define VIRTCNTIN_MODE_ABSOLUTE_DIRECT_ENDLESS 5        // Source is absolute. Output position will not
overflow at encoder overflow (absolute encoders).

//=============================================================================================================
// 0x2880 VIRTLATCH_PARAM - Virtual Latch Inputs Parameters
//=============================================================================================================
#define SDOINDEX_VIRTLATCH_PARAM                    0x2880
#define VIRTLATCH_PARAM_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_VIRTLATCH_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTLATCH_PARAM_SRCINDEX(modno,parno)       ((SDOINDEX_VIRTLATCH_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTLATCH_PARAM(modno,parno)                Sysvar[VIRTLATCH_PARAM_INDEX((modno),(parno))]
#define VIRTLATCH_PARAM_MAX                         37

#define VIRTLATCH_MODE                 1            // Latch mode
#define VIRTLATCH_SIM_ZDIST            4            // Distance of latch values in simulator mode
#define VIRTLATCH_LATCHVALID_BITMASK   5            // Latchvalid Bitmask
#define VIRTLATCH_PISRC_COUNTER        10           // Index/subindex of counter source (attached)
#define VIRTLATCH_PISRC_LATCHCNT       11           // Index/subindex of the counter at which the latch module is
looking
#define VIRTLATCH_PISRC_LATCH          12           // Index/subindex of latch source
#define VIRTLATCH_PISRC_LATCHSTAT      13           // Index/subindex of latch status source
#define VIRTLATCH_PISRC_LATCHVALID     14           // Index / Subindex of latch valid information
#define VIRTLATCH_PISRC_LATCHFIFO_AMOUNT 15           // Index/subindex of LatchFifo information (amount)
#define VIRTLATCH_PISRC_LATCHFIFO_READ 16           // Index/subindex of Latchfifo informtaion (read register)
#define VIRTLATCH_CNTW_CLEAR           30           // Control Word Clear Command
#define VIRTLATCH_CNTW_ACTIVE          31           // Control Word Active Command
#define VIRTLATCH_STAT_HIT             33           // Value which signals that latch happened
#define VIRTLATCH_OFFSET               34           // Offset which is added to latch value
#define VIRTLATCH_UUFACT_UNITNO        35           // Conversion Factor - number of user units
#define VIRTLATCH_UUFACT_INCNO         36           // Conversion Factor - number of increments

    // Values for VIRTLATCH_MODE:
    #define VIRTLATCH_MODE_HARDWARE        0        // Standard mode with HW Latch modules.
    #define VIRTLATCH_MODE_SOFTWARE        1        // Latching is done in software (latency up to 1ms), using
VIRTLATCH_PISRC_LATCHVALID and VIRTLATCH_LATCHVALID_BITMASK.
    #define VIRTLATCH_MODE_SIMULATOR       2        // Used for simulated latches. The value from
VIRTLATCH_PISRC_COUNTER will be compared to the current PO_VIRTLATCH_VALUE. If the distance is bigger than
VIRTLATCH_SIM_ZDIST, a new latch will be generated at PO_VIRTLATCH_VALUE + VIRTLATCH_SIM_ZDIST.
    #define VIRTLATCH_MODE_16BIT           3        // See Mode 0, but source is a 16bit value.
    #define VIRTLATCH_MODE_DIRECT          4        // If the value from VIRTLATCH_PISRC_LATCHVALID ANDed with
VIRTLATCH_VALID_BITMASK result != 0, the value from VIRTLATCH_PISRC_COUNTER will be written directly in
PO_VIRTLATCH_VALUE. Can be used for bus drives.

//=============================================================================================================
// 0x28C0 VIRTAMP_PARAM - Virtual Amplifier Parameters
//=============================================================================================================
#define SDOINDEX_VIRTAMP_PARAM                      0x28C0
#define VIRTAMP_PARAM_INDEX(modno,parno)            ((0x01000000 | (SDOINDEX_VIRTAMP_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define VIRTAMP_PARAM_SRCINDEX(modno,parno)         ((SDOINDEX_VIRTAMP_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define VIRTAMP_PARAM(modno,parno)                  Sysvar[VIRTAMP_PARAM_INDEX((modno),(parno))]
#define VIRTAMP_PARAM_MAX                           54

#define VIRTAMP_MODE                   1            // Mode of operation
```

```
#define VIRTAMP_PISRC_CMDWORD          10          // Index/subindex of command word
#define VIRTAMP_PISRC_REFPOS           11          // Index/subindex of position reference
#define VIRTAMP_PISRC_REFVEL           12          // Index/subindex of velocity reference
#define VIRTAMP_PISRC_REFACC           13          // Index/subindex of acceleration/current reference
#define VIRTAMP_PISRC_CURRENT          20          // Index/subindex of current
#define VIRTAMP_PISRC_STATUS           21          // Index/subindex of status
#define VIRTAMP_DRIVETYPE              30          // Predefined drivetype
#define VIRTAMP_CNTRLW_PWROFF          31          // Controlword to use in power off state
#define VIRTAMP_CNTRLW_PWRONDIS        32          // Controlword to use in power on disable state
#define VIRTAMP_CNTRLW_PWRONENP        33          // Controlword to use in power on enable positive direction
state
#define VIRTAMP_CNTRLW_PWRONENN        34          // Controlword to use in power on enable negative direction
state (could be same as above)
#define VIRTAMP_CNTRLW_QUICKSTOP       35          // Controlword to stop drive with internal ramps
#define VIRTAMP_CNTRLW_RESET           36          // Controlword to reset from error
#define VIRTAMP_REFTYPE                37          // Type of reference
#define VIRTAMP_REFOUTP                38          // Digital output for positive direction signal
#define VIRTAMP_REFOUTN                39          // Number of output which must be set if reference is negative
(reset if positive)
#define VIRTAMP_REF100PERC             40          // Output reference value to use at 100 percent
#define VIRTAMP_REFLIMIT               41          // Output reference limit
#define VIRTAMP_REFOFFSET              42          // Offset to add to output reference
#define VIRTAMP_INVERT                 43          // Output reference reversion
#define VIRTAMP_I2TTIME                44          // I2T Time interval for filtering
#define VIRTAMP_STOPDELAY              45          // Delay after a STOP command
#define VIRTAMP_REVERSE                46          // Reversing allowed
#define VIRTAMP_ERROR_BITMASK          47          // Bitmask to detect error condition
#define VIRTAMP_ERROR_POLARITY         48          // Polarity of Error Bit
#define VIRTAMP_I2TLIMIT               49          // I2T Limit value
#define VIRTAMP_REFACC100PERC          50          // Acceleration reference value (PO_VIRTAMP_REFACC) at full
acceleration
#define VIRTAMP_READY_BITMASK          51          // Bitmask to detect drive ready condition
#define VIRTAMP_READY_POLARITY         52          // Polarity of Ready Bit
#define VIRTAMP_TORQUE_CONST           53          // Represents the motor’s torque constant

    // Values for VIRTAMP_MODE:
    #define VIRTAMP_MODE_DISABLE           0       // Disabled
    #define VIRTAMP_MODE_ENABLE            1       // Enabled

//==========================================================================================================
// 0x2900 VIRTDIGIN_PARAM - Virtual Digital Input Parameters
//==========================================================================================================
#define SDOINDEX_VIRTDIGIN_PARAM                   0x2900
#define VIRTDIGIN_PARAM_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_VIRTDIGIN_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTDIGIN_PARAM_SRCINDEX(modno,parno)      ((SDOINDEX_VIRTDIGIN_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTDIGIN_PARAM(modno,parno)               Sysvar[VIRTDIGIN_PARAM_INDEX((modno),(parno))]
#define VIRTDIGIN_PARAM_MAX                        14

#define VIRTDIGIN_PISRC_MAP1           10          // First Mapping entry for outputs (like CanOpen)
#define VIRTDIGIN_PISRC_MAP2           11          // Second Mapping entry for outputs
#define VIRTDIGIN_PISRC_MAP3           12          // Third Mapping entry for outputs
#define VIRTDIGIN_PISRC_MAP4           13          // Fourth Mapping entry for outputs

//==========================================================================================================
// 0x2940 VIRTDIGOUT_PARAM - Virtual Digital Output Parameters
//==========================================================================================================
#define SDOINDEX_VIRTDIGOUT_PARAM                  0x2940
#define VIRTDIGOUT_PARAM_INDEX(modno,parno)        ((0x01000000 | (SDOINDEX_VIRTDIGOUT_PARAM<<8)) | (((long)
```

```
(modno))<<8) | ((long) (parno)))
#define VIRTDIGOUT_PARAM_SRCINDEX(modno,parno)      ((SDOINDEX_VIRTDIGOUT_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTDIGOUT_PARAM(modno,parno)               Sysvar[VIRTDIGOUT_PARAM_INDEX((modno),(parno))]
#define VIRTDIGOUT_PARAM_MAX                        2

//=============================================================================================================
// 0x2980 VIRTANIN_PARAM - Virtual Analog Input Parameters
//=============================================================================================================
#define SDOINDEX_VIRTANIN_PARAM                     0x2980
#define VIRTANIN_PARAM_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_VIRTANIN_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTANIN_PARAM_SRCINDEX(modno,parno)        ((SDOINDEX_VIRTANIN_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define VIRTANIN_PARAM(modno,parno)                 Sysvar[VIRTANIN_PARAM_INDEX((modno),(parno))]
#define VIRTANIN_PARAM_MAX                          33

#define VIRTANIN_PISRC_VALUE           10           // Index/subindex of source value
#define VIRTANIN_UUFACT_UNITNO         30           // Scale factor
#define VIRTANIN_UUFACT_DIGNO          31           // Scale factor
#define VIRTANIN_DIG_OFFSET            32           // Offset value

//=============================================================================================================
// 0x29C0 VIRTANOUT_PARAM - Virtual Analog Output Parameters
//=============================================================================================================
#define SDOINDEX_VIRTANOUT_PARAM                    0x29C0
#define VIRTANOUT_PARAM_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_VIRTANOUT_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTANOUT_PARAM_SRCINDEX(modno,parno)       ((SDOINDEX_VIRTANOUT_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTANOUT_PARAM(modno,parno)                Sysvar[VIRTANOUT_PARAM_INDEX((modno),(parno))]
#define VIRTANOUT_PARAM_MAX                         2

//=============================================================================================================
// 0x2A00 VIRTCUSTOM_PARAM - Virtual Customer Module Parameters
//=============================================================================================================
#define SDOINDEX_VIRTCUSTOM_PARAM                   0x2A00
#define VIRTCUSTOM_PARAM_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_VIRTCUSTOM_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTCUSTOM_PARAM_SRCINDEX(modno,parno)      ((SDOINDEX_VIRTCUSTOM_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTCUSTOM_PARAM(modno,parno)               Sysvar[VIRTCUSTOM_PARAM_INDEX((modno),(parno))]
#define VIRTCUSTOM_PARAM_MAX                        20

#define VIRTCUSTOM_PAR_1               1            // Custom parameters
#define VIRTCUSTOM_PAR_2               2            // Custom parameters
#define VIRTCUSTOM_PAR_3               3            // Custom parameters
#define VIRTCUSTOM_PAR_4               4            // Custom parameters
#define VIRTCUSTOM_PAR_5               5            // Custom parameters
#define VIRTCUSTOM_PAR_6               6            // Custom parameters
#define VIRTCUSTOM_PAR_7               7            // Custom parameters
#define VIRTCUSTOM_PAR_8               8            // Custom parameters
#define VIRTCUSTOM_PAR_9               9            // Custom parameters
#define VIRTCUSTOM_PISRC_1             10           // Custom sources
#define VIRTCUSTOM_PISRC_2             11           // Custom sources
#define VIRTCUSTOM_PISRC_3             12           // Custom sources
#define VIRTCUSTOM_PISRC_4             13           // Custom sources
#define VIRTCUSTOM_PISRC_5             14           // Custom sources
#define VIRTCUSTOM_PISRC_6             15           // Custom sources
#define VIRTCUSTOM_PISRC_7             16           // Custom sources
```

```
#define VIRTCUSTOM_PISRC_8              17          // Custom sources
#define VIRTCUSTOM_PISRC_9              18          // Custom sources
#define VIRTCUSTOM_PISRC_10             19          // Custom sources

//=========================================================================================================
// 0x2A40 VIRTMATH_PARAM - Virtual Math Module Parameters
//=========================================================================================================
#define SDOINDEX_VIRTMATH_PARAM                     0x2A40
#define VIRTMATH_PARAM_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_VIRTMATH_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTMATH_PARAM_SRCINDEX(modno,parno)        ((SDOINDEX_VIRTMATH_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define VIRTMATH_PARAM(modno,parno)                 Sysvar[VIRTMATH_PARAM_INDEX((modno),(parno))]
#define VIRTMATH_PARAM_MAX                          34

#define VIRTMATH_MODE                   1           // Mode of the module
#define VIRTMATH_INVERT                 2           // Inverts inputs and outputs
#define VIRTMATH_LOW_LIMIT              3           // Output low limit
#define VIRTMATH_HIGH_LIMIT             4           // Output high limit
#define VIRTMATH_PISRC_1                10          // Sources 1-4
#define VIRTMATH_PISRC_2                11          // Sources 1-4
#define VIRTMATH_PISRC_3                12          // Sources 1-4
#define VIRTMATH_PISRC_4                13          // Sources 1-4
#define VIRTMATH_POSRC_1                20          // Output source 1-2
#define VIRTMATH_POSRC_2                21          // Output source 1-2
#define VIRTMATH_MODE_BASED_1           30          // Mode based configuration 1 - 4
#define VIRTMATH_MODE_BASED_2           31          // Mode based configuration 1 - 4
#define VIRTMATH_MODE_BASED_3           32          // Mode based configuration 1 - 4
#define VIRTMATH_MODE_BASED_4           33          // Mode based configuration 1 - 4

    // Values for VIRTMATH_MODE:
    #define VIRTMATH_MODE_OFF               0       // Module Disabled
    #define VIRTMATH_MODE_ADD               1       // Output = Input1 + Input2 + Input3 + Input4
    #define VIRTMATH_MODE_GAIN              2       // Output = (Input1 / Input2) * (Input3 / Input4)
    #define VIRTMATH_MODE_SINW              10      // Output = Input1 / Input2 * Sin(2*PI*t/Input3) + Input4
    #define VIRTMATH_MODE_COSW              11      // Output = Input1 / Input2 * Cos(2*PI*t/Input3) + Input4
    #define VIRTMATH_MODE_SIN               12      // Output = Input1 / Input2 * Sin(Input3/1000) + Input4
    #define VIRTMATH_MODE_COS               13      // Output = Input1 / Input2 * Cos(Input3/1000) + Input4
    #define VIRTMATH_MODE_LP1_FILTER        20      // Output =   (Input1 - Output) / Input2 + Output
    #define VIRTMATH_MODE_PID               30      // Output = Pid(): Setpoint = Input1, measurement = Input2,
Configuration = VIRTMATH_MODE_BASED1-4

    // Values for VIRTMATH_MODE_BASED:
    #define VIRTMATH_MODE_PID_KP            30      // Mode VIRTMATH_MODE_PID: Controller gain proportional
    #define VIRTMATH_MODE_PID_KI            31      // Mode VIRTMATH_MODE_PID: Controller gain integral
    #define VIRTMATH_MODE_PID_KD            32      // Mode VIRTMATH_MODE_PID: Controller gain differential
    #define VIRTMATH_MODE_PID_KILIM         33      // Mode VIRTMATH_MODE_PID: Integral limit

//=========================================================================================================
// 0x2C00 VIRTMAST_PROCESS - Virtual Master Process Data
//=========================================================================================================
#define SDOINDEX_VIRTMAST_PROCESS                   0x2C00
#define VIRTMAST_PROCESS_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_VIRTMAST_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTMAST_PROCESS_SRCINDEX(modno,parno)      ((SDOINDEX_VIRTMAST_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTMAST_PROCESS(modno,parno)               Sysvar[VIRTMAST_PROCESS_INDEX((modno),(parno))]
#define VIRTMAST_PROCESS_MAX                        7

#define PO_VIRTMAST_VEL                 1           // Output velocity in increments per line
```

```
#define PO_VIRTMAST_POS                2            // Cumulated position value of the master
#define PO_VIRTMAST_TVEL_INC           3            // scaled internal value
#define PO_VIRTMAST_ACC_INC            4            // scaled internal value
#define PO_VIRTMAST_DEC_INC            5            // scaled internal value
#define PO_VIRTMAST_POS16              6            // Cumulated position value of the master in qc

//=============================================================================================
// 0x2C40 VIRTCOUNTIN_PROCESS - Virtual Counter Inputs Process Data
//=============================================================================================
#define SDOINDEX_VIRTCOUNTIN_PROCESS                0x2C40
#define VIRTCOUNTIN_PROCESS_INDEX(modno,parno)      ((0x01000000 | (SDOINDEX_VIRTCOUNTIN_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTCOUNTIN_PROCESS_SRCINDEX(modno,parno)   ((SDOINDEX_VIRTCOUNTIN_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTCOUNTIN_PROCESS(modno,parno)            Sysvar[VIRTCOUNTIN_PROCESS_INDEX((modno),(parno))]
#define VIRTCOUNTIN_PROCESS_MAX                     13

#define PO_VIRTCNTIN_VALUE             1            // Output of counter module
#define PO_VIRTCNTIN_VELOCITY          2            // Velocity
#define PO_VIRTCNTIN_FRACT             4            // Fractional part of the current position
#define PO_VIRTCNTIN_OVFLS             5            // Number of overflows
#define PO_VIRTCNTIN_LASTSRCPOS        6            // Last Source Position
#define PO_VIRTCNTIN_LASTVELOCITY      7            // Last measured velocity
#define PO_VIRTCNTIN_OVFL_AMOUNT       8            // When an overflow has happend, this value represents the
amount of it
#define PO_VIRTCNTIN_STAT              9            // Status bits
#define PO_VIRTCNTIN_ERR_COUNT         10           // How many errors were detected
#define PO_VIRTCNTIN_VALUE_FILTERED_FRACTIONAL 11          //
#define PO_VIRTCNTIN_VALUE_FILTERED_INTEGRAL 12           //

//=============================================================================================
// 0x2C80 VIRTLATCH_PROCESS - Virtual Latch Inputs Process Data
//=============================================================================================
#define SDOINDEX_VIRTLATCH_PROCESS                  0x2C80
#define VIRTLATCH_PROCESS_INDEX(modno,parno)        ((0x01000000 | (SDOINDEX_VIRTLATCH_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTLATCH_PROCESS_SRCINDEX(modno,parno)     ((SDOINDEX_VIRTLATCH_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTLATCH_PROCESS(modno,parno)              Sysvar[VIRTLATCH_PROCESS_INDEX((modno),(parno))]
#define VIRTLATCH_PROCESS_MAX                       16

#define PO_VIRTLATCH_VALUE             1            // Latch position
#define PO_VIRTLATCH_COUNTER           2            // Latch counter
#define PO_VIRTLATCH_LASTSRCLATCHPOS   4            // Last source latch position
#define PO_VIRTLATCH_FLAG              5            // Latch flag, cleared by writing
#define PO_VIRTLATCH_CMDWORD           6            // Command word
#define PO_VIRTLATCH_FIFO_READ         7            // Latch position read from FIFO
#define PO_VIRTLATCH_FIFO_VALID        8            // Indicates if the latch from the FIFO is valid
#define PO_VIRTLATCH_FIFO_WRITEINDEX   9            // Position of the FIFO write index
#define PO_VIRTLATCH_FIFO_READINDEX    10           // Position of the FIFO read index
#define PO_VIRTLATCH_FIFO_LATCHAMOUNT  11           // Number of latches in the FIFO
#define PO_VIRTLATCH_FIFO_TOTALCNT     12           // Number of latches ever processed through FIFO
#define PO_VIRTLATCH_FIFO_OVERFLOWCNT  13           // Number of overflows
#define PO_VIRTLATCH_FIFO_PEEK         14           // Peek the latest element from the FIFO
#define PO_VIRTLATCH_VALID             15           // Valid flag

//=============================================================================================
// 0x2CC0 VIRTAMP_PROCESS - Virtual Amplifier Process Data
//=============================================================================================
#define SDOINDEX_VIRTAMP_PROCESS                    0x2CC0
```

```
#define VIRTAMP_PROCESS_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_VIRTAMP_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTAMP_PROCESS_SRCINDEX(modno,parno)       ((SDOINDEX_VIRTAMP_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTAMP_PROCESS(modno,parno)                Sysvar[VIRTAMP_PROCESS_INDEX((modno),(parno))]
#define VIRTAMP_PROCESS_MAX                         16

#define PO_VIRTAMP_STATUS              1            // Status
#define PO_VIRTAMP_CMDWORD             2            // Output command word
#define PO_VIRTAMP_REFPOS              3            // Output reference position
#define PO_VIRTAMP_REFVEL              4            // Output reference velocity
#define PO_VIRTAMP_REFACC              5            // Output reference acceleration
#define PO_VIRTAMP_DIRPOS              7            // Direction information
#define PO_VIRTAMP_DIRNEG              8            // Direction information
#define PO_VIRTAMP_CURRENT             10           // Current value
#define PO_VIRTAMP_ACTSTOPDELAY        12           // Internally used counter
#define PO_VIRTAMP_I2TVALUE            13           // Current sum of I2T calculation
#define PO_VIRTAMP_ERROR               14           // Error status
#define PO_VIRTAMP_REMOTE_AMP          15           // Indicates if remote amplifier is in use

    // Values for PO_VIRTAMP_ERROR:
    #define VIRTAMP_ERROR_HWAMP            0x0001   // HW amplifier reports error
    #define VIRTAMP_ERROR_I2T              0x0002   // Virtual amplifier reports I2T error
    #define VIRTAMP_ERROR_NOTREADY         0x0004   // HW amplifier not ready

//=============================================================================================================
// 0x2D00 VIRTDIGIN_PROCESS - Virtual Digital Input Process Data
//=============================================================================================================
#define SDOINDEX_VIRTDIGIN_PROCESS                  0x2D00
#define VIRTDIGIN_PROCESS_INDEX(modno,parno)        ((0x01000000 | (SDOINDEX_VIRTDIGIN_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTDIGIN_PROCESS_SRCINDEX(modno,parno)     ((SDOINDEX_VIRTDIGIN_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTDIGIN_PROCESS(modno,parno)              Sysvar[VIRTDIGIN_PROCESS_INDEX((modno),(parno))]
#define VIRTDIGIN_PROCESS_MAX                       8

#define PO_VIRTDIGIN_VALLONG           1            // Returns the full 32bit output value
#define PO_VIRTDIGIN_VALWORD1          2            // Returns 16bit of the output value
#define PO_VIRTDIGIN_VALWORD2          3            // Returns 16bit of the output value
#define PO_VIRTDIGIN_VALBYTE1          4            // Returns 8bit of the output value.
#define PO_VIRTDIGIN_VALBYTE2          5            // Returns 8bit of the output value.
#define PO_VIRTDIGIN_VALBYTE3          6            // Returns 8bit of the output value.
#define PO_VIRTDIGIN_VALBYTE4          7            // Returns 8bit of the output value.

//=============================================================================================================
// 0x2D40 VIRTDIGOUT_PROCESS - Virtual Digital Output Process Data
//=============================================================================================================
#define SDOINDEX_VIRTDIGOUT_PROCESS                 0x2D40
#define VIRTDIGOUT_PROCESS_INDEX(modno,parno)       ((0x01000000 | (SDOINDEX_VIRTDIGOUT_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTDIGOUT_PROCESS_SRCINDEX(modno,parno)    ((SDOINDEX_VIRTDIGOUT_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTDIGOUT_PROCESS(modno,parno)             Sysvar[VIRTDIGOUT_PROCESS_INDEX((modno),(parno))]
#define VIRTDIGOUT_PROCESS_MAX                      8

#define PO_VIRTDIGOUT_VALLONG          1            // Returns all 32bit of the output value.
#define PO_VIRTDIGOUT_VALWORD1         2            // Returns 16 bit of the output value.
#define PO_VIRTDIGOUT_VALWORD2         3            // Returns 16 bit of the output value.
#define PO_VIRTDIGOUT_VALBYTE1         4            // Returns 8bit of the output value
#define PO_VIRTDIGOUT_VALBYTE2         5            // Returns 8bit of the output value
```

```
#define PO_VIRTDIGOUT_VALBYTE3          6           // Returns 8bit of the output value
#define PO_VIRTDIGOUT_VALBYTE4          7           // Returns 8bit of the output value

//==========================================================================================
// 0x2D80 VIRTANIN_PROCESS - Virtual Analog Input Process Data
//==========================================================================================
#define SDOINDEX_VIRTANIN_PROCESS                   0x2D80
#define VIRTANIN_PROCESS_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_VIRTANIN_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTANIN_PROCESS_SRCINDEX(modno,parno)      ((SDOINDEX_VIRTANIN_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTANIN_PROCESS(modno,parno)               Sysvar[VIRTANIN_PROCESS_INDEX((modno),(parno))]
#define VIRTANIN_PROCESS_MAX                        8

#define PO_VIRTANIN_VALUE               1           // Output value of the module
#define PO_VIRTANIN_MAXVAL              2           // Maximum measured input value
#define PO_VIRTANIN_MINVAL              3           // Minimum measured input value
#define PO_VIRTANIN_SUM                 4           // Integrated value of PO_VIRTANIN_VALUE
#define PO_VIRTANIN_COUNT               6           // Integration time
#define PO_VIRTANIN_AVG                 7           // Average value

//==========================================================================================
// 0x2DC0 VIRTANOUT_PROCESS - Virtual Analog Output Process Data
//==========================================================================================
#define SDOINDEX_VIRTANOUT_PROCESS                  0x2DC0
#define VIRTANOUT_PROCESS_INDEX(modno,parno)        ((0x01000000 | (SDOINDEX_VIRTANOUT_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTANOUT_PROCESS_SRCINDEX(modno,parno)     ((SDOINDEX_VIRTANOUT_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTANOUT_PROCESS(modno,parno)              Sysvar[VIRTANOUT_PROCESS_INDEX((modno),(parno))]
#define VIRTANOUT_PROCESS_MAX                       4

#define PO_VIRTANOUT_VALUE              1           // Returns all 32bit of the output value.
#define PO_VIRTANOUT_VALWORD1           2           // Returns 16bit of the output value.
#define PO_VIRTANOUT_VALWORD2           3           // Returns 16bit of the output value.

//==========================================================================================
// 0x2E00 VIRTCUSTOM_PROCESS - Virtual Customer Module Process Data
//==========================================================================================
#define SDOINDEX_VIRTCUSTOM_PROCESS                 0x2E00
#define VIRTCUSTOM_PROCESS_INDEX(modno,parno)       ((0x01000000 | (SDOINDEX_VIRTCUSTOM_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTCUSTOM_PROCESS_SRCINDEX(modno,parno)    ((SDOINDEX_VIRTCUSTOM_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTCUSTOM_PROCESS(modno,parno)             Sysvar[VIRTCUSTOM_PROCESS_INDEX((modno),(parno))]
#define VIRTCUSTOM_PROCESS_MAX                      21

#define PO_VIRTCUSTOM_1                 1           // Custom Output Values
#define PO_VIRTCUSTOM_2                 2           // Custom Output Values
#define PO_VIRTCUSTOM_3                 3           // Custom Output Values
#define PO_VIRTCUSTOM_4                 4           // Custom Output Values
#define PO_VIRTCUSTOM_5                 5           // Custom Output Values
#define PO_VIRTCUSTOM_6                 6           // Custom Output Values
#define PO_VIRTCUSTOM_7                 7           // Custom Output Values
#define PO_VIRTCUSTOM_8                 8           // Custom Output Values
#define PO_VIRTCUSTOM_9                 9           // Custom Output Values
#define PO_VIRTCUSTOM_10                10          // Custom Output Values
#define PO_VIRTCUSTOM_11                11          // Custom Output Values
#define PO_VIRTCUSTOM_12                12          // Custom Output Values
#define PO_VIRTCUSTOM_13                13          // Custom Output Values
```

```
#define PO_VIRTCUSTOM_14                14          // Custom Output Values
#define PO_VIRTCUSTOM_15                15          // Custom Output Values
#define PO_VIRTCUSTOM_16                16          // Custom Output Values
#define PO_VIRTCUSTOM_17                17          // Custom Output Values
#define PO_VIRTCUSTOM_18                18          // Custom Output Values
#define PO_VIRTCUSTOM_19                19          // Custom Output Values
#define PO_VIRTCUSTOM_20                20          // Custom Output Values

//=============================================================================================================
// 0x2E40 VIRTMATH_PROCESS - Virtual Math Module Process Data
//=============================================================================================================
#define SDOINDEX_VIRTMATH_PROCESS                   0x2E40
#define VIRTMATH_PROCESS_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_VIRTMATH_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define VIRTMATH_PROCESS_SRCINDEX(modno,parno)      ((SDOINDEX_VIRTMATH_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define VIRTMATH_PROCESS(modno,parno)               Sysvar[VIRTMATH_PROCESS_INDEX((modno),(parno))]
#define VIRTMATH_PROCESS_MAX                        7

#define PO_VIRTMATH_VALUE               1           // Output value
#define PO_VIRTMATH_MODE_BASED_VALUE_1 2            // Mode dependent process value 1 - 5
#define PO_VIRTMATH_MODE_BASED_VALUE_2 3            // Mode dependent process value 1 - 5
#define PO_VIRTMATH_MODE_BASED_VALUE_3 4            // Mode dependent process value 1 - 5
#define PO_VIRTMATH_MODE_BASED_VALUE_4 5            // Mode dependent process value 1 - 5
#define PO_VIRTMATH_MODE_BASED_VALUE_5 6            // Mode dependent process value 1 - 5

    // Values for PO_VIRTMATH_MODE_BASED_VALUE:
    #define VIRTMATH_MODE_PID_PROP_VALUE   2        // Mode VIRTMATH_MODE_PID: Proportional value
    #define VIRTMATH_MODE_PID_INT_VALUE    3        // Mode VIRTMATH_MODE_PID: Integral value
    #define VIRTMATH_MODE_PID_DIFF_VALUE   4        // Mode VIRTMATH_MODE_PID: Differential value
    #define VIRTMATH_MODE_PID_PREV_ERROR   5        // Mode VIRTMATH_MODE_PID: Previous error
    #define VIRTMATH_MODE_PID_INT_SUM      6        // Mode VIRTMATH_MODE_PID: Actual integral sum

//=============================================================================================================
// 0x3301 ECAT_SLAVE_PDO1_1_255 -
//=============================================================================================================
#define SDOINDEX_ECAT_SLAVE_PDO1_1_255              0x3301
#define ECAT_SLAVE_PDO1_1_255_INDEX(parno)          ((0x01000000 | (SDOINDEX_ECAT_SLAVE_PDO1_1_255<<8)) | ((long)
(parno)))
#define ECAT_SLAVE_PDO1_1_255_SRCINDEX(parno)       ((SDOINDEX_ECAT_SLAVE_PDO1_1_255<<8) | ((long) (parno)))
#define ECAT_SLAVE_PDO1_1_255(parno)                Sysvar[ECAT_SLAVE_PDO1_1_255_INDEX(parno)]
#define ECAT_SLAVE_PDO1_1_255_MAX                   256

//=============================================================================================================
// 0x3302 ECAT_SLAVE_PDO1_256_510 -
//=============================================================================================================
#define SDOINDEX_ECAT_SLAVE_PDO1_256_510            0x3302
#define ECAT_SLAVE_PDO1_256_510_INDEX(parno)        ((0x01000000 | (SDOINDEX_ECAT_SLAVE_PDO1_256_510<<8)) | ((long)
(parno)))
#define ECAT_SLAVE_PDO1_256_510_SRCINDEX(parno)     ((SDOINDEX_ECAT_SLAVE_PDO1_256_510<<8) | ((long) (parno)))
#define ECAT_SLAVE_PDO1_256_510(parno)              Sysvar[ECAT_SLAVE_PDO1_256_510_INDEX(parno)]
#define ECAT_SLAVE_PDO1_256_510_MAX                 256

//=============================================================================================================
// 0x3303 ECAT_SLAVE_PDO1_511_765 -
//=============================================================================================================
#define SDOINDEX_ECAT_SLAVE_PDO1_511_765            0x3303
#define ECAT_SLAVE_PDO1_511_765_INDEX(parno)        ((0x01000000 | (SDOINDEX_ECAT_SLAVE_PDO1_511_765<<8)) | ((long)
(parno)))
#define ECAT_SLAVE_PDO1_511_765_SRCINDEX(parno)     ((SDOINDEX_ECAT_SLAVE_PDO1_511_765<<8) | ((long) (parno)))
```

```
#define ECAT_SLAVE_PDO1_511_765(parno)              Sysvar[ECAT_SLAVE_PDO1_511_765_INDEX(parno)]
#define ECAT_SLAVE_PDO1_511_765_MAX                 256

//==========================================================================================
// 0x3304 ECAT_SLAVE_PDO1_766_1020 -
//==========================================================================================
#define SDOINDEX_ECAT_SLAVE_PDO1_766_1020           0x3304
#define ECAT_SLAVE_PDO1_766_1020_INDEX(parno)       ((0x01000000 | (SDOINDEX_ECAT_SLAVE_PDO1_766_1020<<8)) |
((long) (parno)))
#define ECAT_SLAVE_PDO1_766_1020_SRCINDEX(parno)    ((SDOINDEX_ECAT_SLAVE_PDO1_766_1020<<8) | ((long) (parno)))
#define ECAT_SLAVE_PDO1_766_1020(parno)             Sysvar[ECAT_SLAVE_PDO1_766_1020_INDEX(parno)]
#define ECAT_SLAVE_PDO1_766_1020_MAX                256

//==========================================================================================
// 0x3305 ECAT_SLAVE_PDO2_1_255 -
//==========================================================================================
#define SDOINDEX_ECAT_SLAVE_PDO2_1_255              0x3305
#define ECAT_SLAVE_PDO2_1_255_INDEX(parno)          ((0x01000000 | (SDOINDEX_ECAT_SLAVE_PDO2_1_255<<8)) | ((long)
(parno)))
#define ECAT_SLAVE_PDO2_1_255_SRCINDEX(parno)       ((SDOINDEX_ECAT_SLAVE_PDO2_1_255<<8) | ((long) (parno)))
#define ECAT_SLAVE_PDO2_1_255(parno)                Sysvar[ECAT_SLAVE_PDO2_1_255_INDEX(parno)]
#define ECAT_SLAVE_PDO2_1_255_MAX                   256

//==========================================================================================
// 0x3306 ECAT_SLAVE_PDO2_256_510 -
//==========================================================================================
#define SDOINDEX_ECAT_SLAVE_PDO2_256_510            0x3306
#define ECAT_SLAVE_PDO2_256_510_INDEX(parno)        ((0x01000000 | (SDOINDEX_ECAT_SLAVE_PDO2_256_510<<8)) | ((long)
(parno)))
#define ECAT_SLAVE_PDO2_256_510_SRCINDEX(parno)     ((SDOINDEX_ECAT_SLAVE_PDO2_256_510<<8) | ((long) (parno)))
#define ECAT_SLAVE_PDO2_256_510(parno)              Sysvar[ECAT_SLAVE_PDO2_256_510_INDEX(parno)]
#define ECAT_SLAVE_PDO2_256_510_MAX                 256

//==========================================================================================
// 0x3307 ECAT_SLAVE_PDO2_511_765 -
//==========================================================================================
#define SDOINDEX_ECAT_SLAVE_PDO2_511_765            0x3307
#define ECAT_SLAVE_PDO2_511_765_INDEX(parno)        ((0x01000000 | (SDOINDEX_ECAT_SLAVE_PDO2_511_765<<8)) | ((long)
(parno)))
#define ECAT_SLAVE_PDO2_511_765_SRCINDEX(parno)     ((SDOINDEX_ECAT_SLAVE_PDO2_511_765<<8) | ((long) (parno)))
#define ECAT_SLAVE_PDO2_511_765(parno)              Sysvar[ECAT_SLAVE_PDO2_511_765_INDEX(parno)]
#define ECAT_SLAVE_PDO2_511_765_MAX                 256

//==========================================================================================
// 0x3308 ECAT_SLAVE_PDO2_766_1020 -
//==========================================================================================
#define SDOINDEX_ECAT_SLAVE_PDO2_766_1020           0x3308
#define ECAT_SLAVE_PDO2_766_1020_INDEX(parno)       ((0x01000000 | (SDOINDEX_ECAT_SLAVE_PDO2_766_1020<<8)) |
((long) (parno)))
#define ECAT_SLAVE_PDO2_766_1020_SRCINDEX(parno)    ((SDOINDEX_ECAT_SLAVE_PDO2_766_1020<<8) | ((long) (parno)))
#define ECAT_SLAVE_PDO2_766_1020(parno)             Sysvar[ECAT_SLAVE_PDO2_766_1020_INDEX(parno)]
#define ECAT_SLAVE_PDO2_766_1020_MAX                256

//==========================================================================================
// 0x3600 SMGLOBAL_PROCESS - Global State Machine Process Data
//==========================================================================================
#define SDOINDEX_SMGLOBAL_PROCESS                   0x3600
#define SMGLOBAL_PROCESS_INDEX(parno)               ((0x01000000 | (SDOINDEX_SMGLOBAL_PROCESS<<8)) | ((long)
(parno)))
#define SMGLOBAL_PROCESS_SRCINDEX(parno)            ((SDOINDEX_SMGLOBAL_PROCESS<<8) | ((long) (parno)))
```

```
#define SMGLOBAL_PROCESS(parno)                     Sysvar[SMGLOBAL_PROCESS_INDEX(parno)]
#define SMGLOBAL_PROCESS_MAX                        48

