# Table of Contents

## Introduction

Manual Contents .............................................................. 1

## Book I - Debugger Engine

Book I Contents ............................................................. 3

1 **Introduction** ........................................................................... 5
   Freescale Debugger .......................................................... 5
   Debugger Application ......................................................... 5
   Debugger Features ............................................................ 6
   Demo Version Limitations on Components .......................... 6

2 **Debugger Interface** ............................................................ 7
   Introduction ........................................................................ 7
   Application Programs ......................................................... 7
   Starting the Debugger ....................................................... 8
   Starting from within the IDE .............................................. 8
   Starting Debug from the Project Window ........................... 8
   Starting Debug from the Main Window Menu Bar .............. 8
   Debugger Command Line Start ........................................... 9
   Command Line Options ..................................................... 9
   Order of Commands ......................................................... 11
   Debugger Main Window ..................................................... 11
   Debugger Main Window Toolbar ....................................... 12
   Debugger Main Window Status Bar .................................... 13
Table of Contents

Main Window Menu Bar ................................................. 13
File Menu ................................................................. 14
Preferences Window .................................................... 15
View Menu ................................................................. 17
Customizing the Toolbar ............................................. 17
Run Menu ................................................................. 20
Connection Menu ....................................................... 22
Loading a Connection .................................................. 23
Connection Command File Window ............................... 25
Component Menu ........................................................ 27
Window Menu ............................................................. 29
Help Menu ................................................................. 31
About Box ................................................................. 31
Component Associated Menus ...................................... 32
Component Main Menu ................................................ 32
Component Files ......................................................... 33
Component Windows Object Info Bar ............................ 33
Component Popup Menu .............................................. 33
Highlights of the User Interface ................................. 34
Activating Services with Drag and Drop. .................... 34
To Drag and Drop an Object ....................................... 36
Drag and Drop Combinations ..................................... 37
Dragging from Assembly Component Window ............... 37
Dragging from Data Component Window ...................... 37
Dragging from Source Component Window ................... 38
Dragging from the Memory Component Window ............ 39
Dragging from Procedure Component Window ............... 40
Dragging from Register Component Window .................. 40
Dragging from Module Component Window ................... 41
Selection Dialog Box ................................................... 41

3 Debugger Components ................................................. 43
Component Introduction .............................................. 43
CPU Components ....................................................... 43
Window Components .................................................. 43
## Table of Contents

- Connection Components ........................................... 44
- Loading Component Windows ........................................ 44
- General Debugger Components ....................................... 46
  - Assembly Component ............................................. 47
    - Assembly Menu .................................................. 47
    - Setting Breakpoints .......................................... 48
    - Associated Popup Menu ....................................... 49
  - Command Line Component ......................................... 52
    - Command Menu .................................................. 54
    - Cache Size ..................................................... 55
  - Coverage Component ............................................... 56
    - Coverage Operations ......................................... 57
    - Coverage Menu ................................................ 57
    - Output File .................................................... 58
    - Split View Associated Popup Menu ........................... 59
  - DA-C Link Component ............................................... 61
    - DA-C Link Operation .......................................... 61
    - DA-C Link Menu ............................................... 61
    - Drag Out ....................................................... 62
    - Drop Into ..................................................... 62
    - Demo Version Limitations ................................... 62
  - Data Component .................................................... 63
    - Data Operations ............................................... 63
    - Expression Editor ............................................. 64
    - Expression Command file ..................................... 65
    - Data Menu ..................................................... 66
    - Scope Submenu ................................................. 66
    - Format Submenu ................................................ 67
    - Format Selected & All Sub Menu ............................... 68
    - Mode Submenu ................................................ 68
    - Options Submenu .............................................. 70
    - Zoom and Sort Submenus ...................................... 72
    - Associated Popup Menu ....................................... 72
  - Memory Component ................................................ 76
    - Memory Operations ............................................ 77
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Menu</td>
<td>77</td>
</tr>
<tr>
<td>Display Submenu</td>
<td>80</td>
</tr>
<tr>
<td>Fill Memory</td>
<td>81</td>
</tr>
<tr>
<td>Display Address</td>
<td>81</td>
</tr>
<tr>
<td>CopyMem Submenu</td>
<td>83</td>
</tr>
<tr>
<td>Search Pattern</td>
<td>83</td>
</tr>
<tr>
<td>Update Rate</td>
<td>84</td>
</tr>
<tr>
<td>Associated Popup Menu</td>
<td>85</td>
</tr>
<tr>
<td>MicroC Component</td>
<td>88</td>
</tr>
<tr>
<td>MicroC Link Menu</td>
<td>88</td>
</tr>
<tr>
<td>MicroC DLLs</td>
<td>89</td>
</tr>
<tr>
<td>Module Component</td>
<td>92</td>
</tr>
<tr>
<td>Module Operations</td>
<td>92</td>
</tr>
<tr>
<td>Module Menu</td>
<td>92</td>
</tr>
<tr>
<td>Procedure Component</td>
<td>94</td>
</tr>
<tr>
<td>Procedure Operations</td>
<td>94</td>
</tr>
<tr>
<td>Procedure Menu</td>
<td>94</td>
</tr>
<tr>
<td>Profiler Component</td>
<td>96</td>
</tr>
<tr>
<td>Profiler Operations</td>
<td>97</td>
</tr>
<tr>
<td>Profiler Output File Functions</td>
<td>98</td>
</tr>
<tr>
<td>Recorder Component</td>
<td>99</td>
</tr>
<tr>
<td>Recorder Operations</td>
<td>100</td>
</tr>
<tr>
<td>Recorder Menu</td>
<td>100</td>
</tr>
<tr>
<td>Register Component</td>
<td>102</td>
</tr>
<tr>
<td>Status Register Bits</td>
<td>102</td>
</tr>
<tr>
<td>Editing Registers</td>
<td>102</td>
</tr>
<tr>
<td>Register Menu (Format Submenu)</td>
<td>103</td>
</tr>
<tr>
<td>Drop Into:</td>
<td>104</td>
</tr>
<tr>
<td>SoftTrace Component</td>
<td>105</td>
</tr>
<tr>
<td>SoftTrace Operations</td>
<td>105</td>
</tr>
<tr>
<td>SoftTrace Menu</td>
<td>105</td>
</tr>
<tr>
<td>Associated Popup Menu</td>
<td>106</td>
</tr>
<tr>
<td>Source Component</td>
<td>108</td>
</tr>
<tr>
<td>Folding and Unfolding</td>
<td>111</td>
</tr>
<tr>
<td>Source Menus</td>
<td>112</td>
</tr>
</tbody>
</table>
## Table of Contents

Associate a Command with a Breakpoint .......................... 156
Demo Version Limitations ........................................... 156

Watchpoints .......................................................... 157
Watchpoints Tab ..................................................... 159
Multiple Selections ................................................... 160
Checking Syntax ....................................................... 160

Setting Watchpoints .................................................. 161
Setting a Read Watchpoint ........................................... 161
Setting a Write Watchpoint ........................................... 162
Defining a Read/Write Watchpoint ................................. 162
Defining a Counting Watchpoint ................................... 163
Defining a Conditional Watchpoint ................................. 164
Deleting a Watchpoint ............................................... 165
Associate a Command with a Watchpoint ......................... 165
Demo Version Limitations ........................................... 166

Markpoints ........................................................... 167
Markpoints Tab ......................................................... 169

Setting Markpoints .................................................... 170
Setting a Source Markpoint ......................................... 170
Setting a Data Markpoint ............................................ 170
Setting a Memory Markpoint ........................................ 171
Deleting a Markpoint ................................................. 171

Halting on a Control Point .......................................... 172

## 5 Real Time Kernel Awareness ................................... 173

Introduction ........................................................ 173
Inspecting Task State ............................................... 174
RTK Interface ......................................................... 174
Task Description Language ......................................... 174
Application Example ................................................. 176
Inspecting Kernel Data Structures ................................. 177
RTK Awareness Register Assignments ............................. 178
OSEK Kernel Awareness ............................................ 179
OSEK ORTI ........................................................... 179
ORTI File and Filename ............................................. 179
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORTI File Structure</td>
<td>180</td>
</tr>
<tr>
<td>OSEK RTK Inspector Component</td>
<td>181</td>
</tr>
<tr>
<td>Inspector Task</td>
<td>183</td>
</tr>
<tr>
<td>Inspector Stack</td>
<td>184</td>
</tr>
<tr>
<td>Inspector SystemTimer</td>
<td>184</td>
</tr>
<tr>
<td>Inspector Alarm</td>
<td>185</td>
</tr>
<tr>
<td>Inspector Message</td>
<td>186</td>
</tr>
</tbody>
</table>

6 How To ... 187

How To Configure the Debugger                      188
For Use from Desktop (Win 95, Win 98, Win NT4.0 or Win2000) 188
Defining the Default Directory in the MCUTOOLS.INI 188
How To Start the Debugger                          189
From WinEdit                                       189
Automating Debugger Startup                        190
How To Load an Application                         191
How To Start an Application                        192
How To Stop an Application                         192
How To Step in the Application                     193
On Source Level                                     193
On the Next Source Instruction                     193
Step Over a Function Call (Flat Step)              194
Step Out from a Function Call                      194
Step on Assembly Level                              195
How To Work on Variables                           195
Display Local Variable from a Function             195
Display Global Variable from a Module              195
Change Format for Variable Value Display           196
Modify a Variable Value                             197
Get the Address Where a Variable is Allocated      198
Inspect Memory Starting at a Variable Location Address 198
Load an Address Register with the Address of a Variable 198
How To Work on the Register                        199
Change Format of Register Display                  199
Modify a Register Content                           199
Table of Contents

- Modify Index or Accumulator Register Content ........................................... 199
- Modify Bit Register Content ................................................................. 200
- Start Memory Dump at Address Where Register Is Pointing ...................... 200
- Modify Content of Memory Address ....................................................... 201
- How to Consult Assembler Instructions Generated by a Source Statement ... 201
- How To View Code ..................................................................................... 202
- How to Communicate with the Application ............................................... 203
- About startup.cmd, reset.cmd, preload.cmd, postload.cmd ...................... 203

7 CodeWarrior Integration ............................................................................. 205
   Debugger Configuration ............................................................................ 205

8 Debugger DDE Capabilities ....................................................................... 207
   Introduction ............................................................................................. 207
   DDE Implementation ................................................................................ 207
   Driving Debugger through DDE .............................................................. 207

9 Synchronized Debugging Through DA-C IDE ......................................... 209
   Configuring DA-C IDE for Freescale Tool Kit ........................................ 209
   Create New Project ................................................................................... 210
   Configure Working Directories ............................................................. 210
      Configure File Types ............................................................................ 212
      Configure Library Path ........................................................................ 212
   Adding Files to Project ........................................................................... 214
   Building The Database ........................................................................... 215
   Configuring The Tools ............................................................................. 217
      Compiler Configuration ........................................................................ 217
      Linker Configuration ............................................................................ 218
      Maker Configuration ............................................................................. 219
   Debugger Interface ................................................................................... 220
      DA-C IDE and Debugger Communication ........................................... 221
      Communication DLL Installation ......................................................... 221
      Debugger Properties Configuration ..................................................... 222
      Debugger Project File Configuration ................................................... 223
   Synchronized Debugging ......................................................................... 225
# Table of Contents

## Book II - HC(S)12(X) Debug Connections

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book II Contents</td>
<td>229</td>
</tr>
<tr>
<td>10 HC(S)12(X) Full Chip Simulation Connection</td>
<td>231</td>
</tr>
<tr>
<td>Full Chip Simulation Menu</td>
<td>231</td>
</tr>
<tr>
<td>Debugger Status Bar with Full Chip Simulation</td>
<td>232</td>
</tr>
<tr>
<td>Open I/O Component Dialog Box</td>
<td>233</td>
</tr>
<tr>
<td>Command File Window</td>
<td>234</td>
</tr>
<tr>
<td>Memory Configuration</td>
<td>235</td>
</tr>
<tr>
<td>Memory Configuration Dialog Box Features</td>
<td>235</td>
</tr>
<tr>
<td>Access Details Dialog Box</td>
<td>240</td>
</tr>
<tr>
<td>Output</td>
<td>241</td>
</tr>
<tr>
<td>Clock Frequency Setup</td>
<td>241</td>
</tr>
<tr>
<td>Bus Trace</td>
<td>242</td>
</tr>
<tr>
<td>Full Chip Simulation Warnings</td>
<td>243</td>
</tr>
<tr>
<td>WARNING_SETUP Command</td>
<td>244</td>
</tr>
<tr>
<td>MESSAGE_HIDE_ID Command</td>
<td>245</td>
</tr>
<tr>
<td>MESSAGE_SHOW_ID Command</td>
<td>246</td>
</tr>
<tr>
<td>MESSAGE_HIDE_RESET Command</td>
<td>246</td>
</tr>
<tr>
<td>FCS and Silicon On-chip Peripherals Simulation</td>
<td>247</td>
</tr>
<tr>
<td>Supported Derivatives</td>
<td>248</td>
</tr>
<tr>
<td>Communication Modules</td>
<td>257</td>
</tr>
<tr>
<td>Byteflight (BF)</td>
<td>257</td>
</tr>
<tr>
<td>J1850 Bus (BLCD)</td>
<td>257</td>
</tr>
<tr>
<td>Motorola Scalable CAN (MSCAN)</td>
<td>257</td>
</tr>
<tr>
<td>Inter-IC Bus (IIC)</td>
<td>258</td>
</tr>
<tr>
<td>Serial Communication Interface</td>
<td>258</td>
</tr>
<tr>
<td>Serial Peripheral Interface</td>
<td>260</td>
</tr>
<tr>
<td>Converter Modules</td>
<td>260</td>
</tr>
<tr>
<td>Analog to Digital Converter</td>
<td>260</td>
</tr>
<tr>
<td>Memory Modules</td>
<td>262</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEPROM (EETS)</td>
<td>262</td>
</tr>
<tr>
<td>Flash (FTS)</td>
<td>262</td>
</tr>
<tr>
<td>Miscellaneous Modules</td>
<td>262</td>
</tr>
<tr>
<td>Voltage Regulator (VREG)</td>
<td>262</td>
</tr>
<tr>
<td>Debug Module (DBG)</td>
<td>263</td>
</tr>
<tr>
<td>S12X_INT</td>
<td>263</td>
</tr>
<tr>
<td>XGATE</td>
<td>263</td>
</tr>
<tr>
<td>Port I/O Modules</td>
<td>264</td>
</tr>
<tr>
<td>External Bus Interface (EBI)</td>
<td>264</td>
</tr>
<tr>
<td>Module Mapping Control (MMC)</td>
<td>265</td>
</tr>
<tr>
<td>Multiplexed External Bus Interface (MEBI)</td>
<td>265</td>
</tr>
<tr>
<td>Port Integration Module (PIM)</td>
<td>266</td>
</tr>
<tr>
<td>Timer Modules</td>
<td>266</td>
</tr>
<tr>
<td>Clock and Reset Generator (CRG)</td>
<td>266</td>
</tr>
<tr>
<td>Blocks:</td>
<td>266</td>
</tr>
<tr>
<td>Enhanced Capture Timer (ECT)</td>
<td>268</td>
</tr>
<tr>
<td>Periodic Interrupt Timer (PIT)</td>
<td>271</td>
</tr>
<tr>
<td>Pulse Width Modulator (PWM)</td>
<td>271</td>
</tr>
<tr>
<td>Timer Module (TIM)</td>
<td>273</td>
</tr>
<tr>
<td>Legacy HC12 (CPU12) Derivatives Simulation</td>
<td>273</td>
</tr>
<tr>
<td>MC68HC812A4</td>
<td>273</td>
</tr>
<tr>
<td>HC912DG128x, HC912DT128x</td>
<td>290</td>
</tr>
<tr>
<td>FCS Visualization Utilities</td>
<td>295</td>
</tr>
<tr>
<td>Analog Meter Component</td>
<td>295</td>
</tr>
<tr>
<td>Analog Meter Operations</td>
<td>296</td>
</tr>
<tr>
<td>Analog Meter Menu</td>
<td>296</td>
</tr>
<tr>
<td>IO_LED Component</td>
<td>297</td>
</tr>
<tr>
<td>IO_LED Operations</td>
<td>297</td>
</tr>
<tr>
<td>IO_LED Menu</td>
<td>297</td>
</tr>
<tr>
<td>Demo Version Limitations</td>
<td>298</td>
</tr>
<tr>
<td>LED Component</td>
<td>299</td>
</tr>
<tr>
<td>LED Operations</td>
<td>299</td>
</tr>
<tr>
<td>LED Menu</td>
<td>299</td>
</tr>
<tr>
<td>Phone Component</td>
<td>301</td>
</tr>
<tr>
<td>Phone Operations</td>
<td>302</td>
</tr>
</tbody>
</table>
# Table of Contents

ADC/DAC Component ........................................... 303  
ADC/DAC Menu .................................................. 305  
ADC/DAC - Setup Dialog Box .................................. 305  
Conversion Parameters Dialog Box .......................... 306  
Display Properties Dialog Box ............................... 307  
IT_Keyboard Component ....................................... 308  
IT_Keyboard Menu ................................................ 309  
Interruption Keyboard Setup .................................. 309  
LCD Component ................................................... 311  
LCD Operation .................................................... 311  
Instruction Listing ............................................. 312  
The Initialization Step ......................................... 314  
LCD Menus ....................................................... 315  
LCD Display ..................................................... 316  
Monitor Component ............................................. 317  
Monitor Menu ..................................................... 317  
Add Channel ...................................................... 318  
Monitor Settings ............................................... 318  
Change Colors ................................................... 319  
Push Buttons Component ...................................... 321  
Push Buttons Menu .............................................. 321  
Push Buttons Setup ............................................. 321  
Use with IO_Ports ............................................... 322  
Use with LEDs Component .................................... 322  
Programmable IO_Ports Component .......................... 323  
Programmable IO_Ports Menu ................................. 323  
Port Address ..................................................... 324  
7-Segments Display Component .............................. 325  
7-Segments Display Menu ...................................... 326  
7-Segments Display Setup ..................................... 326  
Stimulation Component ........................................ 328  
Stimulation Popup Menu ....................................... 328  
Example of a Stimulation File ............................... 329  
TestTerm Component .......................................... 331  
Output Redirection ............................................. 332
# Table of Contents

- How to Redirect .................................................. 333
- Using TestTerm ...................................................... 333
- TestTerm Menu ...................................................... 334
- Terminal Component ............................................... 336
  - Configure Terminal Connections ...................... 337
  - Input and Output File .................................... 338
  - File Control Commands .................................. 338
  - How to Use Virtual SCI .................................. 340
- Wagon Component .................................................. 341
  - Wagon Menu .................................................... 341
  - Wagon Setup .................................................... 341
- True Time I/O Stimulation ........................................ 343
  - Stimulation Program Examples .......................... 343
    - Running an Example Program Without Stimulation .... 343
    - Example Program with Periodical Stimulation of a Variable ...... 345
    - Example Program with Stimulated Interrupt ............. 346
    - Example of a Larger Stimulation File ................ 348
  - Stimulation Input File Syntax .......................... 351
  - EBNF .......................................................... 351
- Electrical Signal Generators and Signals Application to Device Pins .... 353
  - Signal IO Component ....................................... 353
  - Signal Description File EBNF .......................... 353
    - Signal File Format ..................................... 353
    - File Example 1 .......................................... 354
    - File Example 2 .......................................... 355
    - File Parameters ......................................... 355
    - Signal IO Usage ........................................ 356
    - Signal Commands ........................................ 356
    - Remarks ................................................... 357
  - Base Signal Files Provided .............................. 357
  - Virtual Wire Connections with the Pinconn IO Component .............. 358
    - Pinconn IO ................................................ 358
  - Command Set to Apply Signal on ATD Pin .................. 359
  - FCS Tutorials ................................................ 360
    - Guess the Number ....................................... 360
# Table of Contents

Step 1 - Environment Setup ........................................... 360
Step 2 - Creating the project ........................................ 360
Step 3 - ‘Target CPU’ Window ....................................... 361
Step 4 - ‘Bean Selector’ Window .................................... 362
Step 5 - ‘Project Panel’ Window .................................... 363
Step 6 - ‘Bean Inspector AS1:AsynchroSerial’ Window ............. 364
Step 7 - Generation of Driver Code .................................. 365
Step 8 - Verification of Files Created ............................... 365
Step 9 - Entering User Code ........................................... 366
Step 10 - Run .............................................................. 368

PWM Channel 0 .......................................................... 369
Step 1 - Environment Setup ........................................... 369
Step 2 - Creating Project .............................................. 369
Step 3 - ‘Target CPU’ Window ....................................... 369
Step 4 - Creating PWM Bean ......................................... 370
Step 5 - ‘Project Panel’ Window .................................... 370
Step 6 - ‘Bean Inspector PWM8.PWM .............................. 370
Step 7 - Generate Driver Code. ....................................... 370
Step 8 - Verification of Files Created ............................... 370
Step 9 - Entering User Code ........................................... 371
Step 10 - Run .............................................................. 372
VisualizationTool Properties ........................................ 372
‘Chart’ Properties ....................................................... 372
Period ‘Bar Properties’ ............................................... 373
Duty Time ‘Bar Properties’ ........................................... 373

## 11 ICD-12 Connection

Introduction ............................................................. 375
Installation Notice ..................................................... 375
Interfacing Your System and ICD-12 ................................. 376
Loading the ICD-12 Connection Component .......................... 377
Default Connection Setup ............................................ 378
ICD-12 Default Environment ......................................... 379
ICDPORT Variable ...................................................... 379
BMDELAY Variable ..................................................... 380
Table of Contents

Command Files and ICD-12 Commands ........................................... 381
Memory Configuration ................................................................. 381

12 Softec inDart HCS12 Connection ............................ 383

Technical Considerations .......................................................... 383
CodeWarrior and SofTec HCS12 Connection ................................. 383
First Steps Using the Stationery Wizard ...................................... 384
First Steps From Within an Existing Project ............................... 388

inDart-HCS12 Menu Options ....................................................... 390
MCU Configuration Option ......................................................... 391
User’s Manual Option ............................................................... 391
About Option ............................................................................ 391
MCU Configuration Dialog Box .................................................. 391
Communication Settings Dialog Box .......................................... 392
About Dialog Box ..................................................................... 393

13 HCS12 Serial Monitor Connection .............................. 395

Serial Monitor Technical Considerations ................................. 395
CodeWarrior and Serial Monitor Connection .............................. 395
First Steps Using the Stationery Wizard ...................................... 396
First Steps From Within an Existing Project ............................... 401

MONITOR-HCS12 Menu Options .................................................. 405
Monitor Communication............................................................ 405
Vector Mirroring Setup.............................................................. 405
Erase Flash .............................................................................. 405
Trigger Module Settings ............................................................ 405
Bus Trace ................................................................................ 406
Select Derivative ..................................................................... 406
Monitor Setup Window ............................................................... 406
Monitor Communication Tab ...................................................... 406
Vector Table Mirroring Tab ......................................................... 407
Derivative Selection Dialog Box ................................................. 408

14 BDIK Connection ................................................................. 409

BDIK Highlights ...................................................................... 410