#define SM_PROC_RUNFLAGS              1             // Runtime flags
#define SM_PROC_PARAMCNT              2             // Maximum number of event parameters in event record
#define SM_PROC_EVENTCNT              3             // Number of events
#define SM_PROC_STATECNT              4             // Number of states
#define SM_PROC_MACHINECNT            5             // Number of state machine specifications
#define SM_PROC_POOLSIZE              6             // Number of records in event pool
#define SM_PROC_QUEUESIZE             7             // Total size of all event queues
#define SM_PROC_STATEDEPTH            8             // Depth of deepest state machine tree
#define SM_PROC_TIMERMAX              9             // Maximum number of timers
#define SM_PROC_SUBSIZE               10            // Number of records in subscribe pool
#define SM_PROC_PRMSIZE               11            // Number of records in parameter pool
#define SM_PROC_POSSIZE               12            // Number of records in position pool
#define SM_PROC_SYSSIZE               13            // Number of records in system event pool.
#define SM_PROC_SYSCNT                14            // Maximum number of system event chains.
#define SM_PROC_SMRUNSTATE            41            // Current state machine running state
#define SM_PROC_SMRUNFLAGS            42            // Current state machine run flags
#define SM_PROC_MACHINE               43            // Current state machine index
#define SM_PROC_POOLUSED              44            // Number of used records in event pool
#define SM_PROC_POOLFREE              45            // Number of free records in event pool
#define SM_PROC_POOLMAX               46            // Maximum number of records used in event pool
#define SM_PROC_HISTORY               47            // History recording enabled

//==========================================================================================================
// 0x3601 SMMACHINE_PROCESS - State Machine Process Data
//==========================================================================================================
#define SDOINDEX_SMMACHINE_PROCESS                  0x3601
#define SMMACHINE_PROCESS_INDEX(modno,parno)        ((0x01000000 | (SDOINDEX_SMMACHINE_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define SMMACHINE_PROCESS_SRCINDEX(modno,parno)     ((SDOINDEX_SMMACHINE_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define SMMACHINE_PROCESS(modno,parno)              Sysvar[SMMACHINE_PROCESS_INDEX((modno),(parno))]
#define SMMACHINE_PROCESS_MAX                       121

#define SM_MACH_ID                    1             // User defined state machine ID
#define SM_MACH_ARRAYCNT              2             // Number of state machines in array
#define SM_MACH_QUEUESIZE             3             // Size of event queue
#define SM_MACH_DATASIZE              4             // Size of private data array
#define SM_MACH_ROOTSTATE             5             // Unique ID of root state
#define SM_MACH_FLAGS                 41            // State machine flags
#define SM_MACH_STATE                 42            // Unique ID of current state
#define SM_MACH_QUEUEUSED             43            // Number of used records in event queue
#define SM_MACH_QUEUEFREE             44            // Number of free records in event queue
#define SM_MACH_QUEUEMAX              45            // Maximum number of records used in event queue
#define SM_MACH_DATA_1                81            // Private data
#define SM_MACH_DATA_2                82            // Private data
#define SM_MACH_DATA_3                83            // Private data
#define SM_MACH_DATA_4                84            // Private data
#define SM_MACH_DATA_5                85            // Private data
#define SM_MACH_DATA_6                86            // Private data
#define SM_MACH_DATA_7                87            // Private data
#define SM_MACH_DATA_8                88            // Private data
#define SM_MACH_DATA_9                89            // Private data
#define SM_MACH_DATA_10               90            // Private data
#define SM_MACH_DATA_11               91            // Private data
#define SM_MACH_DATA_12               92            // Private data
#define SM_MACH_DATA_13               93            // Private data
#define SM_MACH_DATA_14               94            // Private data
```

```
#define SM_MACH_DATA_15                95          // Private data
#define SM_MACH_DATA_16                96          // Private data
#define SM_MACH_DATA_17                97          // Private data
#define SM_MACH_DATA_18                98          // Private data
#define SM_MACH_DATA_19                99          // Private data
#define SM_MACH_DATA_20                100         // Private data
#define SM_MACH_DATA_21                101         // Private data
#define SM_MACH_DATA_22                102         // Private data
#define SM_MACH_DATA_23                103         // Private data
#define SM_MACH_DATA_24                104         // Private data
#define SM_MACH_DATA_25                105         // Private data
#define SM_MACH_DATA_26                106         // Private data
#define SM_MACH_DATA_27                107         // Private data
#define SM_MACH_DATA_28                108         // Private data
#define SM_MACH_DATA_29                109         // Private data
#define SM_MACH_DATA_30                110         // Private data
#define SM_MACH_DATA_31                111         // Private data
#define SM_MACH_DATA_32                112         // Private data
#define SM_MACH_DATA_33                113         // Private data
#define SM_MACH_DATA_34                114         // Private data
#define SM_MACH_DATA_35                115         // Private data
#define SM_MACH_DATA_36                116         // Private data
#define SM_MACH_DATA_37                117         // Private data
#define SM_MACH_DATA_38                118         // Private data
#define SM_MACH_DATA_39                119         // Private data
#define SM_MACH_DATA_40                120         // Private data

//==========================================================================================================
// 0x4000 HWAMP_PARAM - HW Amplifier Parameters
//==========================================================================================================
#define SDOINDEX_HWAMP_PARAM                       0x4000
#define HWAMP_PARAM_INDEX(modno,parno)             ((0x01000000 | (SDOINDEX_HWAMP_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWAMP_PARAM_SRCINDEX(modno,parno)          ((SDOINDEX_HWAMP_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWAMP_PARAM(modno,parno)                   Sysvar[HWAMP_PARAM_INDEX((modno),(parno))]
#define HWAMP_PARAM_MAX                            93

#define HWAMP_MODE                     1           // Mode of operation
#define HWAMP_PISRC_CMDWORD            10          // Source Index/subindex of command word
#define HWAMP_PISRC_REFPOS             11          // Source Index/subindex of position reference
#define HWAMP_PISRC_REFVEL             12          // Source Index/subindex of velocity reference
#define HWAMP_PISRC_REFACC             13          // Source Index/subindex of acceleration reference
#define HWAMP_PISRC_ACTPOS             14          // Source for actual position
#define HWAMP_PISRC_ACTVEL             15          // Not used
#define HWAMP_PISRC_MAX                20          //
#define HWAMP_VELKPROP                 30          // Proportional factor of velocity controller
#define HWAMP_VELKINT                  31          // Integral factor of velocity controller
#define HWAMP_VELKILIM                 32          // Integral limit of velocity controller
#define HWAMP_CURKPROP                 33          // Proportional factor of current controller
#define HWAMP_CURKINT                  34          // Integral factor of current controller
#define HWAMP_CURKILIM                 35          // Integral limit of current controller
#define HWAMP_COMMTYPE                 36          // Commutation type in use
#define HWAMP_PWMFREQ                  37          // Deprecated: Frequency of the PWM output
#define HWAMP_POLES                    38          // Number of pole pairs
#define HWAMP_AHEADANGLE               39          // Field ahead angle (not used)
#define HWAMP_I2TTIME                  40          // Not used
#define HWAMP_CURFILTTIME              41          // Filter time for current measurement returned by
PO_HWAMP_CURRENT, not used in current controller, so current controller uses allways unfiltered values.
#define HWAMP_ENCRES                   42          // Resolution of the encoder for position feed back in
```

```
increments (quadcounts)
#define HWAMP_MAXCUR                    43          // Maximal current allowed
#define HWAMP_POSDIR                    44          // Position input reverse
#define HWAMP_MAXRPM                    45          // Maximum velocity in RPM
#define HWAMP_USERDEFINED_DATA0         46          // Selects what data should be mapped into
PO_HWAMP_USERDEFINED_DATAn (only MACS5)
#define HWAMP_USERDEFINED_DATA1         47          // Selects what data should be mapped into
PO_HWAMP_USERDEFINED_DATAn (only MACS5)
#define HWAMP_USERDEFINED_DATA2         48          // Selects what data should be mapped into
PO_HWAMP_USERDEFINED_DATAn (only MACS5)
#define HWAMP_USERDEFINED_DATA3         49          // Selects what data should be mapped into
PO_HWAMP_USERDEFINED_DATAn (only MACS5)
#define HWAMP_ELPOL                     51          // encoder polarity vs. electrical polarity
#define HWAMP_I2T_TRIPVALUE             53          // I2T monitoring trip value
#define HWAMP_I2T_CONT                  54          // I2T value of continuously allowed current
#define HWAMP_CHOP_UDCMIN               55          // Voltage Level for low chopper function
#define HWAMP_CHOP_UDCMAX               56          // Voltage Level for high chopper function
#define HWAMP_VELPI_REFLIM              59          // Reference Limit for velocity controller
#define HWAMP_VELPI_OUTLIM              60          // Output Limit for velocity controller
#define HWAMP_CURPI0_REFLIM             61          // Reference Limit for current controller 0
#define HWAMP_CURPI0_OUTLIM             62          // Output Limit for current controller 0
#define HWAMP_CURPI1_REFLIM             63          // Reference Limit for current controller 1
#define HWAMP_CURPI1_OUTLIM             64          // Output Limit for current controller 1
#define HWAMP_ELPOS_LAGMULT             65          // Lag compensation multiplier value
#define HWAMP_TRIP_CURRENT              66          // Trip current value
#define HWAMP_TEMP_MAX                  67          // Maximum board temperature
#define HWAMP_VOLT_MIN                  68          // Minimum voltage for amplifier
#define HWAMP_VOLT_MAX                  69          // Maximum voltage for amplifier
#define HWAMP_CURPI_MODE                70          // Current regulator mode. 0 = MODE_CLASSIC, 1 =
MODE_LOADESTIMATE
#define HWAMP_VELPI_MODE                71          // Velocity regulator mode. 0 = MODE_CLASSIC, 1 =
MODE_LOADESTIMATE
#define HWAMP_PROG                      72          // Amplifier Program Execution
#define HWAMP_PROG_PARAMVAL             73          // Amplifier Program Parameter Value
#define HWAMP_PROG_PARAMIND             74          // Amplifier Program Parameter Index
#define HWAMP_PROG_RETVAL               75          // Amplifier Program Return Value
#define HWAMP_HALL_SIMULATOR            86          // Only for testing!
#define HWAMP_HALL_ERROR_LIMIT          87          // Limit for hall errors
#define HWAMP_HALL_ALIGNMENT            88          // Hall Alignment
#define HWAMP_SW_VERSION                89          //
#define HWAMP_POSEL_OFFSET              90          // Sets the offset of the electric field in relation to an
absolute encoder.
#define HWAMP_DEFINE_PO_DATA_SET        91          // Defines a set of process data
#define HWAMP_IDQ_D_REF                 92          // Defines Idq d-Part  reference in mA

    // Values for HWAMP_MODE:
    #define HWAMP_MODE_POS_VEL_CUR          0       // Pos -> Vel -> Cur -> PWM
    #define HWAMP_MODE_POS_VEL              1       // Pos -> Vel -> PWM
    #define HWAMP_MODE_POS_CUR              2       // Pos -> Cur -> PWM
    #define HWAMP_MODE_POS                  3       // Pos -> PWM
    #define HWAMP_MODE_PFG_CUR_PWM          4       // Mode for all STEP and PMSM in open loop, with current
control
    #define HWAMP_MODE_PFG_PWM              5       // Mode for all STEP and PMSM in open loop, without current
control
    #define HWAMP_MODE_INT_CUR_PWM          6       // Mode only for HWAMP_COMMTYPE_CHOPPER: with current control
    #define HWAMP_MODE_INT_PWM              7       // Mode only for HWAMP_COMMTYPE_CHOPPER: without current
control

    // Values for HWAMP_COMMTYPE:
    #define HWAMP_COMMTYPE_NONE             0       // Amplifier disabled
```

```
    #define HWAMP_COMMTYPE_DC              1        // DC motor, two phases
    #define HWAMP_COMMTYPE_BLDC            2        // Brushless, three phases, block commutation, requires HALL
sensor
    #define HWAMP_COMMTYPE_BLDC_120        2        // Brushless, three phases, block commutation, 120° hall
sensors, Deprecated
    #define HWAMP_COMMTYPE_BLDC_60         3        // Brushless, three phases, block commutation, 60° hall
sensors, Deprecated
    #define HWAMP_COMMTYPE_STEP            4        // Stepper motor, two phases bipolar
    #define HWAMP_COMMTYPE_HALL_PMSM       5        // Alignment with HALL sensor, else like PMSM
    #define HWAMP_COMMTYPE_DC_TWIN         6        // DC motor  twin mode
    #define HWAMP_COMMTYPE_PMSM            7        // Brushless, three phases, sinusoidal commutation
    #define HWAMP_COMMTYPE_BLDC_TWIN       8        // Brushless, block commutation, requires HALL sensor,  twin
mode
    #define HWAMP_COMMTYPE_BLDC_120_TWIN   8        // Brushless, block commutation, 120° hall sensors, twin mode,
Deprecated
    #define HWAMP_COMMTYPE_BLDC_60_TWIN    9        // Brushless, block commutation, 60° hall sensors, twin mode,
Deprecated
    #define HWAMP_COMMTYPE_STEP_TWIN       10       // Stepper motor, two phases bipolar,  twin mode
    #define HWAMP_COMMTYPE_HALL_PMSM_TWIN  11       // Alignment with HALL sensor, else like PMSM_TWIN
    #define HWAMP_COMMTYPE_PMSM_TWIN       12       // Brushless, three phases, sinusoidal commutation, twin mode
    #define HWAMP_COMMTYPE_CHOPPER         13       // Brake chopper, RL load required
    #define HWAMP_COMMTYPE_DC_ALT          14       // Only for internal use
    #define HWAMP_COMMTYPE_DC_100          15       // Only for internal use

    // Values for HWAMP_ENCRES:
    #define HWAMP_ENCRES_MICROSTEP_RES     1024     // This is a constant factor, which is used to calculate the
value of the parameter HWAMP_ENCRES

    // Values for HWAMP_POSDIR:
    #define HWAMP_POSDIR_NORMAL            1        // Direction normal
    #define HWAMP_POSDIR_REVERSE           -1       // Direction reverse

    // Values for HWAMP_ELPOL:
    #define HWAMP_ELPOL_REGULAR            1        // Electrical polarity is equal encoder's polarity
    #define HWAMP_ELPOL_INVERS             -1       // Deprecated, wrong spelling, Use HWAMP_ELPOL_INVERSE
    #define HWAMP_ELPOL_INVERSE            -1       // Electrical polarity is inverse to encoder's polarity

    // Values for HWAMP_HALL_ALIGNMENT:
    #define HALL_ALIGNMENT_BOTTOM          0        // First element
    #define HALL_ALIGNMENT_360120240       0        // 120° CCW: H1 at 360°, H2 at 120°, H3 at 240°
    #define HALL_ALIGNMENT_060180300       1        // 120° CCW: H1 at 060°, H2 at 180°, H3 at 300°
    #define HALL_ALIGNMENT_120240360       2        // 120° CCW: H1 at 120°, H2 at 240°, H3 at 360°
    #define HALL_ALIGNMENT_180300060       3        // 120° CCW: H1 at 180°, H2 at 300°, H3 at 060°
    #define HALL_ALIGNMENT_240360120       4        // 120° CCW: H1 at 240°, H2 at 360°, H3 at 120°
    #define HALL_ALIGNMENT_300060180       5        // 120° CCW: H1 at 300°, H2 at 060°, H3 at 180°
    #define HALL_ALIGNMENT_120DEGREE       5        // 120° standard: same as ALIGNMENT_300060180
    #define HALL_ALIGNMENT_360240120       6        // 120° CW:  H1 at 360°, H2 at 240°, H3 at 120°
    #define HALL_ALIGNMENT_060300180       7        // 120° CW:  H1 at 060°, H2 at 300°, H3 at 180°
    #define HALL_ALIGNMENT_120360240       8        // 120° CW:  H1 at 120°, H2 at 360°, H3 at 240°
    #define HALL_ALIGNMENT_180060300       9        // 120° CW:  H1 at 180°, H2 at 060°, H3 at 300°
    #define HALL_ALIGNMENT_240120360       10       // 120° CW:  H1 at 240°, H2 at 120°, H3 at 360°
    #define HALL_ALIGNMENT_300180060       11       // 120° CW:  H1 at 300°, H2 at 180°, H3 at 060°
    #define HALL_ALIGNMENT_360060120       12       // 60° CCW: H1 at 360°, H2 at 060°, H3 at 120°
    #define HALL_ALIGNMENT_060120180       13       // 60° CCW: H1 at 060°, H2 at 120°, H3 at 180°
    #define HALL_ALIGNMENT_120180240       14       // 60° CCW: H1 at 120°, H2 at 180°, H3 at 240°
    #define HALL_ALIGNMENT_180240300       15       // 60° CCW: H1 at 180°, H2 at 240°, H3 at 300°
    #define HALL_ALIGNMENT_240300360       16       // 60° CCW: H1 at 240°, H2 at 300°, H3 at 360°
    #define HALL_ALIGNMENT_300360060       17       // 60° CCW: H1 at 300°, H2 at 360°, H3 at 060°
    #define HALL_ALIGNMENT_360120060       18       // 60° CW:  H1 at 360°, H2 at 120°, H3 at 060°
    #define HALL_ALIGNMENT_060180120       19       // 60° CW:  H1 at 060°, H2 at 180°, H3 at 120°
```

```
    #define HALL_ALIGNMENT_120240180       20       // 60° CW:  H1 at 120°, H2 at 240°, H3 at 180°
    #define HALL_ALIGNMENT_180300240       21       // 60° CW:  H1 at 180°, H2 at 300°, H3 at 240°
    #define HALL_ALIGNMENT_240360300       22       // 60° CW:  H1 at 240°, H2 at 360°, H3 at 300°
    #define HALL_ALIGNMENT_060DEGREE       22       // 60° standard: same as ALIGNMENT_240360300
    #define HALL_ALIGNMENT_300060360       23       // 60° CW:  H1 at 300°, H2 at 060°, H3 at 360°
    #define HALL_ALIGNMENT_TOP               23       // Last valid element
    #define HALL_ALIGNMENT_ERROR             24       // No valid alignment

    // Values for HWAMP_DEFINE_PO_DATA_SET:
    #define HWAMP_PO_DATA_SET_FIRST        1        // First set
    #define HWAMP_PO_DATA_SET_LAST         9        // Last set
    #define HWAMP_PO_DATA_SET_1_OFFSET     1        // Offset of SET1, see example
    #define HWAMP_PO_DATA_SET_2_OFFSET     1000     // Offset of SET2, see example
    #define HWAMP_PO_DATA_SET_MAX_LENGTH   6        // Max length of a set. SET1 and SET2 have each this length.
    #define HWAMP_PO_DATA_SET_IMEAS        1        // Actual not filtered half bridge current in [mA], order:
depends on COMMTYPE. This is the default set for set 1.
    #define HWAMP_PO_DATA_SET_IMEASFIL     2        // Actual filtered half bridge current in [mA], order: depends
on COMMTYPE. This is the default set for set 2.
    #define HWAMP_PO_DATA_SET_CUR_CALIB_OFFSET 3        // Used current calibration offset in [mA], order: depends
on COMMTYPE
    #define HWAMP_PO_DATA_SET_CUR_CALIB_STATE 4        // State of current calibrartion, see CurCalibState, order:
depends on COMMTYPE
    #define HWAMP_PO_DATA_SET_CUR_CALIB_GAIN 5        // Used current calibration gain, value range
-10000...10000, order: depends on COMMTYPE
    #define HWAMP_PO_DATA_SET_DEVICE_SPECIFIC_1 6        // Depends on device, returns 0 if nothing implemented
    #define HWAMP_PO_DATA_SET_DEVICE_SPECIFIC_2 7        // Depends on device, returns 0 if nothing implemented
    #define HWAMP_PO_DATA_SET_I2T_ACT      8        // Actual half bridge I2T value, order: depends on COMMTYPE
    #define HWAMP_PO_DATA_SET_CUR_RAW      9        // Actual raw value of ADC's current sense, order: depends on
COMMTYPE

//=============================================================================================================
// 0x4040 HWENC_PARAM - HW Encoder Parameters
//=============================================================================================================
#define SDOINDEX_HWENC_PARAM                        0x4040
#define HWENC_PARAM_INDEX(modno,parno)              ((0x01000000 | (SDOINDEX_HWENC_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWENC_PARAM_SRCINDEX(modno,parno)           ((SDOINDEX_HWENC_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWENC_PARAM(modno,parno)                    Sysvar[HWENC_PARAM_INDEX((modno),(parno))]
#define HWENC_PARAM_MAX                             44

#define HWENCODER_MODE                1            // Encoder type
#define HWENCODER_ISABSOLUTE          2            // Defines, if the used encoder delivers an absolute value
#define HWENCODER_PISRC_ENCOUT        10           // Source for the encoder signal
#define HWENCODER_PISRC_REFOUT        11           // Source for the reference signal
#define HWENCODER_GLITCHFILT          30           // Filter for encoder signals
#define HWENCODER_CLOCKFREQ           31           // Clock frequency for SSI encoder
#define HWENCODER_DATLEN              32           // Number of bits for SSI communication
#define HWENCODER_DELAY               33           // Delay for SSI
#define HWENCODER_TERM                34           // Termination (not used)
#define HWENCODER_BUSID               35           // ID for serial communication (not used)
#define HWENCODER_BAUDRATE            36           // Baud rate for serial communication (not used)
#define HWENCODER_PARITY              37           // Parity for serial communication (not used)
#define HWENCODER_MONITORING          38           // Encoder Monitoring Enable
#define HWENCODER_CLOCK_ACTIVE        39           // SSI clock active or not
#define HWENCODER_FAST_UPDATE         40           // Encoder update rate (Not on MACS5)
#define HWENCODER_REVERSE_DIRECTION   41           // Encoder counting direction reverse (Only MACS5)
#define HWENCODER_POSBITS             42           // Number of SSI bits with position information (Not on MACS5)
#define HWENCODER_TRAILBITS           43           // Number of trailling SSI bits with no position information
(Not on MACS5)
```

```
    // Values for HWENCODER_MODE:
    #define HWENCODER_MODE_INCREMENTAL     0        // Incremental Encoder Input
    #define HWENCODER_MODE_INCROUTPUT      1        // Incremental Encoder Output
    #define HWENCODER_MODE_SSI_ACTIVE      2        // SSI Encoder active clock
    #define HWENCODER_MODE_SSI_PASSIVE     3        // SSI Encoder passive clock
    #define HWENCODER_MODE_HIPERFACE_RX    4        // Hiperface Encoder receiving
    #define HWENCODER_MODE_HIPERFACE_TX    5        // Hiperface Encoder sending
    #define HWENCODER_MODE_SINCOS          6        // Sin/Cos Encoder
    #define HWENCODER_MODE_HALL            7        // HALL Sensor
    #define HWENCODER_MODE_ENDAT           9        // EnDat Encoder

    // Values for HWENCODER_FAST_UPDATE:
    #define HWENCODER_FAST_UPDATE_ENABLE   1
    #define HWENCODER_FAST_UPDATE_DISABLE  0

    // Values for HWENCODER_REVERSE_DIRECTION:
    #define HWENCODER_REVERSE_DIRECTION_NORMAL 0        // Normal counting direction
    #define HWENCODER_REVERSE_DIRECTION_INVERSE 1        // Inverse counting direction

//==========================================================================================================
// 0x4070 HWHALL_PARAM - HW Hall Sensor Parameters
//==========================================================================================================
#define SDOINDEX_HWHALL_PARAM                       0x4070
#define HWHALL_PARAM_INDEX(modno,parno)             ((0x01000000 | (SDOINDEX_HWHALL_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWHALL_PARAM_SRCINDEX(modno,parno)          ((SDOINDEX_HWHALL_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWHALL_PARAM(modno,parno)                   Sysvar[HWHALL_PARAM_INDEX((modno),(parno))]
#define HWHALL_PARAM_MAX                            11

#define HWHALL_MODE                    1            // Enable
#define HWHALL_PISRC_ENCOUT            10           // Source for the HALL Information

    // Values for HWHALL_MODE:
    #define HWHALL_MODE_DISABLE            0        // Disable hall port
    #define HWHALL_MODE_ENABLE             1        // Enable hall port
    #define HWHALL_MODE_ENABLE_VEL         2        // Enable hall with velocity measurement

//==========================================================================================================
// 0x4080 HWCOUNTINC_PARAM - HW Counter-inc Parameters
//==========================================================================================================
#define SDOINDEX_HWCOUNTINC_PARAM                   0x4080
#define HWCOUNTINC_PARAM_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_HWCOUNTINC_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWCOUNTINC_PARAM_SRCINDEX(modno,parno)      ((SDOINDEX_HWCOUNTINC_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWCOUNTINC_PARAM(modno,parno)               Sysvar[HWCOUNTINC_PARAM_INDEX((modno),(parno))]
#define HWCOUNTINC_PARAM_MAX                        36

#define HWCNTINC_MODE                  1            // not used
#define HWCNTINC_PISRC_COUNTER         10           // Source for counter signal
#define HWCNTINC_PISRC_TRIGGER         11           // Source for reset signal
#define HWCNTINC_SLOPE                 30           // Slope selector for reset signal
#define HWCNTINC_INTERPOL              31           // Interpolation factor
#define HWCNTINC_SCALE                 32           // Scale value (not used)
#define HWCNTINC_UPDATERATE            33           // Update frequency of counter (not used)
#define HWCNTINC_RESOLUTION            34           // Resolution of counter per revolution
#define HWCNTINC_SINCOS_AB_OFFSET      35           // Offset values for SinCOS - Channel A & B
```

```
    // Values for HWCNTINC_PISRC_TRIGGER:
    #define HWCNTINC_PISRC_TRIGGER_ENCZ     0x000    // Encoder Index line
    #define HWCNTINC_PISRC_TRIGGER_VMZ      0x040    // Virtual Master
    #define HWCNTINC_PISRC_TRIGGER_DDSZ     0x080    // DDS
    #define HWCNTINC_PISRC_TRIGGER_BRDA     0x0C0    // Bridge A Signals
    #define HWCNTINC_PISRC_TRIGGER_BRDB     0x100    // Bridge B Signals
    #define HWCNTINC_PISRC_TRIGGER_DINP     0x140    // Digital Input
    #define HWCNTINC_PISRC_TRIGGER_DOUT     0x180    // Digital Output
    #define HWCNTINC_PISRC_TRIGGER_CMPG     0x1C0    // Compare GreaterThan
    #define HWCNTINC_PISRC_TRIGGER_CMPL     0x200    // Compare LessThan
    #define HWCNTINC_PISRC_TRIGGER_PULS     0x240    // Pulse Generator
    #define HWCNTINC_PISRC_TRIGGER_SHIFT    0x280    // Shift Register
    #define HWCNTINC_PISRC_TRIGGER_IRQ      0x2C0    // SOC Interrupt
    #define HWCNTINC_PISRC_TRIGGER_NOT      0xFFC0   // No trigger

    // Values for HWCNTINC_SLOPE:
    #define HWCNTINC_SLOPE_CONTINUOUS       0        // Continuous reset signal
    #define HWCNTINC_SLOPE_RISING           1        // Rising reset signal
    #define HWCNTINC_SLOPE_FALLING          2        // Falling reset signal
    #define HWCNTINC_SLOPE_BOTH             3        // Rising/ Falling reset signal

//==========================================================================================================
// 0x40C0 HWCOUNTABS_PARAM - HW Counter-abs Parameters
//==========================================================================================================
#define SDOINDEX_HWCOUNTABS_PARAM                   0x40C0
#define HWCOUNTABS_PARAM_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_HWCOUNTABS_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWCOUNTABS_PARAM_SRCINDEX(modno,parno)      ((SDOINDEX_HWCOUNTABS_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWCOUNTABS_PARAM(modno,parno)               Sysvar[HWCOUNTABS_PARAM_INDEX((modno),(parno))]
#define HWCOUNTABS_PARAM_MAX                        11

#define HWCNTABS_MODE                   1           // Not used
#define HWCNTABS_CODING                 2           // Data coding
#define HWCNTABS_PISCR_COUNTER          10          // not used

    // Values for HWCNTABS_CODING:
    #define HWCNTABS_CODING_GREY            0        // Gray code (typo)
    #define HWCNTABS_CODING_GRAY            0        // Gray code (default)
    #define HWCNTABS_CODING_NONE            1        // Binary

//==========================================================================================================
// 0x4100 HWCOUNTUNI_PARAM - HW Counter-universal Parameters
//==========================================================================================================
#define SDOINDEX_HWCOUNTUNI_PARAM                   0x4100
#define HWCOUNTUNI_PARAM_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_HWCOUNTUNI_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWCOUNTUNI_PARAM_SRCINDEX(modno,parno)      ((SDOINDEX_HWCOUNTUNI_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWCOUNTUNI_PARAM(modno,parno)               Sysvar[HWCOUNTUNI_PARAM_INDEX((modno),(parno))]
#define HWCOUNTUNI_PARAM_MAX                        32

#define HWCNTUNI_MODE                   1           // Encoder / Reference Input
#define HWCNTUNI_PISRC_COUNTER          10          // Source of the counter signal
#define HWCNTUNI_PISRC_CLEARSIG         11          // Source of reset signal
#define HWCNTUNI_CLRSLOPE               30          // Slope selector for reset signal
#define HWCNTUNI_SRCSLOPE               31          // Slope selector for counter signal

    // Values for HWCNTUNI_MODE:
    #define HW_CNTUNI_MODE_ENCSRC           0        // Encoder source /+/-)
```

```
    #define HW_CNTUNI_MODE_REFSRC          1        // Reference source (+)
    #define HW_CNTUNI_MODE_SINGLENCA       2        // Single encoder source A (+)
    #define HW_CNTUNI_MODE_SINGLENCB       3        // Single encoder source B (+)
    #define HW_CNTUNI_MODE_RESETOFFSET     0x8000   // Clear (add this value to mode to clear)

//==========================================================================================
// 0x4120 HWCMPUNI_PARAM - HW Comparator-universal Parameters
//==========================================================================================
#define SDOINDEX_HWCMPUNI_PARAM                     0x4120
#define HWCMPUNI_PARAM_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_HWCMPUNI_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWCMPUNI_PARAM_SRCINDEX(modno,parno)        ((SDOINDEX_HWCMPUNI_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWCMPUNI_PARAM(modno,parno)                 Sysvar[HWCMPUNI_PARAM_INDEX((modno),(parno))]
#define HWCMPUNI_PARAM_MAX                          34

#define HWCMPUNI_MODE                  1            // not used
#define HWCMPUNI_PISRC_COUNTER         10           // Counter to be observed
#define HWCMPUNI_POLUL                 30           // Polarity of output at upper limit
#define HWCMPUNI_POLLL                 31           // Polarity of output at lower limit
#define HWCMPUNI_UPLIMIT               32           // Value of upper limit
#define HWCMPUNI_LOLIMIT               33           // Value of lower limit

//==========================================================================================
// 0x4140 HWLATCH_PARAM - HW Latch Register Parameters
//==========================================================================================
#define SDOINDEX_HWLATCH_PARAM                      0x4140
#define HWLATCH_PARAM_INDEX(modno,parno)            ((0x01000000 | (SDOINDEX_HWLATCH_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWLATCH_PARAM_SRCINDEX(modno,parno)         ((SDOINDEX_HWLATCH_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWLATCH_PARAM(modno,parno)                  Sysvar[HWLATCH_PARAM_INDEX((modno),(parno))]
#define HWLATCH_PARAM_MAX                           33

#define HWLATCH_MODE                   1            // not used
#define HWLATCH_PISRC_COUNTER          10           // Source for counter to latch
#define HWLATCH_PISRC_TRIGGER          11           // Source for trigger signal
#define HWLATCH_PISRC_CMDWORD          12           // Source Index/subindex of command word
#define HWLATCH_SLOPE                  30           // Slope selector for trigger signal
#define HWLATCH_PAR_DIGIN_VALIDATOR    31           // Selects a digital input to validate the latch signal
#define HWLATCH_STARTADC               32           // If this is true, on some platforms the ADC will be started
when a latch arrives

    // Values for HWLATCH_PISRC_TRIGGER:
    #define HWLATCH_PISRC_TRIGGER_ENCZ     0x0000   // Encoder Index line
    #define HWLATCH_PISRC_TRIGGER_VMZ      0x0040   // Virtual Master
    #define HWLATCH_PISRC_TRIGGER_DDSZ     0x0080   // DDS
    #define HWLATCH_PISRC_TRIGGER_BRDA     0x00C0   // Bridge A Signals
    #define HWLATCH_PISRC_TRIGGER_BRDB     0x0100   // Bridge B Signals
    #define HWLATCH_PISRC_TRIGGER_DINP     0x0140   // Digital Input
    #define HWLATCH_PISRC_TRIGGER_DOUT     0x0180   // Digital Output
    #define HWLATCH_PISRC_TRIGGER_CMPG     0x01C0   // Compare GreaterThan
    #define HWLATCH_PISRC_TRIGGER_CMPL     0x0200   // Compare LessThan
    #define HWLATCH_PISRC_TRIGGER_PULS     0x0240   // Pulse Generator
    #define HWLATCH_PISRC_TRIGGER_SHIFT    0x0280   // Shift Register
    #define HWLATCH_PISRC_TRIGGER_IRQ      0x02C0   // SOC Interrupt
    #define HWLATCH_PISRC_TRIGGER_NOT      0xFFC0   // No trigger

    // Values for HWLATCH_SLOPE:
    #define HWLATCH_SLOPE_CONTINUOUS       0        // Continuous trigger signal
```

```
    #define HWLATCH_SLOPE_RISING           1        // Rising trigger signal
    #define HWLATCH_SLOPE_FALLING          2        // Falling trigger signal
    #define HWLATCH_SLOPE_BOTH             3        // Rising/ Falling trigger signal

    // Values for HWLATCH_PAR_DIGIN_VALIDATOR:
    #define HWLATCH_PAR_DIGIN_VALIDATOR_NONE 0        // Always accept
    #define HWLATCH_PAR_DIGIN_VALIDATOR_HIGH_DINP1 1        // High Din 1
    #define HWLATCH_PAR_DIGIN_VALIDATOR_HIGH_DINP2 2        // High Din 2
    #define HWLATCH_PAR_DIGIN_VALIDATOR_LOW_DINP1 -1       // Low Din 1
    #define HWLATCH_PAR_DIGIN_VALIDATOR_LOW_DINP2 -2       // Low Din 2

//==========================================================================================================
// 0x4180 HWSIGGEN_PARAM - HW Signalgen (Enc/Pulse) Parameters
//==========================================================================================================
#define SDOINDEX_HWSIGGEN_PARAM                     0x4180
#define HWSIGGEN_PARAM_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_HWSIGGEN_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWSIGGEN_PARAM_SRCINDEX(modno,parno)        ((SDOINDEX_HWSIGGEN_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWSIGGEN_PARAM(modno,parno)                 Sysvar[HWSIGGEN_PARAM_INDEX((modno),(parno))]
#define HWSIGGEN_PARAM_MAX                          32

#define HWSIGGEN_MODE                  1            // Enable
#define HWSIGGEN_PISRC_VELOCITY        10           // Source for frequency reference
#define HWSIGGEN_PISRC_SYNC            11           // Source for the sync information (not used)
#define HWSIGGEN_POLARITY              30           // Polarity of the reference signal
#define HWSIGGEN_SIGDIST               31           // Distance between two reference signals

    // Values for HWSIGGEN_MODE:
    #define HWSIGGEN_MODE_DISABLE          0        // Disable the signal generator
    #define HWSIGGEN_MODE_ENABLE           1        // Enable the signal generator
    #define HWSIGGEN_MODE_ENABLE_SPECIAL   2        // Enable the signal generator additionally to an incremental
encoder (only trigger signal)

//==========================================================================================================
// 0x41C0 HWDDS_PARAM - HW DDS Parameters
//==========================================================================================================
#define SDOINDEX_HWDDS_PARAM                        0x41C0
#define HWDDS_PARAM_INDEX(modno,parno)              ((0x01000000 | (SDOINDEX_HWDDS_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWDDS_PARAM_SRCINDEX(modno,parno)           ((SDOINDEX_HWDDS_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWDDS_PARAM(modno,parno)                    Sysvar[HWDDS_PARAM_INDEX((modno),(parno))]
#define HWDDS_PARAM_MAX                             35

#define HWDDS_MODE                     1            // Mode
#define HWDDS_PISRC_ENCSIG             10           // Source for encoder signal
#define HWDDS_PISRC_SYNC               11           // Source for the sync information
#define HWDDS_POLARITY                 30           // Polarity of the reference signal
#define HWDDS_SIGDIST                  31           // Distance between two reference signals
#define HWDDS_NUMERATOR                32           // Numerator of the divisor. Must be smaller than the
denominator below.
#define HWDDS_DENOMINATOR              33           // Denominator of the divisor
#define HWDDS_ENAREVMCOMP              34           // Reverse compensation

//==========================================================================================================
// 0x41D0 HWPWMGEN_PARAM - HW PWM Generator Parameters
//==========================================================================================================
#define SDOINDEX_HWPWMGEN_PARAM                     0x41D0
#define HWPWMGEN_PARAM_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_HWPWMGEN_PARAM<<8)) | (((long)
```

```
(modno))<<8) | ((long) (parno)))
#define HWPWMGEN_PARAM_SRCINDEX(modno,parno)        ((SDOINDEX_HWPWMGEN_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWPWMGEN_PARAM(modno,parno)                 Sysvar[HWPWMGEN_PARAM_INDEX((modno),(parno))]
#define HWPWMGEN_PARAM_MAX                          11

#define HWPWMGEN_MODE                  1            // Mode
#define HWPWMGEN_POLARITY              2            // Polarity of the reference signal
#define HWPWMGEN_FREQUENCY             3            // Frequency of the PWM output
#define HWPWMGEN_DUTYCYCLE_RANGE       4            // Duty cycle range
#define HWPWMGEN_PISRC_DUTYCYCLE       10           // Source for pwm duty cycle