HC(S)12(X) Debugger Manual xiv
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDIK Requirements</td>
<td>410</td>
</tr>
<tr>
<td>BDIK Connection Introduction</td>
<td>410</td>
</tr>
<tr>
<td>Interfacing BDIK and Your System</td>
<td>411</td>
</tr>
<tr>
<td>BDI Interface Software Setup</td>
<td>412</td>
</tr>
<tr>
<td>Running the ABATRON Configuration Tool</td>
<td>412</td>
</tr>
<tr>
<td>Example with B10C12.EXE Configuration Tool</td>
<td>412</td>
</tr>
<tr>
<td>Firmware Loading</td>
<td>413</td>
</tr>
<tr>
<td>Initialization List (Startup Init List) Loading</td>
<td>414</td>
</tr>
<tr>
<td>Communication with the Debugger Setup</td>
<td>415</td>
</tr>
<tr>
<td>BDI Working Mode and Setup/List Transmission</td>
<td>416</td>
</tr>
<tr>
<td>Loading the BDIK Connection</td>
<td>416</td>
</tr>
<tr>
<td>BDIK Connection Menu Entries</td>
<td>419</td>
</tr>
<tr>
<td>Load</td>
<td>419</td>
</tr>
<tr>
<td>Reset</td>
<td>419</td>
</tr>
<tr>
<td>Communication... or Connect...</td>
<td>419</td>
</tr>
<tr>
<td>Setup</td>
<td>420</td>
</tr>
<tr>
<td>Configure BDI Box</td>
<td>420</td>
</tr>
<tr>
<td>Set Bank</td>
<td>420</td>
</tr>
<tr>
<td>Command Files</td>
<td>420</td>
</tr>
<tr>
<td>Help</td>
<td>420</td>
</tr>
<tr>
<td>BDIK Connection Windows, Edit and Dialog Boxes</td>
<td>420</td>
</tr>
<tr>
<td>Communication Device Specification Edit Box</td>
<td>421</td>
</tr>
<tr>
<td>Setup Dialog Box</td>
<td>422</td>
</tr>
<tr>
<td>BDIK Status Bar Information</td>
<td>423</td>
</tr>
<tr>
<td>Status Messages</td>
<td>423</td>
</tr>
<tr>
<td>BDI ready V x.xx</td>
<td>423</td>
</tr>
<tr>
<td>No Link To Target</td>
<td>423</td>
</tr>
<tr>
<td>RUNNING</td>
<td>423</td>
</tr>
<tr>
<td>HALTED</td>
<td>424</td>
</tr>
<tr>
<td>RESET</td>
<td>424</td>
</tr>
<tr>
<td>Stepping and Breakpoint Messages</td>
<td>424</td>
</tr>
<tr>
<td>STEPPED</td>
<td>424</td>
</tr>
<tr>
<td>STEPPED OVER</td>
<td>424</td>
</tr>
<tr>
<td>STOPPED</td>
<td>424</td>
</tr>
<tr>
<td>TRACED</td>
<td>424</td>
</tr>
</tbody>
</table>
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREAKPOINT</td>
<td>424</td>
</tr>
<tr>
<td>WATCHPOINT</td>
<td>425</td>
</tr>
<tr>
<td>Terminal Emulation</td>
<td>425</td>
</tr>
<tr>
<td>Example for CPU12 Targets</td>
<td>425</td>
</tr>
<tr>
<td>Flash Programming</td>
<td>427</td>
</tr>
<tr>
<td>Examples of HC12/CPU12 Direct Commands</td>
<td>427</td>
</tr>
<tr>
<td>BDIK Connection Environment</td>
<td>429</td>
</tr>
<tr>
<td>Default Target Setup</td>
<td>429</td>
</tr>
<tr>
<td>Example</td>
<td>431</td>
</tr>
<tr>
<td>BDIK Connection Commands</td>
<td>435</td>
</tr>
<tr>
<td>HC12 and HCS12 Banked Memory support</td>
<td>438</td>
</tr>
<tr>
<td>Banked Memory Location Window</td>
<td>438</td>
</tr>
<tr>
<td>PPAGE Tab</td>
<td>439</td>
</tr>
<tr>
<td>DPAGE Tab</td>
<td>439</td>
</tr>
<tr>
<td>EPAGE Tab</td>
<td>440</td>
</tr>
<tr>
<td>Various Tab (Not For All Connections)</td>
<td>441</td>
</tr>
<tr>
<td>Banked Memory Location-associated Commands</td>
<td>441</td>
</tr>
<tr>
<td>BANKWINDOW Examples</td>
<td>443</td>
</tr>
<tr>
<td>Banked Memory Location-associated Environment Variables</td>
<td>444</td>
</tr>
<tr>
<td>BDIK Connection Command Files</td>
<td>446</td>
</tr>
<tr>
<td>Startup Command File</td>
<td>447</td>
</tr>
<tr>
<td>Reset Command File</td>
<td>447</td>
</tr>
<tr>
<td>Preload Command File</td>
<td>448</td>
</tr>
<tr>
<td>Postload Command File</td>
<td>448</td>
</tr>
<tr>
<td>Vppon Command File</td>
<td>448</td>
</tr>
<tr>
<td>Vppoff Command File</td>
<td>448</td>
</tr>
<tr>
<td>Associated Commands</td>
<td>449</td>
</tr>
<tr>
<td>Target Interface Command Files Window</td>
<td>451</td>
</tr>
<tr>
<td>Associated Environment Variables</td>
<td>452</td>
</tr>
<tr>
<td>BDIK Connection Demo</td>
<td>454</td>
</tr>
<tr>
<td>Debugging with the BDIK Connection</td>
<td>454</td>
</tr>
<tr>
<td>Starting with the BDIK Connection</td>
<td>454</td>
</tr>
<tr>
<td>Debugging an Application in RAM</td>
<td>455</td>
</tr>
</tbody>
</table>
Book III - HC(S)12(X) Debug Connections - Common Features

Book III Contents ......................................................... 457

15 HCS12 On-chip DBG Module ................................. 459
   DBG Features ......................................................... 459
   Specific Connection Menu Options ....................... 460
   Popup Menu Entries in Source, Data, Assembly and Memory Windows .... 460
      Source and Assembly Windows .............................. 460
      Trigger Storing as Markpoints .......................... 463
      Data and Memory Windows ................................. 466
      Expert Triggers ............................................... 467
      Trigger Settings ............................................. 470
      Trigger Module Usage ...................................... 471
   DBG Support Status Bar Item ............................... 471
   Trigger Module Settings Window ......................... 472
      DBG Module Mode Setup .................................. 473
      Automatic Mode (Default) ............................... 473
      Expert Mode .................................................. 474
      Expert Mode Tab ............................................ 475
      Profiling and Coverage Mode ......................... 475
      Disabled Mode ................................................. 476
   “Memory Access” Triggers ................................. 477
      Memory Access at Address A .............................. 477
      Memory Access at Address A or Address B ............ 477
      Memory Access Inside Address A - Address B Range .... 477
      Memory Access at Address A then Memory Access at Address B .... 477
      Memory Access at Address A and Value on Data Bus Match ...... 477
      Memory Access at Address A and Value on Data Bus Mismatch .... 478
   “Instruction” Triggers ............................................. 479
      Instruction at Address A is Executed .................. 479
### Table of Contents

Instruction at Address A or Address B Is Executed .......................... 479  
Instruction Execution Inside Address A - Address B Range" ........... 480  
Instruction Execution Outside Address A - Address B Range" ........... 480  
Instruction at Address A then at Address B Were Executed .............. 480  
Instruction at Address A and Value on Data Bus Match .................. 480  
Instruction at Address A and Value on Data Bus Mismatch ............... 481  
"Capture” Triggers ................................................................. 482  
Capture Read/Write Values at Address B .................................. 482  
Capture Read/Write Values at Address B After Access at Address A . 482  
DBG Module Options ............................................................. 483  
Program Code Change of Flow Recording ................................. 483  
When Data Recording ............................................................ 484  
Trigger Editing ......................................................................... 485  
Trigger Module Settings Window - Display Information ................. 487  
General Settings Tab ............................................................... 488  
Trace Component Window .......................................................... 489  
Instructions Display ............................................................... 490  
Graphical Display .................................................................... 491  
Textual Display ......................................................................... 491  
Column Display and Moving ...................................................... 492  
Dumping Frames to File ............................................................ 493  
Goto Frame ............................................................................. 493  
Clearing Frames ....................................................................... 494  
DBG Module FIFO/Buffer Display .............................................. 494  
Recorded Data Display ............................................................. 495  
Limitations .................................................................................. 496  

16 HCS12X On-chip DBG Module ................................................. 497  
DBG Features ............................................................................ 497  
Specific Connection Menu Options ............................................. 498  
Popup Menu Entries in Source, Data, Assembly and Memory Windows . 498  
Source and Assembly Windows .................................................. 498  
Trigger Storing as Markpoints ..................................................... 501  
Data and Memory Windows ....................................................... 504  
Expert Triggers .......................................................................... 505
### Trigger Settings
- Automatic Mode (Default) .................................................. 511
- Expert Mode .................................................................. 512
- Expert Mode Tab .......................................................... 513
- Profiling and Coverage Mode ............................................... 513
- Disabled Mode .............................................................. 514

### “Memory Access” Triggers
- Memory Access at Address A .............................................. 515
- Memory Access at Address A or Address B ......................... 515
- Memory Access Inside Address A - Address B Range .............. 515
- Memory Access at Address A then Memory Access at Address B .. 515
- Memory Access at Address A and Value on Data Bus Match ....... 515
- Memory Access at Address A and Value on Data Bus Mismatch .... 516

### “Instruction” Triggers
- Instruction at Address A Is Executed ................................. 517
- Instruction at Address A or Address B Is Executed .................. 517
- Instruction Execution Inside Address A - Address B Range" ...... 518
- Instruction Execution Outside Address A - Address B Range" .... 518
- Instruction at Address A then at Address B Were Executed ....... 518
- Instruction at Address A and Value on Data Bus Match .......... 518
- Instruction at Address A and Value on Data Bus Mismatch ....... 519

### “Capture” Triggers
- Capture Read/Write Values at Address B ............................. 520
- Capture Read/Write Values at Address B After Access at Address A .. 520

### DBG Module Options
- Program Code Change of Flow Recording ............................ 521
- When Data Recording ....................................................... 522
- Trigger Editing ............................................................... 523
- Trigger Module Settings Window - Display Information ............ 525
- General Settings Tab ....................................................... 526
- Trace Component Window ............................................... 527
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Display</td>
<td>528</td>
</tr>
<tr>
<td>Graphical Display</td>
<td>529</td>
</tr>
<tr>
<td>Textual Display</td>
<td>529</td>
</tr>
<tr>
<td>Column Display and Moving</td>
<td>530</td>
</tr>
<tr>
<td>Dumping Frames to File</td>
<td>531</td>
</tr>
<tr>
<td>Goto Frame</td>
<td>531</td>
</tr>
<tr>
<td>Clearing Frames</td>
<td>532</td>
</tr>
<tr>
<td>DBG Module FIFO/Buffer Display</td>
<td>532</td>
</tr>
<tr>
<td>Recorded Data Display</td>
<td>533</td>
</tr>
<tr>
<td>Limitations</td>
<td>534</td>
</tr>
<tr>
<td>17 Debugging Memory Map</td>
<td>535</td>
</tr>
<tr>
<td>Introduction</td>
<td>535</td>
</tr>
<tr>
<td>Debugging Memory Map GUI</td>
<td>536</td>
</tr>
<tr>
<td>Edition Dialog Box</td>
<td>537</td>
</tr>
<tr>
<td>Types</td>
<td>538</td>
</tr>
<tr>
<td>Priorities</td>
<td>539</td>
</tr>
<tr>
<td>Remarks</td>
<td>540</td>
</tr>
<tr>
<td>CPU Core Types and Priorities</td>
<td>540</td>
</tr>
<tr>
<td>HC12(CPU12) Core</td>
<td>540</td>
</tr>
<tr>
<td>Priorities:</td>
<td>540</td>
</tr>
<tr>
<td>Types:</td>
<td>541</td>
</tr>
<tr>
<td>HCS12 Core</td>
<td>542</td>
</tr>
<tr>
<td>Priorities:</td>
<td>542</td>
</tr>
<tr>
<td>Types:</td>
<td>543</td>
</tr>
<tr>
<td>HCS12X Core</td>
<td>543</td>
</tr>
<tr>
<td>Priorities:</td>
<td>543</td>
</tr>
<tr>
<td>Types:</td>
<td>543</td>
</tr>
<tr>
<td>DMM Commands</td>
<td>544</td>
</tr>
<tr>
<td>18 HC(S)12(X) Flash Programming</td>
<td>545</td>
</tr>
<tr>
<td>Non-volatile Memory Control Utility Introduction</td>
<td>545</td>
</tr>
<tr>
<td>Automated Application Programming</td>
<td>545</td>
</tr>
<tr>
<td>Setup</td>
<td>546</td>
</tr>
<tr>
<td>Advanced Options: Preventing Erasing</td>
<td>547</td>
</tr>
</tbody>
</table>
## Table of Contents

- NVMC Graphical User Interface ........................................... 548
  - Modules and Module States ............................................. 548
- NVMC Dialog Box .................................................................. 549
- Flash Module Handling ....................................................... 551
- MCU Speed Information ...................................................... 552
- Configuration: FPP File Loading ......................................... 552
- Loading an Application in Flash ......................................... 553
- Preparing, Loading an Application in Flash ......................... 555
- Hardware Considerations .................................................. 556
  - HC12 (CPU12) CPU devices .............................................. 556
    - HC12B32 .................................................................. 556
    - HC12D60 .................................................................. 556
    - HC12DG128 ................................................................ 557
  - HCS12 and HCS12X CPU devices ....................................... 558
    - HCS12 EEPROM’s Relocation ......................................... 559
    - EB386 Compliancy and RAM Moving ............................... 560
  - Legacy Flash Programming Commands in Preload and Postload Command Files ......................................................... 560

### 19 XGATE Debugging Support ............................................. 563
- XGATE Debugging Support .................................................. 563
# Book IV - Commands

## Debugger Engine Commands

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commands Overview</td>
<td>567</td>
</tr>
<tr>
<td>Command Syntax</td>
<td>568</td>
</tr>
<tr>
<td>Available Command Lists</td>
<td>568</td>
</tr>
<tr>
<td>Kernel Commands</td>
<td>568</td>
</tr>
<tr>
<td>Base Commands</td>
<td>569</td>
</tr>
<tr>
<td>Environment Commands</td>
<td>571</td>
</tr>
<tr>
<td>Component Commands</td>
<td>572</td>
</tr>
<tr>
<td>Component Specific Commands</td>
<td>573</td>
</tr>
<tr>
<td>Command Syntax Terms</td>
<td>575</td>
</tr>
<tr>
<td>Module Names</td>
<td>576</td>
</tr>
<tr>
<td>Debugger Commands</td>
<td>576</td>
</tr>
<tr>
<td>A</td>
<td>577</td>
</tr>
<tr>
<td>ACTIVATE</td>
<td>577</td>
</tr>
<tr>
<td>ADDXPR</td>
<td>578</td>
</tr>
<tr>
<td>ATTRIBUTES</td>
<td>578</td>
</tr>
<tr>
<td>In the Command Component</td>
<td>578</td>
</tr>
<tr>
<td>In the Procedure Component</td>
<td>579</td>
</tr>
<tr>
<td>In the Assembly Component</td>
<td>579</td>
</tr>
<tr>
<td>In the Register Component</td>
<td>580</td>
</tr>
<tr>
<td>In the Source Component</td>
<td>582</td>
</tr>
<tr>
<td>In the Data Component</td>
<td>583</td>
</tr>
<tr>
<td>In the Memory Component</td>
<td>584</td>
</tr>
<tr>
<td>In the Inspector Component</td>
<td>586</td>
</tr>
<tr>
<td>AT</td>
<td>588</td>
</tr>
<tr>
<td>AUTOSIZE</td>
<td>588</td>
</tr>
<tr>
<td>BASE</td>
<td>589</td>
</tr>
<tr>
<td>BC</td>
<td>589</td>
</tr>
<tr>
<td>BCKCOLOR</td>
<td>590</td>
</tr>
<tr>
<td>BD</td>
<td>591</td>
</tr>
</tbody>
</table>
Table of Contents

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BS</td>
<td>592</td>
</tr>
<tr>
<td>CALL</td>
<td>594</td>
</tr>
<tr>
<td>CD</td>
<td>595</td>
</tr>
<tr>
<td>CF</td>
<td>596</td>
</tr>
<tr>
<td>CLOCK</td>
<td>598</td>
</tr>
<tr>
<td>CLOSE</td>
<td>598</td>
</tr>
<tr>
<td>COPYMEM</td>
<td>599</td>
</tr>
<tr>
<td>CMDFILE</td>
<td>599</td>
</tr>
<tr>
<td>CR</td>
<td>600</td>
</tr>
<tr>
<td>CYCLE</td>
<td>600</td>
</tr>
<tr>
<td>DASM</td>
<td>601</td>
</tr>
<tr>
<td>DB</td>
<td>602</td>
</tr>
<tr>
<td>DDEPROTOCOL</td>
<td>603</td>
</tr>
<tr>
<td>DEFINE</td>
<td>604</td>
</tr>
<tr>
<td>DETAILS</td>
<td>605</td>
</tr>
<tr>
<td>DL</td>
<td>606</td>
</tr>
<tr>
<td>DUMP</td>
<td>607</td>
</tr>
<tr>
<td>DW</td>
<td>607</td>
</tr>
<tr>
<td>E</td>
<td>608</td>
</tr>
<tr>
<td>ELSE</td>
<td>609</td>
</tr>
<tr>
<td>ELSEIF</td>
<td>609</td>
</tr>
<tr>
<td>ENDFOCUS</td>
<td>610</td>
</tr>
<tr>
<td>ENDFOR</td>
<td>611</td>
</tr>
<tr>
<td>ENDIF</td>
<td>611</td>
</tr>
<tr>
<td>ENDWHILE</td>
<td>612</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>612</td>
</tr>
<tr>
<td>EXIT</td>
<td>613</td>
</tr>
<tr>
<td>FILL</td>
<td>613</td>
</tr>
<tr>
<td>FILTER</td>
<td>614</td>
</tr>
<tr>
<td>FIND</td>
<td>615</td>
</tr>
<tr>
<td>FINDPROC</td>
<td>616</td>
</tr>
<tr>
<td>FOCUS</td>
<td>617</td>
</tr>
<tr>
<td>FOLD</td>
<td>618</td>
</tr>
<tr>
<td>FONT</td>
<td>618</td>
</tr>
<tr>
<td>FOR</td>
<td>619</td>
</tr>
</tbody>
</table>
Table of Contents

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPRINTF</td>
<td>620</td>
</tr>
<tr>
<td>FRAMES</td>
<td>620</td>
</tr>
<tr>
<td>G</td>
<td>621</td>
</tr>
<tr>
<td>GO</td>
<td>622</td>
</tr>
<tr>
<td>GOTO</td>
<td>623</td>
</tr>
<tr>
<td>GOTOIF</td>
<td>624</td>
</tr>
<tr>
<td>GRAPHICS</td>
<td>625</td>
</tr>
<tr>
<td>HELP</td>
<td>625</td>
</tr>
<tr>
<td>IF</td>
<td>626</td>
</tr>
<tr>
<td>INSPECTOROUTPUT</td>
<td>627</td>
</tr>
<tr>
<td>INSPECTORUPDATE</td>
<td>627</td>
</tr>
<tr>
<td>LF</td>
<td>628</td>
</tr>
<tr>
<td>LOAD</td>
<td>629</td>
</tr>
<tr>
<td>LOADCODE</td>
<td>630</td>
</tr>
<tr>
<td>LOADSYMBOLS</td>
<td>630</td>
</tr>
<tr>
<td>LOG</td>
<td>631</td>
</tr>
<tr>
<td>More About Logging of IF, FOR, WHILE and REPEAT</td>
<td>632</td>
</tr>
<tr>
<td>LS</td>
<td>635</td>
</tr>
<tr>
<td>MEM</td>
<td>636</td>
</tr>
<tr>
<td>MS</td>
<td>637</td>
</tr>
<tr>
<td>NB</td>
<td>638</td>
</tr>
<tr>
<td>NOCR</td>
<td>640</td>
</tr>
<tr>
<td>NOLF</td>
<td>640</td>
</tr>
<tr>
<td>OPEN</td>
<td>641</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>642</td>
</tr>
<tr>
<td>P</td>
<td>643</td>
</tr>
<tr>
<td>PAUSETST</td>
<td>644</td>
</tr>
<tr>
<td>PRINTF</td>
<td>644</td>
</tr>
<tr>
<td>PTRARRAY</td>
<td>645</td>
</tr>
<tr>
<td>RD</td>
<td>645</td>
</tr>
<tr>
<td>RECORD</td>
<td>646</td>
</tr>
<tr>
<td>REPEAT</td>
<td>647</td>
</tr>
<tr>
<td>RESET</td>
<td>648</td>
</tr>
<tr>
<td>RESTART</td>
<td>648</td>
</tr>
<tr>
<td>RETURN</td>
<td>649</td>
</tr>
</tbody>
</table>
## Table of Contents

RS .................................................. 650
S ................................................... 651
SAVE ............................................... 652
SAVEBP ........................................... 653
SET ............................................... 654
SETCOLORS ..................................... 654
SLAY .............................................. 655
SLINE ............................................ 655
SMEM ............................................. 656
SMOD ............................................. 657
SPC ............................................... 658
SPROC ........................................... 659
SREC ............................................. 660
STEPINTO ....................................... 661
STEPOUT ........................................ 662
STEPOVER ....................................... 663
STOP ............................................. 664
t ................................................ 665
TESTBOX ........................................ 666
TUPDATE ......................................... 666
UNDEF ........................................... 667
UNFOLD ......................................... 669
UNTIL ............................................ 670
UPDATERATE .................................... 670
VER .............................................. 671
WAIT ............................................. 672
WB ................................................. 673
WHILE ........................................... 674
WL ................................................ 675
WW ............................................... 675
ZOOM ............................................. 676

### 21 Debugger Connection-specific Commands

BDIK Connection Commands ................................ 677
Banked Memory Location-associated Commands .......... 681
Table of Contents

BANKWINDOW Examples ............................................. 684
ICD-12 Commands ...................................................... 686
NVMC Commands ....................................................... 687
FLASH ................................................................. 687
[ <blockNo> ] ......................................................... 690
DMM Commands ....................................................... 693
  Debugging Memory Map Manager Commands ......................... 693
  "DMM" Command .................................................. 693
  "DMM ADD" command .............................................. 693
  "DMM DEL" Command ............................................. 694
  "DMM SAVE" Command ........................................... 694
  "DMM DELETEALLMODULES" Command ............................. 694
  "DMM RELEASECACHES" Command ................................ 695
  "DMM CACHINGON" Command .................................... 695
  "DMM CACHINGOFF" Command .................................. 695
  "DMM HCS12MERHANDLINGON" Command ......................... 695
  "DMM HCS12MERHANDLINGOFF" Command ......................... 695
  "DMM OPENGUI" Command ....................................... 696
Full Chip Simulator Commands ...................................... 696
  FCS-Associated Component-specific Commands ................... 696
ADDCHANNEL ......................................................... 698
ADCPORT ............................................................ 698
CPORT .............................................................. 699
DELCHANNEL ......................................................... 699
ITPORT ............................................................. 700
ITVECT .............................................................. 700
KPORT .............................................................. 701
LCDPORT ............................................................ 701
LINKADDR ........................................................... 702
PPORT ............................................................... 702
PORT ............................................................... 703
PSMODE ............................................................. 703
REGBASE ............................................................ 704
RESETCYCLES ....................................................... 704
RESETMEM .......................................................... 705
## Table of Contents

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESETRAM</td>
<td>706</td>
</tr>
<tr>
<td>RESETSTAT</td>
<td>706</td>
</tr>
<tr>
<td>SEGPORT</td>
<td>707</td>
</tr>
<tr>
<td>SETCONTROL</td>
<td>707</td>
</tr>
<tr>
<td>SETCPU</td>
<td>708</td>
</tr>
<tr>
<td>SHOWCYCLES</td>
<td>708</td>
</tr>
<tr>
<td>WPORT</td>
<td>709</td>
</tr>
</tbody>
</table>
# Book V - Environment Variables

Book V Contents ......................................................... 711

## 22 Debugger Engine Environment Variables  713

- Debugger Environment ............................................. 714
- The Current Directory ............................................. 715
- Global Initialization File (MCUTOOLS.INI - PC Only) ........ 716
- Local Configuration File (usually project.ini) ................ 717
- Default Layout Configuration (PROJECT.INI) ................... 718
- Ini File Activation ................................................ 720
- Environment Variable Paths ....................................... 721
- Line Continuation .................................................. 722
- Environment Variables ............................................. 723
- ABSPATH: Absolute Path .......................................... 724
- DEFAULTDIR: Default Current Directory ....................... 725
- ENVIRONMENT=: Environment File Specification .............. 726
- GENPATH:  
- include “File” Path ............................................. 727
- LIBRARYPATH:  
- include <File>’ Path ............................................ 728
- OBJPATH: Object File Path ........................................ 729
- TMP: Temporary directory ........................................ 730
- USELIBPATH: Using LIBPATH Environment Variable .......... 731
- Search Order for Source Files .................................... 732
- In the Debugger for C Source Files (*.c, *.cpp) ............... 732
- In the Debugger for Assembly Source Files (*.dbg) ........... 732
- In the Debugger for Object Files (HILOADER) .................. 732
- Debugger Files ...................................................... 732

## 23 Connection-specific Environment Variables  735

- Connection-specific Environment Variables .................... 735
- BDIK Connection Environment Variables ......................... 735
- Example ............................................................ 738
- Banked Memory Location-associated Environment Variables . 742
- ICD-12 Default Environment ..................................... 745
Table of Contents

ICD-12 Environment Variables ........................................ 745
ICDPORT Variable ....................................................... 745
BMDELAY Variable ..................................................... 746

Index ................................................................. 747
Introduction

Manual Contents

The HC(S)12(X) Debugger Manual consists of the following books:

Book 1: Debugger Engine - defines the HC12, HCS12 and HC(S)12(X) common and base features, their functionality, and a description of the components that are available in the debugger.