    // Values for HWPWMGEN_MODE:
    #define HWPWMGEN_MODE_SIGNED           0        // Input value is a signed 16bit value
    #define HWPWMGEN_MODE_UNSIGNED         1        // Input value is an unsigned 16bit value

    // Values for HWPWMGEN_POLARITY:
    #define HWPWMGEN_POLARITY_POSITIVE     0        // Output Signal is not inverted
    #define HWPWMGEN_POLARITY_NEGATIVE     1        // Output Signal is inverted

//=================================================================================================
// 0x41E0 HWPULSGEN_PARAM - HW Pulsgen Parameters
//=================================================================================================
#define SDOINDEX_HWPULSGEN_PARAM                    0x41E0
#define HWPULSGEN_PARAM_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_HWPULSGEN_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWPULSGEN_PARAM_SRCINDEX(modno,parno)       ((SDOINDEX_HWPULSGEN_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWPULSGEN_PARAM(modno,parno)                Sysvar[HWPULSGEN_PARAM_INDEX((modno),(parno))]
#define HWPULSGEN_PARAM_MAX                         34

#define HWPULSGEN_MODE                 1            // Mode
#define HWPULSGEN_PISRC_ENCSIG         10           // Source for encoder signal
#define HWPULSGEN_PISRC_SYNC           11           // Source for the sync information
#define HWPULSGEN_POLARITY             30           // Polarity of the output signal
#define HWPULSGEN_SIGDIST              31           // Distance between two output pulses (period)
#define HWPULSGEN_SIGLENGTH            32           // Length of the pulse
#define HWPULSGEN_NOCOMP               33           // Disable reverse motion compensation

//=================================================================================================
// 0x41F0 HWSHIFTREG_PARAM - HW Shift Register Parameters
//=================================================================================================
#define SDOINDEX_HWSHIFTREG_PARAM                   0x41F0
#define HWSHIFTREG_PARAM_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_HWSHIFTREG_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWSHIFTREG_PARAM_SRCINDEX(modno,parno)      ((SDOINDEX_HWSHIFTREG_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWSHIFTREG_PARAM(modno,parno)               Sysvar[HWSHIFTREG_PARAM_INDEX((modno),(parno))]
#define HWSHIFTREG_PARAM_MAX                        31

#define HWSHIFTREG_MODE                1            // Enable
#define HWSHIFTREG_PISRC_ENCSIG        10           // Source for shift clock
#define HWSHIFTREG_PISRC_DATA          11           // Data to be shifted
#define HWSHIFTREG_DELAY               30           // Input delay

//=================================================================================================
// 0x4200 HWDIGIN_PARAM - HW Digital Input Parameters
//=================================================================================================
#define SDOINDEX_HWDIGIN_PARAM                      0x4200
#define HWDIGIN_PARAM_INDEX(modno,parno)            ((0x01000000 | (SDOINDEX_HWDIGIN_PARAM<<8)) | (((long) (modno))
```

```
<<8) | ((long) (parno)))
#define HWDIGIN_PARAM_SRCINDEX(modno,parno)         ((SDOINDEX_HWDIGIN_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWDIGIN_PARAM(modno,parno)                  Sysvar[HWDIGIN_PARAM_INDEX((modno),(parno))]
#define HWDIGIN_PARAM_MAX                           63

#define HWDIGIN_MODE                  1             // not used
#define HWDIGIN_POLARITY              30            // Polarity of input signal
#define HWDIGIN_GLITCHFILT1           31            // Glitch filter
#define HWDIGIN_GLITCHFILT2           32            // Glitch filter
#define HWDIGIN_GLITCHFILT3           33            // Glitch filter
#define HWDIGIN_GLITCHFILT4           34            // Glitch filter
#define HWDIGIN_GLITCHFILT5           35            // Glitch filter
#define HWDIGIN_GLITCHFILT6           36            // Glitch filter
#define HWDIGIN_GLITCHFILT7           37            // Glitch filter
#define HWDIGIN_GLITCHFILT8           38            // Glitch filter
#define HWDIGIN_GLITCHFILT9           39            // Glitch filter
#define HWDIGIN_GLITCHFILT10          40            // Glitch filter
#define HWDIGIN_GLITCHFILT11          41            // Glitch filter
#define HWDIGIN_GLITCHFILT12          42            // Glitch filter
#define HWDIGIN_GLITCHFILT13          43            // Glitch filter
#define HWDIGIN_GLITCHFILT14          44            // Glitch filter
#define HWDIGIN_GLITCHFILT15          45            // Glitch filter
#define HWDIGIN_GLITCHFILT16          46            // Glitch filter
#define HWDIGIN_GLITCHFILT17          47            // Glitch filter
#define HWDIGIN_GLITCHFILT18          48            // Glitch filter
#define HWDIGIN_GLITCHFILT19          49            // Glitch filter
#define HWDIGIN_GLITCHFILT20          50            // Glitch filter
#define HWDIGIN_GLITCHFILT21          51            // Glitch filter
#define HWDIGIN_GLITCHFILT22          52            // Glitch filter
#define HWDIGIN_GLITCHFILT23          53            // Glitch filter
#define HWDIGIN_GLITCHFILT24          54            // Glitch filter
#define HWDIGIN_GLITCHFILT25          55            // Glitch filter
#define HWDIGIN_GLITCHFILT26          56            // Glitch filter
#define HWDIGIN_GLITCHFILT27          57            // Glitch filter
#define HWDIGIN_GLITCHFILT28          58            // Glitch filter
#define HWDIGIN_GLITCHFILT29          59            // Glitch filter
#define HWDIGIN_GLITCHFILT30          60            // Glitch filter
#define HWDIGIN_GLITCHFILT31          61            // Glitch filter
#define HWDIGIN_GLITCHFILT32          62            // Glitch filter

//==========================================================================================================
// 0x4240 HWDIGOUT_PARAM - HW Digital Output Parameters
//==========================================================================================================
#define SDOINDEX_HWDIGOUT_PARAM                     0x4240
#define HWDIGOUT_PARAM_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_HWDIGOUT_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWDIGOUT_PARAM_SRCINDEX(modno,parno)        ((SDOINDEX_HWDIGOUT_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWDIGOUT_PARAM(modno,parno)                 Sysvar[HWDIGOUT_PARAM_INDEX((modno),(parno))]
#define HWDIGOUT_PARAM_MAX                          18

#define HWDIGOUT_MODE                 1             // Mode
#define HWDIGOUT_PISRC_MAP1           10            // Mapping entry for outputs
#define HWDIGOUT_PISRC_MAP2           11            // Mapping entry for outputs
#define HWDIGOUT_PISRC_MAP3           12            // Mapping entry for outputs
#define HWDIGOUT_PISRC_MAP4           13            // Mapping entry for outputs
#define HWDIGOUT_PISRC_BIT1           14            // Source for input signal
#define HWDIGOUT_PISRC_BIT2           15            // Source for input signal
#define HWDIGOUT_PISRC_BIT3           16            // Source for input signal
```

```
#define HWDIGOUT_PISRC_BIT4            17          // Source for input signal

//=================================================================================================
// 0x4280 HWANIN_PARAM - HW Analog Input Parameters
//=================================================================================================
#define SDOINDEX_HWANIN_PARAM                      0x4280
#define HWANIN_PARAM_INDEX(modno,parno)            ((0x01000000 | (SDOINDEX_HWANIN_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWANIN_PARAM_SRCINDEX(modno,parno)         ((SDOINDEX_HWANIN_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWANIN_PARAM(modno,parno)                  Sysvar[HWANIN_PARAM_INDEX((modno),(parno))]
#define HWANIN_PARAM_MAX                           33

#define HWANIN_MODE                    1           // not used
#define HWANIN_OFFSET                  30          // Offset which will be added to the output value
#define HWANIN_FILTTIME                31          // Filter time for analogue input
#define HWANIN_SCALING                 32          // not used

//=================================================================================================
// 0x42C0 HWANOUT_PARAM - HW Analog Output Parameters
//=================================================================================================
#define SDOINDEX_HWANOUT_PARAM                     0x42C0
#define HWANOUT_PARAM_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_HWANOUT_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWANOUT_PARAM_SRCINDEX(modno,parno)        ((SDOINDEX_HWANOUT_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWANOUT_PARAM(modno,parno)                 Sysvar[HWANOUT_PARAM_INDEX((modno),(parno))]
#define HWANOUT_PARAM_MAX                          31

#define HWANOUT_MODE                   1           // Mode
#define HWANOUT_PISRC_VALUE            10          // Index and subindex of source
#define HWANOUT_OFFSET                 30          // Value which is added to analogue value

//=================================================================================================
// 0x4300 HWSIGBUS_PARAM - HW Signalbridges/bus ENC-REF Parameters
//=================================================================================================
#define SDOINDEX_HWSIGBUS_PARAM                    0x4300
#define HWSIGBUS_PARAM_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_HWSIGBUS_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWSIGBUS_PARAM_SRCINDEX(modno,parno)       ((SDOINDEX_HWSIGBUS_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWSIGBUS_PARAM(modno,parno)                Sysvar[HWSIGBUS_PARAM_INDEX((modno),(parno))]
#define HWSIGBUS_PARAM_MAX                         11

#define HWBRENCREF_PISRC_ENCSIG        10          // Source encoder signal

//=================================================================================================
// 0x4310 HWBRREFENC_PARAM - HW Signalbridges/bus REF-ENC Parameters
//=================================================================================================
#define SDOINDEX_HWBRREFENC_PARAM                  0x4310
#define HWBRREFENC_PARAM_INDEX(modno,parno)        ((0x01000000 | (SDOINDEX_HWBRREFENC_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWBRREFENC_PARAM_SRCINDEX(modno,parno)     ((SDOINDEX_HWBRREFENC_PARAM<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWBRREFENC_PARAM(modno,parno)              Sysvar[HWBRREFENC_PARAM_INDEX((modno),(parno))]
#define HWBRREFENC_PARAM_MAX                       12

#define HWBRREFENC_PISRC_REFSIGA       10          // Source for encoder signal A
#define HWBRREFENC_PISRC_REFSIGB       11          // Source for encoder signal B
```

```
//==========================================================================================================
// 0x4320 HWREFBUS_PARAM - HW Reference Bus Parameters
//==========================================================================================================
#define SDOINDEX_HWREFBUS_PARAM                     0x4320
#define HWREFBUS_PARAM_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_HWREFBUS_PARAM<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWREFBUS_PARAM_SRCINDEX(modno,parno)        ((SDOINDEX_HWREFBUS_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWREFBUS_PARAM(modno,parno)                 Sysvar[HWREFBUS_PARAM_INDEX((modno),(parno))]
#define HWREFBUS_PARAM_MAX                          32

//==========================================================================================================
// 0x4400 BUSMOD_PARAM - Bus Module Parameters
//==========================================================================================================
#define SDOINDEX_BUSMOD_PARAM                       0x4400
#define BUSMOD_PARAM_INDEX(modno,parno)             ((0x01000000 | (SDOINDEX_BUSMOD_PARAM<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define BUSMOD_PARAM_SRCINDEX(modno,parno)          ((SDOINDEX_BUSMOD_PARAM<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define BUSMOD_PARAM(modno,parno)                   Sysvar[BUSMOD_PARAM_INDEX((modno),(parno))]
#define BUSMOD_PARAM_MAX                            69

#define BUSMOD_MODE                    1            // Activate / Deactivate
#define BUSMOD_TYPE                    2            // Parameter preset
#define BUSMOD_TYPE_AXES               3            // Axes number used for source setting.
#define BUSMOD_PISRC_INPUT1            10           // Index and subindex of Input value
#define BUSMOD_PISRC_INPUT2            11           // Index and subindex of Input value
#define BUSMOD_PISRC_INPUT3            12           // Index and subindex of Input value
#define BUSMOD_PISRC_INPUT4            13           // Index and subindex of Input value
#define BUSMOD_PISRC_INPUT5            14           // Index and subindex of Input value
#define BUSMOD_PISRC_INPUT6            15           // Index and subindex of Input value
#define BUSMOD_PISRC_INPUT7            16           // Index and subindex of Input value
#define BUSMOD_PISRC_INPUT8            17           // Index and subindex of Input value
#define BUSMOD_LEN_TXPDO1              22           // Length of TxPDO in Bytes
#define BUSMOD_LEN_TXPDO2              23           // Length of TxPDO in Bytes
#define BUSMOD_LEN_TXPDO3              24           // Length of TxPDO in Bytes
#define BUSMOD_LEN_TXPDO4              25           // Length of TxPDO in Bytes
#define BUSMOD_LEN_RXPDO1              26           // Length of RxPDO in Bytes
#define BUSMOD_LEN_RXPDO2              27           // Length of RxPDO in Bytes
#define BUSMOD_LEN_RXPDO3              28           // Length of RxPDO in Bytes
#define BUSMOD_LEN_RXPDO4              29           // Length of RxPDO in Bytes
#define BUSMOD_BUSTYPE                 30           // Bus type
#define BUSMOD_BUSNO                   31           // Bus number
#define BUSMOD_ID                      32           // Slave node id
#define BUSMOD_SYNC                    33           // Sync interval (only for CAN)
#define BUSMOD_GUARDTIME               35           // Guard interval
#define BUSMOD_INHIBITTIME             36           // Inhibit time
#define BUSMOD_EVENTTIME               37           // Event time
#define BUSMOD_TXOFFSET                38           // Offset of the TX PDO
#define BUSMOD_RXOFFSET                39           // Offset of the RX PDO
#define BUSMOD_TXMAP_INPUT1            41           // Mapping for TX PDOs (send to bus)
#define BUSMOD_TXMAP_INPUT2            42           // Mapping for TX PDOs (send to bus)
#define BUSMOD_TXMAP_INPUT3            43           // Mapping for TX PDOs (send to bus)
#define BUSMOD_TXMAP_INPUT4            44           // Mapping for TX PDOs (send to bus)
#define BUSMOD_TXMAP_INPUT5            45           // Mapping for TX PDOs (send to bus)
#define BUSMOD_TXMAP_INPUT6            46           // Mapping for TX PDOs (send to bus)
#define BUSMOD_TXMAP_INPUT7            47           // Mapping for TX PDOs (send to bus)
#define BUSMOD_TXMAP_INPUT8            48           // Mapping for TX PDOs (send to bus)
#define BUSMOD_RXMAP_POVALUE1          61           // Mapping for RX PDOs (receive from bus)
#define BUSMOD_RXMAP_POVALUE2          62           // Mapping for RX PDOs (receive from bus)
```

```
#define BUSMOD_RXMAP_POVALUE3          63          // Mapping for RX PDOs (receive from bus)
#define BUSMOD_RXMAP_POVALUE4          64          // Mapping for RX PDOs (receive from bus)
#define BUSMOD_RXMAP_POVALUE5          65          // Mapping for RX PDOs (receive from bus)
#define BUSMOD_RXMAP_POVALUE6          66          // Mapping for RX PDOs (receive from bus)
#define BUSMOD_RXMAP_POVALUE7          67          // Mapping for RX PDOs (receive from bus)
#define BUSMOD_RXMAP_POVALUE8          68          // Mapping for RX PDOs (receive from bus)

    // Values for BUSMOD_MODE:
    #define BUSMOD_MODE_DEACTIVATE         0       // Deactivate
    #define BUSMOD_MODE_ACTIVATE           1       // Activate
    #define BUSMOD_MODE_ACTIVATE_NOSTOP    2       // Activate (only deleted in case of 0 or newstart mocexe)

    // Values for BUSMOD_BUSTYPE:
    #define BUSMOD_BUSTYPE_CAN             0       // CAN bus
    #define BUSMOD_BUSTYPE_ECAT_S          1       // EtherCAT Slave (not implemented yet)
    #define BUSMOD_BUSTYPE_ECAT_M          2       // EtherCAT Master

    // Values for BUSMOD_BUSNO:
    #define BUSMOD_BUSNO_CAN1              0       // CAN bus number 1
    #define BUSMOD_BUSNO_CAN2              1       // CAN bus number 2

//==========================================================================================================
// 0x4800 HWAMP_PROCESS - HW Amplifier Process Data
//==========================================================================================================
#define SDOINDEX_HWAMP_PROCESS                     0x4800
#define HWAMP_PROCESS_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_HWAMP_PROCESS<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWAMP_PROCESS_SRCINDEX(modno,parno)        ((SDOINDEX_HWAMP_PROCESS<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWAMP_PROCESS(modno,parno)                 Sysvar[HWAMP_PROCESS_INDEX((modno),(parno))]
#define HWAMP_PROCESS_MAX                          86

#define PO_HWAMP_STATUS                1           // Status word of the axis
#define PO_HWAMP_CURRENT               2           // Actual current
#define PO_HWAMP_HALLPOS               6           // Hall position
#define PO_HWAMP_SUPPLYVOLTAGE         8           // not used
#define PO_HWAMP_POWERFAIL             9           // Use PO_HWAMP_STATUS bit UNDERVOLTAGE
#define PO_HWAMP_USERDEFINED_DATA0     13          // The value in this process data can be selected by the
HWAMP_USERDEFINED_DATA parameters (only MACS5)
#define PO_HWAMP_USERDEFINED_DATA1     14          // The value in this process data can be selected by the
HWAMP_USERDEFINED_DATA parameters (only MACS5)
#define PO_HWAMP_USERDEFINED_DATA2     15          // The value in this process data can be selected by the
HWAMP_USERDEFINED_DATA parameters (only MACS5)
#define PO_HWAMP_USERDEFINED_DATA3     16          // The value in this process data can be selected by the
HWAMP_USERDEFINED_DATA parameters (only MACS5)
#define PO_HWAMP_TEMP1                 17          // Temperature Value 1 (On  board)
#define PO_HWAMP_TEMP2                 18          // Temperature Value 2 (On  board, in case of one sensor:
equal PO_HWAMP_TEMP1)
#define PO_HWAMP_VOLT1                 19          // Voltage Value 1 (Power voltage at amplifier)
#define PO_HWAMP_VOLT2                 20          // Voltage Value 2 (Logic voltage, 24V)
#define PO_HWAMP_PABS                  21          // absolute position used to determine electrical position
#define PO_HWAMP_PABSVEL               22          // Velocity
#define PO_HWAMP_HALL_POSEL            23          // Hall electrical position
#define PO_HWAMP_HALL_POSRAW           24          // Hall raw value
#define PO_HWAMP_ELPOS_POSEL           25          // Electrical position in quadcounts
#define PO_HWAMP_ELPOS_POSENC          26          // Last encoder position received
#define PO_HWAMP_COSSIN_COS            27          // Cosine value used for the transformation from ab to dq
#define PO_HWAMP_COSSIN_SIN            28          // Sine value used for the transformation from ab to dq
#define PO_HWAMP_VEL_VEL               29          // Velocity[rpm] = PO_HWAMP_VEL_VEL * 2 * HWAMP_MAXRPM /
0x7FFF
```

```
#define PO_HWAMP_VEL_VELRAW            30          // Velocity
#define PO_HWAMP_IPH_U                 31          // Phase current U
#define PO_HWAMP_IPH_V                 32          // Phase current V
#define PO_HWAMP_IPH_W                 33          // Phase current W
#define PO_HWAMP_IPHFIL_U              34          // Deprecated, Phase current U filtered
#define PO_HWAMP_IPHFIL_V              35          // Deprecated, Phase current V filtered
#define PO_HWAMP_IPHFIL_W              36          // Deprecated, Phase current W filtered
#define PO_HWAMP_IPHI2T_U              37          // Deprecated, Actual I2T value U
#define PO_HWAMP_IPHI2T_V              38          // Deprecated, Actual I2T value V
#define PO_HWAMP_IPHI2T_W              39          // Deprecated, Actual I2T value W
#define PO_HWAMP_IDQ_D                 40          // Currents in d coordinate used by PMSM or Stepper current
controller
#define PO_HWAMP_IDQ_Q                 41          // Currents in q coordinate used by PMSM or Stepper current
controller
#define PO_HWAMP_DCPH_U                42          // Duty cycle in ph coordinates, -0x7FFF...0x7FFF
#define PO_HWAMP_DCPH_V                43          // Duty cycle in ph coordinates, -0x7FFF...0x7FFF
#define PO_HWAMP_DCPH_W                44          // Duty cycle in ph coordinates, -0x7FFF...0x7FFF
#define PO_HWAMP_DCDQ_D                45          // Duty cycle in d coordinate, -0x7FFF...0x7FFF
#define PO_HWAMP_DCDQ_Q                46          // Duty cycle in q coordinate, -0x7FFF...0x7FFF
#define PO_HWAMP_ENPH_U                47          // Enable/disable halfbridge U
#define PO_HWAMP_ENPH_V                48          // Enable/disable halfbridge V
#define PO_HWAMP_ENPH_W                49          // Enable/disable halfbridge W
#define PO_HWAMP_VELPI_ACTUAL          50          // Same as PO_HWAMP_VEL_VEL
#define PO_HWAMP_VELPI_REF             51          // Reference of velocity controller. Reference value[rpm] =
PO_HWAMP_VELPI_REF * 2 * HWAMP_MAXRPM / 0x7FFF
#define PO_HWAMP_VELPI_ERR             52          // Actual regulation error of velocity controller. Error[rpm]
= PO_HWAMP_VELPI_ERR * 2 * HWAMP_MAXRPM / 0x7FFF
#define PO_HWAMP_VELPI_OUT             53          // Actual output of velocity controller. Output[rpm] =
PO_HWAMP_VELPI_OUT * 2 * HWAMP_MAXRPM / 0x7FFF
#define PO_HWAMP_VELPI_ACTINTSUM       54          // Actual integrated sum of velocity controller. Sum[rpm] =
PO_HWAMP_VELPI_ACTINTSUM * 2 * HWAMP_MAXRPM / 0x7FFF
#define PO_HWAMP_CURPI0_ACTUAL         55          // Actual value for current controller
#define PO_HWAMP_CURPI0_REF            56          // Reference value for current controller 0
#define PO_HWAMP_CURPI0_ERR            57          // Actual regulation error of current controller 0
#define PO_HWAMP_CURPI0_OUT            58          // Actual output of current controller 0
#define PO_HWAMP_CURPI0_ACTINTSUM      59          // Actual integrated sum of current controller 0
#define PO_HWAMP_CURPI1_ACTUAL         60          // Actual value for current controller 1
#define PO_HWAMP_CURPI1_REF            61          // Reference value for current controller 1
#define PO_HWAMP_CURPI1_ERR            62          // Actual regulation error of current controller 1
#define PO_HWAMP_CURPI1_OUT            63          // Actual output of current controller 1
#define PO_HWAMP_CURPI1_ACTINTSUM      64          // Actual integrated sum of current controller 1
#define PO_HWAMP_SET1_0                65          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET1_1                66          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET1_2                67          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET1_3                68          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET1_4                69          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET1_5                70          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET2_0                71          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET2_1                72          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET2_2                73          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET2_3                74          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET2_4                75          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_SET2_5                76          // Value depends on HWAMP_DEFINE_PO_DATA_SET
#define PO_HWAMP_ELPOS_LAG             77          // Electrical position lag compensation value
#define PO_HWAMP_ELPOS_DELTA           78          // Electrical position lag actual value
#define PO_HWAMP_STEPPOS_POSEL         79          // Stepper Motor electrical position in cossin units
#define PO_HWAMP_REF                   80          // Reference value
#define PO_HWAMP_FF                    81          // Feed forward component
#define PO_HWAMP_STEPPOS_CURLOOPINDEX  82          // Index counting from 0 to number of current loops within a
millisecond-loop
```

```
#define PO_HWAMP_HALL_ERRORCOUNT       83          // HALL error count
#define PO_HWAMP_TEMP_MOT              84          // Motor's temperature
#define PO_HWAMP_TOTAL_CURRENT         85          // Sum of alll PO_HWAMP_CURRENT

    // Values for PO_HWAMP_STATUS:
    #define PO_HWAMP_STATUS_ENABLED        0x00001  // Axis is enabled, its half bridges cannot be assigned to
other axes
    #define PO_HWAMP_STATUS_RUNNING        0x00002  // Axis is running, velocity and/or current control is
executed
    #define PO_HWAMP_STATUS_UNDERVOLTAGE   0x00004  // DC supply voltage has fallen below the min. level
(HWAMP_VOLT_MIN) while axis is in RUNNING state
    #define PO_HWAMP_STATUS_OVERVOLTAGE    0x00008  // DC supply voltage has risen above the max. level
(HWAMP_VOLT_MAX)
    #define PO_HWAMP_STATUS_HW_OVERCURRENT_ABS 0x00010  // Error detected by the hardware-based overcurrent
protection of any axis
    #define PO_HWAMP_STATUS_SW_OVERCURRENT_ABS 0x00020  // Error detected by the software-based overcurrent
protection
    #define PO_HWAMP_STATUS_SW_OVERCURRENT_I2T 0x00040  // I2T protection has detected an overload incident
    #define PO_HWAMP_STATUS_OVERTEMPERATURE 0x00080  // Internal temperature has risen above the max. level
    #define PO_HWAMP_STATUS_DSP_LINK       0x00100  // MACS5 only: Communication problem between amplifier and
controller
    #define PO_HWAMP_STATUS_MOSFETDRIVER_SUPPLY 0x00200  // MOSFET driver supply failed
    #define PO_HWAMP_STATUS_ADC            0x00400  // MACS5 only: ADC failed
    #define PO_HWAMP_STATUS_HALL           0x00800  // Hall sensor problem
    #define PO_HWAMP_STATUS_HALL_ANGLE     0x01000  // Hall or encoder sensor problem
    #define PO_HWAMP_STATUS_NOT_YET_READY  0x10000  // Only MiniMACS6: Axis not yet ready. After setting COMMTYPE,
it takes 7ms for calibration. Do not call AxisControll() before!

    // Values for PO_HWAMP_TEMP1:
    #define PO_HWAMP_TEMP_INVALID          0x7FFF   // Temperature is invalid. Temperature sensor may be not
connected or defect.

    // Values for PO_HWAMP_TEMP2:
    #define PO_HWAMP__TEMP_INVALID         0x7FFF   // Temperature is invalid. Temperature sensor may be not
connected or defect.

    // Values for PO_HWAMP_HALL_ERRORCOUNT:
    #define PO_HWAMP_HALL_ERRORCOUNT_MAX   0xFFFF   // Max value of PO_HWAMP_HALL_ERRORCOUNT

    // Values for PO_HWAMP_TEMP_MOT:
    #define PO_HWAMP_TEMP_INVALID          0x7FFF   // Temperature is invalid. Temperature sensor may be not
connected or defect.

//==========================================================================================================
// 0x4840 HWENC_PROCESS - HW Encoder Process Data
//==========================================================================================================
#define SDOINDEX_HWENC_PROCESS                      0x4840
#define HWENC_PROCESS_INDEX(modno,parno)            ((0x01000000 | (SDOINDEX_HWENC_PROCESS<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWENC_PROCESS_SRCINDEX(modno,parno)         ((SDOINDEX_HWENC_PROCESS<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWENC_PROCESS(modno,parno)                  Sysvar[HWENC_PROCESS_INDEX((modno),(parno))]
#define HWENC_PROCESS_MAX                           2

#define PO_HWENCODER_ERROR             1            // Encoder error bits

    // Values for PO_HWENCODER_ERROR:
    #define HWENCODER_ERROR_CRC            0x01     // CRC check fault
    #define HWENCODER_ERROR_SYNC           0x02     // EnDat sync fault
    #define HWENCODER_ERROR_TRANSFERTIME   0x04     // Too little time to transfer data, increase encoder
```

```
frequency
    #define HWENCODER_ERROR_SIGNAL         0x08     // Encoder signal quality monitoring

//==========================================================================================
// 0x4870 HWHALL_PROCESS - HW Hall Sensor Process Data
//==========================================================================================
#define SDOINDEX_HWHALL_PROCESS                     0x4870
#define HWHALL_PROCESS_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_HWHALL_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWHALL_PROCESS_SRCINDEX(modno,parno)        ((SDOINDEX_HWHALL_PROCESS<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWHALL_PROCESS(modno,parno)                 Sysvar[HWHALL_PROCESS_INDEX((modno),(parno))]
#define HWHALL_PROCESS_MAX                          4

#define PO_HWHALL_STATUS               1            // Digital input status of HALL sensor (3 bits, 0...7)
#define PO_HWHALL_POS                  2            // Position value (incrementing/decrementing)
#define PO_HWHALL_VEL                  3            // Velocity value

//==========================================================================================
// 0x4880 HWCOUNTINC_PROCESS - HW Counter-inc Process Data
//==========================================================================================
#define SDOINDEX_HWCOUNTINC_PROCESS                 0x4880
#define HWCOUNTINC_PROCESS_INDEX(modno,parno)       ((0x01000000 | (SDOINDEX_HWCOUNTINC_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWCOUNTINC_PROCESS_SRCINDEX(modno,parno)    ((SDOINDEX_HWCOUNTINC_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWCOUNTINC_PROCESS(modno,parno)             Sysvar[HWCOUNTINC_PROCESS_INDEX((modno),(parno))]
#define HWCOUNTINC_PROCESS_MAX                      14

#define PO_HWCNTINC_VALUE              1            // Output of counter module (position value)
#define PO_HWCNTINC_VELOCITY           2            // This value returns the position difference per millisecond
#define PO_HWCNTINC_STATUS             3            // not used (only legacy bits for Zoller)
#define PO_HWCNTINC_LASTVALUE          4            // Last value of the counter
#define PO_HWCNTINC_SINCOSPART         5            // Analog part of the SinCos Signal
#define PO_HWCNTINC_INCPART            8            // Incremental part of a Sin Cos signal
#define PO_HWCNTINC_SINCOS_DEBUG       9            // Debug information for SinCos Encoder
#define PO_HWCNTINC_SINCOS_DEBUG2      10           // Debug Information for SinCos Encoder
#define PO_HWCNTINC_SINCOS_AB_VAL      11           // Analog SinCos values -  Channels A & B
#define PO_HWCNTINC_SINCOS_AB_MIN      12           // Minimum measured SinCos peak values - Channels A & B
#define PO_HWCNTINC_SINCOS_AB_MAX      13           // Maximum measured SinCos peak values - Channels A & B

//==========================================================================================
// 0x48C0 HWCOUNTABS_PROCESS - HW Counter-abs Process Data
//==========================================================================================
#define SDOINDEX_HWCOUNTABS_PROCESS                 0x48C0
#define HWCOUNTABS_PROCESS_INDEX(modno,parno)       ((0x01000000 | (SDOINDEX_HWCOUNTABS_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWCOUNTABS_PROCESS_SRCINDEX(modno,parno)    ((SDOINDEX_HWCOUNTABS_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWCOUNTABS_PROCESS(modno,parno)             Sysvar[HWCOUNTABS_PROCESS_INDEX((modno),(parno))]
#define HWCOUNTABS_PROCESS_MAX                      3

#define PO_HWCNTABS_VALUE              1            // Position output of absolute encoder Counter
#define PO_HWCNTABS_RAW                2            // Raw databits of the transmission (Not on MACS5)

//==========================================================================================
// 0x4900 HWCOUNTUNI_PROCESS - HW Counter-abs Process Data
//==========================================================================================
#define SDOINDEX_HWCOUNTUNI_PROCESS                 0x4900
#define HWCOUNTUNI_PROCESS_INDEX(modno,parno)       ((0x01000000 | (SDOINDEX_HWCOUNTUNI_PROCESS<<8)) | (((long)
```

```
(modno))<<8) | ((long) (parno)))
#define HWCOUNTUNI_PROCESS_SRCINDEX(modno,parno)    ((SDOINDEX_HWCOUNTUNI_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWCOUNTUNI_PROCESS(modno,parno)             Sysvar[HWCOUNTUNI_PROCESS_INDEX((modno),(parno))]
#define HWCOUNTUNI_PROCESS_MAX                      3

#define PO_HWCNTUNI_VALUE              1            // Output of counter module
#define PO_HWCNTUNI_VELOCITY           2            // not used

//==========================================================================================================
// 0x4920 HWCMPUNI_PROCESS - HW Comparator-universal Process Data
//==========================================================================================================
#define SDOINDEX_HWCMPUNI_PROCESS                   0x4920
#define HWCMPUNI_PROCESS_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_HWCMPUNI_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWCMPUNI_PROCESS_SRCINDEX(modno,parno)      ((SDOINDEX_HWCMPUNI_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWCMPUNI_PROCESS(modno,parno)               Sysvar[HWCMPUNI_PROCESS_INDEX((modno),(parno))]
#define HWCMPUNI_PROCESS_MAX                        2

//==========================================================================================================
// 0x4940 HWLATCH_PROCESS - HW Latch Register Process Data
//==========================================================================================================
#define SDOINDEX_HWLATCH_PROCESS                    0x4940
#define HWLATCH_PROCESS_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_HWLATCH_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWLATCH_PROCESS_SRCINDEX(modno,parno)       ((SDOINDEX_HWLATCH_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWLATCH_PROCESS(modno,parno)                Sysvar[HWLATCH_PROCESS_INDEX((modno),(parno))]
#define HWLATCH_PROCESS_MAX                         10

#define PO_HWLATCH_VALUE               1            // Output of latch module
#define PO_HWLATCH_FLAG                2            // Flag signal that a new latch happened
#define PO_HWLATCH_COUNT               3            // Number of latches occurred up until now
#define PO_HWLATCH_AMOUNT_IN_FIFO      4            // Number of latches in FIFO
#define PO_HWLATCH_FIFOREAD            5            // Latch position read from FIFO
#define PO_HWLATCH_FIFO_WRITEINDEX     6            // Write index position of FIFO
#define PO_HWLATCH_FIFO_READINDEX      7            // Read index position of FIFO
#define PO_HWLATCH_FIFO_STAT           8            // Status of HW Latch FIFO
#define PO_HWLATCH_FIFO_PEEK           9            // Peek the latest element from the FIFO

//==========================================================================================================
// 0x4980 HWSIGGEN_PROCESS - HW Signalgen (Enc/Pulse) Process Data
//==========================================================================================================
#define SDOINDEX_HWSIGGEN_PROCESS                   0x4980
#define HWSIGGEN_PROCESS_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_HWSIGGEN_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWSIGGEN_PROCESS_SRCINDEX(modno,parno)      ((SDOINDEX_HWSIGGEN_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWSIGGEN_PROCESS(modno,parno)               Sysvar[HWSIGGEN_PROCESS_INDEX((modno),(parno))]
#define HWSIGGEN_PROCESS_MAX                        4

//==========================================================================================================
// 0x49C0 HWDDS_PROCESS - HW DDS Process Data
//==========================================================================================================
#define SDOINDEX_HWDDS_PROCESS                      0x49C0
#define HWDDS_PROCESS_INDEX(modno,parno)            ((0x01000000 | (SDOINDEX_HWDDS_PROCESS<<8)) | (((long) (modno))
<<8) | ((long) (parno)))
#define HWDDS_PROCESS_SRCINDEX(modno,parno)         ((SDOINDEX_HWDDS_PROCESS<<8) | (((long) (modno))<<8) | ((long)
(parno)))
```

```
#define HWDDS_PROCESS(modno,parno)                  Sysvar[HWDDS_PROCESS_INDEX((modno),(parno))]
#define HWDDS_PROCESS_MAX                           1

//==========================================================================================
// 0x49D0 HWPWMGEN_PROCESS - HW PWM Generators Process Data
//==========================================================================================
#define SDOINDEX_HWPWMGEN_PROCESS                   0x49D0
#define HWPWMGEN_PROCESS_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_HWPWMGEN_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWPWMGEN_PROCESS_SRCINDEX(modno,parno)      ((SDOINDEX_HWPWMGEN_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWPWMGEN_PROCESS(modno,parno)               Sysvar[HWPWMGEN_PROCESS_INDEX((modno),(parno))]
#define HWPWMGEN_PROCESS_MAX                        2

#define PO_HWPWMGEN_VALUE              1            // Current Duty Cycle (0-100%)

//==========================================================================================
// 0x49E0 HWPULSGEN_PROCESS - HW Pulsgen Process Data
//==========================================================================================
#define SDOINDEX_HWPULSGEN_PROCESS                  0x49E0
#define HWPULSGEN_PROCESS_INDEX(modno,parno)        ((0x01000000 | (SDOINDEX_HWPULSGEN_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWPULSGEN_PROCESS_SRCINDEX(modno,parno)     ((SDOINDEX_HWPULSGEN_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWPULSGEN_PROCESS(modno,parno)              Sysvar[HWPULSGEN_PROCESS_INDEX((modno),(parno))]
#define HWPULSGEN_PROCESS_MAX                       1

//==========================================================================================
// 0x49F0 HWSHIFTREG_PROCESS - HW Shift Register Process Data
//==========================================================================================
#define SDOINDEX_HWSHIFTREG_PROCESS                 0x49F0
#define HWSHIFTREG_PROCESS_INDEX(modno,parno)       ((0x01000000 | (SDOINDEX_HWSHIFTREG_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWSHIFTREG_PROCESS_SRCINDEX(modno,parno)    ((SDOINDEX_HWSHIFTREG_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWSHIFTREG_PROCESS(modno,parno)             Sysvar[HWSHIFTREG_PROCESS_INDEX((modno),(parno))]
#define HWSHIFTREG_PROCESS_MAX                      1

//==========================================================================================
// 0x4A00 HWDIGIN_PROCESS - HW Digital Input Process Data
//==========================================================================================
#define SDOINDEX_HWDIGIN_PROCESS                    0x4A00
#define HWDIGIN_PROCESS_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_HWDIGIN_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWDIGIN_PROCESS_SRCINDEX(modno,parno)       ((SDOINDEX_HWDIGIN_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWDIGIN_PROCESS(modno,parno)                Sysvar[HWDIGIN_PROCESS_INDEX((modno),(parno))]
#define HWDIGIN_PROCESS_MAX                         8

#define PO_HWDIGIN_VALLONG             1            // Return 32 bit input states (1 bit for each digital input
signal)
#define PO_HWDIGIN_VALWORD1            2            // Returns 16 bit input state (1 bit for each digtal input)
#define PO_HWDIGIN_VALWORD2            3            // Returns 16 bit input state (1 bit for each digtal input)
#define PO_HWDIGIN_VALBYTE1            4            // Returns 8 bit input state (1 bit for each digital input)
#define PO_HWDIGIN_VALBYTE2            5            // Returns 8 bit input state (1 bit for each digital input)
#define PO_HWDIGIN_VALBYTE3            6            // Returns 8 bit input state (1 bit for each digital input)
#define PO_HWDIGIN_VALBYTE4            7            // Returns 8 bit input state (1 bit for each digital input)

//==========================================================================================
// 0x4A40 HWDIGOUT_PROCESS - HW Digital Output Process Data
```

```
//==========================================================================================
#define SDOINDEX_HWDIGOUT_PROCESS                    0x4A40
#define HWDIGOUT_PROCESS_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_HWDIGOUT_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWDIGOUT_PROCESS_SRCINDEX(modno,parno)       ((SDOINDEX_HWDIGOUT_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWDIGOUT_PROCESS(modno,parno)                Sysvar[HWDIGOUT_PROCESS_INDEX((modno),(parno))]
#define HWDIGOUT_PROCESS_MAX                         3

#define PO_HWDIGOUT_VALUE              1             // The current digital output state (32 bits, each bit
represents one digital output)
#define PO_HWDIGOUT_FORCEVALUE         2             // The output value to be forced.

//==========================================================================================
// 0x4A80 HWANIN_PROCESS - HW Analog Input Process Data
//==========================================================================================
#define SDOINDEX_HWANIN_PROCESS                      0x4A80
#define HWANIN_PROCESS_INDEX(modno,parno)            ((0x01000000 | (SDOINDEX_HWANIN_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWANIN_PROCESS_SRCINDEX(modno,parno)         ((SDOINDEX_HWANIN_PROCESS<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define HWANIN_PROCESS(modno,parno)                  Sysvar[HWANIN_PROCESS_INDEX((modno),(parno))]
#define HWANIN_PROCESS_MAX                           5

#define PO_HWANIN_VALUE                1             // Filtered analog input value
#define PO_HWANIN_UNFILT               2             // Unfiltered analog input value
#define PO_HWANIN_MAXVAL               3             // Maximum value of the filtered value up to now
#define PO_HWANIN_MINVAL               4             // Minimum value of the filtered value up to now

//==========================================================================================
// 0x4AC0 HWANOUT_PROCESS - HW Analog Output Process Data
//==========================================================================================
#define SDOINDEX_HWANOUT_PROCESS                     0x4AC0
#define HWANOUT_PROCESS_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_HWANOUT_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWANOUT_PROCESS_SRCINDEX(modno,parno)        ((SDOINDEX_HWANOUT_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWANOUT_PROCESS(modno,parno)                 Sysvar[HWANOUT_PROCESS_INDEX((modno),(parno))]
#define HWANOUT_PROCESS_MAX                          3

#define PO_HWANOUT_VALUE               1             //
#define PO_HWANOUT_FORCEVALUE          2             // Enable write by setting HWANOUT_MODE to 1

//==========================================================================================
// 0x4B00 HWSIGBUS_PROCESS - HW Signalbridges/bus ENC-REF Process Data
//==========================================================================================
#define SDOINDEX_HWSIGBUS_PROCESS                    0x4B00
#define HWSIGBUS_PROCESS_INDEX(modno,parno)          ((0x01000000 | (SDOINDEX_HWSIGBUS_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWSIGBUS_PROCESS_SRCINDEX(modno,parno)       ((SDOINDEX_HWSIGBUS_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWSIGBUS_PROCESS(modno,parno)                Sysvar[HWSIGBUS_PROCESS_INDEX((modno),(parno))]
#define HWSIGBUS_PROCESS_MAX                         1

//==========================================================================================
// 0x4B10 HWBRREFENC_PROCESS - HW Signalbridges/bus REF-ENC Process Data
//==========================================================================================
#define SDOINDEX_HWBRREFENC_PROCESS                  0x4B10
#define HWBRREFENC_PROCESS_INDEX(modno,parno)        ((0x01000000 | (SDOINDEX_HWBRREFENC_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
```

```
#define HWBRREFENC_PROCESS_SRCINDEX(modno,parno)    ((SDOINDEX_HWBRREFENC_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWBRREFENC_PROCESS(modno,parno)             Sysvar[HWBRREFENC_PROCESS_INDEX((modno),(parno))]
#define HWBRREFENC_PROCESS_MAX                      1

//=============================================================================================================
// 0x4B20 HWREFBUS_PROCESS - HW Reference Bus Process Data
//=============================================================================================================
#define SDOINDEX_HWREFBUS_PROCESS                   0x4B20
#define HWREFBUS_PROCESS_INDEX(modno,parno)         ((0x01000000 | (SDOINDEX_HWREFBUS_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define HWREFBUS_PROCESS_SRCINDEX(modno,parno)      ((SDOINDEX_HWREFBUS_PROCESS<<8) | (((long) (modno))<<8) |
((long) (parno)))
#define HWREFBUS_PROCESS(modno,parno)               Sysvar[HWREFBUS_PROCESS_INDEX((modno),(parno))]
#define HWREFBUS_PROCESS_MAX                        3

#define HWREFBUS_PO_REG0              1             // RefVecBus bits 0-31
#define HWREFBUS_PO_REG1              2             // RefVecBus bits 32-63

//=============================================================================================================
// 0x4C00 BUSMOD_PROCESS - Bus Module Process Data
//=============================================================================================================
#define SDOINDEX_BUSMOD_PROCESS                     0x4C00
#define BUSMOD_PROCESS_INDEX(modno,parno)           ((0x01000000 | (SDOINDEX_BUSMOD_PROCESS<<8)) | (((long)
(modno))<<8) | ((long) (parno)))
#define BUSMOD_PROCESS_SRCINDEX(modno,parno)        ((SDOINDEX_BUSMOD_PROCESS<<8) | (((long) (modno))<<8) | ((long)
(parno)))
#define BUSMOD_PROCESS(modno,parno)                 Sysvar[BUSMOD_PROCESS_INDEX((modno),(parno))]
#define BUSMOD_PROCESS_MAX                          39

#define PO_BUSMOD_VALUE1              1             // Output value of the RX PDOs
#define PO_BUSMOD_VALUE2              2             // Output value of the RX PDOs
#define PO_BUSMOD_VALUE3              3             // Output value of the RX PDOs
#define PO_BUSMOD_VALUE4              4             // Output value of the RX PDOs
#define PO_BUSMOD_VALUE5              5             // Output value of the RX PDOs
#define PO_BUSMOD_VALUE6              6             // Output value of the RX PDOs
#define PO_BUSMOD_VALUE7              7             // Output value of the RX PDOs
#define PO_BUSMOD_VALUE8              8             // Output value of the RX PDOs
#define PO_BUSMOD_LASTERROR           23            // Last error number
#define PO_BUSMOD_LASTRXTIME          24            // Time of last PDO reception (PDO 1-4)
#define PO_BUSMOD_LASTTXTIME          25            // Time of last PDO sending (PDO1-4)
#define PO_BUSMOD_OBJ_TXPDO1          26            // Internal CanObj Number used for PDO
#define PO_BUSMOD_OBJ_TXPDO2          27            // Internal CanObj Number used for PDO
#define PO_BUSMOD_OBJ_TXPDO3          28            // Internal CanObj Number used for PDO
#define PO_BUSMOD_OBJ_TXPDO4          29            // Internal CanObj Number used for PDO
#define PO_BUSMOD_OBJ_RXPDO1          30            // Internal CanObj Number used for PDO
#define PO_BUSMOD_OBJ_RXPDO2          31            // Internal CanObj Number used for PDO
#define PO_BUSMOD_OBJ_RXPDO3          32            // Internal CanObj Number used for PDO
#define PO_BUSMOD_OBJ_RXPDO4          33            // Internal CanObj Number used for PDO
#define PO_BUSMOD_DEVTYPE             34            // Devicetype of connected slave node (only CAN)
#define PO_BUSMOD_TIMER_TXPDO1        35            // Internal counter for event and inhibit time
#define PO_BUSMOD_TIMER_TXPDO2        36            // Internal counter for event and inhibit time
#define PO_BUSMOD_TIMER_TXPDO3        37            // Internal counter for event and inhibit time
#define PO_BUSMOD_TIMER_TXPDO4        38            // Internal counter for event and inhibit time

//=============================================================================================================
```

12.6 Serial PDO

There is a special kind of PDO used for serial communication (USB, Ethernet, RS232). This is the PDO 512. The main use case of this PDO is the ApossIDE oscilloscope. But it can also be used in special applications as a way to transfer cyclic data to and from a host system.

The controllers support a certain amount of CAN or EtherCAT PDOs (usually 4 or 8), see Controller Dependent Feature List. The serial PDO is an additional one, so it can be used in parallel to the CAN or EtherCAT PDOs.

PDO 512 can be configured by the respective configuration SDOs foreseen for PDO 512, which are the SDOs 0x15FF, 0x17FF, 0x19FF, 0x1BFF.

Usually, the customer doesn't have to do anything with this PDO. This information is just for reference.

12.7 Version History

ApossIDE Versions

What's New in Version 7.00

Firmware Versions

What's New in Version 7.2.00
What's New in Version 8.0.00
What's New in Version 9.0.00

12.7.1 ApossIDE

12.7.1.1 What's New in Version 7.00

The most significant changes to the ApossIDE between version 6.13 and 7.00 are as follows:

Name Change

- Prior to version 7.00.00, this software was referred to as the "Aposs Integrated Development Environment", or "Aposs" for short. Starting with version 7.00.00, it will be referred to simply as the "ApossIDE".

Major New Features

- The ApossIDE now supports a subset of the C Programming Language (see ApossC Programming Language for details).

Command Changes

- The ApossC programming language provides the following new commands:
  AxisHomeStart
  AxisHomeStatus
  AxisHomeStop
  AxisMoving
  AxisStop
  AxisTargetReached
  CamArrayIndex
  CamArrayLen
  HallDetectStart

HallDetectStatus
MotorAlignStart
MotorAlignStatus
PathArrayIndex
EthernetGetConnectionStatus
EthernetOpenClient
EthernetOpenServer
EthernetReceiveTelegram
EthernetSendTelegram

Other ApossIDE Changes

- The F12 Command List is no longer available. Upgrading it to support advances in the ApossC and APOSS programming languages became too problematic and the Command List has now been removed.
- The **Controller** → **Memory** → **Save RAM** menu command that was used to save all programs, parameters, and arrays from RAM into EPROM, has been removed. The saving of program is managed by the **Controller** → **Programs** menu command. The saving of parameters is a task that should be done by the application program itself. The saving of arrays should be done either by the application program, by the CAM Editor, or by the Array Editor, whichever is appropriate.
- The **Controller** → **Memory** → **Delete EPROM** menu command that was used to reset all parameters and delete all arrays and programs (i.e. returning the controller to the original factory setting) has been removed. This can be done using the **Controller** → **Reset** → **All** menu command
- The **TESTSETP Oscilloscope** has been renamed to the **Record Oscilloscope**.
- The **Tune Oscilloscope** is no longer useful and has been removed.
- If an ApossC program is being compiled, then the compiler is now able to automatically calculate the stack requirements. See Compiler.
- The **Enable "Find in Files" toolbar buttons** option (see Options) has been removed. These toolbar buttons are now always enabled.
- Binary files are now always supported by the **Controller** → **Programs** menu command.
- The **Development** → **Compile to file** menu command is now always enabled.
- The **Binary file support** option in the **Settings** → **Options** dialog has been removed. This has been replaced with check boxes directly in the **Controller** → **Programs** dialog.
- Folding of the functions is enabled in the editor by default. It can be disabled in **Settings** → **Editor Settings**. Folding can be controlled individually for every function or Fold All/Unfold All from a **Edit** → **Folding** can be used to collapse/expand all functions.

12.7.2 Firmware

12.7.2.1 What's New in Version 8.0.00

Deprecated Axis Parameters

The new virtual and hardware module architecture was introduced in version 7.0.00. In order to provide backwards compatibility, it was necessary for many existing axis parameters to influence these new modules. For example, if ENCODERTYPE was set, then this also automatically set parameters in the HW_ENCODER and HW_AMP modules.

To cleanly separate axis parameters from virtual and hardware modules, the following axis parameters have been deprecated starting from firmware version 8.0.00. Although these parameters are currently still available, application programs which use them should be changed to use the corresponding virtual and hardware module parameters.

If a deprecated parameter is used in an application program, then the Edit Window will display the parameter keyword with an orange background as follows:

```
set encoder 1000
```

AMPCOMMTYPE
AMPENCNO
AMPENCRES
AMPENCRPM
AMPMAXCUR
AMPMODE
AMPPWMFREQ
CURFTIME
CURKILIM
CURKINT
CURKPROP
ENCCONTROL
ENCODER
ENCODERBAUD
ENCODERCLOCK
ENCODERDATLEN
ENCODERDELAY
ENCODERFREQ
ENCODERID
ENCODERPARITY
ENCODERTYPE
MENCCONTROL
MENCODERBAUD
MENCODERCLOCK
MENCODERDATLEN
MENCODERDELAY
MENCODERFREQ
MENCODERID
MENCODERPARITY
MENCODERTERM
MENCODERTYPE
SYNCMTYPM
SYNCMTYPS
VELKILIM
VELKINT
VELKPROP

12.7.2.2 What's New in Version 9.0.00

Support for ApossC

This Firmware Version fully supports ApossC language.

The legacy APOSS language is still supported for backwards compatibility, but it is recommended to change to ApossC if possible.

Removed Axis Parameters

Axis parameters which were deprecated in Version 8.0.0 have been removed for ApossC language. Use the corresponding virtual and hardware module parameters.

The removed parameters are:

AMPCOMMTYPE
AMPENCNO
AMPENCRES
AMPENCRPM
AMPMAXCUR
AMPMODE
AMPPWMFREQ
CURFTIME
CURKILIM
CURKINT
CURKPROP
ENCCONTROL
ENCODER
ENCODERBAUD
ENCODERCLOCK
ENCODERDATLEN
ENCODERDELAY
ENCODERFREQ
ENCODERID
ENCODERPARITY
ENCODERTYPE
MENCCONTROL
MENCODERBAUD
MENCODERCLOCK
MENCODERDATLEN
MENCODERDELAY
MENCODERFREQ
MENCODERID
MENCODERPARITY
MENCODERTERM
MENCODERTYPE
SYNCMTYPM
SYNCMTYPS
VELKILIM
VELKINT
VELKPROP

12.8 Controller Types

The specifications of each controller can be found in the respective hardware reference manual. Here is a list of individual properties of the controllers.

- MACS5
- MACS5-Compact
- MasterMACS
- MiniMACS
- MiniMACS6-AMP-4

The Controller Dependent Feature List shows various features that are available on the different controllers.

12.8.1 MACS5

Analog inputs The scaling for the analog inputs with the AnalogInput command is as follows:

No option mounted: 0 = 200mV, 32768 = 10V
The voltage range from 0-200mV cannot be measured due to hardware limitations. The scaling is set up so that it starts measuring at 200mV. The value will be 0 at 0-200 mV, and rising from there to 32768 at 10 V.

Option IO1 (3AOut/1AIn): 32768 = +10V, -32768 = -10V

Option IO2 (6AIn 13bit): 0 = 0V, 32768 = 10V

Analog outputs The scaling for the analog outputs (option board necessary) with the AnalogOutput command is as follows:

32768 = +10V

-32768 = -10V

Data Types The **double** datatype has a precision of 64bit.

The ApossIDE library functions (see Function Reference) only calculate with 32-bit precision.

Serial interface RS232 / RS485

Ethernet Ports SXP 23

Ethernet Ports SSP 23

12.8.2 MACS5-Compact

Analog inputs The scaling for the analog inputs with the AnalogInput command is as follows:

No option mounted: 0 = 200mV, 32768 = 10V
The voltage range from 0-200mV cannot be measured due to hardware limitations. The scaling is set up so that it starts measuring at 200mV. The value will be 0 at 0-200 mV, and rising from there to 32768 at 10 V.

Option IO1 (3AOut/1AIn): 32768 = +10V, -32768 = -10V

Option IO2 (6AIn 13bit): 0 = 0V, 32768 = 10V

Analog outputs The scaling for the analog outputs (option board necessary) with the AnalogOutput command is as follows:

32768 = +10V

-32768 = -10V

Data Types The **double** datatype has a precision of 64bit.

The ApossIDE library functions (see Function Reference) only calculate with 32-bit precision.

Analog inputs The scaling for the analog inputs with the AnalogInput command is as follows:

No option mounted: 0 = 200mV, 32768 = 10V
The voltage range from 0-200mV cannot be measured due to hardware limitations. The scaling is set up so that it starts measuring at 200mV. The value will be 0 at 0-200 mV, and rising from there to 32768 at 10 V.

Option IO1 (3AOut/1AIn): 32768 = +10V, -32768 = -10V

Option IO2 (6AIn 13bit): 0 = 0V, 32768 = 10V

Serial interface RS232 / RS485

Ethernet Ports SXP 23

Ethernet Ports SSP 23

12.8.3 MiniMACS

Analog inputs The scaling for the analog inputs with the AnalogInput command is as follows:

No option mounted: 0 = 200mV, 32768 = 10V

Option IO1 (3AOut/1AIn): 32767 = +10V, -32768 = -10V

Option IO2 (6AIn 13bit): 0 = 0V, 32767 = 10V

Analog outputs The scaling for the analog outputs (option board necessary) with the AnalogOutput command is as follows:

32767 = +10V

-32768 = -10V

Data Types The **double** datatype has a precision of 64bit.

The ApossIDE library functions (see Function Reference) only calculate with 32-bit precision.

Serial interface -

Ethernet Ports SXP 23

Ethernet Ports SSP 23

12.8.4 MasterMACS

Analog inputs The scaling for the analog inputs with the AnalogInput command is as follows:

0 = 0V, 32768 = 10V

Analog outputs -

Data Types The **double** datatype has a precision of 64bit.

The ApossIDE library functions (see Function Reference) will also calculate with 64-bit precision.

Serial interface RS232 / RS485

By default, the CAN2 interface is active. The RS485/ CAN2 interface cannot be used

Analog inputs The scaling for the analog inputs with the AnalogInput command is as follows:

0 = 0V, 32768 = 10V

concurrently. To activate RS485, SDO 0x2203 / 16 must be written with the value 0x000B0001.

Ethernet Ports SXP 23

Ethernet Ports SSP 23, 26, 1914, 1915, 1916, 1917, 1918

12.8.5 MiniMACS6-AMP-4

Analog inputs The scaling for the analog inputs with the AnalogInput command is as follows:

No option mounted: 0 = V, 32768 = 10V
The value will be 0 at 0V, and rising from there to 32768 at 10V.

Analog outputs -

Data Types The **double** datatype has a precision of 64bit.

The ApossIDE library functions (see Function Reference) will also calculate with 64-bit precision.

Serial interface -

Ethernet Ports SXP 23

Ethernet Ports SSP 23, 26

12.8.6 Controller Dependent Feature List

The tables are based on the latest firmware.

General Functionalities

General functionalities as an overview of communication features.

	MACS5	MACS5-AMP-1/2	MACS5-AMP3	MiniMACS	MasterMACS	MiniMACS6-AMP-4	PC-Encoderinterface	MicroMACS6
Webserver / FTP	X	X	X		X			
SD Card Functions	X	X			X			
USB Memory Stick Functions						X		
Memorydump command is non-blocking	X	X			X			
Ethernet SXP Protocol Port	23	23	23	23	23	23	23	23
Ethernet SSP Protocol Number of Sockets	1 Port	1 Port	1 Port	1 Port	7 Ports	2 Ports	2	2 Ports
Ethernet ApossC Commands (Number of Sockets)				X (1)	X (2)	X (2)	X (2)	X (2)
Ethernet ApossC Commands max. telegram size				30 bytes	1000 bytes	300 bytes	300 bytes	300 bytes
Encoder Signal output	X	X	X			X		
EtherCAT Master	X	X	X		X			
Kinematics	X	X	X		X	X		X
Fast Recording (24kHz)						X		
Powerdown Save	X	X	X		X	X	X	
EtherCAT Slave (FoE)	X (X)	X (X)			X	X		
Watchdog	X	X	X					

	MACS5	MACS5-AMP-1/2	MACS5-AMP3	MiniMACS	MasterMACS	MiniMACS6-AMP-4	PC-Encoderinterface	MicroMACS6
Emergency Reset	X	X		X	X	X		X
Real Time Clock					?			
Memorydump 0x2212 / 2 (Dump Log Ringbuffer)					X	X		
RS232	X	X			X			
RS485	X	X			X			
EtherCAT slave max PDO size (bytes)	340	340	340		340	200		
Serial SXP protocol max PDO size (bytes)	247	247	247	247	340	200	247	247
TimeHW unit	1us	1us	1us	1us	1.5625us	1us	1us	1us
HALL Sensor Velocity Measurement						X		
Number of CanOpen PDOs	4	4	4	4	8	8	8	8
Number of Virtual Math Modules					32	10	10	10

Features marked with 'X' are implemented.

Features marked with '?' can beimplemented on request.

MACS Amplifier

	MACS5-AMP1	MACS5-AMP2	MACS5-AMP3	MiniMACS6-AMP-4
HWAMP_COMMTYPE_DC	6	6	3	6
HWAMP_COMMTYPE_DC_TWIN	3	3	1	3
HWAMP_COMMTYPE_BLDC	4	4	2	4
HWAMP_COMMTYPE_BLDC_TWIN	2	2	1	2
HWAMP_COMMTYPE_PMSM	4	4	2	4
HWAMP_COMMTYPE_PMSM_TWIN	2	2	1	2
HWAMP_COMMTYPE_HALL_PMSM	?	?	?	4
HWAMP_COMMTYPE_HALL_PMSM_TWIN	?	?	?	2
HWAMP_COMMTYPE_STEP	3	3	1	3
HWAMP_COMMTYPE_STEP_TWIN	1	1	0	?
HWAMP_COMMTYPE_CHOPPER	1	1	1	1

Reading examples: One controller MiniMACS6 AMP-4 supports up to 4 EC-Motors when using commutation type HWAMP_COMMTYPE_PMSM

It is possible to combine different commutation types on the same controller. Example: One controller MiniMACS6 AMP-4 supports 3 DC-Motors when using commutation type HWAMP_COMMTYPE_DC and 2 EC-Motors when using HWAMP_COMMTYPE_HALL_PMSM. Specific information on each controller is given in the hardware reference manual.

Features marked with '?' can be implemented on request.

Limitation for PRINTF and SPRINTF command

	MACS5	MiniMACS	MasterMACS	MiniMACS6	MicroMACS6
Maximum number of parameters (after the format string)	10	5	10	10	10
Maximum length of buffer per parameter	30	30	150	30	30

Limitation for DIM arrays

	MACS5	MiniMACS	MasterMACS	MiniMACS6	MicroMACS6
Maximum number of arrays	300	300	1000	300	300

Encoder type support (HWENCODER_MODE)

The following table shows which encoder ports support which encoder types.

	MACS5 Ports	MiniMACS Ports	MasterMACS Ports	MiniMACS6 Ports	MicroMACS6 (no encoder)
HWENCODER_MODE_INCREMENTAL	1-6	1	1	1-4	
HWENCODER_MODE_INCROUTPUT	4-6				
HWENCODER_MODE_SSI_ACTIVE	4-6			1-4	
HWENCODER_MODE_SSI_PASSIVE	4-6				
HWENCODER_MODE_HIPERFACE_RX	4-6				
HWENCODER_MODE_HIPERFACE_TX	4-6				
HWENCODER_MODE_SINCOS	1-3			1-4	
HWENCODER_MODE_HALL	1-6			4 dedicated ports	
HWENCODER_MODE_ENDAT					

12.8.7 MicroMACS6

Analog inputs The scaling for the analog inputs with the AnalogInput command is as follows:

No option mounted: 0 = V, 32768 = 10V
The value will be 0 at 0V, and rising from there to 32768 at 10V.

Analog outputs -

Data Types The **double** datatype has a precision of 64bit.

The ApossIDE library functions (see Function Reference) will also calculate with 64-bit precision.

Serial interface -

Ethernet Port SXP 23

Ethernet Port SSP 23, 26

12.9 Developer Tools

12.9.1 Introduction to ZbMoc

What services are provided?

The ZbMoc Library offers a collection of simple to use functions for the

- Configuration
- Control
- Supervising

of zub controllers in a production environment. All of the required functions can be accessed by a Windows

application program.

The ZbMoc Library offers a consistent application programming interface (API) for communicating with the different controllers of zub machine control AG. The programming interface is independent of the hardware interfaces and bus systems used by each application and controller.

The slight differences in control and functional range resulting from different hardware and software versions of controllers are handled within the ZbMoc Library.

The library makes it possible to address a controller offering a new interface or bus system, without significant modifications to existing application programs.

The ZbMoc Library is a highly efficient tool which allows the application programmer to focus on the application programming itself rather than working through all details of bus systems, error handling, and so forth.

Developing with the Library

The ZbMoc Library is a Dynamic Link Library (DLL) available for 32-bit Windows (WIN32). It offers full driver support for all supported interfaces. Usage of the DLL requires that the ApossIDE software be installed. Installing the ApossIDE will correctly install and configure the low-level device drivers used by the ZbMoc Library.

Application development depends on the development environment used by your application program. For example, if Microsoft Visual Studio is used, it is sufficient to include the header 'ZbMoc.h' and link the base library (ZbMoc.lib) in order to use all functions of the library. The corresponding 'ZbMoc.dll' must be placed somewhere in the search path which is used by the program.

If the ApossIDE software has been installed, then a small sample program will be installed in the 'Development\ZbMoc' subdirectory under the ApossIDE install directory. This sample program shows ZbMoc Library integration for the Microsoft Visual Studio 2005 and Visual Studio 2010 development environments. Please read the 'ReadMe.txt' file in this directory for details about how to build and execute the sample program.

General usage of the Library

All functions in the library use C calling conventions. Most of the functions will return a signed 16-bit integer error code. This error code will be 0 for success and a negative value for failure. The meanings of the error codes can be found in the 'ZbMocErr.h' header file. Please refer to this file for a current list of possible errors.

Standardized data types defined in the 'ZbTypes.h' header file are used to improve portability of the library. They are:

Signed Integer, 8, 16 and 32 bits long:

- `SIGNED8`
- `SIGNED16`
- `SIGNED32`

Unsigned Integer, 8, 16 and 32 bits long:

- `UNSIGNED8`
- `UNSIGNED16`
- `UNSIGNED32`

Boolean value, 0 or 1 (8 bits long):

- `BOOLEAN`

Character string using UTF-8 encoding:

- STRUTF8

To communicate with one or more controllers, a 'connection interface' must first be opened by calling one of the 'ZbMocOpen...' functions. These routines will return a 'connection handle' indicating that a connection has been established. This handle is then passed as a parameter, along with the ID of the controller, to the other ZbMoc Library functions. Call one of the 'ZbMocClose...' functions when finished with the Library.

12.9.1.1 Function Groups

Opening and Closing Interfaces

To communicate with a controller, a connection 'interface' must first be opened by calling one of the 'ZbMocOpen...' functions. The 'ZbMocOpen...' functions are responsible for the following internal steps:

1. Availability, compatibility, and fitness-for-use check of the hardware, as well as the required drivers for the selected interface.
2. Setting of all required parameters necessary to open the interface.
3. Scan of all controllers which are connected and match the parameters specified for the interface.
4. Returning an interface handle (0...n) or a negative error code.

The following functions are available for opening the interface:

ZbMocOpenCanLpt	CAN BUS Centronics Adapter
ZbMocOpenCanUsb	CAN USB Stick
ZbMocOpenSerial	V24 Serial Port
ZbMocOpenVlt	VLT Serial Port
ZbMocOpenUsb	USB Port
ZbMocOpenTcp	Ethernet
ZbMocOpenEtherCat	EtherCAT
ZbMocOpenSim	MACS Simulator

All of the 'ZbMocOpen...' functions collect and save information about the connected controllers. The application can retrieve this information by calling either the ZbMocMoconInfo or ZbMocMoconInfoIdx function. These functions return the information in a ZbMocMoconInfoS structure which is defined in 'ZbMoc.h'.

If the application program is being closed, then all open interfaces must first be closed. If an interface must be re-opened with new parameters, then the interface must first be closed before being re-opened. Closing an interface may be done by calling one of the following functions:

ZbMocClose	Close a specified interface.
ZbMocCloseAll	Close all open interfaces.

Other functions related to accessing controllers on an interface (particularly USB interfaces):

ZbMocProbe	Search for controllers on an interface.
ZbMocConnect	Complete a pending connection or re-establish a

	disconnected connection.
ZbMocDisconnect	Disconnect a single controller without closing an interface.

Controller Information

Information about ZbMoc, open interfaces, and specific controllers can be queried used the following functions:

ZbMocGetZbMocVersion	Return the version string of the active ZbMoc DLL.
ZbMocGetIFDriverID	Get the version string of the low level interface driver.
ZbMocGetNumber	Query the number of open controllers.
ZbMocMoconFind	Find a controller ID based on one of several search criteria.
ZbMocMoconState	Query the connection state of a controller.
ZbMocMoconIdent	Query identification information about a specified controller.
ZbMocMoconInfo	Query general information about a specified controller.
ZbMocMoconInfoIdx	Query general information about a controller identified by index.
ZbMocVersionCounts	Return version-specific controller counts.
ZbMocSDOGetUsage	Query whether or not SDO-based communication may be used.
ZbMocSDOGetUsageDetails	Query whether or not SDO-based communication would be used (if possible).
ZbMocMemoryLock	Query whether or not memory has been locked.

Controller/Interface Status

Information about the status of any open controllers can be queried used the following functions:

ZbMocMoconExecutionStatus	Query the execution state (e.g. Is the controller still executing a program?).
ZbMocMoconBusyWait	Wait until a controller is no longer busy.
ZbMocGetAxisPosition	Query the position of an axis of a controller.
ZbMocGetError	Query the error state of an interface (i.e. not of a controller).
ZbMocClearError	Clear the error state of an interface (i.e. not of a controller).

Controller Configuration

The following functions relate to the configuration of the controller:

ZbMocMoconClearParameters	Reset all parameters of a controller to the factory settings.
ZbMocMoconNameSet	Set the name of a controller.
ZbMocParamRead	Read the global parameters of a controller.
ZbMocParamWrite	Set the global parameters of a controller.
ZbMocAxisParamRead(DEPRECATED)	Read the axis parameters of a controller.
ZbMocAxisParamWrite(DEPRECATED)	Set the axis parameters of a controller.
ZbMocUserParamReadRaw	Read user parameters. User parameters are a set of 100 values intended to hold information used by the application program.
ZbMocUserParamWriteRaw	Write user parameters. User parameters are a set of 100 values intended to hold information used by the application program.
ZbMocReadUserArray	Read DIM Arrays.
ZbMocWriteUserArray	Write DIM Arrays.
ZbMocMemoryDump	Read an array from a controller and write it into a file.
ZbMocCNFSaveToFile	Read all parameter settings and array definitions from the controller and save them in a configuration file (.zbc) on disk.
ZbMocCNFRestoreFromFile	Read all parameter settings and array definitions from a configuration file (.zbc) and download them to the controller.

The controller's global parameters are read and written using the ZbMocParamS structure. The controller's axis parameters are read and written using the ZbMocAxisParamS structure. Both of these structures are defined in 'ZbMoc.h'.

The number of global and axis parameters implemented in a specific controller depends on the software version of the controller. Use the ZbMocVersionCounts or ZbMocMoconInfo function to retrieve the number of parameters supported by a specific controller. As well, some of the parameters have different meanings, depending on the controller chip and software version. For the exact meaning of individual parameters, please see the parameter's documentation.

SDO/PDO Handling

The following functions may be used to handle SDOs and PDOs (if a controller supports them):

ZbMocCanOpenStart	Configure and start a controller as a CAN open slave.
ZbMocCanOpenReadSDO	Read data from an SDO of a controller.

ZbMocCanOpenWriteSDO	Transmit data to an SDO of a controller.
ZbMocCanOpenUploadSDO	Read all subindices of an SDO from a controller.
ZbMocCanOpenDownloadSDO	Write all subindices of an SDO to a controller.
ZbMocCanOpenWritePDO	Transmit a PDO message to a controller.
ZbMocCanOpenInhibitPDO	Set the "inhibit timeout" for the ZbMocCanOpenQueuePDO routine.
ZbMocCanOpenQueuePDO	Transmit a PDO message to a controller.
ZbMocCanOpenPollPDO	Poll for any PDO that has been received.
ZbMocSetCallbackPDO	Defines a pointer to a callback function which will receive PDO messages.

ApossIDE Program Handling

The following functions relate to program handling:

ZbMocProgramList	List all programs permanently stored in the controller.
ZbMocProgramLoad	Download a binary (i.e. compiled) program to a controller using a download program.
ZbMocProgramLoadBinary	Download a binary (i.e. compiled) program to a controller.
ZbMocProgramReadBinary	Upload a binary (i.e. compiled) program from a controller.
ZbMocProgramSave	Store the temporarily downloaded program with a given name. A program number is assigned automatically to it.
ZbMocProgramSaveAs	Store the temporarily downloaded program with a given name and program number.
ZbMocProgramDelete	Delete a program with the given program number.
ZbMocMoconClearPrograms	Delete all programs in a controller.
ZbMocExec	Execute a stored program identified by a program number.
ZbMocExecTemp	Execute the temporary program. It is possible to start the temporary program directly after downloading without first having to save the program.
ZbMocBreak	Stop a running program as well as clear a controller error state. The controller is set to a defined state.

ZbMocBreakAll	Identical to ZbMocBreak, but affects all controllers.
ZbMocContinue	Continue a program that was previously interrupted by a "Break" command.
ZbMocPollMessage	Retrieves the next message generated by a "print" command in the running application program or by a system message generated by the controller itself.
ZbMocAutoSet	Set the 'Autostart' flag for a given stored program. The program with the 'Autostart' flag, starts up automatically at power-up of the controller.
ZbMocAutoClear	Erase the 'Autostart' flag. No program is executed automatically at power-up of the controller.
ZbMocSourceSaveInController	Write the source code in the given file into a controller.
ZbMocSourceRestoreFromController	Read the source code of the given program from a controller.
ZbMocSourceDeleteFromController	Delete the source code of the given program from a controller.

Memory Functions

The following memory-related functions are available:

ZbMocMoconClearFactory	Reset a controller to the factory settings.
ZbMocMoconDeleteEeprom	Erase the content of the non-volatile memory (EEPROM) in a controller.
ZbMocMoconSaveRam	Save programs and parameters in non-volatile memory. This is necessary only for controllers without battery backed-up RAM.
ZbMocMoconSaveRamSection	Save the content of the specified RAM section in non-volatile memory.
ZbMocBlobInfoRequest	Return the blob information headers of all saved blobs.
ZbMocBlobRead	Read blob data that was saved in a controller.
ZbMocBlobWrite	Save blob data in a controller.

Other Functions

The following miscellaneous functions are also available:

ZbMocSetCallbackProgress	Defines a callback pointer to a function called to provide progress monitoring.
ZbMocCanReadRaw	Read a single (raw) CAN message of a controller.
ZbMocCanWriteRaw	Transmit a single CAN message to a controller.
ZbMocMoconSetCanParameters	Define the CAN communication parameters of a controller.
ZbMocMoconWaitForString	Wait until the given string is received via serial port of a controller.
ZbMocReadMemPage	Read a memory page out of a controller.
ZbMocReadUserVar	Read a data message from a controller.
ZbMocWriteUserVar	Transmit a data message to a controller.
ZbMocV24ReadRawChar	Read a single character received via the serial interface.
ZbMocV24ReadRawString	Read a string received via the serial interface.
ZbMocV24WriteRaw	Write a string to the serial interface buffer for transmission.

Debug Functions

The following functions may be useful for developers to debug communication issues related to the use of the ZbMoc Library:

ZbMocSetDebugImmediate	Start a debug log immediately using default level and flag settings.
ZbMocSetDebugFile	Define the log file in which debugging messages are written.
ZbMocSetDebugFlags	Define the debugging flags.
ZbMocSetDebugLevel	Define the debugging level.
ZbMocSetDebugHistory	Set the maximum size of the debug history.
ZbMocSetDebugRetain	Set the number of days to retain debug log files.
ZbMocSetDebugMaxLen	Set the maximum length debug log files.
ZbMocDebugPrint	Insert a string into the debug log file.
ZbMocDebugPrintw	Insert a Unicode string into the debug log file.

12.9.1.2 ZbMoc Structures

12.9.1.2.1 ZbMocMoconIdentS

The ZbMocMoconIdentS structure contains identification information about a single controller. The 'ZbMocOpen...' functions retrieve this information automatically from all connected controllers after the connection has been established.

The ZbMoc library caches this information for each of the controllers connected and returns a copy of it to the application via the ZbMocMoconIdent function. Note that a call to this function is faster than a call to ZbMocMoconInfo.

```
typedef struct
{
    UNSIGNED16  id;         // Controller ID (same as "can_id" for CAN interfaces,
                            // port number for EtherCAT).
    UNSIGNED16  ctrltype;   // Controller type.  ZBMOC_CONTROLLER_UNKNOWN if not
                            // known.
    UNSIGNED32  serial;     // Serial number.  0 if not known.
    UNSIGNED32  firmware;   // Firmware version number (e.g. 4.5.1 is represented
                            // as 40501).  0 if not known.
    STRUTF8     desc[100];  // V24 banner (sign on) string, USB device description,
                            // TCP application title, EtherCAT revision.  Empty if
                            // not known or not used.
    SIGNED32    location;   // Device location (USB interfaces).  0 if not known or
                            // not used.
    SIGNED32    pid;        // USB product ID (USB interfaces).  0 if not known or
                            // not used.
    UNSIGNED32  ipaddr;     // IP address (Ethernet interfaces).  0 if not known or
                            // not used.
} ZbMocMoconIdentS;
```

12.9.1.2.2 ZbMocMoconInfoS

The ZbMocMoconInfoS structure contains information about a single controller. The 'ZbMocOpen...' functions retrieve this information automatically from all connected controllers after the connection has been established.

The ZbMoc library caches this information for each of the controllers connected and returns a copy of it to the application via the ZbMocMoconInfo and ZbMocMoconInfoIdx functions.