- Chapter 1.1 “Introduction”
- Chapter 1.2 “Debugger Interface”
- Chapter 1.3 “Debugger Components”
- Chapter 1.4 “Control Points”
- Chapter 1.5 “Real Time Kernel Awareness”
- Chapter 1.6 “How To…”
- Chapter 1.7 “CodeWarrior Integration.”
- Chapter 1.8 “Debugger DDE Capabilities.”
- Chapter 1.9 “Synchronized Debugging Through DA-C IDE.”

Book 2: HC(S)12(X) Debugger Connections - defines the connections available for debugging code written for HC12 CPUs.

- Chapter 2.1 “”
- Chapter 2.2 “ICD-12 Connection”
- Chapter 2.3 “Softec inDart HCS12 Connection”
- Chapter 2.4 “HCS12 Serial Monitor Connection”
- Chapter 2.5 “BDIK Connection”

Book 3: HC(S)12(X) Debugger Connections - Common Features

- Chapter 3.1 “HCS12 On-chip DBG Module”
- Chapter 3.2 “HCS12X On-chip DBG Module”
- Chapter 3.3 “Debugging Memory Map”
- Chapter 3.4 “HC(S)12(X) Flash Programming”
Manual Contents

Book 4: Commands
- Chapter 4.1 Debugger Engine Commands
- Chapter 4.2 Debugger Connection-specific Commands

Book 5: Environment Variables
- Chapter 5.1 Debugger Engine Environment Variables
- Chapter 5.2 Connection-specific Environment Variables
Book I - Debugger Engine

Book I Contents

Each section of the Debugger manual includes information to help you become more familiar with the Debugger, to use all its functions and help you understand how to use the environment.

Book I, the Debugger engine, defines the HC12, HCS12 and HCS12X common and base features, their functionality, and gives a description of the components that are available in the debugger.

This book is divided into the following chapters:

- This chapter describes the manual and special features of the Debugger.
- The “Introduction” Chapter introduces the Debugger concept.
- The “Debugger Interface” Chapter provides all details about the Debugger user interface environment i.e., menus, toolbars, status bars and drag and drop facilities.
- The “Debugger Components” Chapter contains descriptions of each basic component and visualization utility.
- The “Control Points” Chapter is dedicated to the control points and associated windows.
- The “Real Time Kernel Awareness” Chapter contains descriptions of the Real Time concept and related applications.
- The “How To ...” Chapter provides answers for common questions and describes how to use advanced features of the Debugger.
- The “CodeWarrior Integration,” chapter explains how to configure the Debugger for use with CodeWarrior.
- The Debugger DDE Capabilities describe the debugger DDE features.
- The Synchronized Debugging Through DA-C IDE chapter explains the use of tools with the DA-C IDE from RistanCase
Introduction

This section is an introduction to the Debugger from Freescale used in 8/16 bit embedded applications.

Freescale Debugger

The Debugger is a member of the tool family for Embedded Development. It is a Multipurpose Tool that you can use for various tasks in the embedded system and industrial control world. Some typical tasks are:

- Simulation and debugging of an embedded application.
- Simulation and debugging of real time embedded applications.
- Simulation and/or cross-debugging of an embedded application.
- Multi-Language Debugging: Assembly, C and C++
- True Time Stimulation
- User Components creation with the Peripheral Builder
- Simulation of a hardware design (e.g., board, processor, I/O chip).
- Building a target application using an object oriented approach.
- Building a host application controlling a plant using an object oriented approach.

Debugger Application

A Debugger Application contains the Debugger Engine and a set of debugger components bound to the task that they should perform (for example a simulation and debugging session). The Debugger Engine is the heart of the system. It monitors and coordinates the tasks of the components. Each Debugger Component has its own functionality (e.g., source level debugging, profiling, I/O stimulation).

You can adapt your Debugger application to your specific needs. Integrating or removing the Debugger Components is very easy. You can add additional Debugger Components (for example, for simulation of a specific I/O peripheral chip) and integrate them with your Debugger Application.

You can also open several components of the same type.
Debugger Features

- True 32-bit application
- Powerful features for embedded debugging
- Special features for real time embedded debugging
- Powerful features for True Time Simulation
- Various and Same look Target Interfaces
- User Interface
- Versatile and intuitive drag and drop functions between components
- Folding and unfolding of objects like functions, structures, classes
- Graphical editing of user defined objects
- Visualization functions
- Smart interactions with objects
- Extensibility function
- Both Powerful Simulation & Debugger
- Show Me How Tool
- GUI (graphical user interface) version including command line
- Context sensitive help
- Configurable GUI with Tool Bar
- Smooth integration into third party tools
- Supports both Freescale and ELF/DWARF Object File Format and S-Records.

Demo Version Limitations on Components

When the Debugger is started in demo mode or with an invalid engine license, then all components that are protected with FLEXlm are in demo mode. The limitations of all components are described in their respective chapter.
Debugger Interface

This chapter describes the Debugger Graphic User Interface (GUI). Click any of the following links to jump to the corresponding section of this chapter:

- Introduction
- Application Programs
- Starting the Debugger
- Debugger Main Window
- Component Associated Menus
- Highlights of the User Interface

Introduction

The CodeWarrior IDE main window acts as a container for windows of all the debugger components. The main window provides a main menu bar, a tool bar, a status bar for status information, and object information bars for several components.

The Debugger main window allows you to manage the layout of the different component windows (Window menu of the Debugger application). Component windows are organized as follows:

- Tiled arrangement - Auto tiled, component windows are automatically resized when the main window is resized
- Component windows are overlapped
- Component windows that are currently minimized are Debugger Main window icons

Application Programs

The CodeWarrior installer places executable programs in the prog subdirectory of the CodeWarrior installation directory. For example, if you installed the CodeWarrior IDE software in C:\Freescale, you would find all program files in the folder C:\Freescale\prog.

The following list is an overview of files that CodeWarrior uses for C/C++ debugging.

- hiwave.exe  Debugger executable file
Starting the Debugger

This section explains how you can start the debugger from within the Codewarrior IDE or from a DOS command line.

Starting from within the IDE

There are two ways to start the debugger from within the IDE, from a Project window icon, or from the IDE Main Window menu bar.

Starting Debug from the Project Window

To start the debugger from the Project window, click the Debug icon (Figure 2.1), at the top of the Project window.

Figure 2.1 Project Window Make and Debug Icons

Starting Debug from the Main Window Menu Bar

You can also start the debugger from the main menu bar of the CodeWarrior IDE. To start the debugger from the main menu bar, select Debug from the Project menu: (Project > Debug.)
Debugger Command Line Start

You can start the debugger from a DOS command line. The command syntax is as follows:

HIWAVE.EXE [<AbsFileName> {-<options>}]

where AbsFileName is the name of the application to load in the debugger. Precede each option with a dash.

Command Line Options

DOS command line options are:

-T=<time>: Test mode

The debugger terminates after the specified time (in seconds). The default value is 300 seconds. For example:

c:\Freescale\prog\hiwave.exe -T=10

The above example instructs the debugger to terminate after 10 seconds.

-Target=<targetname>
This option sets the specified connection. For example:

C:\Freescale\prog\hiwave.exe
C:\Freescale\demo\hc12\sim\fibo.abs -w -Target=sim

The command in the above example starts the debugger and loads fibo.abs file.

-W: Wait mode
Debugger will wait even when a <exeName> is specified.

-Instance=%currentTargetName
This option defines a build instance name. When a build instance is defined, the same one will be used. For example:

  c:\\Freescale\\prog\\hiwave.exe -Instance=%currentTargetName

If you attempt to start the debugger again, the existing instance of the debugger is brought to the foreground.

-Prod= <fileName>
This option specifies the project directory and/or project file to be used at start-up. For example:

  c:\\Freescale\\prog\\hiwave.exe -Prod=c:\demoproject\test.pjt

-Nodefaults
Debugger will not load the default layout (see section 4 of the Project file Activation). For example:

  c:\\Freescale\\prog\\hiwave.exe -nodefaults

-Cmd = <Command>
This option specifies a command to be executed at start-up: -cmd = "" {characters}. For example:

  c:\\Freescale\\prog\\hiwave.exe -cmd="open recorder"

-C <cmdFile>
This option specifies a command file to be executed at start-up. For example:

  c:\\Freescale\\prog\\hiwave.exe -c c:\temp\mycommandfile.txt

-ENVpath: "-Env" <Environment Variable> "=" <Variable Setting>
This option sets an environment variable. This environment variable may be used to overwrite system environment variables. For example:
Debugger Interface

Debugger Main Window

\[c:\Freescale\prog\hiwave.exe -EnvOBJPATH=c:\sources\obj\]

**NOTE** Options are not case sensitive.

---

**Order of Commands**

Commands specified by options are executed in the following order:

1. Load (activate) the project file (see below). If the project file is not specified, “project.ini” is used by default.
2. Load <exeFile> if available and start* running unless option \|{W} was specified
3. Execute command file <cmdFile> if specified
4. Execute command if specified
5. *Start running unless option \|{W} was specified

**NOTE** * In version 6.0 of the debugger, the loaded program is started after all command and command files are executed.

**NOTE** The function \Open in the File menu will interpret any file without an \ini extension as a command file and not a project file.

---

**Example**

\[C:\Freescale\PROG \DEMO\TEST.ABS -w -d\]

---

**Debugger Main Window**

Once you start the Debugger, the True Time Simulator & Real Time Debugger window opens in the right side of the IDE Main Window.
Debugger Main Window Toolbar

The Debugger Main Window toolbar is the default toolbar. Most of the Main Window menu commands have a related shortcut icon on this toolbar. Figure 2.4 identifies each default icon.
Debugger Interface
Debugger Main Window

Figure 2.4 Debugger Main Window Toolbar

A tool tip is available when you point the mouse at an icon.

Debugger Main Window Status Bar

The status bar at the bottom of the Debugger Main Window, shown in Figure 2.5 contains a context sensitive help line for connection specific information, e.g., number of CPU cycles for the Simulator connection and execution status.

Figure 2.5 The Debugger Status Bar

Main Window Menu Bar

The Debugger Main Window Menu Bar, shown in Figure 2.6 is associated with the main function of the debugger application, connection, and selected windows.

Figure 2.6 Debugger Window Menu Bar

NOTE You can select menu commands by pressing the ALT key to select the menu bar, then pressing the key corresponding to the underlined letter in the menu command.

Table 2.1 describes menu entries available in the menu bar.
Table 2.1 Description of the Main Menu Toolbar Entries

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Contains entries to manage debugger configuration files.</td>
</tr>
<tr>
<td>View</td>
<td>Contains entries to configure the toolbar.</td>
</tr>
<tr>
<td>Run</td>
<td>Contains entries to monitor a simulation or debug session.</td>
</tr>
<tr>
<td>Connection</td>
<td>Contains entries to select the debugger connection. Once a connection has been selected, the name of this heading changes.</td>
</tr>
<tr>
<td>Component</td>
<td>Contains entries to select and configure extra component window</td>
</tr>
<tr>
<td>Data</td>
<td>Contains entries to select Data component functions.</td>
</tr>
<tr>
<td>Window</td>
<td>Contains entries to set the component windows.</td>
</tr>
<tr>
<td>Help</td>
<td>A standard Windows Help menu.</td>
</tr>
</tbody>
</table>

File Menu

The File menu shown in Figure 2.7 is dedicated to the debugger project.

Figure 2.7 File Menu
Debugger Interface
Debugger Main Window

Table 2.2 describes File Menu entries.

Table 2.2 File Menu Entry Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>Creates a new project.</td>
</tr>
<tr>
<td>Load Application</td>
<td>Loads an executable file (or debugger connection if nothing is selected).</td>
</tr>
<tr>
<td>...\restart.abs</td>
<td>Recent applications list</td>
</tr>
<tr>
<td>...\await.abs</td>
<td></td>
</tr>
<tr>
<td>Open Configuration</td>
<td>Opens the debugger project window. You can load a project file .PJT or .INI. Additionally you can load an existing .HWC file corresponding to a debugger configuration file. You can load a project .INI file containing component names, associated window positions and parameters, window parameters (fonts, background colors, etc.), connection name e.g., Simulator and the .ABS application file to load.</td>
</tr>
<tr>
<td>Save Configuration</td>
<td>Saves the project file</td>
</tr>
<tr>
<td>Save Project As</td>
<td>Opens the debugger project window to save the project file under a different path and name, and format (PJT; INI...).</td>
</tr>
<tr>
<td>Configuration</td>
<td>Opens the Preferences window to set environment variables for current project.</td>
</tr>
<tr>
<td>1.Project.ini</td>
<td>Recent project file list</td>
</tr>
<tr>
<td>2.Test.ini</td>
<td></td>
</tr>
<tr>
<td>3...</td>
<td></td>
</tr>
<tr>
<td>Exit</td>
<td>Quits the Debugger.</td>
</tr>
</tbody>
</table>

You can shortcut some of these functions by clicking toolbar icons (refer to the Debugger Main Window Toolbar section).

Preferences Window

Open the Preferences window by selecting Configuration from the Files menu. With this window (Figure 2.8) it is possible to set up environment variables for the current project. New variables are saved in the current project file when you click the OK button.
NOTE The corresponding menu entry (File>Configuration) is only enabled if a project file is loaded.

Figure 2.8 Preferences Window

The Preferences Window contains the following controls:

- A list box containing all available environment variables. You can select a variable with the mouse or Up/Down buttons.
- Command Line Arguments are displayed in the text box. You can add, delete, or modify options, and specify a directory with the browse button (...).
- A second list box contains the arguments for all of the environment variables defined in the corresponding Environment section. Select a variable with the mouse or Up/Down buttons.

Command Buttons:

- **OK**: Changes are confirmed and saved in current project file.
- **Cancel**: Closes dialog box without saving changes.
- **Help**: Opens the help file.
Debugger Interface
Debugger Main Window

View Menu

In the Main Window View menu (Figure 2.9) you can choose to show or hide the toolbar, status bar, window component titles and headlines (see the Component Windows Object Info Bar). You can select smaller window borders and customize the toolbar. Table 2.3 describes the View Menu entries.

![Figure 2.9 View Menu](image)

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toolbar</td>
<td>Check / uncheck Toolbar if you want to display or hide it.</td>
</tr>
<tr>
<td>Status Bar</td>
<td>Check / uncheck Status Bar if you want to display or hide it.</td>
</tr>
<tr>
<td>Hide Title</td>
<td>Check / uncheck Hide Title if you want to hide or display the window title.</td>
</tr>
<tr>
<td>Hide Headline</td>
<td>Check / uncheck Hide Headline if you want to hide or display the headline.</td>
</tr>
<tr>
<td>Small Borders</td>
<td>Check / uncheck Small Border if you want to display or hide small window borders.</td>
</tr>
<tr>
<td>Customize</td>
<td>Opens the debugger Customize Toolbar window.</td>
</tr>
</tbody>
</table>

Customizing the Toolbar

When you select Customize from the View menu, the Customize Toolbar dialog box appears. You can customize the toolbar of the Debugger, adding and removing component shortcuts and action shortcuts in this dialog box. You can also insert separators to separate icons. Almost all functions in View, Run and Window menus are available as shortcut buttons, as shown in Figure 2.10.
Debugger Interface

Debugger Main Window

Figure 2.10 Customize Toolbar Dialog Box

- Select the desired shortcut button in the Available buttons list box and click Add to install it in the toolbar.
- Select a button in the Toolbar buttons list box and click Remove to remove it from the toolbar.

Demo Version Limitations

The default toolbar cannot be configured.

Examples of View Menu Options

Figure 2.11 shows a typical component window display.

Figure 2.11 Typical Component Window Display

```c
#include <stdio.h> /* for EnableInterruption macro */
#include "derivative.h" /* include peripheral declarations */

void main(void) {
    EnableInterrupts; /* enable interrupts */
    /* include your code here */

    for (;;) {
        /* PRINT_MESSAGE(); /* print the dog */
        /* loop forever */
        /* please make sure that you never leave this function */
    }
}
Figure 2.12 shows a component window without a title and headline.

**Figure 2.12 Component Window without Title and Headline**

```c
i = 2;
while (i <= n) {
    fibo = fib1 + fib2;
    fib1 = fib2;
    fib2 = fibo;
    i++;
}
return(fibo);
void main(void)
```

Figure 2.13 shows a component window without a title and headline, and with a small border.

**Figure 2.13 Component Window without Title and Headline, and with Small Border**