```
typedef struct ZbMocMoconInfoS_
{
    UNSIGNED16  state;                  // Connection state.
    UNSIGNED16  controller;             // Controller type.
    UNSIGNED16  interf;                 // Interface type.
    UNSIGNED16  channelno;              // Channel number.
    UNSIGNED32  full_version;           // Full version number
                                        // (e.g. 4.5.1 is represented as 40501).
    UNSIGNED16  major_version;          // Major version number.
    UNSIGNED16  minor_version;          // Minor version number.
    UNSIGNED16  micro_version;          // Micro version number.
    UNSIGNED8   cpu_type;               // CPU type.
    UNSIGNED16  axis_controller_type;   // Type of axis controller chip
                                        // ('L'=National, 'H'=HCTL, 'S'=Software).
    UNSIGNED8   number_axis;            // Number of axes.
    UNSIGNED8   option_code;            // Option code.
    UNSIGNED8   board_revision;         // Board version.
    UNSIGNED16  bus_id;                 // Controller ID (same as "can_id" for
                                        // CAN interfaces).
```

```
    UNSIGNED16  can_id;                 // CAN ID (0 if not known).
    STRUTF8     unit_name[24+1];        // Name of controller (trimmed and
                                        // terminated).
    UNSIGNED32  regspeed;               // Regulator speed (8MHz==800).
    UNSIGNED8   sdo_compatible;         // Level of firmware SDO compatibility.
    UNSIGNED16  globalcnt;              // Number of global parameters.
    UNSIGNED16  usercnt;                // Number of user parameters.
    UNSIGNED16  axiscnt;                // Number of axis parameters.
    UNSIGNED32  arraymax;               // Maximum length of arrays.
    double      timebase;               // Length of one controller "tick" in
                                        // milliseconds.
    UNSIGNED16  devclass;               // Device class.

    UNSIGNED16  pid;                    // USB product ID.
    STRUTF8     serial[50];             // Serial number (if known).
    STRUTF8     description[100];       // Device description.
    STRUTF8     banner[100];            // Banner (sign on) string
                                        // (old V24 controllers only).

    UNSIGNED32  number_amp;             // Number of amplifiers.
    UNSIGNED32  ipaddr;                 // IP address (TCP controller only).

    UNSIGNED32  features;               // Controller feature flags.
} ZbMocMoconInfoS;
```

12.9.1.2.3 ZbMocParamS

The ZbMocParamS structure contains the controller's "global" parameters. They can be read with the ZbMocParamRead function and writen with the ZbMocParamWrite function.

The number of parameters implemented in a specific controller depends on the software version of the controller. See the controller documentation for details. The number of parameters supported by a specific controller, can be found in the ZbMocMoconInfoS structure.

```
typedef struct ZbMocParamS_
{
    UNSIGNED32 can_nr;          // #100
    UNSIGNED32 can_baud;        // #101
    UNSIGNED32 prgpar;          // #102
    UNSIGNED32 i_prgstart;      // #103
    UNSIGNED32 i_prgchoice;     // #104
    UNSIGNED32 i_break;         // #105
    UNSIGNED32 i_continue;      // #106
    UNSIGNED32 i_errclr;        // #107
    UNSIGNED32 o_error;         // #108
    UNSIGNED32 ainftime;        // #109 (MACS3 >= 6.4.xx only)
    UNSIGNED32 external24v;     // #110 (VLT >= 6.4.xx only)
    UNSIGNED32 rstermination;   // #111 (VLT >= 6.4.xx and MACS3 >= 6.4.xx only)
    UNSIGNED32 rsbaudrate;      // #112 (VLT >= 6.4.xx and MACS3 >= 6.4.xx only)
    UNSIGNED32 iomode;          // #113 (VLT >= 6.4.xx only)
    UNSIGNED32 cansynctimer;    // #114 (6.5.16)
    UNSIGNED32 canguardtimer;   // #115 (6.6.18)
    UNSIGNED32 vmenctype;       // #116 (6.6.18)
    UNSIGNED32 vmencres;        // #117 (6.6.18)
    UNSIGNED32 vmencmtyp;       // #118 (6.6.18)
    UNSIGNED32 ipaddressmode;   // #119 (7.0.23)
```

```
    UNSIGNED32 ipsubnet;         // #120 (7.0.23)
    UNSIGNED32 reserved1;        // #121 (7.1.32)
    UNSIGNED32 reserved2;        // #122 (7.1.32)
    UNSIGNED32 ipmask;           // #123 (7.1.32)
    UNSIGNED32 ipdefaultgw;      // #124 (7.1.32)
} ZbMocParamS;
```

12.9.1.2.4 ZbMocAxisParamS

The ZbMocAxisParamS structure contains the controller's "axis" parameters. They can be read with the ZbMocAxisParamRead(DEPRECATED) function and writen with the ZbMocAxisParamWrite(DEPRECATED) function.

The number of parameters implemented in a specific controller depends on the software version of the controller. See the controller documentation for details. The number of parameters supported by a specific controller, can be found in the ZbMocMoconInfoS structure. Please note that some of the parameters have different meanings depending on the controller's chip and software version.

It is recommended that the ApossIDE software be used to set axis parameters and permanently save them to controller's memory. The ApossIDE automatically handles all hardware and software controller versions correctly.

```
typedef struct ZbMocAxisParamS_
{
    UNSIGNED32 function_inputs;     // #0
    UNSIGNED32 syncmftime;          // #18
      SIGNED32 sensor_counts;       // #2
    UNSIGNED32 max_velocity;        // #9
    UNSIGNED32 max_acceleration;    // #10
    UNSIGNED32 timer;               // #14
    UNSIGNED32 gain;                // #11
    UNSIGNED32 zero;                // #12
    UNSIGNED32 pole;                // #13
    UNSIGNED32 pullgap;             // #15
    UNSIGNED32 test_window;         // #8
    UNSIGNED32 syncofftime;         // #16
    UNSIGNED32 syncmfpar;           // #17
    UNSIGNED32 home_velocity;       // #7
    UNSIGNED32 home_enforce;        // #3
    UNSIGNED32 negative_limit;      // #4
    UNSIGNED32 positive_limit;      // #5
    UNSIGNED32 integration_limit;   // #21
    UNSIGNED32 use_neg_limit;       // #19
    UNSIGNED32 use_pos_limit;       // #20
    UNSIGNED32 velocity_divisor;    // #22
    UNSIGNED32 user_factor;         // #23
    UNSIGNED32 rpm;                 // #1
    UNSIGNED32 mnu_point;           // #6
    UNSIGNED32 test_duration;       // #24
    UNSIGNED32 test_limit;          // #25
    UNSIGNED32 usrteil;             // ???
    UNSIGNED32 encodertyp;          // #27
    UNSIGNED32 posdrct;             // #28
    UNSIGNED32 posunits;            // #29
    UNSIGNED32 mencoder;            // #30
    UNSIGNED32 mencodertype;        // #67
    UNSIGNED32 ramp_min;            // #31
    UNSIGNED32 ramptype;            // #32
```

```
UNSIGNED32 dfltvel;             // #33
UNSIGNED32 dfltacc;             // #34
UNSIGNED32 bandwidth;           // #35
UNSIGNED32 ffvel;               // #33
UNSIGNED32 ffacc;               // #37
UNSIGNED32 regwmax;             // #38
UNSIGNED32 regwmin;             // #39
UNSIGNED32 hometype;            // #40
UNSIGNED32 home_ramp;           // #41
UNSIGNED32 originoffs;          // #42
UNSIGNED32 errcond;             // #43
UNSIGNED32 endswmod;            // #44
UNSIGNED32 i_refswitch;         // #45
UNSIGNED32 i_poslimitsw;        // #46
UNSIGNED32 i_neglimitsw;        // #47
UNSIGNED32 o_brake;             // #48
UNSIGNED32 syncfactm;           // #49
UNSIGNED32 syncfacts;           // #50
UNSIGNED32 synctype;            // #51
UNSIGNED32 syncmarkm;           // #52
UNSIGNED32 syncmarks;           // #53
UNSIGNED32 syncposoffs;         // #54
UNSIGNED32 syncaccuracy;        // #55
UNSIGNED32 syncready;           // #56
UNSIGNED32 syncfault;           // #57
UNSIGNED32 syncmpulsm;          // #58
UNSIGNED32 syncmpulss;          // #59
UNSIGNED32 syncmtypm;           // #60
UNSIGNED32 syncmtyps;           // #61
UNSIGNED32 syncmstart;          // #62
UNSIGNED32 syncvftime;          // #65
UNSIGNED32 syncvelrel;          // #66
UNSIGNED32 revers;              // #63
UNSIGNED32 o_axmove;            // #64
UNSIGNED32 posfact_n;           // #26
UNSIGNED32 syncmwinm;           // #68
UNSIGNED32 syncmwins;           // #69
UNSIGNED32 esccond;             // #70
UNSIGNED32 encoderdatlen;       // #71  (VLT >= 6.4.xx only)
UNSIGNED32 encoderterm;         // #72  (VLT >= 6.4.xx only)
UNSIGNED32 encoderclock;        // #73  (VLT >= 6.4.xx only)
UNSIGNED32 encoderfreq;         // #74  (VLT >= 6.4.xx only)
UNSIGNED32 mencoderdatlen;      // #75  (VLT >= 6.4.xx only)
UNSIGNED32 mencoderterm;        // #76  (VLT >= 6.4.xx only)
UNSIGNED32 mencoderclock;       // #77  (VLT >= 6.4.xx only)
UNSIGNED32 mencoderfreq;        // #78  (VLT >= 6.4.xx only)
UNSIGNED32 statusmonitor;       // #79  (VLT >= 6.4.xx only)
UNSIGNED32 encoderdelay;        // #80  (VLT >= 6.4.xx only)
UNSIGNED32 mencoderdelay;       // #81  (VLT >= 6.4.xx only)
UNSIGNED32 encodermonitoring;   // #82  (VLT >= 6.4.xx only)
UNSIGNED32 mencodermonitoring;  // #83  (VLT >= 6.4.xx only)
UNSIGNED32 encoderabsturns;     // #84  (VLT >= 6.4.xx only)
UNSIGNED32 mencoderabsturns;    // #85  (VLT >= 6.4.xx only)
UNSIGNED32 encoderabstype;      // #86  (VLT >= 6.4.xx only)
UNSIGNED32 mencoderabstype;     // #87  (VLT >= 6.4.xx only)
UNSIGNED32 encoderabsres;       // #88  (VLT >= 6.4.xx only)
UNSIGNED32 mencoderabsres;      // #89  (VLT >= 6.4.xx only)
UNSIGNED32 velkprop;            // #90  (6.4.63 zub)
UNSIGNED32 velkint;             // #91  (6.4.63 zub)
```

```
    UNSIGNED32 curkprop;            // #92  (6.4.63 zub)
    UNSIGNED32 curkint;             // #93  (6.4.63 zub)
    UNSIGNED32 ampmaxcur;           // #94  (6.4.63 zub)
    UNSIGNED32 ampcommtyp;          // #95  (6.4.63 zub)
    UNSIGNED32 amppwmfreq;          // #96  (6.4.63 zub)
    UNSIGNED32 ampmode;             // #97  (6.4.63 zub)
    UNSIGNED32 jerkmin;             // #98  (6.4.63 zub)
    UNSIGNED32 spiencoder;          // #99  (6.4.72 MOC305 only)
    UNSIGNED32 jerkmin2;            // #100 (6.4.72 zub)
    UNSIGNED32 jerkmin3;            // #101 (6.4.72 zub)
    UNSIGNED32 jerkmin4;            // #102 (6.4.72 zub)
    UNSIGNED32 velkilim;            // #103 (6.6.18 zub)
    UNSIGNED32 curkilim;            // #104 (6.6.18 zub)
    UNSIGNED32 kilimtime;           // #105 (6.6.18 zub)
    UNSIGNED32 syncsftime;          // #106 (6.6.18 zub)
    UNSIGNED32 enccontrol;          // #107 (6.6.18 zub)
    UNSIGNED32 menccontrol;         // #108 (6.6.18 zub)
    UNSIGNED32 syncffvel;           // #109 (6.6.24 zub)
    UNSIGNED32 syncvflimit;         // #110 (6.6.24 zub)
    UNSIGNED32 poserrtime;          // #111 (6.6.32 zub)
    UNSIGNED32 feeddist;            // #112 (6.6.51 zub)
    UNSIGNED32 feedrev;             // #113 (6.6.51 zub)
    UNSIGNED32 posencqc;            // #114 (6.6.51 zub)
    UNSIGNED32 posencrev;           // #115 (6.6.51 zub)
    UNSIGNED32 homezvel;            // #116 (6.6.51 zub)
    UNSIGNED32 i2tftime;            // #117 (6.7.19 zub)
    UNSIGNED32 i2tlimit;            // #118 (6.7.19 zub)
    UNSIGNED32 curftime;            // #119 (6.7.19 zub)
    UNSIGNED32 encoderid;           // #120 (6.7.40 MOC305 only)
    UNSIGNED32 encoderbaud;         // #121 (6.7.40 MOC305 only)
    UNSIGNED32 encoderparity;       // #122 (6.7.40 MOC305 only)
    UNSIGNED32 mencoderid;          // #123 (6.7.40 MOC305 only)
    UNSIGNED32 mencoderbaud;        // #124 (6.7.40 MOC305 only)
    UNSIGNED32 mencoderparity;      // #125 (6.7.40 MOC305 only)
    UNSIGNED32 ampencno;            // #126 (6.7.43 zub)
    UNSIGNED32 ampencres;           // #127 (6.7.43 zub)
    UNSIGNED32 ampencrpm;           // #128 (6.7.43 zub)
    UNSIGNED32 mencoderno;          // #129 (6.7.44 zub)
    UNSIGNED32 pisrc_svirtcount;    // #130 (7.4.18 zub)
    UNSIGNED32 pisrc_mvirtcount;    // #131 (7.4.18 zub)
    UNSIGNED32 pisrc_svirtlatch;    // #130 (7.4.18 zub)
    UNSIGNED32 pisrc_mvirtlatch;    // #131 (7.4.18 zub)
    UNSIGNED32 kffacc;              // #132 (8.0.08 zub)
    UNSIGNED32 kffdec;              // #133 (8.0.08 zub)
} ZbMocAxisParamS;
```

12.9.1.3 ZbMoc Error Codes

ZbMoc Library error codes are defined in 'ZbMocErr.h'. This is included by 'ZbMoc.h'.

ZbMoc routines will normally return ZBMOC_OK (i.e. 0) if the function succeeded and a non-0 value if the function failed for any reason. However, a few routines (in particular, the "Open" routines and the routines that return counts) will return a non-negative number indicating success.

All errors generated by the ZbMoc Library routines themselves, are negative. All errors (or other status indications) returned by the controller, will be positive. See Error Messages for the errors returned by the controller. #defines for these errors can be found in 'SysDef.mi' (see SysDef Defines).

General Errors

Value	#define	Description
0	ZBMOC_OK	Success.
-500	ZBMOC_E_USERABORT	User abort.
-501	ZBMOC_E_NOT_RESPONDING	Lost connection with controller (controller is not responding).
-100	ZBMOC_E_UNAVAILABLE	Operation not available.
-101	ZBMOC_E_ALLBUSY	No new handles are available.
-102	ZBMOC_E_BADHANDLE	Given handle is not valid.
-103	ZBMOC_E_BADINDEX	Given index is not valid.
-104	ZBMOC_E_CANONLY	Function is for CAN connections only.
-105	ZBMOC_E_PARAMETERS	Illegal parameters.
-106	ZBMOC_E_PARAMETER_READOUT	Error in reading parameters.
-107	ZBMOC_E_IS_BUSY	Controller is busy.
-108	ZBMOC_E_FUNCTION_UNAVAILABLE	Function not available.
-109	ZBMOC_E_BADID	ID does not exist.
-110	ZBMOC_E_NOCONTROLLER	No controller found.
-111	ZBMOC_E_COMPRESSION_ERROR	Compression error.
-112	ZBMOC_E_BADBLOB	Problem with Blob (e.g. Blob number is not valid).
-113	ZBMOC_E_PDO_TOO_LONG	PDO length is too long for this interface type.
-114	ZBMOC_E_BADARRAYNO	Array does not exist.
-115	ZBMOC_E_BADARRAYSIZE	Array size mismatch.
-116	ZBMOC_E_BREAK_REQUIRED	A 'Break' command is required for execution of the requested function.
-117	ZBMOC_E_CONTROLLER_MISSING	Not all control units respond to a request.
-118	ZBMOC_E_RXBUFFER_TOO_SMALL	Size of provided buffer is too small.
-119	ZBMOC_E_FILEACCESS	File access error, e.g. file not found.
-120	ZBMOC_E_FILECONTENT	Content of file not valid.
-121	ZBMOC_E_LOWMEMORY	Running out of memory.
-122	ZBMOC_E_ARRAYTOOBIG	Array size is too big.
-123	ZBMOC_E_BADPOINTER	Bad pointer (e.g. NULL pointer).

Value	#define	Description
-124	ZBMOC_E_BADPARAMNUM	Bad parameter number.
-125	ZBMOC_E_PROGRAMTOOBIG	Program is to large to download or too large to upload.
-126	ZBMOC_E_BLOCKED	Controller is blocked by a previously executing command.
-127	ZBMOC_E_NO_PROGRAM_SLOT	There is no available program slot to save another program.
-129	ZBMOC_E_PENDINGCONNECTION	Connection (probably USB) is pending.
-130	ZBMOC_E_NAME_TOO_LONG	Name is too long.
-131	ZBMOC_E_CANNOT_PACK	String contains characters that cannot be packed.
-132	ZBMOC_E_CRC_MISMATCH	CRC values did not match.
-133	ZBMOC_E_ALREADYOPEN	Interface is already open.
-134	ZBMOC_E_OPENERROR	Error trying to open interface (see log file).
-135	ZBMOC_E_CNFINCOMPLETE	Configuration data is incomplete.
-136	ZBMOC_E_TIME_FAILURE	Could not read PC time.

CAN-specific error messages concerning CAN bus problems

Value	#define	Description
-202	ZBMOC_E_CAN_TIMEOUT	Access timeout to CAN.
-203	ZBMOC_E_CAN_NO_RESET	No reset.
-204	ZBMOC_E_CAN_NO_HALT	No halt.
-205	ZBMOC_E_CAN_NO_START	No start.
-206	ZBMOC_E_CAN_NO_HARDWARE	No CAN hardware found.
-207	ZBMOC_E_CAN_NO_MEMORY	Not enough memory.
-208	ZBMOC_E_CAN_NO_USER	No user found.
-209	ZBMOC_E_CAN_NO_CPU_ACCESS	No access to CPU.
-210	ZBMOC_E_CAN_NO_POLLSLAVES	No pollslaves.
-211	ZBMOC_E_CAN_TOO_MANY_SLAVES	Too many slaves.
-212	ZBMOC_E_CAN_NO_SLAVE	No slaves found.
-213	ZBMOC_E_CAN_NO_INIT	Not initialized.
-214	ZBMOC_E_CAN_SLAVES_EXIST	Slave exists.

Value	#define	Description
-215	ZBMOC_E_CAN_NO_POLLING	Polling is not active.
-216	ZBMOC_E_CAN_POLLING_IS_RUN	Polling is active.

VLT-specific error messages

Value	#define
-217	ZBMOC_E_VLT_SCAN_IN_PROGRESS
-218	ZBMOC_E_VLT_SCAN_ERROR
-219	ZBMOC_E_VLT_SCAN_NO_SLAVES
-221	ZBMOC_E_VLT_ILLEGAL_ID
-222	ZBMOC_E_VLT_OPEN
-223	ZBMOC_E_VLT_TRANSMIT
-224	ZBMOC_E_VLT_FRAMETYPE
-225	ZBMOC_E_VLT_MESSAGETYPE
-226	ZBMOC_E_VLT_STATUS_READ_ERROR
-227	ZBMOC_E_VLT_RXBCC
-228	ZBMOC_E_VLT_NORXBUFFER
-229	ZBMOC_E_VLT_INTERNAL
-230	ZBMOC_E_VLT_RXBADLENGTH
-231	ZBMOC_E_VLT_UNKNOWN
-232	ZBMOC_E_VLT_TIMEOUT
-233	ZBMOC_E_VLT_LOSTCONNECTION

COM port specific error messages

Value	#define
-234	ZBMOC_E_COM_ILLEGALPORT
-235	ZBMOC_E_COM_OPENERROR
-236	ZBMOC_E_COM_TXERROR
-237	ZBMOC_E_COM_TIMEOUT
-238	ZBMOC_E_COM_ERROR

CANopen error messages

Value	#define
-239	ZBMOC_E_CAN_COMERROR

Value	#define
-240	ZBMOC_E_CAN_TIMER_UNAVAILABLE
-241	ZBMOC_E_CAN_STATUS_TIMEOUT
-242	ZBMOC_E_COM_XON_ERROR
-243	ZBMOC_E_COM_CLOSED
-244	ZBMOC_E_COM_COMMAND
-245	ZBMOC_E_COM_PROTOCOL_LEN
-246	ZBMOC_E_COM_GLOBAL_TIMEOUT
-247	ZBMOC_E_COM_COMMAND_TIMEOUT
-249	ZBMOC_E_NULL_POINTER
-250	ZBMOC_E_SLAVE_NOT_FOUND
-251	ZBMOC_E_NO_PARAMETERS
-252	ZBMOC_MESSAGE_READ
-253	ZBMOC_E_COM_FRAME_FORMAT_ERROR
-254	ZBMOC_E_COM_FRAME_TEMP_OVERFLOW
-255	ZBMOC_E_ETHERCAT_ERROR
-256	ZBMOC_E_CAN_NOT_ZUB_DEVICE
-257	ZBMOC_E_CAN_LISTEN_ONLY
-261	ZBMOC_E_USB_CONNECT_FAILED

TCP error messages

Value	#define	Description
-281	ZBMOC_E_TCP_INVALID_ADDRESS	Invalid TCP/IP address.
-282	ZBMOC_E_TCP_SOCKET_ERROR	A TCP/IP socket error has occured.
-283	ZBMOC_E_TCP_CONNECTION_LOST	Connection dropped by Server.
-284	ZBMOC_E_TCP_NOT_CONNECTED	Connection was previously dropped.
-285	ZBMOC_E_TCP_BAD_RESP_LENGTH	Invalid response length received. Connection dropped.
-286	ZBMOC_E_TCP_BAD_COMMAND	Invalid command received by Server.

CANopen error messages

Value	#define
-301	ZBMOC_E_CANOPEN_ServerAckTimeout
-302	ZBMOC_E_CANOPEN_ResponseNotExpected

Value	#define
-303	ZBMOC_E_CANOPEN_TxTimeout
-304	ZBMOC_E_CANOPEN_ArraySizeMismatch
-305	ZBMOC_E_CANOPEN_UploadLastSegment

CANopen error messages for "ServerAbortDomainTransfer" errors

Value	#define	Description
-350	ZBMOC_E_ABORT_Unknown	Unknown abort code.
-351	ZBMOC_E_ABORT_ToggleBitNotAlternated	Toggle bit not alternated.
-352	ZBMOC_E_ABORT_SdoProtocolTimedOut	SDO protocol timed out.
-353	ZBMOC_E_ABORT_CommandSpecifierNotValid	Client/server command specifier not valid or unknown.
-354	ZBMOC_E_ABORT_InvalidBlockSize	Invalid block size (block mode only).
-355	ZBMOC_E_ABORT_InvalidSequenceNumber	Invalid sequence number (block mode only).
-356	ZBMOC_E_ABORT_CrcError	CRC error (block mode only).
-357	ZBMOC_E_ABORT_OutOfMemory	Out of memory.
-358	ZBMOC_E_ABORT_UnsupportedAccessToAnObject	Unsupported access to an object.
-359	ZBMOC_E_ABORT_AttemptToReadWriteOnlyObject	Attempt to

Value	#define	Description
		read a write only object.
-360	ZBMOC_E_ABORT_AttemptToWriteReadOnlyObject	Attempt to write a read only object.
-361	ZBMOC_E_ABORT_ObjectDoesNotExist	Object does not exist in the object dictionary.
-362	ZBMOC_E_ABORT_ObjectCannotBeMappedToThePDO	Object cannot be mapped to the PDO.
-363	ZBMOC_E_ABORT_MappedObjectsExceedPDOlength	The number and length of the objects to be mapped would exceed PDO length.
-364	ZBMOC_E_ABORT_GeneralParameterIncompatibility	General parameter incompatibility.
-365	ZBMOC_E_ABORT_GeneralInternalIncompatibility	General internal incompatibility in the device.
-366	ZBMOC_E_ABORT_AccessFailedDueToAnHardwareError	Access failed due to a hardware error.
-367	ZBMOC_E_ABORT_DataTypeConflict	
-368	ZBMOC_E_ABORT_LengthOfServiceParameterDoesNotMatch	Data type does not match, length of service

Value	#define	Description
		parameter does not match.
-369	ZBMOC_E_ABORT_LengthOfServiceParameterTooHigh	Data type does not match, length of service parameter too high.
-370	ZBMOC_E_ABORT_LengthOfServiceParameterTooLow	Data type does not match, length of service parameter too low.
-371	ZBMOC_E_ABORT_ObjectAttributeInconsistent	
-372	ZBMOC_E_ABORT_SubIndexDoesNotExist	Sub-index does not exist.
-373	ZBMOC_E_ABORT_ParamValueRangeExceeded	Value range of parameter exceeded (only for write access).
-374	ZBMOC_E_ABORT_ParamValueWrittenTooHigh	Value of parameter written too high.
-375	ZBMOC_E_ABORT_ParamValueWrittenTooLow	Value of parameter written too low.
-376	ZBMOC_E_ABORT_MaxValueIsLessThanMinValue	Maximum value is less than minimum

Value	#define	Description
		value.
-377	ZBMOC_E_ABORT_GeneralError	General error.
-378	ZBMOC_E_ABORT_DataCannotBeTransferredOrStored	Data cannot be transferred or stored to the application.
-379	ZBMOC_E_ABORT_DataCannotBeTransferredOrStoredLocalControl	Data cannot be transferred or stored to the application because of local control.
-380	ZBMOC_E_ABORT_DataCannotBeTransferredOrStoredDeviceState	Data cannot be transferred or stored to the application because of the present device state.
-381	ZBMOC_E_ABORT_ObjectDictionaryGenerationFails	Object dictionary dynamic generation fails or no object dictionary is present (e.g. object dictionary is generated from file and generation fails because of an file error).

Value	#define	Description

12.9.1.4 Function Descriptions

12.9.1.4.1 ZbMocAutoClear

```
SIGNED16 ZbMocAutoClear(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Clear the "Autostart" flag of the controller with the given ID. When the "Autostart" flag is clear, no program is started automatically after power-on.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks If the "Autostart" flag was already clear when this function is called, then the function will do nothing. It is not an error to clear the flag when it is already clear.

Portability Firmware ≥ 2.7

12.9.1.4.2 ZbMocAutoSet

```
SIGNED16 ZbMocAutoSet(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 prgnr);
```

Summary Set the "Autostart" flag of the given controller for the program defined by the given program number.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
prgnr = Program number.

Return Value ZbMoc Error Codes

Remarks If the "Autostart" flag was previously set for another program, then this flag is removed. Only one program per controller can be tagged with an "Autostart" flag. The program with the "Autostart" flag starts automatically after power-on or reset of the controller.

Portability Firmware ≥ 2.7

12.9.1.4.3 ZbMocAxisParamRead(DEPRECATED)

```
SIGNED16 ZbMocAxisParamRead(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 axnr,
                            ZbMocAxisParamS *para);
```

Summary Read the axis parameters from the selected controller and axis number. The parameters are decoded and copied into the ZbMocAxisParamS structure. **Deprecated from firmware 9.00.00**

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
axnr = Axis number (1...n) of the controller.
para = Pointer to a structure taking the read out axis parameters.

Return Value ZbMoc Error Codes

Remarks If the controller supports multiple axes, then each axis will maintain its own set of axis parameters.

This function reads all axis parameters supported by the selected controller. All parameters in ZbMocAxisParamS that are not supported by the controller, are set to zero.

If any conversion is necessary to get compatible results corresponding to the version of the controller, then this is done internally.

Portability 9.00 > Firmware ≥ 2.7

12.9.1.4.4 ZbMocAxisParamWrite(DEPRECATED)

```
SIGNED16 ZbMocAxisParamWrite(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 axnr,
                             ZbMocAxisParamS *para);
```

Summary Write the axis parameters of the given axis number to the selected controller. The parameters are taken from the ZbMocAxisParamS structure, prepocessed if necessary, and saved in the controller. **Deprecated from firmware 9.00.00**

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
axnr = Axis number (1...n) of the controller.
para = Pointer to a structure holding the parameters.

Return Value ZbMoc Error Codes

Remarks If the controller supports multiple axes, then each axis will maintain its own set of axis parameters.

This function sets all axis parameters supported by the selected controller. All parameters in ZbMocAxisParamS that are not supported by the controller, are ignored.

If any conversion is necessary to get compatible results corresponding to the version of the controller, this is done internally before writing the settings.

Portability 9.00 > Firmware ≥ 2.7

12.9.1.4.5 ZbMocBlobInfoRequest

```
SIGNED16 ZbMocBlobInfoRequest(UNSIGNED16 h, UNSIGNED16 id, ZbMocBlobInfoS *blobinfos,
                              UNSIGNED16 *numentries, UNSIGNED16 maxentries);
```

Summary Return the blob information headers of all saved blobs.

Parameter h = Handle of the interface.

id = Identification number (ID) of the controller.
blobinfos = Pointer to an array of structures taking the blob information.
numentries = Pointer to a variable taking the actual number of blob information headers returned.
maxentries = Maximum number of blob information structures in the array.

Return Value ZbMoc Error Codes

Remarks Any kind of binary data can be stored in the controller in "blobs" (i.e. binary large objects). Each blob is identified by a unique blob number which can be found in the blob information header returned by this function. The blob number is used by ZbMocBlobRead to access this stored data.

If more than "maxentries" blobs have been saved in the controller, then only the first "maxentries" headers are returned. Note that "numentries" is the number of headers returned and NOT the number of headers that are stored on the controller.

This function is not available for serial (V24) communication below controller version 6.0.1.

Portability Firmware ≥ 5.17

12.9.1.4.6 ZbMocBlobRead

```
SIGNED16 ZbMocBlobRead(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 blobnum,
                       UNSIGNED8 *blobdata, UNSIGNED32 *blobdatalen,
                       UNSIGNED32 buflen);
```

Summary Read blob data that was saved in the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
blobnum = Identification number of the blob.
blobdata = Pointer to an array taking the received blob data.
blobdatalen = Pointer to a variable receiving the length of the blob in bytes or the required length if the provided buffer was too small.
buflen = Size of the array provided for taking the blob data.

Return Value ZbMoc Error Codes

Remarks The parameter "blobnum" is used as an identification number of the blob (i.e. binary large object) holding the data. This number is part of the structure that is returned by the ZbMocBlobInfoRequest function.

The provided buffer must be at least 8 bytes larger than the actual blob size. If the provided buffer is too small, then the ZBMOC_E_RXBUFFER_TOO_SMALL error code is returned. The actual blob size is given in "blobdatalen".

This function is not available for serial (V24) communication below controller version 6.0.1.

Portability Firmware ≥ 5.17

12.9.1.4.7 ZbMocBlobWrite

```
SIGNED16 ZbMocBlobWrite(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 blobnum,
                        UNSIGNED8 *blobdata, UNSIGNED32 blobdatalen);
```

Summary Save blob data in the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
blobnum = Identification number of the blob.
blobdata = Pointer to an array holding the blob data to be written.
blobdatalen = Length of the blob data (in bytes).

Return Value ZbMoc Error Codes

Remarks The parameter "blobnum" is used as an identification number of the blob (i.e. binary large object) holding the data. This number is part of the structure that is returned by the ZbMocBlobInfoRequest function.

The application program uses the '`ZbMocBlob`...' functions to manage the storage of source code in the controller (i.e. source code is stored as blobs). The ApossIDE associates source code with the corresponding saved binary code by using the program number as the blob number. If program source is being saved on the controller (either by the ApossIDE or by the ZbMocSourceSaveInController function), then the user application must avoid using blob numbers in the range 0 to MOC_MAXPROGCNT-1 (see MOC_MAXPROGCNT in 'ZbMoc.h' for more information).

Blob numbers in the range 4096 to 65535 are reserved and should not be used.

This function is not available for serial (V24) communication below controller version 6.0.1.

Portability Firmware ≥ 5.17

12.9.1.4.8 ZbMocBreak

```
SIGNED16 ZbMocBreak(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Transmit a "Break" command to the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks This initiates the following actions:

- Stops program execution (and also autostart).
- Stops all moving axes.
- Clears an error state if the controller is in an error state.

Depending on the type of interface and controller, it might be necessary to call this function repeatedly to stop a program which has the "Autostart" flag set. This is

necessary since the first "Break" command results in a restart. A "Break" command acting on an "Autostart" program results in error state 19.

Portability Firmware ≥ 2.7

12.9.1.4.9 ZbMocBreakAll

```
SIGNED16 ZbMocBreakAll(UNSIGNED16 h);
```

Summary Transmit a "Break" command to ALL controllers connected to the selected interface. From the point of view of a single controller, this function is identical to ZbMocBreak.

Parameter h = Handle of the interface.

Return Value ZbMoc Error Codes

Remarks Controllers connected on a CAN bus or other multi-point interfaces are only affected if their ID's are within the range defined in the '`ZbMocOpen`...' function. Other controllers which may be connected to the same bus are not affected. If any controller is blocked by a previous request, then ZBMOC_E_BLOCKED is returned and no break command is sent to any controller. Otherwise, an attempt is made to break all controllers even if some attempts fail. If any attempts fail, then the first non-ZBMOC_OK return code is returned and subsequent non-ZBMOC_OK return codes are discarded.

Portability Firmware ≥ 2.7

12.9.1.4.10 ZbMocCanOpenDownloadSDO

```
SIGNED16 ZbMocCanOpenDownloadSDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 index,
                                 SIGNED32 *value, SIGNED32 cnt);
```

Summary Write all subindices of an SDO to the selected controller. The controller must be configured as a CAN open slave.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
index = Index of the SDO.
value = Pointer to a buffer containing the SDO data.
cnt = Number of subindices to be written.

Return Value ZbMoc Error Codes

Remarks "cnt" may be less than the actual number of subindices in the SDO. If the SDO contains subindices after "cnt", then they are not modified. Subindex 0 is expected to be the highest subindex number and will not be written by this routine.

Portability Firmware ≥ 5.21

12.9.1.4.11 ZbMocCanOpenInhibitPDO

```
SIGNED16 ZbMocCanOpenInhibitPDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED32 timeout);
```

```
SIGNED16 ZbMocCanOpenSlaveInhibitPDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED32 timeout,
                                     void (*callback)(UNSIGNED16 h, UNSIGNED16 id));
```

Summary Set the "inhibit timeout" for the ZbMocCanOpenQueuePDO routine.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
timeout = Timeout in milliseconds.

Return Value ZbMoc Error Codes

Remarks PDO's queued with the ZbMocCanOpenQueuePDO routine will only be transmitted once the "inhibit timeout" expires.

ZbMocCanOpenSlaveInhibitPDO is an alternate version of this routine that includes a callback. If this version is used and a callback routine is specified, then it is called after each PDO is sent.

Portability Firmware ≥ 5.21

12.9.1.4.12 ZbMocCanOpenPollPDO

```
SIGNED16 ZbMocCanOpenPollPDO(UNSIGNED16 *h, UNSIGNED16 *id, double *time,
                             SIGNED32 *pdo, SIGNED32 *len, UNSIGNED8 *dat,
                             SIGNED32 buflen);
```

Summary Poll for any PDO that has been received.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
time = Time at which the PDO was received (in seconds since ZbMoc was started).
pdo = Type of PDO: 1 - PDO1, 2 - PDO2, etc.
len = Length of the PDO data (in bytes).
dat = Pointer to array of bytes which is to receive the PDO data.
buflen = Length of the "dat" buffer (in bytes).

Return Value 0 - PDO queue is empty. No PDO is returned.
>0 - Number of PDOs in queue (before poll). The oldest PDO in the queue is returned.
<0 - ZbMoc Error Codes

Remarks If a PDO callback routine has NOT been defined (i.e. by calling ZbMocSetCallbackPDO), then incoming PDO messages are queued by ZbMoc. Messages in the queue may be retreived by calling this routine. The return code will be the number of PDO messages in the queue when the routine is called. So 0 will be returned if the queue is empty. If the queue is not empty, then the oldest PDO message will be returned and that message will be removed from the queue. ZbMoc will queue up to 1000 PDO messages. After that, the oldest PDO message is discarded whenever a new PDO message is received. Note that all of the parameters (except "buflen") point to variables that will receive information about the PDO. "len" will be the actual number of data bytes received in the PDO and the PDO data will be copied into "dat". The smaller of "len" and "buflen" bytes will be copied.

Portability Firmware ≥ 5.21

12.9.1.4.13 ZbMocCanOpenQueuePDO

```
SIGNED16 ZbMocCanOpenQueuePDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 type,
                              UNSIGNED8 *inarray, UNSIGNED8 len);
```

Summary Transmit a PDO message to the selected controller. The controller must be configured as a CAN open slave. The message is only transmitted when the "inhibit timeout" expires.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
type = Type of PDO: 1 - PDO1, 2 - PDO2, etc.
inarray = Pointer to array of bytes which is to be transmitted.
len = Size of the array (in bytes).

Return Value ZbMoc Error Codes

Remarks The maximum length of a PDO message depends on the type of interface being used. A PDO message queued by this routine will only be sent when the "inhibit timout" expires. This timeout can be set by the ZbMocCanOpenInhibitPDO routine. Only one PDO message is queued; if a second message is queued before the timeout expires, then the previous message is overwritten.

Portability Firmware ≥ 5.21

12.9.1.4.14 ZbMocCanOpenReadSegmentedSDO

```
SIGNED16 ZbMocCanOpenReadSegmentedSDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 index,
                              UNSIGNED8 subindex, UNSIGNED8 *value,SIGNED32 *size);
```

Summary Read data using segmented transfer from an SDO of the selected controller. The controller must be configured as a CAN open slave.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
index = Index of the SDO.
subindex = Subindex of the SDO.
value = Pointer to the received SDO data.
size = Pointer to a variable to receive the size of data

Return Value ZbMoc Error Codes

Portability Firmware ≥ 5.21

12.9.1.4.15 ZbMocCanOpenReadSDO

```
SIGNED16 ZbMocCanOpenReadSDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 index,
                              UNSIGNED8 subindex, SIGNED32 *value);
```

Summary Read data from an SDO of the selected controller. The controller must be configured as a CAN open slave.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
index = Index of the SDO.
subindex = Subindex of the SDO.
value = Pointer to the received SDO data.

Return Value ZbMoc Error Codes

Remarks The size of the "value" array (i.e. "len") will typically be 4 bytes. If the actual number of bytes received for the SDO (i.e. "len_rx_data") is larger than "len", then the additonal received bytes are ignored.

Portability Firmware ≥ 5.21

12.9.1.4.16 ZbMocCanOpenStart

```
SIGNED16 ZbMocCanOpenStart(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Configure and start the selected controller as a CAN open slave.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks This routine will configure the controller as a CANopen slave. It must be called before the other SDO/PDO routines can be called.

Portability Firmware ≥ 5.21

12.9.1.4.17 ZbMocCanOpenUploadSDO

```
SIGNED16 ZbMocCanOpenUploadSDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 index,
                               SIGNED32 *value, SIGNED32 *cnt);
```

Summary Read all subindices of an SDO from the selected controller. The controller must be configured as a CAN open slave.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
index = Index of the SDO.
value = Pointer to a buffer to receive the SDO data.
cnt = Pointer to a variable to receive the number of subindices that were received.

Return Value ZbMoc Error Codes

Remarks The size of the "value" array MUST be 255 elements long. This is the maximum number of subindices that may be returned for any SDO (i.e. subindex 0-254).

Portability Firmware ≥ 5.21

12.9.1.4.18 ZbMocCanOpenWritePDO

```
SIGNED16 ZbMocCanOpenWritePDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 type,
                              UNSIGNED8 *value, UNSIGNED8 len);
```

Summary Transmit a PDO message to the selected controller. The controller must be configured as a CAN open slave.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
type = Type of PDO: 1 - PDO1, 2 - PDO2, etc.
value = Pointer to array of bytes which is to be transmitted.
len = Size of the array (in bytes).

Return Value ZbMoc Error Codes

Remarks The maximum length of a PDO message depends on the type of interface being used.

Portability Firmware ≥ 5.21

12.9.1.4.19 ZbMocCanOpenWriteSegmentedSDO

```
SIGNED16 ZbMocCanOpenWriteSegmentedSDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 index,
                              UNSIGNED8 subindex, UNSIGNED8* value, SIGNED32 size);
```

Summary Transmit data using segmented transfer to an SDO of the selected controller. The controller must be configured as a CAN open slave.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
index = Index of the SDO.
subindex = Subindex of the SDO.
value = Value to be transmitted to the SDO.
size = The size of data to be written

Return Value ZbMoc Error Codes

Portability Firmware ≥ 5.21

12.9.1.4.20 ZbMocCanOpenWriteSDO

```
SIGNED16 ZbMocCanOpenWriteSDO(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 index,
                              UNSIGNED8 subindex, SIGNED32 value);
```

Summary Transmit data to an SDO of the selected controller. The controller must be configured as a CAN open slave.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

index = Index of the SDO.
subindex = Subindex of the SDO.
value = Value to be transmitted to the SDO.

Return Value ZbMoc Error Codes

Remarks The size of the "value" array (i.e. "len") must be 4 bytes. Other lengths are not supported.

Portability Firmware ≥ 5.21

12.9.1.4.21 ZbMocCanReadRaw

```
SIGNED16 ZbMocCanReadRaw(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 *msg,
                         UNSIGNED32 timeout);
```

Summary Read a single (raw) CAN message of the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
msg = Pointer to a data section of 8 bytes length taking the received CAN message data.
timeout = Maximum period of time [ms] for receiving the message. 0 = Do not wait (i.e. a message must present immediately). If there is no message received within the defined time slot, then the function returns an error code.

Return Value ZBMOC_OK - Message was received and is copied to the data buffer.
ZBMOC_E_CAN_TIMEOUT - No message received within time slot.
ZbMoc Error Codes

Remarks This function is only available for CAN bus based communication.

The user data section of the CAN message has a length of 8 bytes. The data is copied into the "msg" buffer exactly the way it was received.

Portability Firmware ≥ 2.7

12.9.1.4.22 ZbMocCanWriteRaw

```
SIGNED16 ZbMocCanWriteRaw(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 *msg);
```

Summary Transmit a single CAN message to the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
msg = Pointer to a data section of 8 bytes length holding the user data of the CAN message to send.

Return Value ZbMoc Error Codes

Remarks This function is only available for CAN bus based communication.

The user data section of the CAN message has a length of 8 bytes. The data is

transmitted exactly the way it is given in the "msg" data buffer.

Portability Firmware ≥ 2.7

12.9.1.4.23 ZbMocClearError

```
SIGNED16 ZbMocClearError(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Clear an error state in the communication layer of the given interface.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks If an error state was present, then the communication layer is reset (i.e. the transmit and receive buffers of the interface are cleared).

Portability Firmware ≥ 2.7

12.9.1.4.24 ZbMocClose

```
SIGNED16 ZbMocClose(UNSIGNED16 h);
```

Summary Close the given interface.

Parameter h = Handle of the interface.

Return Value ZbMoc Error Codes

Remarks This function should be called before the application program exits.

12.9.1.4.25 ZbMocCloseAll

```
SIGNED16 ZbMocCloseAll(void);
```

Summary Close all open interfaces.

Return Value ZbMoc Error Codes

Remarks This function can be used instead of ZbMocClose.

12.9.1.4.26 ZbMocCNFRestoreFromFile

```
SIGNED16 ZbMocCNFRestoreFromFile(UNSIGNED16 h, UNSIGNED16 id,
                                 const STRUTF8 *filename, BOOLEAN prmflg);
```

Summary Read a configuration (*.zbc/*.cnf) file and download the data (parameters and arrays) to the selected controller.

Parameter h = Handle of the interface.

id = Identification number (ID) of the controller.
filename = Pointer to a string holding the filename and path of the configuration file to be read. This should include the file extension.
prmflg = Flag indicating which parameters to restore. If this is FALSE, then only user parameters and DIM arrays are restored. Otherwise, all parameters (i.e. global parameters, axis parameters, etc.) are restored.

Return Value ZbMoc Error Codes

Remarks With zub controllers that support SDO-based communication, this function can be executed regardless of the controller's execution state. However, saving large arrays (i.e. totalling more than 1000 elements) can influence the speed of program execution significantly because there is no ApossIDE command, interrupt, or error handling processed during the final persistent saving of the arrays. It is recommended that this routine not be used when a user program is executing.

The filename may be either UTF-8 or ANSI encoded.

Portability Firmware ≥ 2.7

12.9.1.4.27 ZbMocCNFSaveToFile

```
SIGNED16 ZbMocCNFSaveToFile(UNSIGNED16 h, UNSIGNED16 id, const STRUTF8 *filename);
```

Summary Read the parameter settings and array definitions from the given controller and write them into a configuration (*.zbc) file.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
filename = Pointer to a string holding the filename and path of the configuration file to be written. This should include the file extension.

Return Value ZbMoc Error Codes

Remarks If the given configuration file exists, then it is overwritten!

The filename may be either UTF-8 or ANSI encoded.

Portability Firmware ≥ 2.7

12.9.1.4.28 ZbMocConnect

```
SIGNED16 ZbMocConnect(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Complete a pending connection or re-establish a disconnected connection.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks When a USB interface is opened using ZbMocOpenUsb, the controllers on the interface are not physically opened for communication. Instead, they will be in a "pending" state

(see the ZbMocMoconInfoS structure). The **ZbMocConnect** routine must be called to finialize the connection before communication with the controller can be started.

12.9.1.4.29 ZbMocContinue

```
SIGNED16 ZbMocContinue(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Continue a program that was previously interrupted by a "Break" command.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks

Portability Firmware ≥ 2.7

12.9.1.4.30 ZbMocDebugPrint

```
void ZbMocDebugPrint(long level, const char *str);
```

Summary Insert a string into the debug log file.

Parameter level = Debug level.
str = Text to be written to the debug file.

Remarks

12.9.1.4.31 ZbMocDebugPrintw

```
void ZbMocDebugPrintw(long level, const wchar_t *str);
```

Summary Insert a Unicode string into the debug log file.

Parameter level = Debug level.
str = Text to be written to the debug file.

Remarks

12.9.1.4.32 ZbMocDisconnect

```
SIGNED16 ZbMocDisconnect(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Disconnect a controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks This can be used to disconnect a USB controller.

12.9.1.4.33 ZbMocExec

```
SIGNED16 ZbMocExec(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 prgnr);
```

Summary Start the program with the given program number on the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
prgnr = Program number.

Return Value ZbMoc Error Codes

Remarks

Portability Firmware ≥ 2.7

12.9.1.4.34 ZbMocExecTemp

```
SIGNED16 ZbMocExecTemp(UNSIGNED16 h, UNSIGNED16 id, BOOLEAN init);
```

Summary Start the previously downloaded "temporary" program.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
init = TRUE (recommended) if internal variables (e.g. MNU point, etc.) must be initialized before starting.

Return Value ZbMoc Error Codes

Remarks The temporary program is the program that was downloaded last and is normally not yet stored under a program number.

See also the ZbMocProgramSave and ZbMocProgramSaveAs functions.

Portability Firmware ≥ 2.7

12.9.1.4.35 ZbMocGetAxisPosition

```
SIGNED16 ZbMocGetAxisPosition(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 axnr,
                              SIGNED32 encoder, SIGNED32 *position);
```

Summary Read the position of the given axis of the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
axnr = Number of the axis: 1...n.
encoder = Encoder to be used: 0 - Slave, 1 - Master.
position = Pointer to a variable taking the returned position value.

Return Value ZbMoc Error Codes

Remarks The application program calling up this function is responsible for ensuring that the given axis number exists in the selected controller.

Portability Firmware ≥ 2.7

12.9.1.4.36 ZbMocGetError

```
SIGNED16 ZbMocGetError(UNSIGNED16 h);
```

Summary Return the error state of the communication layer of the given interface.

Parameter h = Handle of the interface.

Return Value ZbMoc Error Codes

Remarks An error state of the communication layer can be cleared by using the ZbMocClearError function. This also resets the transmit and receive buffers. Note that this is the PC and communication-based error state; it is NOT the error state of any controller on the interface.

12.9.1.4.37 ZbMocGetIFDriverID

```
SIGNED16 ZbMocGetIFDriverID(UNSIGNED16 h, STRUTF8 *driverid);
```

Summary Get the version string of the low level interface driver currently in use (if available).

Parameter h = Handle of the interface.
driverid = Pointer to a string of at least 80 characters taking the identification string of the low level driver.

Return Value ZbMoc Error Codes

Remarks The given identification string depends on the interface specified by the connection handle. A ZBMOC_E_BADHANDLE error is returned if this handle is invalid.

A maximum of 80 bytes (including the '\0'-terminator) will be returned. The string will always be '\0'-terminated.

12.9.1.4.38 ZbMocGetNumber

```
UNSIGNED16 ZbMocGetNumber(UNSIGNED16 h);
```

Summary Return the number of connected controllers on the given interface.

Parameter h = Handle of the interface.

Return Value Number of connected controllers.

Remarks Controllers that have an ID that was not within the ID range defined during execution of the '`ZbMocOpen...`' function for the interface, are not included in the returned number. ID's within the range but for which no controller could be found, are also not

included in the returned number. However, ID's for controllers that were found but that failed to respond properly (and therefore are not accessible) ARE included in the returned number. Note that the ZbMocMoconInfo and ZbMocMoconInfoIdx routines will return FALSE for these controllers.

For the serial (V24) interface, the number of controllers returned is always 1.

If the interface handle is invalid, then 0 is returned.

12.9.1.4.39 ZbMocGetZbMocVersion

```
void ZbMocGetZbMocVersion(STRUTF8 *version);
```

Summary Return the version string of the active ZbMoc DLL.

Parameter version = Pointer to a character string to receive the version string.

Remarks A maximum of 100 bytes (including the '\0'-terminator) will be returned. The string will always be '\0'-terminated. Although the interface specifies a UTF-8 encoded string, no UTF-8 characters will ever be returned; only characters in the range 0-127 will be returned.

12.9.1.4.40 ZbMocMemoryDump

```
SIGNED16 ZbMocMemoryDump(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 arrnr,
                         const STRUTF8 *filename);
```

Summary Read the specified array from the given controller and write it into a file. Note that the file will have the same format as files created by the MemoryDump command.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
arrnr = Number of the array (0...n or 0xFFFF for dynamic memory).
filename = Pointer to a string holding the filename and path of the file to be written.

Return Value ZbMoc Error Codes

Remarks If the given file exists, then it is overwritten!

The filename may be either UTF-8 or ANSI encoded.

12.9.1.4.41 ZbMocMemoryLock

```
SIGNED16 ZbMocMemoryLock(UNSIGNED16 h, UNSIGNED16 id, BOOLEAN *locked);
```

Summary Test if memory has been locked.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
locked = Pointer to variable taking the result flag.

Return Value ZbMoc Error Codes

Remarks This function is intended for use only with MCO305 controllers. For all other controllers, FALSE will be returned.

A program cannot be executing when this routine is called. If a program is executing, then the routine will return a ZBMOC_E_BREAK_REQUIRED error.

If an error return code is returned, then "locked" is NOT returned.

Portability Firmware ≥ 5.17

12.9.1.4.42 ZbMocMoconBusyWait

```
SIGNED16 ZbMocMoconBusyWait(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED32 timeout);
```

Summary This routine will wait until a controller is no longer busy.

Parameter h = Handle of the interface.
id = Identification number (ID) of the control unit
timeout = Length of time to wait (in ms).

Return Value ZBMOC_OK - Controller is not busy.
ZBMOC_E_IS_BUSY - Controller is still busy after timeout.
ZbMoc Error Codes

Remarks This routine waits in increments of up to 100ms.

12.9.1.4.43 ZbMocMoconClearFactory

```
SIGNED16 ZbMocMoconClearFactory(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Reset the selected controller to the factory settings. Every parameter is overwritten by default values and all programs are erased.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks It is *not* recommended that this function be used on MACS or VLT/SyncPos controllers! Due to hardware limitations of the MACS and VLT/SyncPos controllers, it is not possible to reset them completely using a software command. If this function is used on one of these controllers, then the controller must be switched off and on again before any other function is executed!

If the complete memory content of a controller is to be reset, then it is recommended that it be done by calling ZbMocMoconDeleteEeprom and then switching the controller off and on again!

Portability Firmware ≥ 2.7

12.9.1.4.44 ZbMocMoconClearParameters

```
SIGNED16 ZbMocMoconClearParameters(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Reset all global and axis parameters on the controller to the factory settings.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks Every parameter on the controller is overwritten by default values.

Portability Firmware ≥ 2.7

12.9.1.4.45 ZbMocMoconClearPrograms

```
SIGNED16 ZbMocMoconClearPrograms(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Erase all programs on the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks

Portability Firmware ≥ 2.7

12.9.1.4.46 ZbMocMoconDeleteEeprom

```
SIGNED16 ZbMocMoconDeleteEeprom(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Erase the content of the non-volatile memory (EEPROM) of the controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks It is recommended that the controller be switched off and on again after calling **ZbMocMoconDeleteEeprom**. This function does not erase the content of the volatile memory (RAM). Switching the controller off and on again, makes sure that the erased content of the EEPROM is copied into the RAM again and both memory units are consistent. Otherwise, there is the risk that the existing content of the RAM is copied again into the non-volatile memory (EEPROM) by accident (e.g. by saving a program).

Portability Firmware ≥ 2.7

12.9.1.4.47 ZbMocMoconExecutionStatus

```
SIGNED16 ZbMocMoconExecutionStatus(UNSIGNED16 h, UNSIGNED16 id,
                                   BOOLEAN refresh_first, BOOLEAN *is_executing,
                                   SIGNED16 *errnum, UNSIGNED32 *status);
```

Summary Return the program execution state of the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
refresh_first = Refresh request:
- TRUE - Wait until an updated state is available.
- FALSE - Return buffered state.
is_executing = Pointer to a variable taking the program execution state:
- TRUE - Controller is executing a program.
- FALSE - Controller does not execute a program.
errnum = Pointer to a variable taking the error code of the controller (or 0 if controller is not in error state).
status = Pointer to a variable taking the 4 byte status.

Return Value ZbMoc Error Codes

Remarks ZbMoc caches the current state of the controller. This state is updated periodically during the normal operation of ZbMoc whether or not the application is calling any ZbMoc functions. The frequency of this update depends on several factors including the type of interface, the rate at which print messages are being polled, and which other ZbMoc functions are being called by the application. If an up-to-date status is not critical to the calling application, then "refresh_first" should be set to FALSE. This will return the cached status and avoid putting any additional load of the controller. If an up-to-date status is critical, then "refresh_first" should be set to TRUE. This will make an explicit request to the controller for status information and will not return to the calling application until the controller has replied. Note that in the case of a connection to a Danfoss VLT5000 device, "refresh_first" should always be set to TRUE.

Portability Firmware ≥ 2.7: No support of the V24 serial interface.
Firmware ≥ 5.18: Support for all types of interfaces

12.9.1.4.48 ZbMocMoconFind

```
#define ZBMOC_FIND_INDEX  0    // Search by index.
#define ZBMOC_FIND_SERIAL 1    // Search by serial number.
#define ZBMOC_FIND_IPADDR 2    // Search by IP address.

SIGNED32 ZbMocMoconFind(UNSIGNED16 h, UNSIGNED16 how, UNSIGNED32 value);
```

Summary Find a controller ID based on one of several search criteria.

Parameter h = Handle of the interface.
how = Search method:
0 - Search by index.
1 - Search by serial number.
2 - Search by IP address.

value = Search value corresponding to search method.

Return Value ≥0 - ID of the controller.
<0 - ZbMoc Error Codes

Remarks The returned controller ID can be used with the ZbMocMoconInfo routine or any of the other ZbMoc routines. Note that not all search methods can be used with all interface types. For example, searching by IP address will find no controllers on a USB interface.

In certain cases, this routine may find a controller that was not available when the interface was first opened. For example, when a TCP interface is being used and a controller is powered on after the interface is opened, then this routine may find the controller if the search is being made by IP address.

12.9.1.4.49 ZbMocMoconIdent

```
SIGNED16 ZbMocMoconIdent(UNSIGNED16 h, UNSIGNED16 id, ZbMocMoconIdentS *ident);
```

Summary Return identification information about the specified controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
ident = ZbMocMoconIdentS structure that will received a copy of the data.

Return Value ZbMoc Error Codes

Remarks The returned information is read out of the controller during execution of the '`ZbMocOpen`...' function. This routine can be used when the ID's of the controllers are known. If the controller ID's are not known, then ZbMocMoconInfoIdx can be used. Note that a call to this routine is faster than a call to ZbMocMoconInfo.

12.9.1.4.50 ZbMocMoconInfo

```
SIGNED16 ZbMocMoconInfo(UNSIGNED16 h, UNSIGNED16 id, ZbMocMoconInfoS *info);
```

Summary Query general information about the specified controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
info = ZbMocMoconInfoS structure that will received a copy of the info data.

Return Value ZbMoc Error Codes

Remarks The returned information is read out of the controller during execution of the '`ZbMocOpen`...' function. This routine can be used when the ID's of the controllers are known. If the controller ID's are not known, then ZbMocMoconInfoIdx can be used.

12.9.1.4.51 ZbMocMoconInfoIdx

```
SIGNED16 ZbMocMoconInfoIdx(UNSIGNED16 h, UNSIGNED16 idx, ZbMocMoconInfoS *info);
```

Summary Return information about the selected controller.

Parameter h = Handle of the interface.
idx = Index of the controller: 0...ZbMocGetNumber()-1
info = Structure that will received a copy of the info data.

Return Value ZbMoc Error Codes

Remarks The returned information is read out of the controller during execution of the '`ZbMocOpen`...' function. This routine can be used when the ID's of the controllers are not known. If the controller ID's are known, then ZbMocMoconInfo can be used.

12.9.1.4.52 ZbMocMoconNameSet

```
SIGNED16 ZbMocMoconNameSet(UNSIGNED16 h, UNSIGNED16 id, const STRUTF8 *name);
```

Summary Set the ASCII-based project name of the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
name = Pointer to a data buffer holding the name.

Return Value ZbMoc Error Codes

Remarks The name can be a maximum of 10 characters long and should not contain any special (i.e. non-printable) characters. If the name is less than 10 characters long, then it will be padded with space characters. If the name is longer than 10 characters, then it will be truncated. For newer SDO-based controllers, then name cannot be longer than 8 characters.

Portability Firmware ≥ 2.7

12.9.1.4.53 ZbMocMoconSaveRam

```
SIGNED16 ZbMocMoconSaveRam(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Save the content of the RAM in non-volatile memory.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ZbMoc Error Codes

Remarks The execution of this command may take some time (up to 20 seconds) depending on the type of non-volatile memory used in the controller.

If the controller has battery-buffered RAM, then it is not necessary to execute this function.

Portability Firmware ≥ 2.7

12.9.1.4.54 ZbMocMoconSaveRamSection

```
#define MOC_RAMSECTION_AXISPAR      0   // Save axis parameters.
#define MOC_RAMSECTION_GLOBALPAR    1   // Save global parameters.
#define MOC_RAMSECTION_PROGRAMS     2   // Save programs.
#define MOC_RAMSECTION_ARRAYS       3   // Save arrays.
#define MOC_RAMSECTION_ALL          4   // Save all.
#define MOC_RAMSECTION_USERPAR      5   // Save user parameters.
#define MOC_RAMSECTION_IAXPAR       6   // Save internal axis parameters.
```

```
SIGNED16 ZbMocMoconSaveRamSection(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 section);
```

Summary Save the content of the specified RAM section in non-volatile memory.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
section = ID of the RAM section to be saved (see above).

Return Value ZbMoc Error Codes

Remarks This function and the selective saving of a RAM section, is available only for controllers with a firmware version of 6.4.3 or later. For controllers older than this, ZbMocMoconSaveRam must be used.

Note that saving RAM strongly influences the speed of program execution. There is no ApossIDE command, interrupt, or error handling processed during execution of this function. This means that the program might not react for some seconds depending on the selected RAM section. It is strongly recommended that this function NOT be called for saving programs, arrays, or the complete RAM while the controller is executing a program! Use the ZbMocBreak function to stop execution before saving RAM!

The execution of this command might take some seconds, depending on the type of non-volatile memory used in the controller and the RAM section selected.

If the controller has battery-buffered RAM, then it is not necessary to execute this function.

Portability Firmware ≥ 6.0.1

12.9.1.4.55 ZbMocMoconSetCanParameters

```
SIGNED16 ZbMocMoconSetCanParameters(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 cannr,
                                    UNSIGNED8 baud);
```

Summary Define the CAN communication parameters of the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
cannr = New CAN ID.
baud = New CAN baud rate:

no	baud rate
1	10kbps
2	20kbps
3	50kbps

4 | 100kbps
5 | 125kbps
6 | 250kbps
7 | 500kbps
8 | 1000kbps

Return Value ZbMoc Error Codes

Remarks Any modifications to the CAN parameters are applied immediately. As a result, it is recommended that this function be used only if a non-CAN (e.g. serial) interface is being used. If a CAN interface is being used, then changing the CAN parameters will result in the connection to the controller being lost. In this case, it will be necessary to clos and reopen the ZbMoc connection before the controller can be accessed again.

Portability Firmware ≥ 2.7

12.9.1.4.56 ZbMocMoconState

```
SIGNED16 ZbMocMoconState(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Return connection state of the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value ≥ 0 - Connection state of controller.
< 0 - ZbMoc Error Codes

Remarks

12.9.1.4.57 ZbMocMoconWaitForString

```
SIGNED16 ZbMocMoconWaitForString(UNSIGNED16 h, UNSIGNED16 id, char *target,
                                 UNSIGNED32 timeout);
```

Summary Wait until the given string is received via serial port of the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
target = Pointer to a character array holding the string to wait for.
timeout = Maximum period of time [ms] for receiving the given string. 0 = Do not wait (i.e. a string must present immediately). If the string is not received within the defined time slot, the function returns an error code.

Return Value ZbMoc Error Codes

Remarks Strings differing from the defined one are ignored.

This function is only available if serial (V24) communication is used.

Portability Firmware ≥ 2.7

12.9.1.4.58 ZbMocOpenCanLpt

```
#define ZBMOC_CAN_SDOALT    0x0001  // Switch to alternate SDO CAN ID (0x680/0x100)
                                    // if possible.
#define ZBMOC_CAN_NONMT     0x0002  // Do not send NMT when interface is opened.
#define ZBMOC_CAN_SDOPROBE  0x0004  // Probe non-responsive ID's for CANopen devices.
#define ZBMOC_CAN_LISTEN    0x0008  // Listen for CAN messages from all ID's.
#define ZBMOC_CAN_SDOFORCE  0x0010  // Force use of alternate SDO CAN ID
                                    // (0x680/0x100).
#define ZBMOC_CAN_DEMOTE    0x0020  // Demote device types (to ZBMOC_CAN_SDOPROBE or
                                    // ZBMOC_CAN_LISTEN).

typedef struct
{
    UNSIGNED16  lptnr;
    UNSIGNED16  irq;
    UNSIGNED16  mode;
    UNSIGNED16  baud;
    UNSIGNED16  scanfrom;
    UNSIGNED16  scanto;
    UNSIGNED32  glbdelay;
    UNSIGNED32  snddelay;
    UNSIGNED32  flags;
} ZbMocOpenCanLptParamS;

SIGNED16 ZbMocOpenCanLpt(ZbMocOpenCanLptParamS *param);
```

Summary Open the CAN interface using CAN-adapter for PC's auxiliary (centronics) port.

Parameter lptnr = Number of the auxiliary (centronics) port:
0 - LPT1
1 - LPT2
-1 - Check all ports automatically and evaluate corresponding IRQ.
irq = Interrupt used by the auxiliary (centronics) port.
mode = CAN card mode: 0 - SPP, 1 - EPP, 2 - PS2, 3 - Auto
baud = Baud rate of the CAN bus:
1 - 10kbps
2 - 20kbps
3 - 50kbps
4 - 100kbps
5 - 125kbps
6 - 250kbps
7 - 500kbps
8 - 1000kbps
scanfrom = First CAN ID to be checked for the presence of a device.
scanto = Last CAN ID to be checked for the presence of a device.
glbdelay = Delay (in ms) after a global message has been sent before another message may be sent.
snddelay = Delay (in ms) between sent messages for controllers older than firmware version 6.4.0.
flags = Flags:
Name | Bit | Description

---|---|---
ZBMOC_CAN_SDOALT | 0x0001 | Use alternate SDO CAN ID (0x680/0x100).
ZBMOC_CAN_NONMT | 0x0002 | Do not send NMT when interface is opened.
ZBMOC_CAN_SDOPROBE | 0x0004 | Probe non-responsive ID's for CANopen devices.
ZBMOC_CAN_LISTEN | 0x0008 | Listen for CAN messages from all ID's.
ZBMOC_CAN_SDOFORCE | 0x0010 | Force use of alternate SDO CAN ID (0x680/0x100).
ZBMOC_CAN_DEMOTE | 0x0020 | Demote device types (to ZBMOC_CAN_SDOPROBE or ZBMOC_CAN_LISTEN).

Return Value ≥ 0 - Handle of the interface connection.
< 0 - ZbMoc Error Codes

Remarks It is not possible to simultaneously open more than one CAN-LPT interface.

All controllers (within the defined ID range) are addressed and configured automatically for usage by ZbMoc. The version information and configuration of each controller is read and saved for later access by the application program via the ZbMocMoconInfo function. Controllers which have an ID outside the defined range are ignored.

If at least one controller is found, then the interface will be opened successfully and the handle of the interface that was established is returned. If no controller at all responds, then a ZBMOC_E_CAN_NO_SLAVE error code is returned and the connection is closed again.

ZbMoc will normally find and connect to only zub controllers. However, if the ZBMOC_CAN_SDOPROBE flag is specified, then ZbMoc will also search for other CANopen-compliant devices on the CAN bus. A subset of ZbMoc functionality (i.e. SDO and PDO usage) will be avilable for these devices. Also, if the ZBMOC_CAN_LISTEN flag is specified, then ZbMoc will listen for incoming PDO messages from ALL device ID's in the scan range (i.e. including devices that are not CANopen-compliant). In some situations, it may be useful to connect to a zub controller as if it were a simple CANopen device (or even a non-CANopen device). In this case, the ZBMOC_CAN_DEMOTE flag can be specified.

Portability Firmware ≥ 2.7

12.9.1.4.59 ZbMocOpenCanUsb

```
typedef struct
{
    UNSIGNED16  port;
    UNSIGNED16  baud;
    UNSIGNED16  scanfrom;
    UNSIGNED16  scanto;
    UNSIGNED32  flags;
} ZbMocOpenCanUsbParamS;

SIGNED16 ZbMocOpenCanUsb(ZbMocOpenCanUsbParamS *param);
```

Summary Open the CAN-USB interface for controller communication.

Parameter port = Com port: 1 - COM1, 2 - COM2, etc.

baud = Baud rate of the CAN bus:
- 1 - 10kbps
- 2 - 20kbps
- 3 - 50kbps
- 4 - 100kbps
- 5 - 125kbps
- 6 - 250kbps
- 7 - 500kbps
- 8 - 1000kbps

scanfrom = First CAN ID to be checked for the presence of a device.
scanto = Last CAN ID to be checked for the presence of a device.
flags = Flags (same as ZbMocOpenCanLpt).

Return Value ≥ 0 - Handle of the interface connection.
< 0 - ZbMoc Error Codes

Remarks All controllers found on USB connections are configured automatically for usage by ZbMoc. The version information and configuration of each controller is read and saved for later access by the application program via the ZbMocMoconInfo function. Controllers which have an ID outside the defined range are ignored.

If at least one controller is found, then the interface will be opened successfully and the handle of the interface that was established is returned. If no controller at all responds, then a ZBMOC_E_CAN_NO_SLAVE error code is returned and the connection is closed again.

Portability Firmware ≥ 2.7

12.9.1.4.60 ZbMocOpenEtherCat

```
#define ZBMOC_CAT_SCANALL  0x0001   // Scan all ports.

typedef struct
{
    UNSIGNED8    netid[6];
    UNSIGNED16   scanfrom;
    UNSIGNED16   scanto;
    UNSIGNED32   flags;
    UNSIGNED32   mskb;
} ZbMocOpenEtherCatParamS;

SIGNED16 ZbMocOpenEtherCat(ZbMocOpenEtherCatParamS *param);
```

Summary Open the EtherCAT interface for controller communication.

Parameter netid = AMS Net ID of controller on the EtherCAT bus.
scanfrom = First port to be checked for the presence of a device.
scanto = Last port to be checked for the presence of a device.
flags = Flags:

Name	Bit	Meaning
ZBMOC_CAT_SCANALL	0x0001	Scan all ports.

mskb = Milliseconds/kilobyte timeout value for download/upload.

Return Value ≥ 0 - Handle of the interface connection.
< 0 - ZbMoc Error Codes

Remarks All controllers found on EtherCAT connections are configured automatically for usage by ZbMoc. The version information and configuration of each controller is read and saved for later access by the application program via the ZbMocMoconInfo function. Controllers which have an ID outside the defined range are ignored.

If at least one controller is found, then the interface will be opened successfully and the handle of the interface that was established is returned. If no controller at all responds, then a ZBMOC_E_NOCONTROLLER" error code is returned and the connection is closed again. If the EtherCAT master itself is not turned on, then a ZBMOC_E_COM_OPENERROR error code is returned and the connection is also closed.

Portability All controllers that contain an EtherCAT port.

12.9.1.4.61 ZbMocOpenSerial

```
#define ZBMOC_SER_SXP    0x0001    // Enable switching to SXP.
#define ZBMOC_SER_SDO2   0x0002    // Enable SDO2 communication.
#define ZBMOC_SER_XON    0x0004    // Enable Xon/Xoff support.
#define ZBMOC_SER_BREAK  0x0008    // Send "break" if no response.
#define ZBMOC_SER_RS485  0x0010    // RS485 connection.

typedef struct
{
    UNSIGNED16  comnr;
    UNSIGNED32  baud;
    UNSIGNED16  parity;
    UNSIGNED16  retry;
    UNSIGNED32  timeout;
    UNSIGNED32  overfeed;
    UNSIGNED32  flags;
} ZbMocOpenSerialParamS;

SIGNED16 ZbMocOpenSerial(ZbMocOpenSerialParamS *param, BOOLEAN *prot);
```

Summary Open an enhanced serial interface using COM communication. This will initially create a V24 connection at 9600 baud. If this is successful and the controller can support the X-protocol, then the interface will be automatically switched to an SXP interface at the specified baud rate.

Parameter comnr = Number of the COM port: 0 - COM1, 1 - COM2, etc.
baud = Baud rate [bps] of the communication: 9600, 115200, etc.
parity = Parity: 0 = None, 1 = Odd, 2 = Even, 3 = Mark, 4 = Space
retry = The number of times to attempt to send an X-protocol message before assuming a loss of connection.
timeout = The timeout (in milliseconds) between X-protocol message send retries.
overfeed = The timeout (in milliseconds) between the end of a received X-protocol message and the start of the next sent message. This is used to allow a half-duplex RS485 converter time to switch modes.

flags = Flags:
Name	Bit	Meaning
ZBMOC_SER_SXP	0x0001	Enable switching to SXP.
ZBMOC_SER_SDO2	0x0002	Enable SDO2 communication.
ZBMOC_SER_XON	0x0004	Enable Xon/Xoff support.
ZBMOC_SER_BREAK	0x0008	Send "break" if no response.
ZBMOC_SER_RS485	0x0010	RS485 connection.
prot = Final protocol. This will be FALSE if the final connection is a V24 connection and TRUE if the final connection is an X-protocol connection.

Return Value ≥ 0 - Handle of the interface connection.
< 0 - ZbMoc Error Codes

Remarks The controller is configured automatically for usage by ZbMoc. The version information and configuration of the controller is read and saved for later access by the application program via the ZbMocMoconInfo function. Controllers which have an ID outside the defined range are ignored.

If a controller is found, then the interface will be opened successfully and the handle of the interface that was established is returned. If no controller responds, then a ZBMOC_E_NOCONTROLLER error code is returned and the connection is closed again.

Portability Firmware ≥ 2.7

12.9.1.4.62 ZbMocOpenTcp

```
#define ZBMOC_TCP_TCP      0x0001   // Use TCP (rather than UDP).
#define ZBMOC_TCP_START    0x0002   // Reserved.
#define ZBMOC_TCP_SDO2     0x0004   // Enable SDO2 communication.
#define ZBMOC_TCP_SEARCH   0x0008   // Search for all controllers.

typedef struct
{
    BOOLEAN     local;
    UNSIGNED32  address;
    UNSIGNED32  port;
    UNSIGNED16  retry;
    UNSIGNED32  timeout;
    UNSIGNED32  flags;
} ZbMocOpenTcpParamS;

SIGNED16 ZbMocOpenTcp(ZbMocOpenTcpParamS *param);
```

Summary Opens the TCP interface for control unit communication.

Parameter local = Reserved. Set to FALSE.
address = TCP address of host.
port = TCP port to use.
retry = The number of times to attempt to send a message before assuming a loss of connection.
timeout = The timeout (in milliseconds) between message send retries.
flags = Flags:

Name	Bit	Meaning
ZBMOC_TCP_TCP	0x0001	Use TCP. This must be set.
ZBMOC_TCP_START	0x0002	Reserved.
ZBMOC_TCP_SDO2	0x0004	Enable SDO2 communication.
ZBMOC_TCP_SEARCH	0x0008	Search for all controllers.

Return Value ≥ 0 - Handle of the interface connection.
< 0 - ZbMoc Error Codes

Remarks If the ZBMOC_TCP_SEARCH flag is set, then all control units found on TCP connections are configured automatically for usage by the library. If this flag is not set, then only the specified IP address will be used. Note that ZBMOC_TCP_SEARCH will not find controllers with firmware older than 7.1.16. To connect to older controllers, the IP address must be explicitly specified.

If no controller at all is responding a ZBMOC_E_NOCONTROLLER error code is returned and the connection is closed again. Otherwise, the handle of the connection that was established is returned.

Portability All controllers that contain an Ethernet port.