```c
i = 2;
while (i <= n) {
    fibo = fib1 + fib2;
    fib1 = fib2;
    fib2 = fibo;
    i++;
}
return(fibo);
void main(void)
```

Figure 2.14 shows a component window without headline and small border.
Run Menu

The Main Window Run menu, shown in Figure 2.15 is associated with the debug session. You can monitor a simulation or debug session from this menu. Run menu entries are described in Table 2.4.

Figure 2.15 Run Menu
Table 2.4 Run Menu Description

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start/Continue</td>
<td>Starts or continues execution of the loaded application from the current program counter (PC) until a breakpoint or watchpoint is reached, runtime error is detected, or user stops the application by selecting Run -&gt; Halt.</td>
</tr>
<tr>
<td></td>
<td>Shortcut: <strong>F5 key</strong></td>
</tr>
<tr>
<td>Restart</td>
<td>Starts execution of the loaded application from its entry point.</td>
</tr>
<tr>
<td></td>
<td>Shortcut: <strong>CTRL + Shift + F5 keys</strong></td>
</tr>
<tr>
<td>Halt</td>
<td>Interrupts and halts a running application. You can examine the state of each variable in the application, set breakpoints, watchpoints, and inspect source code.</td>
</tr>
<tr>
<td></td>
<td>Shortcut: <strong>F6 key</strong></td>
</tr>
<tr>
<td>Single Step</td>
<td>If the application is halted, this command performs a single step at the source level. Execution continues until the next source reference is reached. If the current statement is a procedure call, the debugger “steps into” that procedure. The Single Step command does not treat a function call as one statement, therefore it steps into the function.</td>
</tr>
<tr>
<td></td>
<td>Shortcut: <strong>F11 key</strong></td>
</tr>
<tr>
<td>Step Over</td>
<td>Similar to the Single Step command, but does not step into called functions. A function call is treated as one statement.</td>
</tr>
<tr>
<td></td>
<td>Shortcut: <strong>F10 key</strong></td>
</tr>
<tr>
<td>Step Out</td>
<td>If the application is halted inside of a function, this command continues execution and then stops at the instruction following the current function invocation. If no function calls are present, then the Step Out command is not performed.</td>
</tr>
<tr>
<td></td>
<td>Shortcut: <strong>Shift + F11 keys</strong></td>
</tr>
<tr>
<td>Assembly Step</td>
<td>If the application is halted, this command performs a single step at the assembly level. Execution continues for one CPU instruction from the point it was halted. This command is similar to the Single Step command, but executes one machine instruction rather than a high level language statement.</td>
</tr>
<tr>
<td></td>
<td>Shortcut: <strong>CTRL + F11 keys</strong></td>
</tr>
</tbody>
</table>
You can provide shortcuts for some of these functions using the toolbar. Refer to the Debugger Main Window Toolbar and Customizing the Toolbar sections for details.

You can also set breakpoints and watchpoints from within the Source and Assembly component windows.

NOTE For more information about breakpoints and watchpoints, refer to the Control Points chapter.

### Connection Menu

This menu entry (Figure 2.16) appears between the Run and Component menus when no connection is specified in the PROJECT.INI file and no connection has been set. The Connection name is replaced by an actual connection name when the connection is set. If a connection has been set, the number of menu entries is expanded, depending on the connection. To set the connection, select Component>Set Connection... Refer to the Component Menu section for details.
Figure 2.16 Connection Menu

Table 2.5 describes the Connection Menu entries.

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Loads a connection</td>
</tr>
<tr>
<td>Reset</td>
<td>Resets the current connection</td>
</tr>
</tbody>
</table>

**Loading a Connection**

Use the Connection menu to load a debugger connection.

1. Choose **Connection>Load...**

The Load Executable File window shown in Figure 2.17 is displayed:

**Load Executable File Window**

From the Connection menu, choose **Load...** to open the Load Executable File window, shown in Figure 2.17, then set the load options and choose a Simulation Execution Framework (an .ABS application file).
Debugger Interface

Debugger Main Window

Figure 2.17 Load Executable File Window

Open Button
When this button is pressed, the application code and symbols are loaded.

Load Options Buttons
These three buttons allow you to select which part of the executable file will be loaded:

- **Load Code Button**: Loads the application code only. Only the application is loaded into the target system. This button can be used if no debugging is needed.

- **Load Symbols Button**: Loads symbols only. Only debugging information is loaded. This button can be used if the code is already loaded into the target system or programmed into a non-volatile memory device (ROM/FLASH).

- **Verify Code Button**: Loader loads no data into memory. However, it reads back current data matching the same areas from the target memory and compares all data with the data from the selected file.

Open and Load Code Options Area
The checkboxes and buttons of this area of the Load Executable File window offer the following options:

- An Automatically erase and program into FLASH and EEPROM checkbox.
- A Verify memory image after loading code checkbox, with two radio buttons that let you define the memory image.
- Run after successful load checkbox.
- A Stop at Function checkbox with a textbox that lets you define the function.

Command Buttons:

- **OK**: Changes are confirmed and saved in current project file.
- **Cancel**: Closes dialog box without saving changes.
- **Help**: Opens the help file.

## Connection Command File Window

From the Connection menu, choose **Command File** to open the Connection Command File window. Each tab of this window, shown in Figure 2.18, corresponds to an event on which a command file can be automatically run from the . See the **Startup Command File**, **Reset Command File**, **Preload Command File**, and **Postload Command File** sections that follow.

![Connection Command File Window](image)

Figure 2.18 Connection Command File Window

The command file in the edit box is executed when the corresponding event occurs. Click the **Browse** button to set the path and name of the command file.
The Enable Command File check box allows you to enable/disable a command file on an event. By default, all command files are enabled:

- The default Startup command file is STARTUP.CMD,
- The default Reset command file is RESET.CMD,
- The default Preload command file is PRELOAD.CMD,
- The default Postload command file is POSTLOAD.CMD.

**NOTE** Startup settings performed in this dialog are stored for subsequent debugging sessions in the [Simulator] section of the PROJECT file using the variable CMDFILE0.

**NOTE** When a CPU is set, the settings performed in this dialog are stored for subsequent debugging sessions in the [Simulator XXX] (where XXX is the processor) section of the PROJECT file using variables CMDFILE0, CMDFILE1,... CMDFILEn.

### Startup Command File

The Startup command file is executed by the after the connection has been loaded. The Startup command file full name and status (enable/disable) can be specified either with the CMDFILE STARTUP Command Line command or using the Startup property tab of the Connection Command File Window.

By default the STARTUP.CMD file located in the current project directory is enabled as the current Startup command file.

### Reset Command File

The Reset command file is executed by the after the reset button, menu entry or Command Line command has been selected. The Reset command file full name and status (enable/disable) can be specified either with the CMDFILE RESET Command Line command or using the Reset property tab of the Connection Command File Window.

By default the RESET.CMD file located in the current project directory is enabled as the current Reset command file.

### Preload Command File

The Preload command file is executed by the before an application is loaded to the target system through the connection.
The `Preload` command file full name and status (enable/disable) can be specified either with the `CMDFILE PRELOAD` Command Line command or using the `Preload` property tab of the Connection Command File Window.

By default the `PRELOAD.CMD` file located in the current project directory is enabled as the current `Preload` command file.

**Postload Command File**

The `Postload` command file is executed by the debugger after an application has been loaded to the target system through the connection.

The `Postload` command file full name and status (enable/disable) can be specified either with the `CMDFILE POSTLOAD` Command Line command or using the `Postload` property tab of the Connection Command File Window.

By default the `POSTLOAD.CMD` file located in the current project directory is enabled as the current `Postload` command file.

**Component Menu**

The Component menu is shown in Figure 2.19, Component Menu.

Figure 2.19 Component Menu

Table 2.6 describes the Component Menu entries.
NOTE
For a readable display, we recommend using a proportional font (e.g., Courier, Terminal, etc.).

Select **Component>Open...** to load an extra component window that has not been loaded by the Debugger at startup. The popup dialog presents a set of different components that are introduced in **Debugger Components**.

Select **Component>Set Connection...** and the **Set Connection** dialog box shown in **Figure 2.20** is opened.

### Table 2.6 Component Menu Description

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Loads an extra component window that has not been loaded by the Debugger at startup. The popup dialog presents a set of different components that are introduced in the <strong>Typical Component Window Display</strong> section.</td>
</tr>
<tr>
<td>Set Connection</td>
<td>Sets the Debugger connection.</td>
</tr>
<tr>
<td>Fonts</td>
<td>Opens a standard Font Selection dialog, where you can set the font used by Debugger components.</td>
</tr>
<tr>
<td>Background Color</td>
<td>Opens a standard Color Selection dialog, where you can set the background color used by the Debugger component windows.</td>
</tr>
</tbody>
</table>
Figure 2.20 Set Connection Dialog Box

2. Use the Processor list popup to select the desired processor.
3. Use the connection list popup to select the desired connection.
   A text panel displays information about the selected connection.

**NOTE** When a connection cannot be loaded, the combo box displays the path where you should install missing dll.

4. Click OK to load connection in debugger.

**NOTE** For more information about which connection to load and how to set/reset a connection, refer to the Debugger connection books in Sections II and III of this manual.

**Window Menu**

In this menu, shown in Figure 2.21, you can set the component windows general arrangement. The Submenu Window>Options is shown in Figure 2.22 and the Submenu Window>Layout in Figure 2.23.
Table 2.7 specifies the Window Menu entries.

Table 2.7 Window Menu Description

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cascade</td>
<td>Option to arrange all open windows in cascade (so they overlap).</td>
</tr>
<tr>
<td>Tile</td>
<td>Option to display all open windows in tile format (non overlapping).</td>
</tr>
<tr>
<td>Arrange Icons</td>
<td>Arranges icons at the bottom of windows.</td>
</tr>
<tr>
<td>Options - Autosize</td>
<td>Component windows always fit into the debugger window whenever you modify the debugger window size.</td>
</tr>
<tr>
<td>Options - Component Menu</td>
<td>When a component window is selected, the associated menu is displayed in the main menu. For example if you select the Source window, the Source menu is displayed in the main menu.</td>
</tr>
<tr>
<td>Layout - Load/Store</td>
<td>Option to Load / Store your arrangements from a .HWL file.</td>
</tr>
</tbody>
</table>
NOTE Autosize and Component Menu are checked by default.

Help Menu
This is the Debugger Main window Help menu (Figure 2.24). Table 2.8 shows menu entries.

Figure 2.24 Help Menu

![Help Menu](image)

Table 2.8 Help Menu Description

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Help Topics</td>
<td>Choose Help Topics in the menu for online help or if you need specific information about a topic.</td>
</tr>
<tr>
<td>About ...</td>
<td>Information about the debugger version and copyright, and license information is displayed.</td>
</tr>
</tbody>
</table>

About Box
Select Help>about to display the About box, shown in Figure 2.25. The about box lists directories for the current project, system information, program information, version number and copyright. It contains information to send for Registration: you can copy this information and send to license@Freescale.com.
Debugger Interface
Component Associated Menus

Figure 2.25 About Box

For more information on all components, click on the Extended Information button.

Two hypertext links allow you to send an E-mail for a license request or information, and open the Freescale internet home page.

Click on OK to close this dialog box.

Component Associated Menus

Various Debugger Component windows are shown in Figure 2.3. Each component window loaded by default or that you have loaded has two menus. One menu is in the main menu and the other one is a popup menu (also called “Associated Popup Menu”) that you can open by right-clicking in a window component. Note that before right-clicking, the component window has to be active.

Component Main Menu

This menu, shown in Figure 2.26 is always between the Component entry and the Window entry of the Debugger main window toolbar. It contains general entries of the current active component. You can hide this menu by unchecking Window>Options>Component Menu.
Component Files

Each component is a windows file with a .wnd extension.

Component Windows Object Info Bar

The object info bar of the debugger window, as shown in Figure 2.27, provides information about the selected object.

Component Popup Menu

The popup menu is a dynamic context sensitive menu. It contains entries for additional facilities available in the current component. Depending on the position of the mouse in the window and what is being pointed to, popup menu entries will differ.
Debug Interface

Highlights of the User Interface

Figure 2.28 Example of Component Popup Menu

For example, if you click the mouse on a breakpoint, menu options allow you to delete, enable, or disable the breakpoint.

Highlights of the User Interface

This section describes some of the main features of the Debugger user interface.

Activating Services with Drag and Drop

You can activate services by dragging objects from one component window to another. This is known as drag and drop, an example is shown in Figure 2.29.
When the destination of a dragged item is not possible, the following cursor symbol is displayed:

Example:
You can activate the display of coverage information on assembler and C statements by dragging the chosen procedure name from the Coverage component to the Source and Assembly components (Figure 2.30).
Debugger Interface

Highlights of the User Interface

Figure 2.30 Dragging Procedure Name from Coverage to Source Component Window.

You can display the memory layout corresponding to the address held in a register by dragging the address from the Register Component to the Memory Component.

To Drag and Drop an Object

To drag an object from one component window to another:

1. Select the component containing the object you want to drag.
2. Make sure the destination component window where you want to drag the object is visible.
3. Select the object you want.
4. Pressing and holding the left mouse button, drag the object into the destination component window and then release the mouse button.
Drag and Drop Combinations

Dragging and dropping objects is possible between different component windows and are introduced in each component description section.

See below, the possible combinations of drag and drop between components and associated actions. When additional components are available, new combinations might be possible and described in the component’s information manual.

Dragging from Assembly Component Window

Table 2.9 summarizes dragging from the Assembly Component.

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Line</td>
<td>The Command Line component appends the address of the “pointed to” instruction to the current command.</td>
</tr>
<tr>
<td>Memory</td>
<td>Dumps memory starting at the selected instruction PC. The PC location is selected in the memory component.</td>
</tr>
<tr>
<td>Register</td>
<td>Loads the destination register with the PC of the selected instruction.</td>
</tr>
<tr>
<td>Source</td>
<td>Source component scrolls up to the source statements and highlights it.</td>
</tr>
</tbody>
</table>

Dragging from Data Component Window

Table 2.10 summarizes dragging from the Data Component.
Debugger Interface

Highlights of the User Interface

Table 2.10 Dragging from the Data Component Window

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Line</td>
<td>Dragging the name appends the address range of the variable to the current command in the Command Line Window. Dragging the value appends the variable value to the current command in the Command Line Window.</td>
</tr>
<tr>
<td>Memory</td>
<td>Dumps memory starting at the address where the selected variable is located. The memory area where the variable is located is selected in the memory component.</td>
</tr>
<tr>
<td>Register</td>
<td>Dragging the name loads the destination register with the address of the selected variable. Dragging the value loads the destination register with the value of the variable.</td>
</tr>
<tr>
<td>Source</td>
<td>Dragging the name of a global variable in the source Windows display the module where the variable is defined and the source text is searched for the first occurrence of the variable and highlighted.</td>
</tr>
</tbody>
</table>

NOTE It is not possible to drag an expression defined with the Expression Editor. The “forbidden” cursor is displayed.

Dragging from Source Component Window

Table 2.11 summarizes dragging from the Source Component.

Table 2.11 Dragging from the Source Component Window

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>Displays disassembled instructions starting at the first high level language instruction selected. The assembler instructions corresponding to the selected high level language instructions are highlighted in the Assembly component</td>
</tr>
<tr>
<td>Register</td>
<td>Loads the destination register with the PC of the first instruction selected.</td>
</tr>
</tbody>
</table>
Dragging from the Memory Component Window

Table 2.12  Dragging from the Memory Component Window

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>Displays disassembled instructions starting at the first address selected. Instructions corresponding to the selected memory area are highlighted in the Assembly component.</td>
</tr>
<tr>
<td>Command Line</td>
<td>Appends the selected memory range to the Command Line window</td>
</tr>
<tr>
<td>Register</td>
<td>Loads the destination register with the start address of the selected memory block.</td>
</tr>
<tr>
<td>Source</td>
<td>Displays high level language source code starting at the first address selected. Instructions corresponding to the selected memory area are greyed in the source component.</td>
</tr>
</tbody>
</table>
## Debugger Interface

*Highlights of the User Interface*

### Dragging from Procedure Component Window

Table 2.13 summarizes dragging from the Procedure Component.

**Table 2.13 Dragging from the Procedure Component Window**

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data &gt; Local</td>
<td>Displays local variables from the selected procedure in the data component</td>
</tr>
<tr>
<td>Source</td>
<td>Displays source code of the selected procedure. Current instruction inside the procedure is highlighted in the Source component.</td>
</tr>
<tr>
<td>Assembly</td>
<td>The current assembly statement inside the procedure is highlighted in the Assembly component.</td>
</tr>
</tbody>
</table>

### Dragging from Register Component Window

Table 2.14 summarizes dragging from the Register Component Window.

**Table 2.14 Dragging from the Register Component Window**

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>Assembly component receives an address range, scrolls to the corresponding instruction and highlights it.</td>
</tr>
<tr>
<td>Memory</td>
<td>Dumps memory starting at the address stored in the selected register. The corresponding address is selected in the memory component.</td>
</tr>
</tbody>
</table>
Dragging from Module Component Window

Table 2.15 summarizes dragging from the Register Component.

Table 2.15 Dragging from the Module Component Window

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data &gt; Global</td>
<td>Displays global variables from the selected module in the data component</td>
</tr>
<tr>
<td>Memory</td>
<td>Dumps memory starting at the address of the first global variable in the module. The memory area where this variable is located is selected in the memory component.</td>
</tr>
<tr>
<td>Source</td>
<td>Displays source code from selected module.</td>
</tr>
</tbody>
</table>

Selection Dialog Box

This dialog box is used in the Debugger for opening general components or source files. You can select the desired item with the arrow keys or mouse and then the OK button to accept or CANCEL to ignore your choice. The HELP button opens this section in the Help File.

This dialog box is used for the following selections:

- Set Connection
- Open IO component
- Open Source File
- Open Module
- Individual component window
Debugger Components

This chapter explains how the different components of the Debugger work. This chapter contains the following sections:

- Component Introduction
- Loading Component Windows
- General Debugger Components
- Visualization Utilities

Component Introduction

The Debugger kernel includes various components.

CPU Components

CPU components handle processor specific properties such as register naming, instruction decoding (disassembling), stack tracing, etc. A specific implementation of the CPU module has to be provided for each processor type that is supported in the debugger. The CPU related component is not introduced in this section. However, this system component is reflected in the Register component, Memory component, and all other Connection dependent components. The appropriate CPU component is automatically loaded when loading a framework (.ABS file). Therefore it is possible to mix frameworks for different MCUs. The Debugger automatically detects the MCU type and loads the appropriate CPU component, if available on your environment.

Window Components

The Debugger main window components are small applications loaded into the debugger framework at run-time. Window components can access all global facilities of the debugger engine, such as the connection (to communicate with different connections), and the symbol table. The Debugger window components are implemented as dynamic link libraries (DLLs) with extension .WND. These components are introduced in this section.
Debugger Components
Component Introduction

Connection Components
Different debugger connections are available. For example, you can set a CPU awareness to simulate your .ABS application files, and also set a background debugger.

Different connections are available to connect the target system (hardware) to the debugger. For example, the connection may be connected using a Full Chip Simulator, an Emulator, a ROM monitor, a BDM pod cable, or any other supported device.

NOTE Connection components are introduced in their respective manuals.

Loading Component Windows
In the Debugger Main Window Menu Bar, shown in Figure 3.1, you can use the Component menu to load all framework components. Each Debugger component you select will appear as a window in the Debugger main window.

Figure 3.1 Debugger Window Menu Bar

To open the window that lets you choose one or more components:
1. Choose Component>Open...
2. In the Open Window Component window shown in Figure 3.2, select the desired component.

NOTE To open more than one component, select multiple components.
3. In the Open Window Component window, use the mouse to select a component.
4. Click the OK button to open the selected component.

There are three tabs in the Open Window Component window:
- The Icon tab shows components with large icons.
- The List tab shows components with small icons.
- The Details tab shows components with their description.
Multiple Component Windows
If you load a project that targets both HC12 and XGATE cores, the Debugger shows two component windows as follows:

- One Assembly window for the HC12 source code and one assembly window for the XGATE source code.
- One Data window for the HC12 portion of the application and one Data window for the XGATE portion of the application.
- One Procedure window for the HC12 call chain and one Procedure window for the XGATE call chain.
- One Register window for the HC12 core and one Register window for the XGATE core.
- One Source window for the HC12 source code and one Source window for the XGATE source code.

General Debugger Components
This chapter describes the various features and usage of the debugger components.
Assembly Component

The Assembly window, shown in Figure 3.3, displays program code in disassembled form. It has a function very similar to that of the Source component window but on a much lower abstraction level. Thus it is therefore possible to view, change, monitor and control the current location of execution in a program.

Figure 3.3 Assembly Window

This window contains all on-line disassembled instructions generated by the loaded application. Each displayed disassembled line in the window can show the following information: the address, machine code, instruction and absolute address in case of a branch instruction. By default, the user can see the instruction and absolute address.

If breakpoints have been set in the application, they are marked in the Assembly component with a special symbol, depending on the kind of breakpoint.

If execution has stopped, the current position is marked in the Assembly component by highlighting the corresponding instruction.

The Object Info Bar of the component window contains the procedure name, which contains the currently selected instruction. When a procedure is double clicked in the Procedure component, the current assembly statement inside this procedure is highlighted in the Assembly component.

Assembly Menu

The Assembly menu shown in Figure 3.4 contains all functions associated with the assembly component. Theses entries are described in Table 3.1.
Debugger Components
General Debugger Components

Figure 3.4 Assembly Menu

Table 3.1 Assembly Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address...</td>
<td>Opens a dialog box prompting for an address: Show PC.</td>
</tr>
<tr>
<td>Display Code</td>
<td>Displays machine code in front of each disassembled instruction.</td>
</tr>
<tr>
<td>Display Symbolic</td>
<td>Displays symbolic names of objects.</td>
</tr>
<tr>
<td>Display Address</td>
<td>Displays the location address at the beginning of each disassembled instruction.</td>
</tr>
<tr>
<td>Display Absolute Address</td>
<td>In a branch instruction, displays the absolute address at the end of the disassembled instruction.</td>
</tr>
</tbody>
</table>

Setting Breakpoints

Breakpoints can be set, edited and deleted when using the popup menu. Right-click on any statement in the Source component window, then choose Set Breakpoint, Delete Breakpoint, etc.

NOTE For information on using breakpoints, see Control Points chapter.
Show PC Dialog Box
If a hexadecimal address is entered in the Show PC dialog box shown in Figure 3.5, memory contents are interpreted and displayed as assembler instructions starting at the specified address.

Figure 3.5 Show PC Dialog Box

Associated Popup Menu
To open the popup menu right-click in the text area of the Assembly component window. The popup menu contains default menu entries for the Assembly component. It also contains some context dependent menu entries described in Table 3.2; depending on the current state of the debugger.

Figure 3.6 Assembly Popup Menu

Set Breakpoint
Run To Cursor
Show Breakpoints…
Show Location

Set Markpoint
Show Markpoints…
Address…

Display
Format
Freeze
## Table 3.2 Assembly Popup Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Breakpoint</td>
<td>Appears only in the popup menu if no breakpoint is set or disabled on the pointed to instruction. When selected, sets a permanent breakpoint on this instruction. When program execution reaches this instruction, the program is halted and the current program state is displayed in all window components.</td>
</tr>
<tr>
<td>Delete Breakpoint</td>
<td>Appears in popup menu if a breakpoint is set or disabled on the specified instruction. When selected, deletes this breakpoint.</td>
</tr>
<tr>
<td>Enable Breakpoint</td>
<td>Appears only in popup menu if a breakpoint is disabled on an instruction. When selected, enables this breakpoint.</td>
</tr>
<tr>
<td>Disable Breakpoint</td>
<td>Appears in the popup menu if a breakpoint is set on an instruction. When selected, disables this breakpoint.</td>
</tr>
<tr>
<td>Run To Cursor</td>
<td>When selected, sets a temporary breakpoint on a specified instruction and continues execution of the program. If there is a disabled breakpoint at this position, the temporary breakpoint will also be disabled and the program will not halt. Temporary breakpoints are automatically removed when they are reached.</td>
</tr>
<tr>
<td>Show Breakpoints</td>
<td>Opens the Controlpoints Configuration Window Breakpoints Tab and displays list of breakpoints defined in the application (refer to Control Points).</td>
</tr>
<tr>
<td>Show Location</td>
<td>When selected, highlights the source statement that generated the pointed to assembler instruction. The assembler instruction is also highlighted. The memory range corresponding to this assembler instruction is also highlighted in the memory component.</td>
</tr>
<tr>
<td>Set Markpoint</td>
<td>When selected, enables you to set a markpoint at this location.</td>
</tr>
<tr>
<td>Delete Markpoint</td>
<td>Appears in the Popup Menu only if a markpoint is set at the nearest code position (visible with marks). When selected, disables this markpoint.</td>
</tr>
<tr>
<td>Show Markpoints</td>
<td>Opens the Controlpoints Configuration Window Markpoints Tab and displays list of markpoints defined in the application (refer to Control Points).</td>
</tr>
<tr>
<td>Address...</td>
<td>For a description of the remaining popup menu entries see Table 3.1 &quot;Assembly Menu Description&quot;.</td>
</tr>
</tbody>
</table>
Retrieving Source Statement

- Point to an instruction in the Assembly component window, drag and drop it into the Source component window. The Source component window scrolls to the source statement generating this assembly instruction and highlights it.
- Left clicking the mouse and clicking the L key Highlights a code range in the Assembly component window corresponding to the first line of code selected in the Source component window where the operation is performed. This line or code range is also highlighted.

Drag Out:

Table 3.3 shows the drag actions possible from the Assembly component.

Table 3.3 Assembly Component Drag Actions

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Line</td>
<td>The Command Line component appends the address of the pointed to instruction to the current command.</td>
</tr>
<tr>
<td>Memory</td>
<td>Dumps memory starting at the selected instruction PC. The PC location is selected in the memory component.</td>
</tr>
<tr>
<td>Register</td>
<td>Loads the destination register with the PC of the selected instruction.</td>
</tr>
<tr>
<td>Source</td>
<td>Source component scrolls to the source statements and highlights it.</td>
</tr>
</tbody>
</table>
Drop Into:

Table 3.4 shows the drop actions possible in the Assembly component

<table>
<thead>
<tr>
<th>Source Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Displays disassembled instructions starting at the first high level language instruction selected. The assembler instructions corresponding to the selected high level language instructions are highlighted in the Assembly component.</td>
</tr>
<tr>
<td>Memory</td>
<td>Displays disassembled instructions starting at the first address selected. Instructions corresponding to the selected memory area are highlighted in the Assembly component.</td>
</tr>
<tr>
<td>Register</td>
<td>Displays disassembled instructions starting at the address stored in the source register. The instruction starting at the address stored in the register is highlighted.</td>
</tr>
<tr>
<td>Procedure</td>
<td>The current assembly statement inside the procedure is highlighted in the Assembly component.</td>
</tr>
</tbody>
</table>

Demo Version Limitations

No limitation

Associated Commands

Following commands are associated with the Assembly component:

ATTRIBUTES, SMEM, SPC

Command Line Component

The Command Line window shown in Figure 3.7 interprets and executes all Debugger commands and functions. The command entry always occurs in the last line of the Command component. Characters can be input or pasted on the edit line.
Keying In Commands
You can type Debugger commands after the “in>” terminal prompt in the Command Line Component window.

Recalling a Line from the Command Line History
To recall a command in the DOS window use either the up or down arrow, of the F3 function key, to retype the previous command.

Scrolling the Command Component Window Content
Use the left and right arrow keys to move the cursor on the line, the HOME key to move the cursor to the beginning of the line, or the END key to move the cursor to the end of the line. To scroll a page, use the PgDn (scroll down a page) or PgUp keys (scroll up a page).

Clearing the Line or a Character of the Command Line
Selected text can be deleted by pressing the left arrow. To clear the current line, press the ESC key.

Command Interpretation
The component executes the command entered, displays results or error messages, if any. Ten previous commands can be recalled using the up arrow key to scroll up or the down arrow key to scroll down. Commands are displayed in blue. Prompts and command responses are displayed in black. Error messages are displayed in red.

When a command is executed and running from the Command Line component, the component cannot be closed. In this case, if the Command Line component is closed with the window close button (X) or with the Close entry of the system menu, the following message is displayed:

“Command Component is busy. Closing will be delayed”

The Command Line component is closed as soon as command execution is complete. If the CLOSE command is applied to this Command Line component (for example, from...
another Command Line component), the component is closed as soon as command execution is finished.

**Variable Checking in the Command Line**

When specifying a single name as an expression in the command line, this expression is first checked as a local variable in the current procedure. If not found, it is checked as a global variable in the current module. If not found, it is checked as a global variable in the application. If not found, it is checked as a function in the current module. If not found, it is checked as a function in the application, finally if not found an error is generated.

Closing the Command Line during an execution

When a command is executed from a Command Line component, it cannot be closed. If the Command Line component is closed with the close button or with the 'Close' entry of the system menu, the following message is displayed ‘Command Component is busy. Closing will be delayed’ and the Command component is closed as soon as command execution is complete. If the 'Close' command is applied to this Command component, the Command component is closed as soon as command execution is complete.

**Command Menu**

Figure 3.8 shows the Command menu, which is identical to the Command Popup menu.

![Figure 3.8 Command Menu](image)

Clicking **Execute File** opens a dialog where you can select a file containing Debugger commands to be executed. Theses files generally have a **.cmd** default extension.

Selected text in the Command Line window can be copied to the clipboard by:

- selecting the menu entry **Command>Copy**.
- pressing the CTRL + C key.
- clicking the button in the toolbar.

The **Command>Copy** menu entry and the button are only enabled if something is selected in the Command Line window.
The first line of text contained in the clipboard can be pasted where the caret is blinking (end of current line) by:

- Selecting the menu entry **Command>Paste**
- Pressing CTRL + V simultaneously.
- Clicking the icon in the toolbar.

### Cache Size

Select **Cache Size** in the menu to bring up the Size of the Cache dialog box and set the cache size in lines for the Command Line window, as shown in Figure 3.9.

**Figure 3.9 Cache Size Dialog Box**

![Cache Size Dialog Box](image)

This Cache Size dialog box is the same for the Terminal Component and the TestTerm Component.

**Drag Out:**
Nothing can be dragged out.

**Drop Into:**
Memory range, address, and value can be dropped into the Command Line Component window, as described in Table 3.5. The command line component appends corresponding items of the current command.
Debugger Components

General Debugger Components

Demo Version Limitations

Only 20 commands can be entered and then command component is closed and it is no longer possible to open a new one in the same Debugger session.

Command files with more than 20 commands cannot be executed.

Associated Commands

BD, CF, E, HELP, NB, LS, SREC, SAVE.

NOTE For more details about commands, refer to Debugger Engine Commands.

Coverage Component

The Coverage window, shown in Figure 3.10 contains source modules and procedure names as well as percentage values representing the proportion of executed code in a given source module or procedure.

Please note that in cases where in cases of advanced code optimizations (like linker overlapping ROM/code areas) the coverage output/data is affected. In such a case, it is recommended to switch of such linker optimizations.

Table 3.5 Drop Into Command Component

<table>
<thead>
<tr>
<th>Source Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>The Command Line component appends the address of the pointed to instruction to the current command.</td>
</tr>
<tr>
<td>Data</td>
<td>Dragging the name appends the address range of the variable to the current command in the Command Line Window. Dragging the value appends the variable value to the current command in the Command Line Window.</td>
</tr>
<tr>
<td>Memory</td>
<td>Appends the selected memory range to the Command Line window</td>
</tr>
<tr>
<td>Register</td>
<td>The address stored in the pointed to register is appended to the current command.</td>
</tr>
</tbody>
</table>
The Coverage window contains percentage numbers and graphic bars. From this component, you can split views in the Source window and Assembly window, as shown in Figure 3.11. A red check mark is displayed in front of each source or assembler instruction that has been executed. Split views are removed when the Coverage window is closed or by selecting Delete in the split view popup menu.

Coverage Operations
Click the folded/unfolded icons to unfold/fold the source module and display/hide the functions defined.

Coverage Menu
The Coverage menu and submenus are shown in Figure 3.12.
Debugger Components
General Debugger Components

Figure 3.12 Coverage Menu

Table 3.6 Coverage Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Resets all simulator statistic information.</td>
</tr>
<tr>
<td>Details</td>
<td>Opens a split view in the chosen component (Source or Assembly).</td>
</tr>
<tr>
<td>Graphics</td>
<td>Toggles the graphic bars.</td>
</tr>
<tr>
<td>Timer Update</td>
<td>Switches the periodic update on/off. If activated, statistics are updated each second.</td>
</tr>
<tr>
<td>Output File</td>
<td>Opens the Output File options.</td>
</tr>
</tbody>
</table>

Output File

You can redirect Coverage component results to an output file by selecting Output File... > Save As... in the menu or popup menu.

Output File Filter

Select Output Filter... to display the dialog box shown in Figure 3.13. Select what you want to display, i.e. modules only, modules and functions, or modules, functions and code lines. You can also specify a range of coverage to be logged in your file.
Debugger Components
General Debugger Components

Figure 3.13 Output File Filter Dialog Box

Output File Save
The Save As... entry opens a Save As dialog where you can specify the output file name and location, an example is shown in Listing 3.1.

Listing 3.1 Example Output File with Modules and Functions:

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>94.4 %</td>
<td>Application</td>
</tr>
<tr>
<td>FULL</td>
<td>fibo.c</td>
</tr>
<tr>
<td>FULL</td>
<td>Fibonacci()</td>
</tr>
<tr>
<td>FULL</td>
<td>main()</td>
</tr>
<tr>
<td>86.0 %</td>
<td>startup.c</td>
</tr>
<tr>
<td>80.5 %</td>
<td>Init()</td>
</tr>
<tr>
<td>FULL</td>
<td>_Startup()</td>
</tr>
</tbody>
</table>

Split View Associated Popup Menu
The popup menu for the split view (Figure 3.14) contains the Delete entry, which is used to remove the split view.
Debugger Components
General Debugger Components

Figure 3.14 Coverage Split View Associated Popup Menu

Drag Out:
All displayed items can be dragged into a Source or Assembly component. Destination component displays marks in front of the executed source or assembler instruction.

Drop Into:
Nothing can be dropped into the Coverage Component window.

Demo Version Limitations
Only modules are displayed and the Save function is disabled.

Associated Commands
DETAILS, FILTER, GRAPHICS, OUTPUT, RESET, TUPDATE
Debugger Components
General Debugger Components

DA-C Link Component
The DA-C Link window shown in Figure 3.15 is an interface module between the DA-C (Development Assistant for C - from RistanCASE GmbH) and the IDE, allowing synchronized debugging features.

Figure 3.15 DA-C Link Window

DA-C Link Operation
When you load the DA-C Link component, communication is established with DA-C (if open) in order to exchange synchronization information.
The Setup entry of the DA-C Link main menu allows you to define the connection parameters.

NOTE For related information refer to the Chapter Synchronized Debugging Through DA-C IDE.

DA-C Link Menu
Selecting Setup from the DA-C Link menu opens the Connection Specification dialog box.

Figure 3.16 DA-C Link Menu

Table 3.7 DA-C Link Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Opens the Connection Specification dialog box.</td>
</tr>
</tbody>
</table>

Connection Specification Dialog Box
In the Connection Specification dialog box you can set the DA-C debugger name.
The DA-C debugger name must be the same as the one selected in the DA-C IDE. Check the “Show Protocol” checkbox to display the communication protocol in the Command component of the Debugger. To validate the settings, click the **OK** button. A new connection is established and the "Connection Specification" is saved in the current Project.ini file. The **HELP** button opens the help topic for this dialog.

**NOTE** If problems exist, refer to the [Troubleshooting] section in the DA-C documentation.

### Drag Out

Nothing can be dragged out.

### Drop Into

Nothing can be dropped into the DAC Component window.

### Demo Version Limitations

None.
Data Component

The Data window shown in Figure 3.18 contains the names, values and types of global or local variables.

Figure 3.18 Data Window

<table>
<thead>
<tr>
<th>Address: 8030</th>
<th>Size: 24</th>
<th>startup.c</th>
<th>Auto</th>
<th>Symb</th>
<th>Global</th>
</tr>
</thead>
<tbody>
<tr>
<td>$startupData$</td>
<td>&lt;24&gt;</td>
<td>$tagStartup$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>unsigned int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>0x8064 _Func</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stackOffset</td>
<td>8190 unsigned int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nofZeroOut</td>
<td>1 unsigned int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pZeroOut</td>
<td>0x804f * _Range</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*pZeroOut</td>
<td>&lt;4&gt; _Range</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toCopyDownBox</td>
<td>0x80ae * _Copy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nofLibInits</td>
<td>32851 unsigned int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Data window shows all variables present in the current source module or procedure. Changed values are in red.

The Component Windows Object Info Bar contains the address and size of the selected variable. It also contains the module name or procedure name where the displayed variables are defined, the display mode (automatic, locked, etc.), the display format (symbolic, hex, bin, etc.), and current scope (global, local or user variables).

Various display formats, such as symbolic representation (depending on variable types), and hexadecimal, octal, binary, signed and unsigned formats may be selected.

Structures can be expanded to display their member fields.

Pointers can be traversed to display data they are pointing to.

Watchpoints can be set in this component. Refer to Control Points chapter.

Data Operations

- Double-click a variable line to edit the value.
- Click the folded/unfolded icons [ ] to unfold/fold the structured variable.
- Double-click a blank line: Opens the Expression editor to insert an expression in the Data Component window.
- Select a variable in the Data component, and left mouse button + R key to set a “Read” watchpoint on the selected variable. A green vertical bar is displayed on the...
left side of the variables on which a read watchpoint has been defined. If a read access on the variable is detected during execution, the program is halted and the current program state is displayed in all window components.

- Select a variable in the Data component, and left mouse button + W key to set a “Write” watchpoint on the selected variable. A red vertical bar is displayed on the left side of the variables on which a write watchpoint has been defined. If write access is detected on the variable during execution, the program is halted and the current program state is displayed in all window components.

- Select a variable in the Data component, and left mouse button + B key to set a “Read/Write” watchpoint on the selected variable. A yellow vertical bar is displayed for the variables on which a read/write watchpoint has been defined. If the variable is accessed during execution, the program is halted and the current program state is displayed in all window components.

- Select a variable on which a watchpoint was previously defined in the Data component, and left mouse button + D key to delete the watchpoint on the selected variable. The vertical bar previously displayed for the variables is removed.

- Select a variable in the Data component, and left mouse button + S key to set a watchpoint on the selected variable. The Watchpoints Setting dialog box is opened. A grey vertical bar is displayed for the variables on which an watchpoint has been defined.

**Expression Editor**

To add your own expression (in EBNF notation) double-click a blank line in the Data component window to open the Edit Expression dialog box shown in Figure 3.19, or point to a blank line as shown below and right-click to select Add Expression... in the popup menu shown in the figure below.

You may enter a logical or numerical expression in the edit box, using the Ansi-C syntax. In general, this expression is a function of one or several variables from the current Data component window.
**Example:**

With 2 variables `variable_1, variable_2`:

- expression entered: `(variable_1<<variable_2)+ 0xFF) <= 0x1000` results in a boolean type.
- expression entered: `(variable_1>>~variable_2)* 0x1000` will result in an integer type.

**NOTE**  
It is not possible to drag an expression defined with the Expression Editor. The “forbidden” cursor is displayed.

**Expression Command file**

The Expression Command file is automatically generated when a new application is loaded or exiting from the Debugger. User defined expressions are stored in this command file. The name of the expression command file is the name of the application with a `.xpr` extension (.XPR file). When loading a new user application, the debugger executes the matching expression command file to load the user defined expression into the data component.

**Example:**  
When loading `fibo.abs`, the debugger executes `Fibo.xpr`
Debugger Components
General Debugger Components

Data Menu

Figure 3.20 shows the Data component menu, the Zoom submenu is shown in Figure 3.29, the Scope submenu is shown in Figure 3.21, the Format submenu in Figure 3.22, the Mode submenu in Figure 3.24, the Options submenu in Figure 3.26 and the Zoom and Sort submenus in Figure 3.29. Data Menu entries are described in Table 3.8.

Figure 3.20 Data Menu

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoom</td>
<td>Zooms in or out of the selected structure. The member field of the structure replaces the variable list.</td>
</tr>
<tr>
<td>Scope...</td>
<td>Opens a variable display submenu.</td>
</tr>
<tr>
<td>Format...</td>
<td>Symb, Hex (hexadecimal), Oct (octal), Bin (binary), Dec (signed decimal), UDec (unsigned decimal) display format.</td>
</tr>
<tr>
<td>Mode...</td>
<td>Switches between Automatic, Periodical, Locked, and Frozen update mode.</td>
</tr>
<tr>
<td>Options...</td>
<td>Opens an options menu for data, for example, Pointer as Array facility.</td>
</tr>
<tr>
<td>Sort...</td>
<td>Opens a Sort Submenu from which you select criteria by which data can be sorted.</td>
</tr>
</tbody>
</table>

Scope Submenu

The Scope Submenu is activated by highlighting the Scope entry on the Data menu:
Debugger Components

General Debugger Components

Figure 3.21 Scope Submenu

Table 3.9 describes the Scope submenu entries.

Table 3.9 Scope Submenu Entries

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Switches to <strong>Global</strong> variable display in the Data component.</td>
</tr>
<tr>
<td>Local</td>
<td>Switches to <strong>Local</strong> variable display in the Data component.</td>
</tr>
<tr>
<td>User</td>
<td>Switches to <strong>User</strong> variable display in the Data component. Displays user defined expression (variables are erased).</td>
</tr>
</tbody>
</table>

**NOTE** If the data component mode is not automatic, entries are greyed (because it is not allowed to change the scope).

In Local Scope, if the Data component is in Locked or Periodical mode, values of the displayed local variables could be invalid (since these variables are no longer defined in the stack).

**Format Submenu**

The Format Submenu is activated by highlighting the format entry on the Data menu:

Figure 3.22 Format Submenu

Table 3.10 describes the Format submenu entries.

Table 3.10 Format Sub Menu Entries

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected</td>
<td>The changes will be applied to the selection only</td>
</tr>
<tr>
<td>All</td>
<td>The changes will be applied to all items</td>
</tr>
</tbody>
</table>
Debugger Components
General Debugger Components

Format Selected & All Sub Menu

The Format Selected & All Submenu is activated by highlighting this entry on the Data Component menu:

![Figure 3.23 Format Selected & All Submenus](image)

Table 3.11 describes the Format Selected Mode & Format All Mode Sub Menu entries.

### Table 3.11 Format Selected & All Sub Menu

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic</td>
<td>Select the <strong>Symbolic</strong> (display format depends on the variable type) display format. This is the default display.</td>
</tr>
<tr>
<td>Hex</td>
<td>Select the hexadecimal data display format</td>
</tr>
<tr>
<td>Bin</td>
<td>Select the binary data display format</td>
</tr>
<tr>
<td>Oct</td>
<td>Select the octal data display format</td>
</tr>
<tr>
<td>Dec</td>
<td>Select the signed decimal data display format</td>
</tr>
<tr>
<td>UDec</td>
<td>Select the unsigned decimal data display format</td>
</tr>
<tr>
<td>Bit Reverse</td>
<td>Select the bit reverse data display format (Each bit is reversed).</td>
</tr>
</tbody>
</table>

Mode Submenu

The Mode Submenu is activated by highlighting the **Mode** entry on the Data menu:
Figure 3.24 Mode Submenu

Table 3.12 Mode Submenu

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>Switches to <strong>Automatic</strong> mode (default), variables are updated when the connection is stopped. Variables from the currently executed module or procedure are displayed in the data component.</td>
</tr>
<tr>
<td>Periodical</td>
<td>Switches to <strong>Periodical</strong> mode: variables are updated at regular time intervals when the connection is running. The default update rate is 1 second, but can be modified by steps of up to 100 ms using the associated dialog box (see below).</td>
</tr>
<tr>
<td>Locked</td>
<td>Switches to <strong>Locked</strong> mode, value from variables displayed in the data component are updated when the connection is stopped.</td>
</tr>
<tr>
<td>Frozen</td>
<td>Switches to <strong>Frozen</strong> mode: value from variables displayed in the data component are not updated when the connection is stopped.</td>
</tr>
</tbody>
</table>

**NOTE** In Locked and Frozen mode, variables from a specific module are displayed in the data component. The same variables are always displayed in the data component.

Update Rate Dialog Box
The Update Rate dialog box shown in Figure 3.25 allows you to modify the default update rate using steps of 100 ms.
Options Submenu

The Options Submenu is activated by highlighting the Options entry on the Data menu:

Figure 3.26 Options Submenu

- Pointer as Array...
- Name Width...

Pointer as Array Option

In the Data menu’s Options submenu, choose Options...>Pointer as Array... to open the dialog box shown in Figure 3.27.

Figure 3.27 Pointer as Array Dialog Box

Within this dialog box, you can display pointers as arrays, assuming that the pointer points to the first item (pointer[0]). Note that this setup is valid for all pointers displayed in the Data window. Check the Display Pointer as Array checkbox and set the number of items that you want to be displayed as array items.

Name Width Option

In the Data Menu’s Options submenu, choose Options... > Name Width... to open the dialog box shown in Figure 3.28.
Figure 3.28  Edit Name Width Dialog Box

This dialog box allows you to adjust the width of the variable name displayed in the Data window. This string will be cut off if it is longer than 16 characters. Thus, by enlarging the value you can adapt the window to longer names.
Debugger Components
General Debugger Components

Zoom and Sort Submenus

Figure 3.29  Zoom and Sort Submenus

Associated Popup Menu

Figure 3.30  Data Popup Menu

Table 3.13 specifies the Data Popup Menu entries.
Table 3.13 Data Popup Menu

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Module...</td>
<td>Opens the Open Module dialog box.</td>
</tr>
<tr>
<td>Set Watchpoint</td>
<td>Appears only in the popup menu if no watchpoint is set or disabled on the pointed to variable. When selected, sets a read/write watchpoint on this variable. A yellow vertical bar is displayed for the variables on which a read/write watchpoint has been defined. If the variable is accessed during execution, the program is halted and the current program state is displayed in all window components.</td>
</tr>
<tr>
<td>Delete Watchpoint</td>
<td>Appears only in the popup menu if a watchpoint is set or disabled on the pointed to variable. When selected, deletes this watchpoint.</td>
</tr>
<tr>
<td>Enable Watchpoint</td>
<td>Appears only in the popup menu if a watchpoint is disabled on the pointed to variable. When selected, enables this watchpoint.</td>
</tr>
<tr>
<td>Disable Breakpoint</td>
<td>Appears only in the popup menu if a breakpoint is set on the pointed to instruction. When selected, disables this watchpoint.</td>
</tr>
<tr>
<td>Show Watchpoints</td>
<td>Opens the Watchpoints Setting dialog box and allows you to view the list of watchpoints defined in the application. (Refer to Control Points).</td>
</tr>
<tr>
<td>Show location</td>
<td>Forces all open components to display information about the pointed to variable (e.g., the Memory component selects the memory range where the variable is located).</td>
</tr>
</tbody>
</table>

**SUBMENU Open Module**

The dialog shown in Figure 3.31 lists all source files bound to the application. Global variables from the selected module are displayed in the data component. This is only supported when the component is in Global scope mode.
Drag Out:
Table 3.14 describes the drag actions possible from the Data component.

Table 3.14 Dragging Data Possibilities

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command Line</td>
<td>Dragging the name appends the address of the variable to the current command in the Command Line Window. Dragging the value appends the variable value to the current command in the Command Line Window.</td>
</tr>
<tr>
<td>Memory</td>
<td>Dumps memory starting at the address where the selected variable is located. The memory area where the variable is located is selected in the memory component.</td>
</tr>
<tr>
<td>Source</td>
<td>Dragging the name of a global variable in the source Window displays the module where the variable is defined and first occurrence of the variable is highlighted.</td>
</tr>
<tr>
<td>Register</td>
<td>Dragging the name loads the destination register with the address of the selected variable. Dragging the value loads the destination register with the value of the variable.</td>