12.9.1.4.63 ZbMocOpenUsb

```
#define ZBMOC_USB_ALLPID 0x0100    // Allow all known PIDs.

typedef struct
{
    UNSIGNED32  baud;
    UNSIGNED16  retry;
    UNSIGNED32  timeout;
    UNSIGNED32  flags;
    UNSIGNED32  latency;
} ZbMocOpenUsbParamS;

SIGNED16 ZbMocOpenUsb(ZbMocOpenUsbParamS *param);
```

Summary Open the USB serial interface for controller communication.

Parameter baud = Baud rate [bps] of the underlying USB chip interface. This should normally be set to 921600.
retry = The number of times to attempt to send an X-protocol message before assuming a loss of connection.
timeout = The timeout (in milliseconds) between X-protocol message send retries.
flags = Flags:
Name	Bit	Meaning
ZBMOC_SER_SDO2	0x0002	Enable SDO2 communication.
ZBMOC_USB_ALLPID	0x0100	Allow all known PIDs.
latency = Latency timer (ms). This should be set to 1.

Return Value ≥ 0 - Handle of the interface connection.

< 0 - ZbMoc Error Codes

Remarks All controllers found on USB connections are configured automatically for usage by ZbMoc. The version information and configuration of each controller is read and saved for later access by the application program via the ZbMocMoconInfo function.

If at least one controller is found, then the interface will be opened successfully and the handle of the interface that was established is returned. If no controller at all responds, then a ZBMOC_E_VLT_SCAN_NO_SLAVES error code is returned and the connection is closed again.

When a USB interface is opened using **ZbMocOpenUsb**, the controllers on the interface are not physically opened for communication. Instead, they will be in a "pending" state (see the ZbMocMoconInfoS structure). The ZbMocConnect routine must be called to finialize the connection before communication with the controller can be started.

Portability All controllers that contain a USB port.

12.9.1.4.64 ZbMocOpenV24

```
typedef struct
{
    UNSIGNED16  comnr;
    UNSIGNED32  flags;
} ZbMocOpenV24ParamS;

SIGNED16 ZbMocOpenV24(ZbMocOpenV24ParamS *param);
```

Summary Open the V24 serial interface for controller communication. This function is obsolete. It should be replaced with "ZbMocOpenSerial()".

Parameter comnr = Number of the COM port: 0 - COM1, 1 - COM2, etc.
flags = Flags:
Name	Bit	Meaning
ZBMOC_SER_SDO2	0x0002	Enable SDO2 communication.
ZBMOC_SER_XON	0x0004	Enable Xon/Xoff support.
ZBMOC_SER_BREAK	0x0008	Send "break" if no response.

Return Value ≥ 0 - Handle of the interface connection.
< 0 - ZbMoc Error Codes

Remarks The V24 settings must be: 9600 baud, 8 data bits, 1 stop bit, no parity.

The controller is configured automatically for usage by ZbMoc. The version information and configuration of the controller is read and saved for later access by the application program via the ZbMocMoconInfo function.

If a controller is found, then the interface will be opened successfully and the handle of the interface that was established is returned. If no controller responds, then a ZBMOC_E_NOCONTROLLER error code is returned and the connection is closed again.

Portability Firmware ≥ 2.7

12.9.1.4.65 ZbMocOpenVlt

```
#define VLT_STANDARD  0
#define VLT_TSTDCOM   1
#define VLT_USEDCOM   2
#define VLT_SXP       3
#define VLT_USB       4
#define VLT_TCP       5
#define VLT_MCO       6
#define VLT_SIM       7

typedef struct
{
    UNSIGNED16  comnr;
    UNSIGNED32  baud;
    UNSIGNED16  retry;
    UNSIGNED32  timeout;
    UNSIGNED32  overfeed;
    char        drivername[80];
    UNSIGNED16  mode;
    UNSIGNED16  scanfrom;
    UNSIGNED16  scanto;
    UNSIGNED32  connid;
} ZbMocOpenVltParamS;

SIGNED16 ZbMocOpenVlt(ZbMocOpenVltParamS *param);
```

Summary Open the VLT interface using the RS485 or DCOM communication. This interface can only be used with Danfoss VLT500 and FC300-family devices.

Parameter comnr = Number of the COM port: 0 - COM1, 1 - COM2, etc.
baud = Baud rate [bps] of the VLT/RS485 communication: 9600, 115200, etc.
retry = The number of times to attempt to send an X-protocol message before assuming a loss of connection.
timeout = The timeout (in milliseconds) between X-protocol message send retries.
overfeed = The timeout (in milliseconds) between the end of a received X-protocol message and the start of the next sent message. This is used to allow a half-duplex RS485 converter time to switch modes.
drivername = Name of the MCT10 communication driver.
mode = Operating mode of the VLT interface:
no	name	meaning
0	VLT_STANDARD	(VLT/RS485 communication)
1	VLT_TSTDCOM	(Not supported)
2	VLT_USEDCOM	(VLT communication via DCOM)
3	VLT_SXP	(Not supported)
4	VLT_USB	(Not supported)
5	VLT_TCP	(Not supported)
6	VLT_MCO	(Not supported)
7	VLT_SIM	(Not supported)
scanfrom = First VLT ID to be checked for the presence of a device.
scanto = Last VLT ID to be checked for the presence of a device.

connid = Connection ID for MTComm.dll (mode = VLT_USEDCOM).

Return Value ≥ 0 - Handle of the interface connection.
< 0 - ZbMoc Error Codes

Remarks It is not possible to simultaneously open more than one VLT interface.

The default VLT settings typically in use (for the RS485 interface) are: 9600 baud, 8 data bits, 1 stop bit, even parity.

For VLT_STANDARD mode, the parameters "drivername" and "connid" are ignored. For VLT_USEDCOM mode, the parameters "comnr" and "baud" are ignored and "drivername" and "connid" have MCT10-specific values.

All controllers found on the VLT network are configured automatically for usage by ZbMoc. The version information and configuration of each controller is read and saved for later access by the application program via the ZbMocMoconInfo function. Controllers which have an ID outside the defined range are ignored.

If at least one controller is found, then the interface will be opened successfully and the handle of the interface that was established is returned. If no controller at all responds, then a ZBMOC_E_VLT_SCAN_NO_SLAVES error code is returned and the connection is closed again.

Portability Firmware ≥ 2.7

12.9.1.4.66 ZbMocOpenSim

```
#define ZBMOC_SIM_SDO2     0x0002   // Enable SDO2 communication.
#define ZBMOC_SIM_REMOTE   0x0004   // Connect to remote computer (i.e. use UDP
communication).

typedef struct
{
    UNSIGNED32  flags;
} ZbMocOpenSimParamS;

SIGNED16 ZbMocOpenSim(ZbMocOpenSimParamS* param);
```

Summary Open the zub Simulator interface for controller communication.

Parameter flags = Flags:
Name	Bit	Meaning
ZBMOC_SIM_SDO2	0x0002	Enable SDO2 communication.
ZBMOC_SIM_REMOTE	0x0004	Connect to remote computer.

Return Value ≥ 0 - Handle of the interface connection.
< 0 - ZbMoc Error Codes

Remarks The controller is configured automatically for usage by ZbMoc. The version information and configuration of the controller is read and saved for later access by the application program via the ZbMocMoconInfo function. Controllers which have an ID outside the defined range are ignored.

If a controller is found, then the interface will be opened successfully and the handle of the interface that was established is returned. If no controller responds, then a ZBMOC_E_NOCONTROLLER error code is returned and the connection is closed again.

Portability ApossIDE 7.00.84

12.9.1.4.67 ZbMocParamRead

```
SIGNED16 ZbMocParamRead(UNSIGNED16 h, UNSIGNED16 id, ZbMocParamS *para);
```

Summary Read all global parameters from the selected controller. The parameters are decoded and copied into the ZbMocParamS structure.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
para = Pointer to a structure taking the parameters.

Return Value ZbMoc Error Codes

Remarks This function reads all global parameters supported by the selected controller. All parameters in ZbMocParamS that are not supported by the controller, are set to zero.

If any conversion is necessary to get compatible results corresponding to the version of the controller, then this is done internally.

Portability Firmware ≥ 2.7

12.9.1.4.68 ZbMocParamWrite

```
SIGNED16 ZbMocParamWrite(UNSIGNED16 h, UNSIGNED16 id, ZbMocParamS *para);
```

Summary Write all global parameters to the selected controller. The parameters are taken from the ZbMocParamS structure, preprocessed if necessary, and saved in the controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
para = Pointer to structure holding the parameters to write.

Return Value ZbMoc Error Codes

Remarks This function sets all global parameters supported by the selected controller. All parameters in ZbMocParamS that are not supported by the controller, are ignored.

If any conversion is necessary to get compatible results corresponding to the version of the controller, this is done internally before writing the settings.

Portability Firmware ≥ 2.7

12.9.1.4.69 ZbMocPollMessage

```
SIGNED16 ZbMocPollMessage(UNSIGNED16 h, SIGNED16 id, STRUTF8 *message);
```

Summary Retrieves the next message generated by a "print" command in the running application program or by a system message generated by the selected controller itself.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
message = Pointer to a buffer taking the message.

Return Value 0 - Message returned.
1 - No message available.
<0 - ZbMoc Error Codes

Remarks If "id" is -1, then all controllers are polled up until a message is found. Subsequent controllers are not polled. If a message is read, then it will be returned in "message". Messages can be up to MOC_PRINTMESSAGEBUFFER_LEN bytes long. Any controller which is blocked by a previous request, will be treated as if no message is read.

UTF-8 encoded strings are returned only if the application program executing in the controller was written to return UTF-8 encoded strings.

Portability Firmware ≥ 2.7

12.9.1.4.70 ZbMocProbe

```
SIGNED16 ZbMocProbe(UNSIGNED16 h);
```

Summary Search for controllers on the given interface.

Parameter h = Handle of the interface.

Return Value ZbMoc Error Codes

Remarks This function will search the interface for controllers. This can be useful for interface types that support controllers that appear and disappear as they are powered up and down. For example, this behaviour is normal for USB and Ethernet devices. Note that calling this routine may change the number of controllers returned by ZbMocGetNumber and the order of the controllers returned by ZbMocMoconInfoIdx. Controller identification numbers of any open controllers are NOT changed.

12.9.1.4.71 ZbMocProgramDelete

```
SIGNED16 ZbMocProgramDelete(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 prgnr);
```

Summary Deletes the program in the program list with the given program number.

Parameter h = Handle of the interface.
id = Identification number (ID) of the control unit.
prgnr = Number (0...94) used to register the temporary program in the list.

Return Value ZbMoc Error Codes

Remarks

Portability Firmware ≥ 2.7

12.9.1.4.72 ZbMocProgramList

```
typedef struct ZbMocProgramS_
{
    UNSIGNED16 index;           // Program number.
    STRUTF8    name[24+1];      // Program name.
    BOOLEAN    autostart;       // Autostart: TRUE/FALSE
} ZbMocProgramS;

SIGNED16 ZbMocProgramList(UNSIGNED16 h, UNSIGNED16 id, ZbMocProgramS *pl,
                          UNSIGNED16 maxentries);
```

Summary Return a list of all program stored in the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
pl = Pointer to an array of "ZbMocProgramS" structures taking the program numbers, names, and the state of the "Autostart" flag.
maxentries = Number of elements in the array.

Return Value ≥ 0 - Number of programs returned.
< 0 - ZbMoc Error Codes

Remarks The returned list of programs includes all programs which were previously stored using one of the ZbMocProgramSave or ZbMocProgramSaveAs functions. Any downloaded "temporary" program, is not included.

If more than "maxentries" programs have been saved in the controller, then only the first "maxentries" programs are returned. Note that the return code is the number of programs returned and NOT the number of programs that are stored on the controller. The calling application must set "maxentries" to at least "MOC_MAXPROGCNT" to ensure that all programs are retrieved (see MOC_MAXPROGCNT in 'ZbMoc.h' for more information).

UTF-8 encoded program names are returned only if program names have been strored using UTF-8 encoded names. Program names will always be null-terminated.

Portability Firmware ≥ 2.7

12.9.1.4.73 ZbMocProgramLoad

```
SIGNED16 ZbMocProgramLoad(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED32 prglen,
                          UNSIGNED8 (*prg)(UNSIGNED32));
```

Summary Download a binary (i.e. compiled) program to the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
prglen = Length of the binary program in number of bytes.
prg = Pointer to a function returning one byte of program code.

Return Value ZbMoc Error Codes

Remarks The application source program (*.m) must be compiled into binary code by the ApossIDE compiler before a download can take place. The binary code must then be downloaded to the controller. It is not possible to download application source code directly.

This function calls the function "(*prg)()" to get each byte of binary program code (up to "prglen" bytes). The argument to "(*prg)()" is the index of the requested byte. The return value of "(*prg)()" is the corresponding binary program code byte.

The program is stored as a "temporary" program and has no program number or name. A temporary program can be started using ZbMocExecTemp or it can be saved as a "permanent" program using one of the ZbMocProgramSave or ZbMocProgramSaveAs functions. Depending on the firmware version of the controller, it may also be necessary to call the function ZbMocMoconSaveRam to save the program persistently.

> **ZbMocProgramLoad** is designed to be used in conjunction with the ZbCompReadcode routine in the ZbComp Library.

Portability Firmware ≥ 2.7

12.9.1.4.74 ZbMocProgramLoadBinary

```
SIGNED16 ZbMocProgramLoadBinary(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED32 prglen,
                                UNSIGNED8 *prg);
```

Summary Download a binary (i.e. compiled) program to the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
prglen = Length of the binary program in number of bytes.
prg = Pointer to the binary program code.

Return Value ZbMoc Error Codes

Remarks The application source program (*.m) must be compiled into binary code by the ApossIDE compiler before a download can take place. The binary code must then be downloaded to the controller. It is not possible to download application source code directly.

This function is identical to ZbMocProgramLoad except that it takes a pointer to a buffer containing the entire binary program rather than a pointer to a retrieval function.

The program is stored as a "temporary" program and has no program number or name. A temporary program can be started using ZbMocExecTemp or it can be saved as a "permanent" program using one of the ZbMocProgramSave or ZbMocProgramSaveAs functions. Depending on the firmware version of the controller, it may also be necessary to call the function ZbMocMoconSaveRam to save the program persistently.

Portability Firmware ≥ 2.7

12.9.1.4.75 ZbMocProgramReadBinary

```
SIGNED16 ZbMocProgramReadBinary(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 prgnr,
                                UNSIGNED32 maxlen, UNSIGNED8 *prg,
                                UNSIGNED32 *prglen);
```

Summary Reads (i.e. uploads) a binary program from the selected control unit.

Parameter h = Handle of the interface.
id = Identification number (ID) of the control unit.
prgnr = Number (0...94) of program to read.
maxlen = Length of binary program code buffer.
prg = Pointer to the binary program code buffer.
prglen = Length of the binary program in number of bytes (returned).

Return Value ZbMoc Error Codes

Remarks

12.9.1.4.76 ZbMocProgramSave

```
SIGNED16 ZbMocProgramSave(UNSIGNED16 h, UNSIGNED16 id, const STRUTF8 *name);
```

Summary Save the current "temporary" program using the next available program number.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
name = Pointer to a character array holding the program name.

Return Value ≥0 - Program number used to save program.
< 0 - ZbMoc Error Codes

Remarks A program must be downloaded using either the function ZbMocProgramLoad or ZbMocProgramLoadBinary before it can be saved.

The program name may be either UTF-8 encoded or ANSI encoded. However, it may not consist of more than 8 bytes (not including the null-terminator).

The program is saved as a new program using the next available program number. If the program number must be defined manually or an existing program must be overwritten, then the function ZbMocProgramSaveAs should be used instead.

Depending on the firmware version of the controller, it may also be necessary to call the function ZbMocMoconSaveRam afterwards to save the program persistently.

Portability Firmware ≥ 2.7

12.9.1.4.77 ZbMocProgramSaveAs

```
SIGNED16 ZbMocProgramSaveAs(UNSIGNED16 h, UNSIGNED16 id, const STRUTF8 *name,
                            UNSIGNED16 prgnr);
```

Summary Save the current "temporary" program using the given program number.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
name = Pointer to a character array holding the program name.
prgnr = Program number used to save the program.

Return Value ZbMoc Error Codes

Remarks A program must be downloaded using either the function ZbMocProgramLoad or ZbMocProgramLoadBinary before it can be saved.

The program name may be either UTF-8 encoded or ANSI encoded. However, it may not consist of more than 8 bytes (not including the null-terminator). Program numbers can be in the range 0 to MOC_MAXPROGCNT-1. Note that controllers with firmware versions 7.00.00 and later support more program numbers (see MOC_MAXPROGCNT in 'ZbMoc.h' for more information). If a program with the same program number already exists, then the existing program is erased and replaced with the new program.

Depending on the firmware version of the controller, it may also be necessary to call the function ZbMocMoconSaveRam afterwards to save the program persistently.

Portability Firmware ≥ 2.7

12.9.1.4.78 ZbMocReadMemPage

```
SIGNED16 ZbMocReadMemPage(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED32 pageaddress,
                          UNSIGNED16 *crc, UNSIGNED8 *values);
```

Summary Read a memory page out of the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
pageaddress = Number of the memory.
crc = Pointer to a variable taking the CRC.
values = Pointer to an array taking the received 256 bytes of data.

Return Value ZbMoc Error Codes

Remarks Each memory page contains 256 bytes.

This function is only available if serial (V24) communication is used.

Portability MVS Version only

12.9.1.4.79 ZbMocReadUserArray

```
SIGNED16 ZbMocReadUserArray(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 arrnr,
                            UNSIGNED32 mxlen, SIGNED32 *outarray,
                            UNSIGNED32 *arrsize);
```

Summary Read a user-defined array from the selected controller.

Parameter h = Handle of the interface.

id = Identification number (ID) of the controller.
arrnr = Number of the array (0...n).
mxlen = Number of values in the "values" array (i.e. NOT the byte length).
outarray = Pointer to an array taking the received user data.
arrsize = Pointer to a variable taking the actual number of values in the array on the controller.

Return Value ZbMoc Error Codes

Remarks Arrays are numbered consecutively, starting with 0.

The number of values copied from the array, is the smaller of either "maxlen" or the actual number of values in the array. However, the actual number of values in the array, is always returned in "arrsize". The application can use this to determine whether or not the entire array has been read. If the array does not exist, then "arrsize" will be set to 0. This can be used to detect the last valid array.

Portability Firmware ≥ 2.7

12.9.1.4.80 ZbMocReadUserVar

```
SIGNED16 ZbMocReadUserVar(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 *varnr,
                          SIGNED32 *val, UNSIGNED32 timeout);
```

Summary Read a data message from the selected controller. A message can be sent from an application program by using the OUTMSG command. The message always contains one 16 bit data value and one 32 bit data value.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
varnr = Pointer to a variable taking the 16 bit value.
val = Pointer to a variable taking the 32 bit value.
timeout = Maximum period of time [ms] for receiving the message.
0 = Do not wait.
If there is no message received within the defined time slot, the function returns an error code.

Return Value ZbMoc Error Codes

Remarks If the timeout is set to 0, then a message is expected to be available immediately. If no message is available, then **ZbMocReadUserVar** will wait up to 1 second for a message to become available. If this routine is being used simply to poll whether or not a message is available, then the timeout should be set to 1. The routine will then return after a maximum of 1 ms.

This function is not available for serial (V24) communication!

Portability Firmware ≥ 2.7

12.9.1.4.81 ZbMocSDOGetUsage

```
BOOLEAN ZbMocSDOGetUsage(UNSIGNED16 h, UNSIGNED16 id);
```

Summary Return a flag indicating whether or not SDO-based communication MAY be used internally for ZbMoc function processing.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

Return Value TRUE - Internal SDO-based communication MAY be used.
FALSE - Internal SDO-based communication CANNOT be used.

Remarks When "FALSE" is returned (i.e. SDO-based communication for ZbMoc function processing is not in use), this is usually due to a controller not supporting the zub SDO object dictionary.

More detailed information (i.e. if SDO-based communication would currently be used and if the controller is capable of it) can be retrieved by the function ZbMocSDOGetUsageDetails.

zub controllers starting with firmware version 6.01.00 support SDO-based communication. SDO-based communication allows many functions (reading and writing parameter values in particular) to be executed by the controller regardless of the controller's execution state. Prior to version 6.01.00, many ZbMoc functions would return a ZBMOC_E_IS_BUSY return code if the controller was already busy executing an application program.

12.9.1.4.82 ZbMocSDOGetUsageDetails

```
BOOLEAN ZbMocSDOGetUsageDetails(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 *level,
                                BOOLEAN *sdosupport);
```

Summary Return a flag indicating whether or not SDO-based communication would currently be used (if possible) for ZbMoc function processing of the given controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
level = Pointer to a variable that will return the level of SDO support provided by the controller.
sdosupport = Pointer to a variable that will return a flag indicating whether or not SDO-based communication is supported by the controller.

Return Value TRUE - Internal SDO-based communication would currently be used (if possible).
FALSE - Internal SDO-based communication would not currently be used.

Remarks The return value depends on the following factors:

- SDO-based communication compatibility of the controller.
- Execution status of the controller.

Note that this is a temporary snapshot. Whether or not SDO-based communication is actually used or not for processing of the next ZbMoc function call, depends on the function call and the execution state of the controller at the time of the next function

call.

zub controllers starting with firmware version 6.01.00 support SDO-based communication. SDO-based communication allows many functions (reading and writing parameter values in particular) to be executed by the controller regardless of the controller's execution state. Prior to version 6.01.00, many ZbMoc functions would return a ZBMOC_E_IS_BUSY return code if the controller was already busy executing an application program.

12.9.1.4.83 ZbMocSetCallbackPDO

```
void ZbMocSetCallbackPDO(void (*cb)(UNSIGNED16 h, UNSIGNED16 id, double time,
                                    SIGNED32 pdo, SIGNED32 len, UNSIGNED8 *dat));
```

Summary Defines a pointer to a callback function which will receive PDO messages.

Parameter cb = Pointer to a callback function.

Remarks This function registers a callback function for receiving PDO messages. The ZbMoc library calls this callback function whenever a PDO message is received. See the description of ZbMocCanOpenPollPDO for an explanation of the parameters to the callback function.

Portability Firmware ≥ 5.21

12.9.1.4.84 ZbMocSetCallbackProgress

```
void ZbMocSetCallbackProgress(UNSIGNED32 (*cb)(UNSIGNED32 data1, UNSIGNED32 data2,
                                               UNSIGNED32 code, UNSIGNED32 value),
                              UNSIGNED32 data1, UNSIGNED32 data2);
```

Summary Defines a callback pointer to a function called to provide progress monitoring.

Parameter cb = Pointer to callback function.
data1 = Arbitrary user-defined value that will be passed to the callback function.
data2 = Arbitrary user-defined value that will be passed to the callback function.

Remarks The callback function will have the following arguments:

data1 - An arbitrary value that will be passed back to the caller.
data2 - An arbitrary value that will be passed back to the caller.
code - A code indicating the type of call. This may be any of the following values:
0 - Progress value.
1 - Wait call.
2 - Enable any "Cancel" button. The "Cancel" button should be disabled by default.
3 - Set the "high" value (i.e. the total).
4 - Show the progress dialog. The progress dialog should be hidden by default.
value - A value depending on "code" as follows:
0 - The amount of progress completed. This will go from 0 up to "high".

1 - The value can be ignored.
2 - The value can be ignored.
3 - The "high" value.
4 - The value can be ignored.

The callback function is expected to return the following return codes:

0 - Continue.
1 - Abort the process.

The return code is ignored for all codes except code 0. Note that some processes cannot be aborted.

12.9.1.4.85 ZbMocSetDebugFile

```
void ZbMocSetDebugFile(const STRUTF8 *filename);
```

Summary Define the log file (complete name including path), in which debugging messages are written. If this routine has not been called to set a file name, then no debugging messages are written.

Parameter filename = Path and filename of debug log file.

Remarks The given string holding the filename and path information should not exceed 259 characters in length. Names longer than this will be truncated. Strings may be specified using either ANSI or UTF-8 encoding.

Calling this routine effectively restarts the log system. Hence, care must be taken when calling this routine if the "new file" flag (i.e. 0x00000010) has been set using ZbMocSetDebugFlags since this will cause the previous file to be erased even if the same file name has been specified.

12.9.1.4.86 ZbMocSetDebugFlags

```
void ZbMocSetDebugFlags(UNSIGNED32 flags);
```

Summary Define the debugging flags.

Parameter flags = Debugging flags. This can be a combination of:
Bit	Meaning
0x00000001	Write elapsed seconds since start.
0x00000004	Indent messages.
0x00000008	Flush and close file after every message.
0x00000010	Create a new file, erasing any previous file.
0x00000020	Write timestamps.
0x00000040	Write thread number.

Remarks The default value of the debug flags is 0x00000004 (i.e. indent messages).

If the indent flag is set, then messages will generally be indented more and more, the further execution goes down the routine calling stack.

If the flush flag is set, then the log file is flushed and closed after every message is written. This will ensure that all messages are safely written to the file in the case of a crash and no messages are lost. However, it can significantly increase the execution time of calls.

If the new file flag is set, then any existing log file of the same name will be erased when the first NEW message is written.

12.9.1.4.87 ZbMocSetDebugHistory

```
void ZbMocSetDebugHistory(SIGNED32 max);
```

Summary Set the maximum size of the debug history.

Parameter max = Maximum number of messages to save in history. 0 for none.

Remarks If the history size is set to 0, then all debug messages are written to the log file. If the history size is greater than 0, then all non-error debug messages are saved in the history and are not written to the log file. Only the last "max" messages are saved; older messages will be lost. When an error message is to be written, then all saved messages are written before the error message is written.

12.9.1.4.88 ZbMocSetDebugImmediate

```
void ZbMocSetDebugImmediate(void);
```

Summary Start a ZbMoc debug log immediately using default level and flag settings.

Remarks Calling this routine will start the log system immediately. The log file will be called "ZbMocDebug.dm" and it will be placed in the standard Windows "Application Data" directory. If this routine is used, then the other '`ZbMocSetDebug`...' routines should not be called.

12.9.1.4.89 ZbMocSetDebugLevel

```
void ZbMocSetDebugLevel(SIGNED16 level);
```

Summary Define the debugging level.

Parameter level = Debugging level:
Value	Meaning
0	switch debug messages off.
+ve	the higher the value, the more detailed and the more extensive, are the messages.
-ve	only level 0 messages and messages for the specified level are generated.

Remarks The default value of the debug level is 0 (i.e. no debugging messages are generated).

If the debug level is set to a value higher than 0 (values larger than 9 generally produce no additional messages), debugging messages are written to the log file specified by calling ZbMocSetDebugFile. The log file is an ASCII text format file.

ZbMocSetDebugLevel will write messages to the log file. Hence, if debug flags are to be set (i.e. by ZbMocSetDebugFlags), then they should be set PRIOR to setting the debug level.

The execution time of a program and library functions will be increased by setting the debug level to a non-0 value. This can be significant for larger debug level values.

12.9.1.4.90 ZbMocSetDebugMaxLen

```
void ZbMocSetDebugMaxLen(SIGNED32 maxlen);
```

Summary Set the maximum length debug log files.

Parameter maxlen = Maximum length in MB. 0 for none.

Remarks If the log file exceeds this length, then it is automatically closed and a new log file is opened. If this is set to <= 0, then the log file will have no maximum length. The maximum length cannot be set larger than 2047MB. Larger numbers will be treated as 2047MB.

12.9.1.4.91 ZbMocSetDebugRetain

```
void ZbMocSetDebugRetain(SIGNED32 days);
```

Summary Set the number of days to retain debug log files.

Parameter days = Number of days to save files. -1 for none.

Remarks If this is set to -1, then the log file will have the name specified by the ZbMocSetDebugFile routine. If set to 0 or more, then the log file names will have the data and a sequence number appended to them. Files that match this naming format and that are older than the specified number of days, will be automatically deleted when ZbMocSetDebugFile is called. Note that a value of 0 means "Keep files from today".

12.9.1.4.92 ZbMocSourceDeleteFromController

```
SIGNED16 ZbMocSourceDeleteFromController(UNSIGNED16 h, UNSIGNED16 id,
                                         UNSIGNED16 prognum);
```

Summary Delete the source code (if any) of the given program from the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.

prognum = Program number of source to be deleted.

Return Value ZbMoc Error Codes

Remarks

Portability Firmware ≥ 5.17

12.9.1.4.93 ZbMocSourceRestoreFromController

```
SIGNED16 ZbMocSourceRestoreFromController(UNSIGNED16 h, UNSIGNED16 id,
                                          UNSIGNED16 prognum,
                                          const STRUTF8 *filename,
                                          ZbMocSourceInfoS *sourceinfo);
```

Summary Read the source code of the given program from the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
prognum = Program number of source to be read.
filename = Pointer to a string holding the filename and path of the file into which the restored source code is to be written.
sourceinfo = Pointer to a structure taking information about the source code (i.e. original file name, date, and time of the save).

Return Value ZbMoc Error Codes

Remarks If the given file already exists, then it is overwritten! The filename may be either UTF-8 or ANSI encoded.

Source code must previously be saved in the controller using the function ZbMocSourceSaveInController before it can be restored. It is not possible to restore source code from a binary program.

There is no guarantee that the source code stored in the controller corresponds to the binary program with the same program number. Binary programs are saved using ZbMocProgramSave or ZbMocProgramSaveAs and source is save using ZbMocSourceSaveInController. There is no internal link in between the binary code and the source code. It is the responsibility of the application program to make sure that both versions are consistent.

The source data is stored in a compressed data format in the controller If the decompression fails, then a ZBMOC_E_COMPRESSION_ERROR error code is returned.

Portability Firmware ≥ 5.17

12.9.1.4.94 ZbMocSourceSaveInController

```
SIGNED16 ZbMocSourceSaveInController(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 prognum,
                                     const STRUTF8 *filename);
```

Summary Write the source code in the given file into the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
prognum = Program number used to save source. This should normally be the program number of the binary program corresponding to the source being saved.
filename = Pointer to a string holding the filename and path of the source code file to be stored in the controller.

Return Value ZbMoc Error Codes

Remarks It is not possible to save empty files in the controller (the ZBMOC_E_FILECONTENT error code will be returned).

The file content is saved as compressed data in the controller. If the compression fails, a ZBMOC_E_COMPRESSION_ERROR error code is returned.

There is no internal link between the binary program code and the corresponding source code. The application is responsible for managing the correspondence between binary programs and source code. It is strongly recommended that this be done by using the same by program number for both the binary program and the corresponding source code.

The filename may be either UTF-8 or ANSI encoded.

Portability Firmware ≥ 5.17

12.9.1.4.95 ZbMocUserParamReadRaw

```
SIGNED16 ZbMocUserParamReadRaw(UNSIGNED16 h, UNSIGNED16 id, SIGNED32 *param,
                               UNSIGNED16 first, UNSIGNED16 last);
```

Summary Read the specified number of user parameters from the selected controller. The parameters are copied into the given array without any modifications (i.e. no conversion or decoding takes place).

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
param = Pointer to an array taking the parameter values.
first = Number of the first parameter to read: 0..n-1
last = Number of the last parameter to read: 0..n-1

Return Value ZbMoc Error Codes

Remarks The calling application must ensure that the requested parameters exist for the connected controller. The ZbMocVersionCounts routine can be used to determine this.

The array size must be big enough to hold all requested parameters. The array size is NOT checked by this function!

Portability Firmware ≥ 2.7

12.9.1.4.96 ZbMocUserParamWriteRaw

```
SIGNED16 ZbMocUserParamWriteRaw(UNSIGNED16 h, UNSIGNED16 id, SIGNED32 *param,
```

```
                                UNSIGNED16 first, UNSIGNED16 last);
```

Summary Write the specified number of user parameters to the selected controller. The array values are copied into the user parameters of the controller without any modifications (i.e. no conversion or decoding takes place).

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
param = Pointer to an array holding the parameter values to write.
first = Number of the first parameter to write: 0..n-1
last = Number of the last parameter to write: 0..n-1

Return Value ZbMoc Error Codes

Remarks The calling application must ensure that the requested parameters exist for the connected controller. The ZbMocVersionCounts routine can be used to determine this.

The array size must be big enough to hold all specified parameters. The array size is NOT checked by this function!

Portability Firmware ≥ 2.7

12.9.1.4.97 ZbMocV24ReadRawChar

```
SIGNED16 ZbMocV24ReadRawChar(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 *rxd,
                             UNSIGNED32 timeout);
```

Summary Read a single character received via the serial interface.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
rxd = Pointer to a variable taking the received character.
timeout = Maximum period of time [ms] for receiving a character.
0 = Do not wait (i.e. a character must present immediately).
If there is no character received within the defined time slot, the function returns an error code.

Return Value ZbMoc Error Codes

Remarks This function can used by applications that receive byte or character based data from the controller via the V24 interface.

This function is only available if serial (V24) communication is used.

Portability Firmware ≥ 2.7

12.9.1.4.98 ZbMocV24ReadRawString

```
SIGNED16 ZbMocV24ReadRawString(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 *rxbuf,
                               UNSIGNED16 rxbuflen, UNSIGNED32 timeout);
```

Summary Read a string received via the serial interface.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
rxbuf = Pointer to a data section taking the received string.
rxbuflen = Length of the data buffer.
timeout = Maximum period of time [ms] for receiving a string.
- 0 = Do not wait (i.e. a character must be present immediately).
- If there is no character received within the defined time slot,
- the function returns an error code.

Return Value ZbMoc Error Codes

Remarks This function can used by applications that receive string data from the controller via the V24 interface. The string will be null terminated.

The string can be sent by the controller using the "print" command. The string is terminated on the serial interface line with carriage return and line feed characters.

This function is only available if serial (V24) communication is used.

Portability Firmware ≥ 2.7

12.9.1.4.99 ZbMocV24WriteRaw

```
SIGNED16 ZbMocV24WriteRaw(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED8 *buf,
                          UNSIGNED16 buflen);
```

Summary Write a string to the serial interface buffer for transmission.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
buf = Pointer to a data section holding the characters to transmit.
buflen = Number of characters to transmit.

Return Value ZbMoc Error Codes

Remarks This function can be used by applications that want to transmit data, single characters, or strings to the controller (and/or application program) via the V24 interface.

Exactly "buflen" bytes are transmitted. Null characters are processed like any other characters.

This function is only available if serial (V24) communication is used.

Portability Firmware ≥ 2.7

12.9.1.4.100 ZbMocVersionCounts

```
void ZbMocVersionCounts(UNSIGNED32 version, UNSIGNED16 *glbcnt, UNSIGNED16 *usrcnt,
                        UNSIGNED16 *axscnt, UNSIGNED32 *arrmax);
```

Summary Return version-specific controller counts.

Parameter version = Full version of controller (from "ZbMocMoconInfoS" structure).

glbcnt = Number of global parameters. This value is returned.
usrcnt = Number of user parameters. This value is returned.
axscnt = Number of axis parameters. This value is returned.
arrmax = Maximum length of DIM arrays. This value is returned.

Remarks

12.9.1.4.101 ZbMocWriteUserArray

```
SIGNED16 ZbMocWriteUserArray(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 *arrnr,
                             UNSIGNED32 *arrsize, UNSIGNED32 *rsize,
                             SIGNED32 *inarray);
```

Summary Write a user-defined array to the selected controller.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
arrnr = Pointer to a variable holding the number of the array (0...n).
arrsize = Pointer to a variable holding the number of array values to write.
rsize = Pointer to a variable taking the actual size of the array stored in the controller (if already present).
inarray = Pointer to an array holding the values to write.

Return Value ZbMoc Error Codes

Remarks Arrays are numbered consecutively, starting with 0.

It is possible to both write new arrays or overwrite existing arrays. If a new array is written, then its number must be one more than the array number of the last array already defined in the controller (i.e. arrays number must be consecutive). If enough free memory is available in the controller, then the new array will be allocated. If an existing array is overwritten, then size of the array being written must not be larger than the size of the array already existing on the controller (i.e. arrays cannot be "grown"). The size of the array which already exists on the controller is returned in "rsize".

Portability Firmware ≥ 2.7

12.9.1.4.102 ZbMocWriteUserVar

```
SIGNED16 ZbMocWriteUserVar(UNSIGNED16 h, UNSIGNED16 id, UNSIGNED16 varnr,
                           SIGNED32 val);
```

Summary Transmit a data message to the selected controller. This message can be received by an application program using the INMSG command. The message always contains one 16-bit data value and one 32 bit data value.

Parameter h = Handle of the interface.
id = Identification number (ID) of the controller.
varnr = 16 bit value to transmit.
val = 32 bit value to transmit.

Return Value ZbMoc Error Codes

Remarks This function can used by applications that receive byte or character based data from the controller via the V24 interface.

This function is only available if serial (V24) communication is used.

Portability Firmware ≥ 2.7

12.9.2 Introduction to ZbComp

The ZbComp Library allows Window developers to create their own applications that make direct calls to the ApossIDE compiler to compile application programs. These programs may then be downloaded to the controller using either the ZbMocProgramLoad or ZbMocProgramLoadBinary routine in the ZbMoc Library.

The ZbComp Library is a Dynamic Link Library (DLL) available for 32-bit Windows (WIN32). Application development depends on the development environment used by your application program. However, if Microsoft Visual Studio is used, then it is sufficient to include the header "ZbComp.h" and link the base library (ZbComp.lib) in order to use all functions of the library. The corresponding 'ZbComp.dll' must be placed somewhere in the search path which is used by the program.

If the ApossIDE software has been installed, then the ZbComp Library files will be installed in the "Development\ZbComp" subdirectory under the ApossIDE install directory. The "ZbComp.dll" file itself will be installed directly in the ApossIDE install directory, along with the other ApossIDE DLL files.