</tr>
</tbody>
</table>
NOTE It is important to distinguish between dragging a variable name and dragging a variable value. Both operations are possible. Dragging the name drags the address of the variable. Dragging the variable value drags the value.

NOTE Expressions are evaluated at run time. They do not have a location address, so you cannot drag an expression name into another component. Values of expressions can be dragged to other components.

Drop Into:
Table 3.15 describes the drop actions possible in the Data component.

Table 3.15 Data Drop Possibilities

<table>
<thead>
<tr>
<th>Source Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>A selection in the Source window is considered an expression in the Data window, as if it was entered through the Expression Editor of the Data component. Refer to Data Component, Expression Editor.</td>
</tr>
<tr>
<td>Module</td>
<td>Displays the global variables from the selected module in the data component.</td>
</tr>
</tbody>
</table>

Demo Version Limitations
Only 2 variables can be displayed.
Only 2 members of a structure are visible when unfolded.
Only 1 expression can be defined.

Associated Commands
ADDXPR, ATTRIBUTES, DUMP, PTRARRAY, SMOD, SPROC, UPDATERATE, ZOOM.
Memory Component

The Memory window shown in Figure 3.32 displays unstructured memory content or memory dump, i.e. continuous memory words without distinction between variables.

Figure 3.32 Memory Window

Various data formats (byte, word, double) and data displays (hexadecimal, binary, octal, decimal, unsigned decimal) can be specified for the display and edition of memory content.

Watchpoints can be defined in this component.

**NOTE** Refer to Control Points for more information about watchpoints.

Memory areas can be initialized with a fill pattern using the Fill Memory box.

An ASCII dump can be added/removed on the right side of the numerical dump when checking/unchecking ASCII in the Display menu entry.

The location address may also be added/removed on the left side of the numerical dump when checking/unchecking Address in the Display menu entry.

To specify the start address for the memory dump use the Address menu entry.

The Component Windows Object Info Bar contains the procedure or variable name, structure field and memory range matching the first selected memory word.

"uu" memory value means: not initialized.

"--" memory values mean: not configured (no memory available)

**NOTE** Memory values that have changed since the last refresh status are displayed in red. However, if a memory item is edited or rewritten with the same value, the display for this memory item remains black.
Memory Operations

- Double-click a memory position to edit it. If the memory is not initialized, this operation is not possible.
- Drag the mouse in the memory dump to select a memory range.
- Hold down the left mouse button + A key to jump to a memory address. The pointed to value is interpreted as an address and the memory component dumps memory starting at this address.
- Select a memory range, and hold down the left mouse button + R key to set a “Read” watchpoint for the selected memory area. Memory ranges where a read watchpoint has been defined are underlined in green. If read access on the memory area is detected during execution, the program is halted and the current program state is displayed in all window components.
- Select a memory range, and hold down the left mouse button + W key to set a “Write” watchpoint on the selected memory area. Memory ranges where a write watchpoint has been defined are underlined in red. If write access on the memory area is detected during execution, the program is halted and the current program state is displayed in all window components.
- Select a memory range, and hold down the left mouse button + B key to set a “Read/Write” watchpoint on the selected memory area. Memory ranges where a read/write watchpoint has been defined are underlined in black. If the memory area is exceeded during execution, the program is halted and the current program state is displayed in all window components.
- Select a memory range on which a watchpoint was previously defined, and hold down the left mouse button + D key to delete the watchpoint on the selected memory area. The memory area is no longer underlined.
- Select a memory range, and hold down the left mouse button + S key to set a watchpoint on the selected memory area. The Watchpoints Setting dialog box is opened. Memory ranges where a watchpoint has been defined are underlined in black.

Memory Menu

The Memory Menu shown in Figure 3.33 provides access to memory commands. Table 3.16 describes the menu entries.
Figure 3.33 Memory Menu

Table 3.16 Memory Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word size</td>
<td>Opens a submenu to specify the display unit size.</td>
</tr>
<tr>
<td>Format</td>
<td>Opens a submenu to select the format to display items.</td>
</tr>
<tr>
<td>Mode</td>
<td>Opens a submenu to choose the update mode.</td>
</tr>
<tr>
<td>Display</td>
<td>Opens a submenu to toggle the display of addresses and ASCII dump.</td>
</tr>
<tr>
<td>Fill...</td>
<td>Opens the Fill Memory to fill a memory range with a bit pattern.</td>
</tr>
<tr>
<td>Address...</td>
<td>Opens the memory dialog and prompts for an address.</td>
</tr>
<tr>
<td>CopyMem</td>
<td>Opens the CopyMem dialog box that allows you to copy memory range values to a specific location.</td>
</tr>
<tr>
<td>Search Pattern</td>
<td>Opens the Search Pattern dialog box.</td>
</tr>
</tbody>
</table>

Word Size Submenu

With the Word Size submenu shown in Figure 3.34, you can set the memory display unit. Table 3.17 describes the menu entries.
Figure 3.34  Word Size Submenu

Table 3.17  Word Size Submenu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>Sets display unit to byte size.</td>
</tr>
<tr>
<td>Word</td>
<td>Sets display unit to word size (=2 bytes).</td>
</tr>
<tr>
<td>LWord</td>
<td>Sets display unit to Lword size (=4 bytes).</td>
</tr>
</tbody>
</table>

Format Submenu

With the Format Submenu shown in Figure 3.35, you can set the memory display format. Table 3.18 describes the menu entries.

Figure 3.35  Format Submenu

Table 3.18  Format Submenu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>Selects the hexadecimal memory display format</td>
</tr>
<tr>
<td>Bin</td>
<td>Selects the binary memory display format</td>
</tr>
<tr>
<td>Oct</td>
<td>Selects the octal memory display format</td>
</tr>
<tr>
<td>Dec</td>
<td>Selects the signed decimal memory display format</td>
</tr>
<tr>
<td>BitReverse</td>
<td></td>
</tr>
</tbody>
</table>
### Debugger Components

#### General Debugger Components

**Table 3.18 Format Submenu Description (continued)**

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDec</td>
<td>Selects the unsigned decimal memory display format</td>
</tr>
<tr>
<td>Bit Reverse</td>
<td>Selects the bit reverse memory display format (each bit is reversed).</td>
</tr>
</tbody>
</table>

**Mode Submenu**

With the Mode submenu shown in Figure 3.36, you can set the memory mode format. Table 3.19 describes the menu entries.

**Figure 3.36 Mode Submenu**

- ✔ Automatic
- ✔ Periodical...
- ✔ Frozen

**Table 3.19 Mode Submenu Description**

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic</td>
<td>Selects <strong>Automatic</strong> mode (default), memory dump is updated when the connection is stopped.</td>
</tr>
<tr>
<td>Periodical</td>
<td>Selects the <strong>Periodical</strong> mode, memory dump is updated at regular time intervals when the connection is running. The default update rate is 1 second, but it can be modified by steps of up to 100 ms using the associated dialog box (see below).</td>
</tr>
<tr>
<td>Frozen</td>
<td>Selects the <strong>Frozen</strong> mode, memory dump displayed in the memory component is not updated when the connection is stopped.</td>
</tr>
</tbody>
</table>

**Display Submenu**

With the Display submenu shown in Figure 3.37, you can set the memory display (address/ascii). Table 3.20 describes the menu entries.

**Figure 3.37 Display Submenu**

- ✔ Address
- ✔ ASCII
Fill Memory
The Fill Memory dialog box shown in Figure 3.38 allows you to fill a memory range (from Address edit box and to Address edit box) with a bit pattern (value edit box).

Figure 3.38 Fill Memory Dialog Box

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Allows you to toggle the display of address dump.</td>
</tr>
<tr>
<td>ASCII</td>
<td>Allows you to toggle the display of ASCII dump.</td>
</tr>
</tbody>
</table>

NOTE If “Hex Format” is checked, numbers and letters are interpreted as hexadecimal numbers. Otherwise, expressions can be typed and Hex numbers should be prefixed with “Ox” or “$”.

Display Address
With the Display Address dialog box, shown in Figure 3.39, the memory component dumps memory starting at the specified address.
Figure 3.39 Display Address Dialog Box

NOTE The Show PC dialog box is the same as the Display Address dialog box. In this dialog box, the Assembly component dumps assembly code starting at the specified address.
CopyMem Submenu

The CopyMem dialog box shown in Figure 3.40 allows you to copy a memory range to a specific address.

Figure 3.40 CopyMem Dialog Box

To copy a memory range to a specific address, enter the source range and the destination address. Press the OK button to copy the specified memory range. Press the Cancel button to close the dialog without changes. Press the Help button to open the help file associated with this dialog.

If "Hex Format" is checked, all given values are in Hexadecimal Format. You don't need to add "0x". For instance type 1000 instead of 0x1000.

NOTE If you try to read or write to an unauthorized memory address, an error dialog box appears.

Search Pattern

The Search Pattern dialog box shown in Figure 3.41 allows you to search memory or a memory range for a specific expression.
Debugger Components
General Debugger Components

Figure 3.41 Search Pattern Dialog Box

This dialog box shown in Figure 3.42 allows you to modify the update rate in steps of 100ms.

Figure 3.42 Update Rate Dialog Box

NOTE Periodical mode is not available for all hardware connections or some additional configuration may be required in order to make it work.
## Associated Popup Menu

### Figure 3.43 Memory Popup Menu

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Watchpoint</td>
<td>Appears in the Popup Menu only if no watchpoint is set or disabled on the selected memory range. When selected, sets a Read/Write watchpoint at this memory area. Memory ranges where a read/write watchpoint has been defined are underlined in yellow. If the memory area is accessed during execution of the application, the program is halted and the current program state is displayed in all window components.</td>
</tr>
<tr>
<td>Delete Watchpoint</td>
<td>Appears in the Popup Menu only if a watchpoint is set or disabled on the selected memory range. When selected, deletes this watchpoint.</td>
</tr>
<tr>
<td>Show Watchpoints</td>
<td>When selected, brings up the Controlpoints Configuration Window - Watchpoints Tab. This is the interface through which watchpoints are controlled. (See “Control Points” chapter)</td>
</tr>
</tbody>
</table>
Debugger Components
General Debugger Components

Table 3.21 Memory Popup Menu Description (continued)

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Markpoint</td>
<td>Appears in the Popup Menu only if no watchpoint is set or disabled on the selected memory range. When selected, sets a Read/Write watchpoint at this memory area.</td>
</tr>
<tr>
<td>Show Markpoints</td>
<td>When selected, brings up the Controlpoints Configuration Window - Markpoints Tab. This is the interface through which markpoints are controlled. (See &quot;Control Points&quot; chapter)</td>
</tr>
<tr>
<td>Show Location</td>
<td>Forces all opened windows to display information about the selected memory area.</td>
</tr>
<tr>
<td>Word Size, etc.</td>
<td>The remaining entries in this menu are explained in table 3.17 Memory Menu Description</td>
</tr>
</tbody>
</table>

Drag Out:
Table 3.22 Describes the drag actions possible from the Memory component.

Table 3.22 Memory Component Drag Possibilities

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>Displays disassembled instructions starting at the first address selected. The instructions corresponding to the selected memory area are highlighted in the Assembly component.</td>
</tr>
<tr>
<td>Command Line</td>
<td>Appends the selected memory range to the Command Line window</td>
</tr>
<tr>
<td>Register</td>
<td>Loads the destination register with the start address of the selected memory block.</td>
</tr>
<tr>
<td>Source</td>
<td>Displays high level language source code starting at the first address selected. Instructions corresponding to the selected memory area are greyed in the source component.</td>
</tr>
</tbody>
</table>

Drop Into:
Table 3.23 shows the drop actions possible in the Memory component.
Debugger Components
General Debugger Components

Table 3.23  Memory Component Drop Possibilities

<table>
<thead>
<tr>
<th>Source Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>Dumps memory starting at the selected PC instruction. The PC location is selected in the memory component.</td>
</tr>
<tr>
<td>Data</td>
<td>Dumps memory starting at the address where the selected variable is located. The memory area where the variable is located is selected in the memory component.</td>
</tr>
<tr>
<td>Register</td>
<td>Dumps memory starting at the address stored in the selected register. The corresponding address is selected in the memory component.</td>
</tr>
<tr>
<td>Module</td>
<td>Dumps memory starting at the address of the first global variable in the module. The memory area where this variable is located is selected in the memory component.</td>
</tr>
</tbody>
</table>

Demo Version Limitations
No limitation

Associated Commands
ATTributes, FILL, SMEM, SMOD, SPC, UPDATERATE.
MicroC Component

The MicroC window shown in Figure 3.44 is an interface module for RHAPSODY in MicroC, the analysis, design and implementation tool for embedded systems and software developers from I-LOGIX.

Figure 3.44 MicroC Window

The MicroC component establishes a communication with Rhapsody in MicroC to activate its design-level debugging capabilities. Rhapsody in MicroC drives its debugging animation that communicates with the Debugger environment over TCP/IP. This allows you to execute, stop and run the application, to set step commands, breakpoints, events, and idle states to perform control over the application.

Communication is realized by selecting the Connect entries of the MicroC Link menu. The Setup entry allows you to define the connection parameters.

The functions available allow you to start the currently loaded application, to stop it, to execute a single step in the application, to set and clear a breakpoint, to evaluate an expression and to quit the application interface.

NOTE For more information, refer to the RHAPSODY in MicroC documentation from I-Logix.

NOTE In order to work, MicroC needs to have a copy of the amc_communication_dll.dll in the prog directory from the current installation.

MicroC Link Menu

Figure 3.45 shows the MicroC menu and its entries are described in Table 3.24.
Debugger Components
General Debugger Components

Within the Communication Specification dialog box shown in Figure 3.46, you can set the MicroC Host and ID for communication between the Debugger and RHAPSODY in MicroC. A checkbox allows you to see the communication protocol.

Figure 3.46  Communication Specification

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Opens the communication setup Window.</td>
</tr>
<tr>
<td>Connect</td>
<td>Establishes communication with RHAPSODY in MicroC.</td>
</tr>
</tbody>
</table>

MicroC Communication Specification

Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dropped in.

Demo Version Limitations
The MicroC Component is not available in demo mode.

MicroC DLLs
The RiMC (or MicroC.wnd) component has been updated to make use of the new features that come of the latest release of the communication DLL from I-Logix.
To ensure proper communication between Rhapsody in MicroC and the external debugger (HI-WAVE) from Freescale (formerly HIWARE), two files have to be installed in the 'prog' subdirectory of the CodeWarrior installation:

**microc.wnd**
This is the HI-WAVE component that has to be loaded in order to configure the communication parameters and mode of operation. This component requires the amc_communication_dll.dll to be loaded properly (if this DLL is missing, there will be an error message that a library is missing).

**amc_communication_dll.dll**
This DLL implements the actual protocol (over TCP/IP). This DLL is delivered together with the RiMC and has to be copied into the 'prog' subdirectory of the CodeWarrior installation (this DLL will not be installed with the CodeWarrior product).

The 'Product Version' of this DLL has to be 'RiMC 3.0' of higher.

**Changes and New Features**
The new DLL from I-Logix allows now implementing the Graphical Back Animation with fewer resources on the target system; so only one single breakpoint is required in synchronous mode and even none in asynchronous mode!

- There are now two modes of operation:
  
  **Synchronous:**
  
  This mode corresponds to the legacy implementation and lets RiMC update the state whenever a change of state is detected on the target system. This is implemented by setting a breakpoint on the target on a function that is called whenever that state of the application is changed. When hit, the state is sent to RiMC and the application is resumed immediately. By concept, this procedure will slow down execution of the target application dramatically. Compared to the previous releases, only one single breakpoint is required for this mode.

  **Asynchronous**
  
  This is a new mode introduced in this release. The state of the application will only sampled from time to time. Thus, this mode allows the application to run at full speed but will not update RiMC about each change of state. Also, it does not require any resources on the target system except that the target memory has to be accessible while the application is running. The connections that support this mode are the HC(S)12(X) Freescale Full Chip Simulator and any Host connection (HTI) that uses the BDM of features dual-ported RAM.

- The Setup dialog was extended to reflect that additional modes:
In **Asynchronous mode**: the interval for updating the state can be specified in increments of 100ms. All the settings from this dialog are saved in the current project file and will be used in future sessions automatically.

- There are now command line commands to setup the communication parameters:

  **MCPROTOCOL [ON|OFF]**
  
  Switched on and off the protocol to the Command window (when open at all).

  **MCMODE (SYNC|ASYNC [interval])**
  
  Sets the reporting mode to synchronous or asynchronous. If asynchronous is specified, the interval can be specified too. If the interval is not specified, the previous value will be maintained.

  **MCCONNECT [HostName] [portNumber]**
  
  This command tries to connect to RiMC. The name of the computer where RiMC is expected and/or its port number can be can be specified. If not specified, the previous value will be used.

  Each of these commands will close any pending communication and re-establish communication with the new parameters.

  **In Synchronous mode**: the states are reported not faster than every 10ms. This will avoid overruns in the communication to RiMC when using the simulator as a connection.
Module Component

The Module window shown in Figure 3.48 gives an overview of source modules building the application.

Figure 3.48  Module Window

![Module Window](image)

The Module component displays all source files (source modules) bound to the application. The Module window displays all modules in the order they appear in the absolute file.

Module Operations

Double-clicking a module name forces all open windows to display information about the module: the Source Component window shows the module's source and the global Data Component window displays the module's global variables.

Module Menu

The Module Component window has no menu.

Drag Out:

Table 3.25 shows the drag actions possible from the Module component.
Drop Into:
Nothing can be dropped into the Module Component window.

Demo Version Limitations
Only 2 modules are displayed

Table 3.25 Module Component Drag Possibilities

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data &gt; Global</td>
<td>Displays the global variables from the selected module in the data component</td>
</tr>
<tr>
<td>Memory</td>
<td>Dumps memory starting at the address of the first global variable in the module. The memory area where this variable is located is selected in the memory component.</td>
</tr>
<tr>
<td>Source</td>
<td>Displays the source code from the selected module.</td>
</tr>
</tbody>
</table>
Procedure Component

The Procedure window shown in Figure 3.49 displays the list of procedure or function calls that have been made up to the moment the program was halted. This list is known as the ‘procedure chain’ or the ‘call chain’.

Figure 3.49 Procedure Window

In the Procedure Component window, entries in the call chain are displayed in reverse order from the last (most recent on top) call to the first call (initial on bottom). Types of procedure parameters are also displayed.

The Object Info bar of the component window contains the source module and address of the selected procedure.

Procedure Operations

Double-clicking on a procedure name forces all open windows to display information about that procedure: the Source Component window shows the procedure's source, the local Data Component window displays the local variables and parameters of the selected procedure. The current assembly statement inside this procedure is highlighted in the Assembly component.

NOTE When a procedure of a level greater than 0 (the top most) is double clicked in the Procedure Component, the statement corresponding to the call of the lower procedure is selected in the Source Window and Assembly Window.

Procedure Menu

Figure 3.50 shows the Procedure menu and its entries are described in Table 3.26.
Drag Out:
Table 3.27 shows the drag actions possible from the Procedure component.

Table 3.27 Procedure Component Drag Possibilities.

<table>
<thead>
<tr>
<th>Destination Component</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data &gt; Local</td>
<td>Displays the local variables from the selected procedure in the data component</td>
</tr>
<tr>
<td>Source</td>
<td>Displays source code of the selected procedure. Current instruction inside the procedure is highlighted in the Source component.</td>
</tr>
<tr>
<td>Assembly</td>
<td>The current assembly statement inside the procedure is highlighted in the Assembly component.</td>
</tr>
</tbody>
</table>

Drop Into:
Nothing can be dropped into the Procedure component.

Demo Version Limitations
Only the last two procedures are displayed.

Associated Commands
ATTRIBUTES, FINDPROC
Profiler Component

The Profiler window shown in Figure 3.51 provides information on application profile.

NOTE In cases where in cases of advanced code optimizations (like linker overlapping ROM/code areas) the profiler output/data is affected. In such a case, it is recommended to switch of such linker optimizations.

Figure 3.51 Profiler Window

The Profiler window contains source module and procedure names and percentage values representing the time spent in each source module or procedure. The Profiler component window contains percentages and also graphic bars.

The Profiler window can set a split view in the Source and Assembly windows, as shown in Figure 3.52. To obtain a split view in either the Source or Assembly windows, select: Details>Source or Details>Assembly or both from the Profiler menu and submenu. The split windows effect ends when the Profiler window is closed.

Figure 3.52 Split View in the Source and Assembly Windows

Percentage values representing the time spent in each source or assembler instruction are displayed on the left side of the instruction. The split view can also display graphic bars. Split views are removed when the Coverage component is closed or if you open the split view Popup Menu and select Delete.
The value displayed may reflect percentages from total code or percentages from module code.

**Profiler Operations**

Click the fold/unfold icon to unfold/fold the source module.

**Profiler Menu**

*Figure 3.53* shows the Profiler Menu entries, with the Details submenu and the Base submenu. *Figure 3.54* shows the Profiler Output File submenu. Entries are described in *Table 3.28*.

**Figure 3.53 Profiler Menu and Submenus**

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Resets all statistics.</td>
</tr>
<tr>
<td>Details</td>
<td>Sets a split view in the chosen component (Source or Assembly)</td>
</tr>
<tr>
<td>Base</td>
<td>Sets the base of percentage (total code or module code).</td>
</tr>
<tr>
<td>Graphics</td>
<td>Toggles the display from graphics bar.</td>
</tr>
<tr>
<td>Timer Update</td>
<td>Switches on/off the periodic update of the Coverage component. If activated, statistics are updated each second.</td>
</tr>
<tr>
<td>Output File</td>
<td>Setup the Profiler Output File Functions.</td>
</tr>
</tbody>
</table>

**Figure 3.54 Profiler Output File Submenu**

**Table 3.28 Profiler Menu Entries Description**
Split View Associated Popup Menu

Figure 3.55 shows the Profiler popup menu, the Delete and Graphics menu entries are described in Table 3.29.

Table 3.29 Profiler Split View Associated Popup Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete</td>
<td>Removes the split view from the host component.</td>
</tr>
<tr>
<td>Graphics</td>
<td>Toggles the graphic bars display in the split view.</td>
</tr>
</tbody>
</table>

Profiler Output File Functions

You can redirect the Profiler component results to an output file by choosing Output File... > Save As... in the menu or popup menu.

Output File Filter

By choosing Output Filter..., the dialog box shown in Figure 3.56 lets you select what you want to display, i.e. modules only, modules and functions, or modules and functions and code lines. You can also specify a range of coverage to be logged in your file.

Figure 3.56 Output File Filter Dialog Box
Output File Save
The Save As... entry opens a Save As dialog box where you can specify the output file name and location.

Associated Popup Menu
Identical to menu.

Drag Out:
All displayed items can be dragged out. Destination windows may display information about the time spent in some codes in a split view.

Drop Into:
Nothing can be dropped into the Profiler Component window.

Demo Version Limitations
Only modules are displayed and the Save function is disabled.

Associated Commands:
GRAPHICS, TUPDATE, DETAILS, RESET, BASE.

Recorder Component
The Recorder window shown in Figure 3.57 provides record and replay facilities for debug sessions.

Figure 3.57 Recorder Window

The Recorder window enables the user to record and replay command files. The recorded file may also contain the time at which the command is executed.
Debugger Components
General Debugger Components

Click the buttons shown below to record, play, pause and stop.

- Play
- Record
- Stop
- Pause

An animation occurs during recording, replaying and pausing.

The current action (record, play or pause) and path of the involved file are displayed in the Object Infor bar of the window.

Recorder Operations

When there is no record or play session (e.g., when the window is open), only the record and play buttons are enabled.

When you click the record button, the debugger prompts you to enter a file name. Then a record session starts and the stop button is enabled. Click the stop button to end the record session.

Clicking the replay button prompts for a file name. Command files have a .rec default extension and can be edited. A replay session starts and only the stop and pause buttons are enabled. When the pause button is clicked, file execution stops and the play and stop buttons are enabled. When the play button is clicked, file execution continues from the point it has been stopped. When the stop button is clicked, the replay session stops.

Terminal and TestTerm Record

Data typed in the Terminal component and TestTerm component is recorded during a record session. The resulting file can be replayed only if the time is also recorded (Record Time menu entry of the recorder has to be checked before recording).

Recorder Menu

The Recorder menu shown in Figure 3.58 changes according to the current session. The menu items are described in Table 3.30.
In Listing 3.2, an .abs file is loaded, a breakpoint is set, the assembly component is configured to display the code and addresses. The Data1 component display is switched to local variables, and the application is started and stopped at the breakpoint.

Listing 3.2 Record File Example

```plaintext
at 4537 load C:\Freescale\DEMO\fibo.abs
at 9424 bs 0x1040 P
at 11917 Assembly < attributes code on
at 14481 Assembly < attributes adr on
at 20540 Data:1 < attributes scope local
at 24425 g
wait ;s
```

Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dropped in.

Demo Version Limitations
Only 20 commands are recorded and replayed.
Register Component

The Register window, shown in Figure 3.59, displays the content of registers and status register bits of the target processor.

Figure 3.59  Register Window

Register values can be displayed in binary or hexadecimal format. These values are editable.

Status Register Bits

Set bits are displayed dark, whereas reset bits are displayed grey. Double-click a bit to toggle it. During program execution, contents of registers that have changed since the last refresh are displayed in red, except for status register bits.

The Object Infor bar of the window contains the number of CPU cycles as well as the processor's name.

Editing Registers

Double-click on a register to open an edit box over the register, so that the value can be modified.

Press the ESC key to ignore changes and retain previous content of the register.

If the Enter key is pressed outside the edited register, the new value is validated and the register content is changed.

If the Tab key is pressed, the new value is validated and the register content is changed. The next register value is selected and may be modified.

Double-clicking a status register bit toggles it.

Holding down the left mouse button and pressing the A key: Contents of Source, Assembly and Memory component windows change. The Source window shows the source code located at the address stored in the register. The Assembly window shows the disassembled code starting at the address stored in the register. The Memory window dumps memory starting at the address stored in the register.
Register Menu (Format Submenu)

The Register menu contains the items shown in Figure 3.60. Table 3.31 describes the menu entries.

Figure 3.60 Register Menu

Table 3.31 Register Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>Selects the hexadecimal register display format</td>
</tr>
<tr>
<td>Bin</td>
<td>Selects the binary register display format</td>
</tr>
<tr>
<td>Oct</td>
<td>Selects the octal register display format</td>
</tr>
<tr>
<td>Dec</td>
<td>Selects the signed decimal register display format</td>
</tr>
<tr>
<td>UDec</td>
<td>Selects the unsigned decimal register display format</td>
</tr>
<tr>
<td>Float</td>
<td>Selects the float register display format (all 32/64 bit registers are displayed as floats, all others as hex)</td>
</tr>
<tr>
<td>Auto</td>
<td>Selects the auto register display format (all floating point 32/64 bit registers are displayed as floats, all others as hex)</td>
</tr>
<tr>
<td>Bit Reverse</td>
<td>Selects the bit reverse data display format (Each bit is reversed).</td>
</tr>
</tbody>
</table>

Drag Out:

Table 3.32 contains the drag actions possible from the Register window.
Debugger Components
General Debugger Components

Table 3.32  Register Component Drag Possibilities

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>Assembly component receives an address range, scrolls up to the corresponding instruction and highlights it.</td>
</tr>
<tr>
<td>Memory</td>
<td>Dumps memory starting at the address stored in the selected register. The corresponding address is selected in the memory component.</td>
</tr>
<tr>
<td>Command Line</td>
<td>The address stored in the pointed to register is appended to the current command.</td>
</tr>
</tbody>
</table>

Drop Into:
Table 3.33 shows the drop actions possible into the Register component.

Table 3.33  Register Component Drop Possibilities

<table>
<thead>
<tr>
<th>Source Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>Loads the destination register with the PC of the selected instruction.</td>
</tr>
<tr>
<td>Data</td>
<td>Dragging the name loads the destination register with the start address of the selected variable. Dragging the value loads the destination register with the value of the variable.</td>
</tr>
<tr>
<td>Source</td>
<td>Loads the destination register with the PC of the first instruction selected.</td>
</tr>
<tr>
<td>Memory</td>
<td>Loads the destination register with the start address of the selected memory block.</td>
</tr>
</tbody>
</table>

Demo Version Limitations
No limitation

Associated Commands
ATTRIBUTES.
SoftTrace Component

The SoftTrace window shown in Figure 3.61 records and displays instruction frames and time or cycles.

**Figure 3.61 SoftTrace Window**

![SoftTrace Window](image)

The window’s Object Bar displays the number of recorded frames and the name of the function where the selected frame is located.

**SoftTrace Operations**

Pointing at a frame and dragging the mouse forces all open windows to show the corresponding code or location. Time and cycles of all other frames are evaluated relative to this base.

Holding down the left mouse button and pressing the Z key sets the zero base frame to the pointed frame.

Holding down the left mouse button and pressing the D key forces all open component windows to show the code matching the pointed to frame.

**SoftTrace Menu**

The SoftTrace Menu shown in Figure 3.62 contains the functions described in Table 3.34.
Debugger Components

General Debugger Components

Figure 3.62  SoftTrace Menu

- Record
  - Clock Speed...
  - Max Frames...
- Cycles
- ms
- Reset

Table 3.34  SoftTrace Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record</td>
<td>Switches recording on and off.</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>Sets the clock frequency.</td>
</tr>
<tr>
<td>Max Frames</td>
<td>Sets the maximum number of recorded frames. Therefore you can minimize the amount of memory required to display frames.</td>
</tr>
<tr>
<td>Cycles</td>
<td>Displays cycles instead of time (in ms).</td>
</tr>
<tr>
<td>ms</td>
<td>Displays time (in ms) instead of cycles.</td>
</tr>
<tr>
<td>Reset</td>
<td>Removes all recorded frames.</td>
</tr>
</tbody>
</table>

Associated Popup Menu

The SoftTrace popup menu shown in Figure 3.63 contains functions (described in Table 3.35) associated with the pointed to frame.

Figure 3.63  SoftTrace Associated Popup Menu

- Set Zero Base
- Show Location
- Record
  - Clock Speed...
  - Max Frames...
- Cycles
- ms
- Reset
Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dropped in.

Demo Version Limitations
The number of frames is limited to 50.

Associated Commands
CLOCK, CYCLE, FRAMES, RECORD, RESET.

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Zero Base</td>
<td>Sets the zero base frame to the pointed to frame.</td>
</tr>
<tr>
<td>Show Location</td>
<td>Forces open component windows to show the code corresponding to the pointed to frame.</td>
</tr>
</tbody>
</table>
Debugger Components
General Debugger Components

Source Component

The Source window shown in Figure 3.64 displays the source code of your program, i.e. your application file.

Figure 3.64 Source Window

The Source window allows you to view, change, monitor and control the current execution location in the program. The text displayed in the Source Component window is chroma-coded, i.e. language keywords, comments and strings are emphasized with different colors (respectively blue, green, red). A word can be selected by double-clicking it. A section of code can be selected by holding down the left mouse button and dragging the mouse.

The object info bar displays the line number in the source file of the first visible line that is at the top of the source.

Source code can be folded and unfolded. Marks (places where breakpoints may be set) can be displayed.

When the source statement matching the current PC is selected in this window, (e.g., in a C source: `fibi = fib2`, the matching assembler instruction in the Assembler component window is also selected. This instruction is the next instruction to be executed by the CPU.
If breakpoints have been set in the program, they will be marked in the program source with a special symbol depending on the kind of breakpoint. For information on breakpoints refer to sections in the "Control Points" chapter. If execution has stopped, the current position is marked in the source component by highlighting the corresponding statement.

The complete path of the displayed source file is written in the Object Info bar of this window.

**NOTE** You cannot edit the visible text in the Source window. This is a file viewer only.

### Tool Tips Features

The Debugger source component provides tool tips to display variable values. The tool tip is a small rectangular pop-up window that displays the value of the selected variable (shown in Figure 3.65) or the parameter value and address of the selected procedure. A parameter or procedure can be selected by double-clicking it.

**Figure 3.65 ToolTips Features**

Select **ToolTips>Enable** from the source menu entry to enable or disable the tool tips feature.

Select **ToolTips>Mode** from the source menu entry to select normal or details mode, which provides more information on a selected procedure.

Select **ToolTips>Format** from the source menu entry to select the tool tip display format (Decimal, Hexadecimal, Octal, Binary or ASCII).
On Line Disassembling

For information about performing on line disassembly, refer to section How to Consult Assembler Instructions Generated by a Source Statement.

- Select a range of instructions in the source component and drag it into the assembly component. The corresponding range of code is highlighted in the Assembly component window, as shown in Figure 3.66.

- Holding down the left mouse button and pressing the T key: Highlights a code range in the Assembly component window corresponding to the first line of code selected in the Source component window where the operation is performed. This line or code range is also highlighted.

Figure 3.66 On Line Disassembling

Setting Temporary Breakpoints

For information on how to set breakpoints refer to sections in the "Control Points" chapter.

- Point to an instruction in the Source component Window and click the right mouse button. The Source window popup menu is displayed. Select Run To Cursor from the popup menu. The application continues execution and stops at this location.

- Holding down the left mouse button and pressing the T key: Sets a temporary breakpoint at the nearest code position (visible with marks) thereafter the program runs and breaks at this location, as shown in Figure 3.67.
Setting Permanent Breakpoints

- Point to an instruction in the Source component Window and click the right mouse button. The Source Component popup menu is displayed. Select **Set Breakpoint** from the popup menu. The permanent breakpoint icon is displayed in front of the source statement pointed to.

- Holding down the left mouse button and pressing the P key: Sets a permanent breakpoint at the nearest code position (visible with marks). The permanent breakpoint icon is displayed in front of the source statement pointed to.

Folding and Unfolding

Use this feature to show or hide a section of source code (e.g., source code of a function). For example, if a section is free of bugs, you can hide it. All text is unfolded at loading.

Sections of code that can be folded are enclosed between and .

Sections of code that can be unfolded are hidden under .

Double-click a folding mark or to fold the text located between the marks.

Double-click an unfolding mark to unfold the text that is hidden behind the mark.
Source Menus

The Source Menu is shown in Figure 3.68 and Figure 3.69 shows the functions associated with the Source Popup Menu, while Table 3.36 describes these functions.

Figure 3.68  Source Menu

![Source Menu]

Figure 3.69  Source Associated Popup Menu

![Source Associated Popup Menu]
Table 3.36  Associated Pop - Up Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Breakpoint</td>
<td>Appears only in the Popup Menu if no breakpoint is set or disabled at the nearest code position (visible with marks). When selected, sets a permanent breakpoint at this position. If program execution reaches this statement, the program is halted and the current program state is displayed in all window components.</td>
</tr>
<tr>
<td>Delete Breakpoint</td>
<td>Appears only in the Popup Menu if a breakpoint is set or disabled at the nearest code position (visible with marks). When selected, deletes this breakpoint.</td>
</tr>
<tr>
<td>Enable Breakpoint</td>
<td>Appears only in the Popup Menu if a breakpoint is disabled at the nearest code position (visible with marks). When selected, enables this breakpoint.</td>
</tr>
<tr>
<td>Disable Breakpoint</td>
<td>Appears only in the Popup Menu if a breakpoint is set at the nearest code position (visible with marks). When selected, disables this breakpoint.</td>
</tr>
<tr>
<td>Run To Cursor</td>
<td>When selected, sets a temporary breakpoint at the nearest code position and continues program execution immediately. If there is a disabled breakpoint at this position, the temporary breakpoint will also be disabled and the program will not halt. Temporary breakpoints are automatically removed when they are reached.</td>
</tr>
<tr>
<td>Show Breakpoints</td>
<td>Opens the Controlpoints Configuration Window's Breakpoints Tab and allows you to view the list of breakpoints defined in the application and modify their properties (See “Control Points” chapter).</td>
</tr>
<tr>
<td>Show Location</td>
<td>Highlights a code range in the Assembly component window matching the line or selected source code. The line or the source code range are highlighted as well.</td>
</tr>
<tr>
<td>Set Markpoint</td>
<td>Appears only in the Popup Menu if a markpoint is disabled at the nearest code position (visible with marks). When selected, enables this markpoint.</td>
</tr>
<tr>
<td>Delete Markpoint</td>
<td>Appears only in the Popup Menu if a markpoint is set at the nearest code position (visible with marks). When selected, disabled this markpoint.</td>
</tr>
</tbody>
</table>

### Debugger Components

#### General Debugger Components

**Table 3.36  Associated Pop - Up Menu Description (continued)**

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show Markpoints</td>
<td>Opens the Controlpoints Configuration Window's Markpoints Tab and allows you to view the list of markpoints defined in the application and modify their properties (See &quot;Control Points&quot; chapter).</td>
</tr>
<tr>
<td>Set Program Counter</td>
<td>The Program Counter is set to the address of the selected source code.</td>
</tr>
<tr>
<td>Open Source File</td>
<td>Opens the Source File Dialog if a CPU is loaded (see chapter below).</td>
</tr>
<tr>
<td>Copy (CTRL+C)</td>
<td>Copies the selected area of the source component into the clipboard. You can select a word by double-clicking it. You can select a text area with the mouse by moving the pointer to the left of the lines until it changes to a right-pointing arrow, and then drag up or down; automatic scrolling is activated when the text is not visible in the windows.</td>
</tr>
<tr>
<td>Go to Line (CTRL+G)</td>
<td>Opens a dialog box to scroll the window to a number line (see chapter below).</td>
</tr>
<tr>
<td>Find... (CTRL+F)</td>
<td>Opens a dialog box prompting for a string and then searches the file displayed in the source component. To start searching, click Find Next, the search is started at the current selection or at the first line visible in the source component (see chapter below).</td>
</tr>
<tr>
<td>Find Procedure (CTRL+I)</td>
<td>Opens a dialog box for searching a procedure (see chapter below).</td>
</tr>
<tr>
<td>Foldings</td>
<td>Opens the folding window (see chapter below).</td>
</tr>
<tr>
<td>Marks</td>
<td>Toggles the display of source positions where breakpoints may be set. If this switch is on, these positions are marked by small triangles.</td>
</tr>
<tr>
<td>ToolTips</td>
<td>Allows you to enable or disable the source tool tips feature, to set up the tool tip mode, and tool tip format.</td>
</tr>
</tbody>
</table>
NOTE If some statements do not show marks although the mark display is switched on, the following reasons may be the cause:
- The statement did not produce any code due to optimizations done by the compiler.
- The entire procedure was not linked in the application, because it is never used.

Open Source File

The Open Source File dialog box shown in Figure 3.70 allows you to open the Source File (if a CPU is loaded). A source file is a file that has been used to build the currently loaded absolute file. Assembly file (*.dbg) is searched in the directory given by the OBJPATH and GENPATH variables. C, C++ files (*.c, *.cpp, *.h, ...) are searched in the directories given by the GENPATH variable.

Figure 3.70 Open Source File Dialog Box

Go to Line

This menu entry is only enabled if a source file is loaded. It opens the dialog box shown in Figure 3.71. In this dialog box, enter the line number you want to go to in the source component, the selected line will be displayed at the top of the source window. If the number is not correct, a message is displayed.
Debugger Components
General Debugger Components

Figure 3.71 Go to Line Dialog Box

When this dialog box is open, the line number of the first visible line in the source is displayed and selected in the Enter Line Number edit box.

Find Operations

The Find dialog box, shown in Figure 3.72 is used to perform find operations for text in the Source component. Enter the string you want to search for in the Find what edit box. To start searching, click Find Next, the search starts at the current selection or first line visible in the source component, when nothing is selected.

Use the Up / Down buttons to search backward or forward. If the string is found, the source component selection is positioned at the string. If the string is not found, a message is displayed.

Figure 3.72 Find Dialog Box

This dialog box allows you to specify the following options:

- **Match whole word only**: If this box is checked, only strings separated by special characters will be recognized.
- **Match case**: If this box is checked, the search is case sensitive.

**NOTE**
If an item (single word or source section) has been selected in the Source component window before opening the Find dialog, the first line of the selection will be copied into the “Find what” edit box.
Find Procedure

The Find Procedure dialog box, shown in Figure 3.73, is used to find the procedure name in the currently loaded application. Enter the procedure name you want to search for in the **Find Procedure** edit box. To start searching, click **OK**, the search starts at the current selection or at the first line visible in the source component, when nothing is selected.

![Figure 3.73 Find Procedure Dialog Box](image)

If a valid procedure name is given as a parameter, the source file where the procedure is defined is opened in the Source Component. The procedure’s definition is displayed and the procedure’s title is highlighted.

The drop-down list allows you to access the last searched items (classified from first to older input). Recent search items are stored in the current project file.

Folding Menu

The Folding Menu shown in Figure 3.74 allows you to select the Fold functions described in Table 3.37.

![Figure 3.74 Folding Menu](image)
Table 3.37  Folding Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unfold</td>
<td>Unfolds the displayed source code</td>
</tr>
<tr>
<td>Fold</td>
<td>Folds the displayed source code</td>
</tr>
<tr>
<td>Unfold All Text</td>
<td>Unfolds all displayed source code</td>
</tr>
<tr>
<td>Fold All Text</td>
<td>Folds all displayed source code</td>
</tr>
<tr>
<td>All Text Folded At Loading</td>
<td>Folds all source code at load time</td>
</tr>
</tbody>
</table>

Drag Out:
Table 3.38 shows the drag actions possible from the Source component.

Table 3.38  Source Drag Possibilities

<table>
<thead>
<tr>
<th>Destination Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>Displays disassembled instructions starting at the first high level language instruction selected. The assembler instructions corresponding to the selected high level language instructions are highlighted in the Assembly component</td>
</tr>
<tr>
<td>Register</td>
<td>Loads the destination register with the PC of the first instruction selected.</td>
</tr>
<tr>
<td>Data</td>
<td>A selection in the Source window is considered as an expression in the Data window, as if it was entered through the Expression Editor of the Data component. (please see Data Component or Expression Editor)</td>
</tr>
</tbody>
</table>
Debugger Components
Visualization Utilities

Drop Into:
Table 3.39 shows the drop actions possible into the Source component.

Table 3.39 Source Drop Possibilities

<table>
<thead>
<tr>
<th>Source Component Window</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>Source component scrolls to the source statements corresponding with the pointed to assembly instruction and highlights it.</td>
</tr>
<tr>
<td>Memory</td>
<td>Displays high level language source code starting at the first address selected. Instructions corresponding to the selected memory area are greyed in the source component.</td>
</tr>
<tr>
<td>Module</td>
<td>Displays source code from the selected module.</td>
</tr>
</tbody>
</table>

Demo Version Limitations
Only one source file of the currently loaded application can be displayed.

Associated Commands
ATTRIBUTES, FIND, FOLD, FINDPROC, SPROC, SMOD, SPC, SMEM, UNFOLD.

Visualization Utilities
Besides components that provide the Debugger engine a well-defined service dedicated to the task of application development, the debugger component family includes utility components that extend to the productive phase of applications, such as, the host application builder components, process visualization components, etc.

Among these components, there are visualization utilities that graphically display values, registers, memory cells, etc., or provide an advanced graphical user interface to simulated I/O devices, program variables, and so forth.

The following components of the continuously growing set of visualization utilities belong to the standard Debugger installation.

Inspector Component
The Inspector window shown in Figure 3.75 displays information about several topics. It displays loaded components, the visible stack, pending events, pending exceptions and loaded I/O devices.
Debugger Components
Visualization Utilities

Figure 3.75 Inspector Component Window

The hierarchical content of the items is displayed in a tree structure. If any item is selected on the left side, then additional information is displayed on the right side.

In the figure above, for example, the Object Pool is expanded. The Object Pool contains the TargetObject, which contains the Leds and Swap peripheral devices. The Swap peripheral device is selected and registers of the Swap device are displayed.

Components Icon

When the components icon is selected in the Inspect window, as shown in Figure 3.76, the right side displays various information about all loaded components. A Component is the “unit of dynamic loading”, therefore all windows, the CPU, the connection and maybe the connection-simulator are listed.
Debugger Components
Visualization Utilities

Figure 3.76 Inspect Window Components Icon

Stack Icon
The Stack icon shown in Figure 3.77 displays the current stack trace. Every function on the stack has a separate icon on the trace. In the stack-trace, the content of a local variable is accessible.

Figure 3.77 Inspector Window Stack Icon

Symbol Table
The symbol table shown in Figure 3.78 displays all loaded symbol table information in raw format. There are no stack frames associated with functions. Therefore the content of local variables is not displayed. Global variables and their types are displayed.
Debugger Components
Visualization Utilities

Figure 3.78 Inspector Window Symbol Table

Events Icon
The Inspector window Events icon shown in Figure 3.79 shows all currently installed events. Events are handled by peripheral devices, and notified at a given time. The Event display shows the name of the event and remaining time until the event occurs.

Figure 3.79 Inspector Window Events Icon

Events are only used in the HC(S)12(X) Freescale Full Chip Simulator. This information is used for simulation I/O device development.

When simulating a watchdog/COP, an event with the remaining time is displayed in the Event View.

Exceptions Icon
The Inspector window Exceptions icon shown in Figure 3.80 shows all currently raised exceptions. Exceptions are pending interrupts.
Events are only used in the HC(S)12(X) Freescale Full Chip Simulator. This information is used for simulation I/O device development.

Since interrupts are usually simulated immediately when they are raised, the Exceptions are usually empty. Only when interrupts are disabled or an interrupt is handled, something is visible in this item.

When simulating a watchdog/COP, an Exception is raised as soon as the watchdog time elapses.

Object Pool

The Object Pool shown in Figure 3.81 is a pool of objects. It can contain any number of Objects, which can communicate together and also with other parts of the Debugger.

The most common use of Objects is to simulate special hardware with the I/O development package, however, other connections also use the Object Pool. For example, the Terminal Component exchanges its input and output by the Object Pool. The Terminal Component also operates with some hardware connections.

For the HC(S)12(X) Freescale Full Chip Simulator, the Object Pool usually contains the TargetObject, which represents the address space. All Objects that are loaded are displayed in the Object Pool. The TargetObject additionally shows the objects that are mapped to the address space.
Debugger Components
Visualization Utilities

Inspector Operations

Click the folded/unfolded icons to unfold/fold the tree and display/hide additional information.

Click on any icon or name to see the corresponding information displayed on the right side.

On the right side, some value fields can be edited by double clicking on them. Only values that are accessible can be edited. Usually, if a value is displayed, it can be changed. I/O Devices in the Object Pool do not accept all new values, depending on the I/O Device. Values can be entered in hexadecimal (with preceding 0x), in decimal, in octal (with preceding 0), or in binary (with preceding &).

To see the IO_Led in the Inspector, as shown in Figure 3.82, open the IO_Led with the context menu Component-Open and then open the Inspector. If the Inspector is already loaded, select Update from the context menu in the Inspector. Then click on the Components icon to see the Component list, which now includes the “IO_Led” component.

**Figure 3.82 How to See the IO_Led in the Inspector Window**

Expand Object Pool, to see the Leds icon. Click on the Leds icon. On the right side, the Port_Register and Data_Direction_Register are displayed with their current value. Double click on the values to change them (Figure 3.83).
Figure 3.83 Changing “Data_Direction_Register” Value

Inspector Menu

The Inspector menu contains entries described in Table 3.40.

Table 3.40 Inspector Menu Entries

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>All displayed information is updated. Items that no longer exist are removed and new items are added.</td>
</tr>
</tbody>
</table>

Associated Popup Menu

Commands in the Inspector context menu depend on the selected item. It can contain entries described in Table 3.41.
Debugger Components
Visualization Utilities

Table 3.41 Inspector Popup Menu Entries Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>All items</td>
<td>All displayed information is updated. Items that no longer exist are removed and new items are added.</td>
</tr>
<tr>
<td>Max. Elements</td>
<td>All items</td>
<td>To display large arrays element by element, the maximum number can be configured. It is also possible to display a dialog that prompts the user.</td>
</tr>
<tr>
<td>Format</td>
<td>All items</td>
<td>Numerical values can be displayed in different formats.</td>
</tr>
<tr>
<td>Close</td>
<td>single selected Component only</td>