This is how the ApossIDE compiler is typically used:

Step	Routine	Description
1	ZbCompOptions	Before calling any other routines, call **ZbCompOptions** to set the compiler options.
2	ZbCompFromFile	Call **ZbCompFromFile** to compile the Aposs source code file.
3	ZbCompApossCFromFile	Call **ZbCompApossCFromFile** to compile the ApossC source code file.
4	ZbCompGetErrorCount ZbCompGetError	Check if there has been an error using the function **ZbCompGetErrorCount**. This will return 0 if no error has occured during compilation. If there have been errors, you can retrieve information about the errors using the function **ZbCompGetError**.
5	ZbCompGetMinFirmware	Call **ZbCompGetMinFirmware** to get the minimum firmware version required to execute the code produced by the compiler. This should be checked against the controller's firmware version before attempting to download the compiled code to the controller.
6	ZbCompGetCodesize	Call **ZbCompGetCodesize** to determine how many bytes of executable code have been produced by the compiler.

7	ZbCompReadcode	Call **ZbCompReadcode** to retrieve each individual byte of the executable code.
8	ZbCompFreecode	After retrieving the executable code, call **ZbCompFreecode** to free the compiler's internal memory buffer used to store the executable code.

12.9.2.1 Function Descriptions

12.9.2.1.1 ZbCompFreecode

```
SIGNED16 ZbCompFreecode();
```

Summary Free the data buffers that have been allocated internally during a compilation.

Return Value 0 - Success

Remarks This routine may be called after the generated executable code is no longer needed.

12.9.2.1.2 ZbCompFromFile

```
SIGNED16 ZbCompFromFile(const char *fname, const char *logicalfname,
                        const char *incpath, SIGNED16 mode, UNSIGNED32 flags,
                        const char *reffname);
```

Summary Execute the compiler using the Aposs source code provided in the given file.

Parameter fname - Name of an ASCII file which contains the Aposs application source code.
logicalfname - File name which is inserted into the error messages. This is truncated to CMP_LOGICALNAMELEN characters, if it is too long.
incpath - Directories (separated by ';') to be searched for include files. This can be NULL.
mode - Type of compilation:
CMP_NORMAL - Normal code generation.
CMP_DEBUG - Debug code generation, including debug information.
CMP_DBGINFO - Normal code generation, but including debug information.
flags - Compiler flags (see "ZbComp.h").
reffname - Cross reference filename ("" if none).

Return Value 0 - Compile completed. Use ZbCompGetErrorCount to check for compile errors.
1 - Compiler errors. Use ZbCompGetError to retrieve compile errors.
2 - Could not open file being compiled.
4 - Memory allocation error in compiler.
6 - Internal compiler error.

Remarks ZbCompOptions must be called before this routine can be called. After compilation is complete, call ZbCompGetErrorCount to check if any errors occurred during compilation.

12.9.2.1.3 ZbCompApossCFromFile

```
SIGNED16 ZbCompApossCFromFile(const char *fname, const char *logicalfname,
                              const char *incpath, SIGNED16 mode, UNSIGNED32 flags,
                              const char *reffname);
```

Summary Execute the compiler using the ApossC source code provided in the given file.

Parameter fname - Name of an ASCII file which contains the ApossC application source code.
logicalfname - File name which is inserted into the error messages. This is truncated to CMP_LOGICALNAMELEN characters, if it is too long.
incpath - Directories (separated by ';') to be searched for include files. This can be NULL.
mode - Type of compilation:
CMP_NORMAL - Normal code generation.
CMP_DEBUG - Debug code generation, including debug information.
CMP_DBGINFO - Normal code generation, but including debug information.
flags - Compiler flags (see "ZbComp.h").
reffname - Cross reference filename ("" if none).

Return Value 0 - Compile completed. Use ZbCompGetErrorCount to check for compile errors.
1 - Compiler errors. Use ZbCompGetError to retrieve compile errors.
2 - Could not open file being compiled.
4 - Memory allocation error in compiler.
6 - Internal compiler error.

Remarks ZbCompOptions must be called before this routine can be called. After compilation is complete, call ZbCompGetErrorCount to check if any errors occurred during compilation.

12.9.2.1.4 ZbCompGetCodesize

```
UNSIGNED16 ZbCompGetCodesize(void);
```

Summary Return the size in number of bytes of the executable code produced by the compiler.

Return Value Number of executable code bytes

Remarks The returned number of bytes can be used to read the compiled executable code by calling ZbCompReadcode.

12.9.2.1.5 ZbCompGetError

```
typedef struct ZbCompErrorTag
{
    int  Number;
    char Text[256];
    int  Line;
    int  Column;
    char Filename[CMP_LOGICALNAMELEN];
```

```
} ZbCompError;
```

```
ZbCompError* ZbCompGetError(int n);
```

Summary Return a pointer to a structure containing the error information corresponding to the given error number from the last compilation.

Parameter n = Number of the error (0 ... n-1).

Return Value Pointer to a structure containing error information. NULL will be returned if the error number does not correspond one of the errors generated during the last compilation. This pointer is TEMPORARY and must not be saved for later use.

Remarks The error message text will always have the format "#nnn Message" where "nnn" is the 3-digit error number and "Message" is the actual message text. The number of errors can be retrieved by calling ZbCompGetErrorCount.

12.9.2.1.6 ZbCompGetErrorCount

```
int ZbCompGetErrorCount(void);
```

Summary Return the number of errors which occured during the compilation.

Return Value Number of errors found during the last compilation.

Remarks Call this routine after each compilation to retrieve the number of errors. Detailed information on each error can be retrieved by calling ZbCompGetError.

12.9.2.1.7 ZbCompGetMinFirmware

```
UNSIGNED16 ZbCompGetMinFirmware(void);
```

Summary Return the minimum controller firmware version required to execute the compiled code.

Return Value Minimum controller firmware version (e.g. 70129 for version 7.01.29).

Remarks ZbCompFromFile must be called before this routine will return a valid version number. As well, the compile must complete without errors (warnings are acceptable). Call ZbCompGetErrorCount to check the error count.

12.9.2.1.8 ZbCompGetVersion

```
const char *ZbCompGetVersion(void);
```

Summary Return a pointer to the compiler's version string.

Return Value Pointer to a \0-terminated character string containing the version information.

Remarks A typical version string is similar to this:

Summary Return a pointer to the compiler's version string.
ApossIDE Compiler 6.13.69 (2019/02/19)

12.9.2.1.9 ZbCompOptions

```
typedef struct
{
    SIGNED32   length;          // Length of structure (in bytes).
    UNSIGNED32 fversion;        // Firmware version.
    SIGNED32   maxvars;         // Maximum number of variables available.
    SIGNED32   stksize;         // Stack size.
    SIGNED16   maxaxes;         // Maximum number of axes available.
    SIGNED32   controller;      // Controller type.
    SIGNED32   hwopt;           // Hardware options.
    SIGNED32   language;        // Language: 0 - English, 1 - Deutsch.
    SIGNED32   maxerrs;         // Maximum number of compiler errors.
    SIGNED32   maxwarn;         // Maximum number of compiler warnings.
} ZbCompParams;
```

```
SIGNED16 ZbCompOptions(ZbCompParams *params);
```

Summary Define all necessary parameters for the next compilation.

Parameter params = Pointer to parameter structure. See "ZbComp.h" for a more detailed description of the compiler options.

Return Value 0 - Success
255 - Error (e.g. invalid number of variables, axes, etc.).

12.9.2.1.10 ZbCompReadcode

```
UNSIGNED8 ZbCompReadcode(UNSIGNED32 indx);
```

Summary Return the byte of executable code corresponding to the given index.

Parameter indx = Index of the executable byte (0...n-1).

Return Value The byte of executable code corresponding to the index.

Remarks ZbCompFromFile must be called before this routine will return valid data. As well, the compile must complete without errors (warnings are acceptable). Call ZbCompGetErrorCount to check the error count.

This routine must be called "n" times with index values from 0 to n-1, where "n" is the total number of bytes. This can be retrieved by calling ZbCompGetCodesize. If the index is out of range, then 127 is returned. This corresponds to the end-of-file command in executable code.

After all bytes have been read and there is no longer any need for the generated code, then ZbCompFreecode may be called to free the internal buffers allocated during the compilation.

Summary Return the byte of executable code corresponding to the given index.

The executable code may be downloaded to the controller using either the ZbMocProgramLoad or ZbMocProgramLoadBinary routine in the ZbMoc Library.

> **ZbCompReadcode** is designed to be passed directly to ZbMocProgramLoad as an argument.

13 Glossary

A

ANSI

American National Standards Institute

The **ANSI** character set includes the standard ASCII character set (values 0 to 127), plus an extended character set (values 128 to 255). The **ANSI** character set is used by Windows and refers to "Codepage 1252".

ApossIDE
APOSS Integrated Development Environment

C

CAN-GIN interface
CAN-GIN supported a special card that plugged into the PCI-bus inside the computer and had outputs for connecting to a CAN network. It has been replaced by CAN-LPT and CAN-USB.

CobId
A unique COB-ID (Communication Object Identifier) exists for each communication object in the network.

COE
CAN application protocol over EtherCAT

csv
Comma separated values

F

FPGA
Field programmable gate array

I

Incr.
Encoder increments

L

LSB
Least significant bit

M

MLONG
Maximum encoder position value. The exact value is 1,073,741,823.

MPCMD
sync command position

mqc

Master Quadcount

MSB
Most significant bit

Mt
Master cycle length

N

NMT0
Network CANopen command

P

PDO

Process Data Object

In communication technology, this is a term for process data telegrams containing one or more data values of high priority and short processing time (e.g. speed setpoints, control words, etc.).

PPR
Pulses per revolution

Q

qc
Quadcount

R

R/W
Read/Write

RPM
Revolutions per minute

S

Schleppabstand
Maximum tolerated position error

SDO
Service Data Object

sqc
Slave quadcount

U

uu
User units (scaled position information)

V

VCMD
Command velocity

[

[BE]

[ms]
Millisecond

[mu]
Master units

[sec]
Second

[st]
Sample time, typically 1 ms

[us]
Microsecond

14 Index

Starting the CAM Editor , 195-196
"Record" Array Structure, 812-813
#config, 181-183
#define, #undef, 178-180
#ifdef, #ifndef, #if, 180-181
#include, 178
#pragma, 724-726
abs, 631-633
Acc, 649
Access with FTP, 796
acos, 631-633
Active Column, 263-264
Adding Curves from other Oscilloscope Files, 297-298
Adding Oscilloscope Curves, 282-283
Additional Coordinate Systems, 577-579
Advanced Simulator Topics, 377-378
AmpErrorClear, 649-650
Amplifier Commands, 642
Amplifier Control Loop Design, 604-606
AnalogInput, 650
AnalogOutput, 650
Analyzing the Results, 301-302
ANSI Arrays, 154
ansicpy, 633-634
ansilen, 634
Answer File Structure, 445-446 , 454-456
Apos, 650-652
AposDiff, 652
ApossC Programming Language , 136-137
ApossIDE Help, -1-0
ApossIDE Programming, 136
ApossIDE Tools, 193-194
ApossIDE User Interface, 66
ApossIDE Window, 66-67
APP_SW_VERSION, 112-117

Application Software Version Support, 112-117
Arc Length CAM Curves, 584-587
Array, 382-383
Array Editor, 238
Array Editor Functions, 242-244
Array Editor List, 240-242
Array SDO Download, 795-796
arraycat, 634-636
arraycpy, 636-637
arraylen, 637-638
arrayset, 638
asin, 631-633
atan, 631-633
Autostart, 100-102
Avel, 652
Axis, 394-395
Axis and Information Commands, 640-642
Axis Data Tab, 210-212
Axis Parameters, 781-785
AxisControl, 652-655
AxisCvelStart, 655
AxisCvelStop, 655-656
AxisHomeStart, 656
AxisHomeStatus, 656-657
AxisHomeStop, 657
AxisLinAbsStart, 657-658
AxisLinRelStart, 658
AxisMoving, 658-659
AxisPosAbsStart, 659
AxisPosAbsStartTvel, 659-660
AxisPosRelStart, 660-661
AxisStatus, 661-662
AxisStatusClear, 662
AxisStop, 662-663
AxisTargetReached, 663
AxisWaitReached, 663

Background Fading, 408-409

Background Images, 231-232

Basic Coordinate Systems, 563-566

Batch Mode, 438-439

Batch Mode Examples, 446-448

BinFileMap, 814-816

Bit, 383-384

Bitmap, 385

Block Commutation, 595

Bookmarks, 85

Box, 385-386

Brake Chopper, 598-601

Brake Chopper

Brake Chopper, 598-601

Break [Esc] and Continue, 88

Break all, 93

Button, 386-387

CAM Array Diagram, 810-812

CAM Array Structure, 803-810

CAM Curve Behaviour, 490-492

CAM Curve Diagram, 199-202

CAM Curve Scaling and Positioning, 492-496

CAM Curve Tab, 202-208

CAM Curves, 488-490

CAM Editor, 194-195

CAM Editor Toolbar, 196-198

CamArrayIndex, 663-664

CamArrayLen, 664-665

CanDelete, 665

CanIn, 665-666

CanInit, 666-667

CANINPUT, 160-161

CAN-LPT Connections, 39-42

CAN-LPT Parameters, 119-120

CanOpenRestart, 667

CanOut, 667-668

CAN-USB Connections, 42-45
CAN-USB Parameters, 120-122
Cartesian Kinematics, 546-548
Chart Window, 255-258
Checkbox, 390-391
Checking the Encoder, 61-63
Checking the Motor, 63-65
Child, 391-392
Child Dialogs, 428-430
Clearing Pending Events, 334-335
Close Interface / Close All Interfaces, 95
Command File, 792-793
Command File Structure, 440-445 , 451-454
Command Reference, 639-640
Communication CAN Basic Commands, 643
Communication CANopen Commands, 642-643
Communication EtherCAT Master Commands, 643
Communication Serial Commands, 647
Communication Window, 71-74
Compare to Controller, 91-93
Compile to file, 91
Compiler, 112-117
Configuring and Starting Batch Mode, 439-440
Configuring and Starting Log Mode, 450-451
Connecting to the Controller, 37
Constant, 392
Contact and Legal Information, 26-27
Continuation lines, 177-178
Control Statements, 379
Controller, 387-388 , 396-397 , 392-393
Controller and Interface Columns, 265-266
Controller Dependent Feature List, 875-877
Controller ID Numbers, 440
Controller Menu, 95-97
Controller Types, 872-873
cos, 631-633
Cpos, 668

CposDiff, 668-669
Creating an Array Definition File, 249-251
Current Regulator Calculator, 465-467
Current Ripple and Motor Chokes, 606
Cursor A/B Columns, 264-265
Curve Properties: Axis, 271-272
Curve Properties: Display, 266-271
Curve Properties: Source, 272-273
Curve Selection List, 260
CurvePos, 669-670
CurveType, 803-810
Customizing the Application, 462-463
Cvel, 670
Data Recording Commands, 647
DC Motor, 595-596
Debugging Commands, 88-89
Debugging Programs, 186-188
Debugging State Machines, 356-357
Dec, 670-671
DefCanIn, 671
DefCanOut, 671-672
DefCposOrigin, 672
Defining Handlers for Events, 324
Defining Signals and Event Types, 316-318
Defining State Machine Variables, 320-321
Defining the State Hierarchy, 318-319
DefMasterCpos, 672-673
DefMasterOrigin, 673
DefOrigin, 673-674
Delay, 674
Delayus, 674
Delete, 674-675
Delete Programs all, 94
Delta Kinematics, 557-561
Derived Curve, 290-292
Detecting Axis Status Changes, 352-353

Detecting Changes to Parameters, 347-349
Detecting Digital Input Changes, 349-350
Detecting General Operation Events, 338-339
Detecting Motor Positions, 350-352
Development Menu, 87
Diagnose Bericht, 95-97
Diagnostics, 108-109
Diagnostics Report, 95-97
Differences Between APOSS and ApossC , 142-153
Differences Between Standard C and ApossC , 137-141
DIGINPUT, 162-163 , 675
DigInputByte, 675-676
DigInputWait, 676
DigOutput, 676-677
DigOutputByte, 677
DIM Arrays, 154-156
DimArray, 677-678
DimArrayNum, 678
Display Tab, 208-210 , 225-227
Displaying Analog Input, 430-432
Displaying and Modifying Variables, 191-192
Displaying and Setting Regulation Parameters, 432-435
Displaying and Setting User Parameters, 435-438
Displaying Status Bits, 409-410
Displaying the Executing Line, 192
Double Functions, 631-633
Double Funktionen, 631-633
Download, 388
Download all, 94
Download Firmware, 110-111 , 131-133
Download Menu, 131
Download Mode, 463-465
Download Programs, 133-135
Dual SCARA Kinematics, 555-557
Dynamic CAM Curves, 496-498
Dynamic Connecting Curves, 498-500
Dynamic Systems, 591-593

EC Motor, 594
ECatMasterCommand, 678-680
ECatMasterConfig, 680
ECatMasterInfo, 680-683
Edit Menu, 79-80
Editing CAM Curves, 198-199
Editing Data Arrays, 237-238
Editing Path Curves, 217-219
Editor Settings, 126-128
Emergency Reset, 799-800
Encoder Position Values, 471
ERROR, 161-162
Error Codes, 446
Error Handling, 173-176
Error History, 106-108
Error Messages, 786-789
Error Reference, 786
ErrorAxis, 683
ErrorClear, 683-684
ErrorInfo, 684
ErrorNo, 684
Esc key, 74-75
EtherCAT Connections (Beckhoff), 47-48
EtherCAT Connections (zub), 48-50
EtherCAT Parameters (Beckhoff), 122-123
EtherCAT Parameters (zub), 123-125
ETHERNET, 162
Ethernet Commands, 643-645
Ethernet events, 346-347
Ethernet protocol, 800
EthernetClose, 684
EthernetGetConnectionStatus, 684-685
EthernetOpenClient, 685
EthernetOpenServer, 685-686
EthernetReceiveTelegram, 686
EthernetSendTelegram, 686-687

Event Handling (State machines), 316-318 , 324 , 335-336
Example 1: Basic Synchronization, 503-516
Example 1: Using the Simulator, 516-520
Example 2: Large Sensor Distances, 528-529
Example 2: Problematic Situations, 529-531
Example 2: Synchronization with Marker Correction, 520-528
Example 2: Using the Simulator, 528
Example 3: Engaging and Disengaging, 531-540
Example 4: Slave Synchronization with Marker, 540-544
Example 5: CAM Box, 544-545
Example Monitor Files, 404
Examples of Limited-Jerk Movements, 483-488
Execute, 87-88
Executing a Simple Test Program, 55-58
Exit, 687
Exit Program, 79
Expert mode WITH an Array Definition File, 248-249
Expert mode WITHOUT an Array Definition File, 245-246
External Storage Device (microSD Card/ USB Memory Stick), 790
fabs, 631-633
Fade, 381
File Menu, 78
Find and Replace, 80-82
Find in Files, 82-84
Findbar, 71
Fix Points Tab, 212-215 , 227-231
FoE Firmware Update, 800
Folding, 85
Font, 393-394
Fonts and Text Strings, 404-406
Free Run Oscilloscope, 252-253
FTDI Driver, 802-803
Function Groups, 879-884
Function Keys, 74
Function Reference, 631-633
Functions, 631-633
Funktionen, 631-633

General CAN Message Events, 342-343
Get, 388-389
Getting Started and Safety Tips, 28-29
Global Parameters, 780-781
Global / Axis, 102-103
Glossary, 953-955
Go to, 84-85
grad, 631-633
Grid Column, 261-262
Group, 389
Group Curve, 292-294
HallDetectStart, 687-688
HallDetectStatus, 688-690
Handling General Controller Errors, 335-336
H-Bot/CoreXY Kinematics, 548-550
Help Menu, 131
Homing, 610-611
Homing Method -1 (Current Threshold Positive Speed & Index), 621-622
Homing Method 1 (Negative Limit Switch & Index), 611-612
Homing Method 11 (Home Switch Negative Speed & Index), 614-615
Homing Method 17 (Negative Limit Switch), 615-616
Homing Method 18 (Positive Limit Switch), 616-617
Homing Method -2 (Current Threshold Negative Speed & Index), 622-623
Homing Method 2 (Positive Limit Switch & Index), 612-613
Homing Method 23 (Home Switch Positive Speed), 617-618
Homing Method 27 (Home Switch Negative Speed), 618-619
Homing Method -3 (Current Threshold Positive Speed), 623-624
Homing Method 33 (Index Negative Speed), 619-620
Homing Method 34 (Index Positive Speed), 620-621
Homing Method 37 (Actual Position), 621
Homing Method -4 (Current Threshold Negative Speed), 624-625
Homing Method 7 (Home Switch Positive Speed & Index), 613-614
Homing Methods, 611
How the Control Process Works, 473-474
How to Set Up Resolution and Gear Parameters, 468-471
Illustrations, 813

Immediate Curves, 500
Incremental Encoder, 607
Initializing Temporary Parameters, 779-780
Input/Output Commands, 642
Installing the ApossIDE , 33-34
Integer Functions, 631-633
Integer Funktionen, 631-633
Interface Parameters, 117-118
Internal Amplifier, 593-594
Internal Encoder, 606-607
Interrupt Behaviour, 156-158
Interrupt Functions, 642
Interrupt Handling, 156
Interrupt Routines, 158-160
InterruptDelete, 690
InterruptDisable, 690-691
InterruptEnable, 691-692
InterruptSetup, 692-693
Introduction to the ApossIDE, 26
Introduction to ZbComp, 947-948
Introduction to ZbMoc, 877-879
Ipos, 693-694
IposDiff, 694
JerkFinVel, 694
JerkStopDist, 694-695
Keyboard functions, 76-78
Kinematics Commands, 645-646
Kinematics synchronization, 588-589
KinematicsCpoint, 695
KinematicsSetup, 698-699
KinematicsWorkToMc, 695-697
Language, 112
Last, Minimum, and Maximum Columns, 264
Layout Element Statements, 381-382
Limited Jerk, 480-483
LinkPdo, 699-701
LinkSdo, 701-703

List, 394
Listboxes and SDO Index Offsets, 419-420
ln, 631-633
Load, 397
log, 631-633
Log File Structure, 456-458
Log Mode, 448-450
Logging Service, 790-791
Lookup, 395 , 397-398
MACS5, 873
MACS5-Compact, 873-874
Mapos, 703
MaposDiff, 703
Marker Correction, 816
Master Units [MU], 468
MasterMACS, 874-875
Mavel, 703-704
max, 631-633
Memory Commands, 646-647
Memory Dump Process, 791
MemoryDump, 704-706
MemoryDumpStatus, 706
Message, 395-396
Messages -> Log file, 88
MicroMACS6, 877
microSD Card Files, 461-462
min, 631-633
MiniMACS, 874
MiniMACS6-AMP-4, 875
Mipos, 706-707
MiposDiff, 707-708
Modifying a Curve's Properties, 266
Modifying the CAM Curve Array, 502-503
Module Editor, 305-306
Module Editor Settings, 313
Module Editor Toolbar, 312-313

Module Graphics View, 308-309
Module Linking, 313-315
Module Tree View, 306-308
Module Window, 309-312
Modules Description, 626-630
Monitor, 378-379
MotorAlignStart, 708-709
MotorAlignStatus, 709
Mouse functions, 75-76
Mouse Position Column, 264
Multi-axis Kinematics, 545-546
Multiple Controllers, Polling Load, and Dynamically Selecting Controllers, 424-426
Multiple Paths, 235-236
Name Column, 260-261
Navigation Window, 258-260
Network Connections, 50-53
Network Parameters, 125
Neustart, 95-97
New, 78
Operators, 137-141
OpFlags, 380-381
Options, 128-130
Oscilloscope, 251-252 , 458
Oscilloscope Settings, 302-305
Oscilloscope Versions, 252
Oscilloscope Window, 254-255
Other Functionalities (microSD Card/ USB Memory Stick), 796
Overcurrent Protection (OVC), 601-604
Overlay Files, 232-235
PARAM, 163-165
Parameter Reference, 778-779
Parameters, 102
Paste Array Assignment, 80
Path Curve Diagram, 219-222
Path Curve Sample, 360-371
Path Curve Tab, 222-225
Path Curves, 575-577

Path Fixed Points, 579-581
PathAcc, 709-710
PathArrayIndex, 710-711
PathContinue, 711-712
PathDec, 712-713
PathJerk, 713-714
PathMcToUu, 714
PathMove, 714-716
PathNext, 716-717
Paths and State Machines, 593
PathStart, 717-720
PathStop, 720-721
PathUuToMc, 721-722
PathVel, 722
PC Output Files, 462
PDO, 389-390 , 399 , 723-724
PDO Curve, 286-288
PDO Message Events, 341-342
PERIOD, 165-166
PID Factors, 475-477
Polling SDO Values on Demand, 410-412
PollInterval, 381
Popup Menus, 76
POSA CURVEPOS, 669-670
Position, 166-169
Position Control Loop Design, 474-475
Posting Events to State Machines, 331-332
pow, 631-633
Powerdown Save Functions, 797-798
Predefined Handlers, 324-325
Preparing the Controller, 459
Preprocessor, 176-177
Print, 398 , 726-727
PRINT AT, 727
PRINT CON, 727
PRINT DEV, 727

PRINTAT, 727
PRINTCON, 727
printcontrol, 727
PRINTDEV, 727
printf, 728-729
Profile Generator Events, 343-346
Program Access to Paths, 590-591
Program Configuration of History, 358-359
Program Samples, 183-186
Programs, 97-100
Quick Start Tutorial for Users, 503
rad, 631-633
RADIXSTR, 726-727
Read/Write SDO, 109-110
Reading .csv Files, 279-281
Reading and Writing SDO Values, 414-416
Reading Old Testrun Files, 281-282
Reading Process (microSD Card/ USB Memory Stick), 791-792
Real Time Clock, 799
Reboot, 95-97
Receiving PDO Messages, 412-414
Record Curve, 294-295
Record Oscilloscope, 253-254
RecordDest, 729-730
Recorder, 458
Recorder Limitations, 463
Recorder Settings, 460-461
RecordIndex, 730-731
RecordStart, 731-732
RecordStop, 732-733
RecordTime, 733-734
RecordTimeBase, 734-735
RecordType, 735-736
Regular Expressions, 82-84
Repeating the Test, 302
Report File, 793-794
Required Program Adaptations, 374-375

Reset, 106
ResetOrigin, 736
Restore from file, 104-106
rnd, 631-633
round, 631-633
RS485/RS232 Events, 339-340
Sample, 78-79
Save, 736
Save ALL to files, 106
Save to file, 103-104
Saving and Opening Oscilloscope Files, 279
Scaling, 468
SCARA Kinematics, 550-555
SD Card, 796-797
SDO, 399-400
SDO Curve, 285-286
SDO Macro Selector, 85-86
SdoRead, 736-737
SdoReadSeg, 737-738
SdoReadSegp, 738-740
SdoStatus, 740-741
SdoWrite, 741-742
SdoWriten, 742-743
SdoWriteSeg, 743
Select Controller, 94-95
Serial Interface Events, 340-341
Serial PDO, 868-869
SERIALCHAR, 169
SerialReadChar, 744-745
SerialReadComm, 745
SerialSetup, 744
SerialWriteComm, 745-746
Set, 390
SetMasterOrigin, 746
SetOrigin, 746-747
SETOUT, 169-171

Setting Basic Parameters, 58-61
Setting Bits, 418-419
Setting Controller Behavior, 477-478
Setting Synchronization Parameters, 471-472
Setting Up the ApossIDE Recorder, 458-459
Settings Menu, 112
Showing and Hiding Dialog Fields, 420-424
Sidebar, 69-71
SIG_ENTRY, 316-318
SIG_ERROR, 316-318 , 335-336
SIG_EXIT, 316-318
SIG_IDLE, 316-318
SIG_INIT, 316-318
SIG_KEYPRESSED, 316-318 , 338-339
SIG_START, 316-318
Signal Generator, 607
Simple Polling, 406-408
Simulator, 371-374
Simulator Connections, 53-55
Simulator Files, 374
Simulator Parameters, 125-126
Simulator User Interface, 375-377
sin, 631-633
Sin Cos Encoder, 607
Single Shot Oscilloscope, 253
Single-Stepping, 189-190
Sinusoidal Commutation, 594-595
Slider, 400-401
SmClear, 334-335 , 747
SmConfig, 321-324 , 747
SmEvent, 316-318 , 747
SmHandled, 324-325
SmInput, 349-350 , 747
SmMachine, 320-321 , 747-748
SmNotHandled, 324-325
SmParam, 347-349 , 748
SmPeriod, 336-338 , 748

SmPosition, 350-352 , 748
SmPositionOnce, 350-352
SmPost, 331-332 , 748
SmPublish, 332-334 , 748
SmRun, 330-331 , 748-749
SmState, 318-319 , 749
SmStop, 330-331 , 749
SmSubscribe, 332-334 , 749
SmSystem, 749
SmTimer, 336-338 , 749
SmTrans, 325-328
SmUnsubscribe, 332-334 , 749
SmUrgent, 331-332 , 749
Special Arrays, 153-154
Special Cases, 501-502
Specified Controllers and Interface Types, 426-428
Specifying the Event Types to be Handled by a State, 319-320
Specifying the State Machine Configuration, 321-324
Sprache, 112
sprint, 749-750
sprintf , 750
sqrt, 631-633
SSI Encoder, 607-610
Start all, 93-94
Start and End Directions and Velocities, 581-582
Start Stop Points Tab, 215-217
Starting and Ending the Test, 298-300
Starting and Exiting the ApossIDE, 35-37
Starting and Stopping State Machines, 330-331
Starting the Array Editor, 238-240
Starting the Debugger, 188-189
Starting the Oscilloscope, 273-276
Starting the Recorder on the PC, 459-460
STATBIT, 171-172
State Functions, 328-330
State Machine, 111

State Machine Commands, 647-648
State Machine Errors, 353-356
State Machine Execution, 331
State Machine Timers, 336-338
State Machines, 315-316
State Transitions, 325-328
Status Column, 262-263
Stepper Motor, 596
Stepper Motor - Closed Loop, 597-598
Stepper Motor - Open Loop, 596-597
Stewart Kinematics, 561-563
Stopping the Debugger, 189
Subscribing to Events and Publishing Events, 332-334
SyncCam, 750-752
SyncCamMarkerMaster, 752-753
SyncCamMarkerSlave, 753
SyncCamStart, 753-754
SyncCamStop, 754-755
SyncDefOrigin, 755-758
SyncError, 758
Synchronization Commands, 472-473
SyncMarker, 758-759
SyncMarkerRestart, 759-760
SyncPos, 760-761
SyncSetCurve, 762-763
SyncShiftOrigin, 763
SyncStatus, 763-764
SyncStatusClear, 764
SyncVel, 764-766
Syntax Check, 91
SysDef Defines, 816-868
System Architecture, 29-33
System Commands, 648-649
System Update Without ApossIDE, 794
Sysvar, 766-767
Table, 395 , 401
Tabs, 78

tan, 631-633
Technical Reference, 790
Ten steps for optimum control for MACS controllers, 478-479
Text, 401-402
TFTP, 796-797
The quickest route to optimal control, 478
Thinning Fixed Points, 582-584
TIME, 172-173 , 767
TimeDiffus, 767
TimeHw, 767-768
Toolbar, 67-69
Tools Menu, 111
TrackError, 768-769
TRANSFORM, 566-570
Transformations, 566-570
TransIdent, 769
TransInverse, 769
TransPoint, 769-770
TransReflect, 770
TransRotateXY, 770-771
TransRotateYZ, 771-772
TransRotateZX, 772-773
TransScale, 773
TransStretch, 773-774
TransTranslate, 774-775
Tvel, 775
Twin Mode, 598
Understanding Point Storage Structure, 573-575
Undo Function, 78
Units Column, 265
Updating from Record Files, 279
Usage Notes for Polling Mode, 283-285
Usage Notes for the Record Oscilloscope, 277-279 , 302
Usage Notes for the Single Shot Oscilloscope, 276-277 , 300-301
USB Connections, 45-47
USB Parameters, 122

User Configuration of History, 357-358
User Curves, 295-297
User mode WITH an Array Definition File, 247-248
User mode WITHOUT an Array Definition File, 244-245
User Parameters, 780
User SDO Download , 794-795
UserStatus, 775-776
Using ".zip" Files, 440
Using Breakpoints, 190-191
V24 (Serial) Connections, 37-39
V24 Parameters, 118-119
Value, 398-399 , 402-404
Variable Curve, 288-290
Vel, 776
Version History, 869
Viewing Individual Axes, 236-237
Viewing State Machine History, 359-360
Watch Add / Start / Stop, 89-91
Watchdog, 801-802
WdActivate, 776-777
WdReset, 777
Webserver, 796-797
What to do, if ...?, 479-480
What's New in Version 7.00, 869-870
What's New in Version 8.0.00 , 870-871
What's New in Version 9.0.00 , 871-872
WindowFlags, 380
Windows Menu, 130-131
WindowTitle, 380
WindowWidth and WindowHeight, 379
Wiring the Controller, 34-35
Working with Kinematics, 545
Working with Paths, 589-590
Working with Points, 570-573
Writing Process (microSD Card/ USB Memory Stick), 790
Writing Specific SDO Values and Writing Multiple SDO Values, 416-418
ZbCompApossCFromFile, 948-949

ZbCompFreecode, 948
ZbCompFromFile, 948
ZbCompGetCodesize, 949
ZbCompGetError, 949-950
ZbCompGetErrorCount, 950
ZbCompGetMinFirmware, 950
ZbCompGetVersion, 950-951
ZbCompOptions, 951
ZbCompReadcode, 951-952
ZbMoc Error Codes, 889-898
ZbMocAutoClear, 898
ZbMocAutoSet, 898
ZbMocAxisParamRead(DEPRECATED), 898-899
ZbMocAxisParamS, 887-889
ZbMocAxisParamWrite(DEPRECATED), 899
ZbMocBlobInfoRequest, 899-900
ZbMocBlobRead, 900
ZbMocBlobWrite, 900-901
ZbMocBreak, 901-902
ZbMocBreakAll, 902
ZbMocCanOpenDownloadSDO, 902
ZbMocCanOpenInhibitPDO, 902-903
ZbMocCanOpenPollPDO, 903-904
ZbMocCanOpenQueuePDO, 904
ZbMocCanOpenReadSDO, 904-905
ZbMocCanOpenReadSegmentedSDO, 904
ZbMocCanOpenStart, 905
ZbMocCanOpenUploadSDO, 905-906
ZbMocCanOpenWritePDO, 906
ZbMocCanOpenWriteSDO, 906-907
ZbMocCanOpenWriteSegmentedSDO, 906
ZbMocCanReadRaw, 907
ZbMocCanWriteRaw, 907-908
ZbMocClearError, 908
ZbMocClose, 908
ZbMocCloseAll, 908

ZbMocCNFRestoreFromFile, 908-909
ZbMocCNFSaveToFile, 909
ZbMocConnect, 909-910
ZbMocContinue, 910
ZbMocDebugPrint, 910
ZbMocDebugPrintw, 910
ZbMocDisconnect, 910-911
ZbMocExec, 911
ZbMocExecTemp, 911
ZbMocGetAxisPosition, 911-912
ZbMocGetError, 912
ZbMocGetIFDriverID, 912
ZbMocGetNumber, 912-913
ZbMocGetZbMocVersion, 913
ZbMocMemoryDump, 913
ZbMocMemoryLock, 913-914
ZbMocMoconBusyWait, 914
ZbMocMoconClearFactory, 914
ZbMocMoconClearParameters, 914-915
ZbMocMoconClearPrograms, 915
ZbMocMoconDeleteEeprom, 915
ZbMocMoconExecutionStatus, 915-916
ZbMocMoconFind, 916-917
ZbMocMoconIdent, 917
ZbMocMoconIdentS, 884-885
ZbMocMoconInfo, 917
ZbMocMoconInfoIdx, 917-918
ZbMocMoconInfoS, 885-886
ZbMocMoconNameSet, 918
ZbMocMoconSaveRam, 918
ZbMocMoconSaveRamSection, 918-919
ZbMocMoconSetCanParameters, 919-920
ZbMocMoconState, 920
ZbMocMoconWaitForString, 920
ZbMocOpenCanLpt, 920-922
ZbMocOpenCanUsb, 922-923
ZbMocOpenEtherCat, 923-924

ZbMocOpenSerial, 924-925
ZbMocOpenSim, 929-930
ZbMocOpenTcp, 925-926
ZbMocOpenUsb, 926-927
ZbMocOpenV24, 927
ZbMocOpenVlt, 928-929
ZbMocParamRead, 930
ZbMocParamS, 886-887
ZbMocParamWrite, 930
ZbMocPollMessage, 930-931
ZbMocProbe, 931
ZbMocProgramDelete, 931
ZbMocProgramList, 932
ZbMocProgramLoad, 932-933
ZbMocProgramLoadBinary, 933
ZbMocProgramReadBinary, 934
ZbMocProgramSave, 934
ZbMocProgramSaveAs, 934-935
ZbMocReadMemPage, 935
ZbMocReadUserArray, 935-936
ZbMocReadUserVar, 936
ZbMocSDOGetUsage, 936-937
ZbMocSDOGetUsageDetails, 937-938
ZbMocSetCallbackPDO, 938
ZbMocSetCallbackProgress, 938-939
ZbMocSetDebugFile, 939
ZbMocSetDebugFlags, 939-940
ZbMocSetDebugHistory, 940
ZbMocSetDebugImmediate, 940
ZbMocSetDebugLevel, 940-941
ZbMocSetDebugMaxLen, 941
ZbMocSetDebugRetain, 941
ZbMocSourceDeleteFromController, 941-942
ZbMocSourceRestoreFromController, 942
ZbMocSourceSaveInController, 942-943
ZbMocUserParamReadRaw, 943

ZbMocUserParamWriteRaw, 943-944

ZbMocV24ReadRawChar, 944

ZbMocV24ReadRawString, 944-945

ZbMocV24WriteRaw, 945

ZbMocVersionCounts, 945-946

ZbMocWriteUserArray, 946

ZbMocWriteUserVar, 946-947