<td>Closes the corresponding component</td>
</tr>
</tbody>
</table>

Drag Out:

Items that can be dragged, depends on which icon is selected. Table 3.42 gives a brief description.

Table 3.42 Inspector Component Drag Possibilities

<table>
<thead>
<tr>
<th>Dragging Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>The components cannot be dragged</td>
</tr>
<tr>
<td>Stack</td>
<td>The Stack Icon itself cannot be dragged. All subitems can be dragged the same way as the Symbol Table subitems, described below.</td>
</tr>
<tr>
<td>Symbol Table</td>
<td>The Symbol Table icon cannot be dragged out. Subitems can be dragged depending on their type: Modules: Modules can be dragged to the source and global data window to specify a specific module. Functions: Functions can be dragged to display the function or code range. Variables: Variables can be dragged to display their content in memory. Indirections: Indirections can be dragged to display their content in memory.</td>
</tr>
</tbody>
</table>
Drop Into:
Nothing can be dropped in.

Demo Version Limitations
Only 5 items can be expanded at each location. For remaining items, an icon with the text “Demo Limitation” is displayed, as shown in Figure 3.84.

Figure 3.84 Inspector Component Demo Version Limitations

VisualizationTool Component
The VisualizationTool component is a very convenient tool for presenting your data. For software demonstration, or for your own debugging session, take advantage of all its virtual instruments.

The VisualizationTool window, shown in Figure 3.85, consists of a plain workspace that can be equipped with many different instruments.
Edit Mode and Display Mode

The VisualizationTool may operate in two modes: Display mode or Edit mode. The Edit mode is for designing the workspace to suit your needs. In the Display mode you can then use what you have done in the Edit mode, that is, to view values, interact with your application and instruments, press buttons, etc.

To switch between these two modes, you can use the toolbar, the context menu, or the shortcut Ctrl+E.

Add New Instrument

Use the context menu (VisualizationTool Menu) to add a new instrument.

Instrument Selection

You can select a single instrument by left clicking the mouse on it, and change the selection by pressing the tab-key.

To make multiple selections, hold down the control key and left-click on the desired instruments. You can also left click, hold and move to create a selection rectangle.
Move Instruments

There are two ways to move instruments. First, make your desired selection. You can then use the mouse to drag the instruments, or use the cursor keys to move them step by step (hold down the control key to move the instrument in steps of ten). The move process performed with the mouse can be broken off by pressing the escape key.

Resize Instruments

When you select a instrument, sizing handles appear at the corners and along the edges of the selection rectangle. You can resize an object by dragging its sizing handles, or by using the cursor keys while holding down the shift key. The resize process performed with the mouse can be broken off by pressing the escape key. Only one instrument can be resized at a time. Furthermore, each instrument has its own size minimum.

Visualization Tool Menu

Once the Visualization Tool component has been launched, its menu appears in the debugger menu bar. The menu contains the entries described in Table 3.43.

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties</td>
<td>Displays the properties of the currently selected instrument. Shortcut: &lt;Ctrl+P&gt;</td>
</tr>
<tr>
<td>Add New Instrument</td>
<td>Enables to choose an instrument from the list and add it to the view.</td>
</tr>
<tr>
<td>Paste</td>
<td>Pastes an instrument that has been previously copied. Shortcut: &lt;Ctrl+V&gt;</td>
</tr>
<tr>
<td>Select All</td>
<td>Selects all the instruments of the view. Shortcut: &lt;Ctrl+A&gt;</td>
</tr>
<tr>
<td>Edit mode</td>
<td>Switches between Display mode and Edit mode. In Edit mode, this entry is checked. Shortcut: &lt;Ctrl+E&gt;</td>
</tr>
</tbody>
</table>
Associated Popup Menu

The context menu of the VisualizationTool depends on the current selection. It can contains the entries described in Table 3.44.

Table 3.44 VisualizationTool Popup Menu

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Context</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit mode</td>
<td>Always</td>
<td>Switches between Display mode and Edit mode. In Edit mode, this entry is checked.</td>
</tr>
<tr>
<td>Setup</td>
<td>Always</td>
<td>Shows the Setup dialog of the VisualizationTool.</td>
</tr>
<tr>
<td>Load Layout</td>
<td>Edit mode</td>
<td>Loads a VisualizationTool-Layout (*.vtl).</td>
</tr>
<tr>
<td>Save Layout</td>
<td>Always</td>
<td>Saves the current layout to a file (*.vtl).</td>
</tr>
<tr>
<td>Add New Instrument</td>
<td>Edit mode</td>
<td>Shows a new popup menu with all available instruments.</td>
</tr>
<tr>
<td>Properties</td>
<td>Only one instrument selected</td>
<td>Shows up the property dialog box for the currently selected instrument.</td>
</tr>
<tr>
<td>Remove</td>
<td>At least one selection</td>
<td>Removes all currently selected instruments.</td>
</tr>
<tr>
<td>Copy</td>
<td>At least one selection</td>
<td>Copies the data of the currently selected instruments into the clipboard.</td>
</tr>
<tr>
<td>Menu entry</td>
<td>Context</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Cut          | At least one selection | Cuts the currently selected instruments into the clipboard.  
Shortcut: Ctrl + X |
| Paste        | Edit mode        | Adds instruments, which are temporary stored in the clipboard, to the workspace.  
Shortcut: Ctrl + V |
| Send to Back | At least one selection | Sends the current instrument to the back of the Z-order. |
| Send to Front| At least one selection | Brings the current instrument to the front of the Z-order. |
| Clone Attributes | More than one selection | Clones the common attributes to all selected instruments according to the last selected.  
Shortcut: <Ctrl + Enter> |
| Align        | At least two selections | Gives access to a new menu for alignment. |
| Top          | Align            | Aligns the instruments to the top line of the last selected instrument. |
| Bottom       | Align            | Aligns the instruments to the bottom line of the last selected instrument. |
| Left         | Align            | Aligns the instruments to the left line of the last selected instrument. |
| Right        | Align            | Aligns the instruments to the right line of the last selected instrument. |
| Size         | Align            | Makes the size of all selected instruments the same as the last selected. |
| Vertical Size| Align            | Makes the vertical size of all selected instruments the same as the last selected. |
| Horizontal Size | Align           | Makes the horizontal size of all selected instruments the same as the last selected. |
VisualizationTool Properties

Like other instruments, the VisualizationTool itself has got Properties. There are several configuration possibilities for the VisualizationTool, shown in Table 3.45. To view the property dialog box of the VisualizationTool, use the shortcut <CTRL-P> or double click on the background.

Table 3.45 VisualizationTool Properties

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit mode</td>
<td>Switches from Edit mode to Display mode.</td>
</tr>
<tr>
<td>Display Scrollbars</td>
<td>Switches the scrollbars on, off, or sets it to automatic mode.</td>
</tr>
<tr>
<td>Display Headline</td>
<td>Switches the headline on or off.</td>
</tr>
<tr>
<td>Backgroundcolor</td>
<td>Specifies the background color of the VisualizationTool.</td>
</tr>
<tr>
<td>Grid Mode</td>
<td>Specifies the grid mode. There are four possibilities: ‘Off,’ ‘Show grid but no snap,’ ‘Snap to grid without showing the grid,’ or ‘Show the grid and snap on it.’</td>
</tr>
<tr>
<td>Grid Size</td>
<td>Specifies the distance between two grid points (vertical, horizontal).</td>
</tr>
<tr>
<td>Grid Color</td>
<td>Specifies the color of the grid points.</td>
</tr>
<tr>
<td>Refresh Mode</td>
<td>Specifies the way the window will be refreshed. You may choose between: “Automatic, Periodical, Each access, Cpu Cycles”.</td>
</tr>
</tbody>
</table>

Instruments

When you first add an instrument, it is in “move mode”. Place it at the desired location on the workspace. All new instruments are set to their default attributes. To configure an instrument, right-click on an instrument and choose ‘Properties’, or double click on it. All instruments have the common attributes shown in Table 3.46.

Table 3.46 Instruments Properties Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Position</td>
<td>Specifies the X-coordinate of the upper left corner.</td>
</tr>
<tr>
<td>Y-Position</td>
<td>Specifies the Y-coordinate of the upper left corner.</td>
</tr>
<tr>
<td>Height</td>
<td>Specifies the instruments height.</td>
</tr>
</tbody>
</table>
Analog Instrument

The Analog instrument (Figure 3.86) represents the classical pointer instrument, also known as speedometer, voltage meter...

**Figure 3.86 Analog Instrument**

Analog instrument attributes are shown in the Table 3.47.
## Debugger Components

### Visualization Utilities

#### Table 3.47 Analog Instrument Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Display Value</td>
<td>Defines the zero point of the indicator. The values below this definition will not be displayed.</td>
</tr>
<tr>
<td>High Display Value</td>
<td>Defines the highest position of the indicator. It defines the value on which the indicator reads 100%.</td>
</tr>
<tr>
<td>Indicatorlength</td>
<td>Defines the length of the small indicator. The minimal value is set to 20.</td>
</tr>
<tr>
<td>Indicator</td>
<td>Defines the color of the indicator. The default color is red.</td>
</tr>
<tr>
<td>Marks</td>
<td>Defines the color of the marks. The default color is black.</td>
</tr>
</tbody>
</table>

#### Bar Instrument

Using the Bar instrument, values are displayed by a bar strip. This instrument (See Figure 3.87) may be used as a position state of a water tank.

#### Figure 3.87 Bar Instrument

Bar instrument attributes are shown in the Table 3.48

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Display Value</td>
<td>Defines the zero point of the indicator. The values below this definition will not be displayed.</td>
</tr>
<tr>
<td>High Display Value</td>
<td>Defines the highest position of the indicator. It defines the value on which the indicator reads 100%.</td>
</tr>
<tr>
<td>Bardirection</td>
<td>Sets the desired direction of the bar that displays the value.</td>
</tr>
<tr>
<td>Barcolor</td>
<td>Specifies the color of the bar. Default color is red.</td>
</tr>
</tbody>
</table>
Bitmap Instrument

You can use the Bitmap instrument to give a special look to your visualization, or to display a warning picture.

**Figure 3.88** Bitmap Instrument

Additionally, it can also be used as a bitmap animation. Its attributes are shown in the Table 3.49.

**Table 3.49** Bitmap Instrument Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filename</td>
<td>Specifies the location of the bitmap. With the button behind, you can browse for files.</td>
</tr>
<tr>
<td>AND Mask</td>
<td>Performs a bitwise-AND operation with this value. AND the value of the selected port. Default value is 0.</td>
</tr>
<tr>
<td>EQUAL Mask</td>
<td>This value is compared to the result of the AND operation. The bitmap is displayed only if both values are the same. Default value is 0.</td>
</tr>
</tbody>
</table>

In general, for showing the bitmap, following condition has to be true:

\[(\text{port\_memory} \& \text{ANDmask}) == \text{EQUALmask}\]

A practical example about using the AND and EQUAL masks is following example:

You want to show in the visualization a taillight of a car. For this you need bitmaps (e.g. from a digital camera) of all possible states of the taillight (e.g. flasher on, brake light on, etc.). Usually the status of all lamps are encoded into a port or memory cell in your application, and each bit in this cell describes if a lamp is on or not. E.g. bit 0 says that the flasher is on, where bit 1 says that the brake light is on. So for your simple application you need following bitmaps with their settings:

- no light on bitmap: AND mask 3, EQUAL mask 0
- flasher on bitmap: AND mask 3, EQUAL mask 1
- brake light on bitmap: AND mask 3, EQUAL mask 2
- brake and flasher light on: AND mask 3, EQUAL mask 3
**Debugger Components**

**Visualization Utilities**

### DILSwitch Instrument

The DILSwitch instrument is also known as Dual-in-Line Switch (Figure 3.89). It is mainly used for configuration purpose. You can use it for viewing or setting bits of one to four bytes.

![DILSwitch Instrument](image)

Figure 3.89  DILSwitch Instrument

DILSwitch instrument attributes are listed in the Table 3.50.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display 0/1</td>
<td>When enabled, displays the value of the bit under each plot of the DILSwitch instrument.</td>
</tr>
<tr>
<td>Switch Color</td>
<td>Specifies the color of the switch.</td>
</tr>
</tbody>
</table>

### Knob Instrument

The Knob instrument is normally known as an adjustment instrument. For example, in can simulate the volume control of a radio (Figure 3.90).

![Knob Instrument](image)

Figure 3.90  Knob Instrument

Knob instrument attributes are shown in the Table 3.51.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low Display Value</td>
<td>Defines the zero point of the indicator. The values below this definition will not be displayed.</td>
</tr>
<tr>
<td>High Display Value</td>
<td>Defines the highest position of the indicator. It defines the value on which the indicator reads 100%.</td>
</tr>
</tbody>
</table>
LED Instrument

The LED instrument is used for observing one definite bit of one byte (Figure 3.91). There are only two states: On and Off.

Figure 3.91 Led Instrument

LED instrument attributes are shown in Table 3.52.

Table 3.52 LED Instrument Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitnumber to Display</td>
<td>Defines the bit of the given byte to be displayed.</td>
</tr>
<tr>
<td>Color if Bit = 1</td>
<td>Defines the color if the given bit is set.</td>
</tr>
<tr>
<td>Color if Bit = 0</td>
<td>Defines the color if the given bit is not set.</td>
</tr>
</tbody>
</table>

7-Segment Display Instrument

This is the well known 7-Segment Display instrument for numbers and characters. It has seven segments and one point. These eight units represent eight bits of one byte (Figure 3.92).

Figure 3.92 7-Segment Display Instrument

7 Segment Display instrument attributes are shown in Table 3.53.
Debugger Components
Visualization Utilities

Switch Instrument
Use the Switch instrument to set or view a definite bit (Figure 3.93). The Switch instrument also provides an interesting debugging feature: you can let it simulate bounces, and thus check whether your algorithm is robust enough. Four different looks of the switch are available: slide switch, toggle switch, jumper or push button.

Figure 3.93 Switch Instrument

Switch instrument attributes are shown in Table 3.54.

Table 3.54 Switch Instrument Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitnumber to Display</td>
<td>Specifies the number of the bit you want to display.</td>
</tr>
<tr>
<td>Display 0/1</td>
<td>Enables to display the value of the bit in its upper left corner.</td>
</tr>
<tr>
<td>Top Position is</td>
<td>Specifies if the 'up' position is either zero or one. Especially useful to easily transform the push button into a reset button.</td>
</tr>
</tbody>
</table>
Text Instrument

The Text instrument has several functions: Static Text, Value, Relative Value, and Command (Figure 3.94).

Table 3.54 Switch Instrument Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind of Switch</td>
<td>Changes the look of the instrument. Following kinds of switches are available: Slide Switch, Toggle Switch, Jumper, Push Button. The behavior of the Push Button slightly differs from the others, since it returns to its initial state as soon as it has been released.</td>
</tr>
<tr>
<td>Switch Color</td>
<td>Specifies the color of the switch.</td>
</tr>
<tr>
<td>Bounces</td>
<td>If enabled, gives access to the following other attributes to configure the way the switch will bounce.</td>
</tr>
<tr>
<td>Nb Bounces</td>
<td>Specifies the number of bounces before stabilization.</td>
</tr>
<tr>
<td>Bounces on Edge</td>
<td>Specifies whether the switch will bounce on falling, rising or both edges.</td>
</tr>
<tr>
<td>Type of Unit</td>
<td>Synchronizes the frequency of the bouncing either on the timer of your host machine, or on CPU cycles.</td>
</tr>
<tr>
<td>Pulse Width (100ms)</td>
<td>Defines the duration of one bounce. This attribute should be filled in if you chose “Host Periodical” in the “Type of Unit” attribute.</td>
</tr>
<tr>
<td>CPU Count</td>
<td>This attribute represents the number of CPU cycles to reach before the switch changes its state. It should be filled in if you chose “CPU Cycles” in the “Type of Unit” attribute.</td>
</tr>
</tbody>
</table>

Figure 3.94 Text Instrument

Value: 

Please use ‘Text Mode’ to switch between the five available modes. Text instrument common attributes are shown in the Table 3.55.
Debugger Components
Visualization Utilities

Table 3.55  Text Instrument Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Mode</td>
<td>Specifies the mode. Choose among four modes: Static Text, Value, Relative Value, and Command</td>
</tr>
<tr>
<td>Displayfont</td>
<td>Defines the desired font. All installed Windows fonts are available.</td>
</tr>
<tr>
<td>Horiz. Text Alignment</td>
<td>Specifies the desired horizontal alignment of the text in the given bounding box.</td>
</tr>
<tr>
<td>Vert. Text Alignment</td>
<td>Specifies the desired vertical alignment of the text in the given bounding box.</td>
</tr>
<tr>
<td>Textcolor</td>
<td>Defines the color of the given text.</td>
</tr>
</tbody>
</table>

'Static Text' is used for adding descriptions on the workspace. Its attributes are shown in Table 3.56.

Table 3.56  Static Text Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Description</td>
<td>Contains the text to be displayed.</td>
</tr>
</tbody>
</table>

'Value' is used for displaying a value in different ways (decimal, hexadecimal, octal, or binary). Its attributes are shown in Table 3.57.

Table 3.57  Value Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Description</td>
<td>Contains the additional description that will be displayed in front of the value. Add a colon and/or space as you wish. The default setting is &quot;Value: &quot;</td>
</tr>
<tr>
<td>Format mode</td>
<td>Defines the format. Choose among this list: Decimal, Hexadecimal, Octal, and Binary formats.</td>
</tr>
</tbody>
</table>

'Relative Value' is used for showing a value in a range of 0 up to 100% or 1000‰. Its attributes are shown in Table 3.58.
With this instrument mode you can specify a command that will be executed by clicking on this field. For more information about commands, read the chapter 'Debugger Commands'. Command mode attributes are shown in Table 3.59.

**Table 3.58 Relative Value Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Description</td>
<td>Add the additional description text to be displayed in front of the value. Add a colon and/or space if desired. The default setting is &quot;Value:&quot;</td>
</tr>
<tr>
<td>Low Display Value</td>
<td>Fixes the minimal value that will represent 0%. Values below this definition will appear as an error: #ERROR.</td>
</tr>
<tr>
<td>High Display Value</td>
<td>Fixes the maximal value that will represent 100%. Values above this definition will appear as an error: #ERROR.</td>
</tr>
<tr>
<td>Relative Mode</td>
<td>Switches between percent and permill.</td>
</tr>
</tbody>
</table>

\'{Command}\'. This mode is the same as command, but with one difference: The returned value will be shown as text instead of \'Field Description\'. Its attributes are shown in Table 3.60.

**Table 3.59 Command Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Description</td>
<td>Contains the text that will be displayed on the button.</td>
</tr>
<tr>
<td>Command</td>
<td>Contains the command-line command to be executed after pressing the button.</td>
</tr>
</tbody>
</table>

\'{CMD Callback}\' This mode is the same as command, but with one difference: The returned value will be shown as text instead of \'Field Description\'. Its attributes are shown in Table 3.60.

**Table 3.60 CMD Callback Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Description</td>
<td>Warning: there is no use to fill out this field as the text will be overwritten the first time you execute the specified command.</td>
</tr>
<tr>
<td>Command</td>
<td>Contains the command line command to be executed after pressing the button.</td>
</tr>
</tbody>
</table>
Drop Into:
In Edit mode, the drag and drop functionality supplies a very easy way to automatically configure an instrument.

To assign a variable, simply drag it from the Data Window onto the instrument. The “kind of Port” is immediately set on “Memory” and the “Port to Display” field contains now the address of the variable. Now repeat the drag-and-drop on a bare portion of the VisualizationTool window: a new text instrument is created, with correct port configuration.

Some other components allow this operation:
- The memory window: select bytes and drag-and-drop them onto the instrument.
- The Inspector component: pick an object from the object pool.

Demo Version Limitations
If you work in demo mode, you will only be able to load one VisualizationTool window. The number of instruments is limited to three.
Control Points

This chapter provides an overview of the debugger control points: Breakpoints, Watchpoints, and Markpoints. Click any of the following links to jump to the corresponding section of this chapter:

- **Introduction**
- **Breakpoints**
- **Setting Breakpoints**
- **Watchpoints**
- **Setting Watchpoints**
- **Markpoints**
- **Setting Markpoints**
- **Halting on a Control Point**

### Introduction

There are three kinds of control points:

- **Breakpoints** (also called data breakpoints): Breakpoints are located at an address. They can be temporary or permanent.
- **Watchpoints**: Watchpoints are located at a memory range. They start from an address, have a range, and a read and/or write state.
- **Markpoints**: Are marked points of observation that can be jumped to by the programmer. They can be located in data, source or memory.

You can set or disable a control point, set a condition and an optional command, and set the current count and counting interval, using the popup menu of the Source, Memory or Assembly window.

You can see and edit control point characteristics through the three tabs of the Controlpoints Configuration Window: Breakpoint, Watchpoints and Markpoints tabs. These three tabs have common properties that allow you to interactively perform the following operations on control points:

- Selecting a single control point from a list box and clicking **Delete**.
- Selecting multiple control points from a list box and clicking **Delete**.
Control Points

Breakpoints

- Enabling/disabling a selected control point by checking or unchecking the related checkbox.
- Enabling/disabling multiple control points by checking or unchecking the related checkbox.
- Enter or modify the condition of a selected control point.
- Enabling/disabling the condition of a selected control point by checking/unchecking the related checkbox.
- Enter or modify the command of a selected control point.
- Enabling/disabling the command of a selected control point by checking/unchecking the related checkbox.
- Enabling/disabling multiple control point commands by selecting control points and checking/unchecking the related checkbox.
- Modifying the counter and/or limit of a single control point.

With breakpoints, the following operations are also available:

- Enabling/disabling halting on a single temporary breakpoint by checking/unchecking the matching checkbox.
- Enabling/disabling halting on multiple temporary breakpoints by checking/unchecking the matching checkbox.

With watchpoints, the following operations are also available:

- Enabling/disabling halting on a single read and/or write access by checking/unchecking the corresponding checkboxes.
- Enabling/disabling halting on multiple read and/or write accesses by checking/unchecking the corresponding checkboxes.
- Defining the memory range controlled by the watchpoint.

Breakpoints

Breakpoints are control points associated with a PC value. That is, program execution is stopped as soon as the PC reaches the value defined in a breakpoint. The Debugger supports four different types of breakpoints:

- Temporary breakpoints, which are activated next time the instruction is executed.
- Permanent breakpoints, which are activated each time the instruction is executed.
- Counting breakpoints, which are activated after the instruction has been executed a certain number of times.
- Conditional breakpoints, which are activated when a given condition is TRUE.
Breakpoints are controlled through the Breakpoints tab of the Controlpoints Configuration window. This window can be opened through the Source Window Popup menu, as described below:

**Figure 4.1 Source Window Popup Menu**

1. Point at a C statement in the Source window, and click the right mouse button.
2. Select **Show Breakpoints** from this menu.

The Controlpoints Configuration Window (Breakpoints Tab) is opened. The Breakpoints tab of this window is shown in Figure 4.2.
Figure 4.2 Controlpoints Configuration Window (Breakpoints Tab)

Breakpoints Tab

The Controlpoints Configuration Window (Breakpoints Tab) contains:

- List box that displays the list of currently defined breakpoints
- “Breakpoint:” group box that displays the address of the currently selected breakpoint, name of procedure in which the breakpoint has been set, state of the breakpoint (disabled or not), and type of breakpoint (temporary or permanent).
- “Condition:” group box that displays the condition string to evaluate, and the state of the condition (disabled or not).
- “Command:” group box that displays the command string to execute and the state of the command (disable or continue after command execution).
- “Counter:” group box that displays the current value of the counter and interval value of the counter.
Control Points

Breakpoints

NOTE Current and Interval values are limited to 2,147,483,647; if entering a number greater than this value, a beep occurs and the character is not appended. When the Interval value is changed, the Counter value is automatically set to the Interval value.

- “Delete” button to remove the currently selected breakpoint.
- “Update” button to Update all modifications in the dialog.
- “Add” button to add new breakpoints; specify the Address (in hexadecimal when Hex format is checked, or as an expression when Hex format is unchecked).
- “OK” button to validate all modifications.
- “Cancel” button to ignore all modifications.
- “Help” button to open related help information.

Multiple Selections in List Box

The list box allows you to select multiple consecutive breakpoints by clicking the first breakpoint then pressing the Shift key and clicking the last breakpoint you want to select.

The list box allows you to select multiple breakpoints that are not consecutive by clicking the first breakpoint then pressing the Ctrl key and clicking another breakpoint.

When multiple breakpoints are selected in the list box, the name of the group box Breakpoint: is changed to Selected Breakpoints:

When selecting multiple breakpoints, the Address (hex), Name:, Condition:, Disable for condition, Command, Current:, and Interval: controls are disabled.

When multiple breakpoints are selected, the Disable and Temporary controls in the Selected breakpoints: group box are enabled and Disable in the Command: group box is enabled.

Checking Expressions

You can enter an expression in the Condition: group edit box. The syntax of the expression is checked when you select another breakpoint in the list box or click OK. The syntax is parameters \(=\) expression. For a register condition the syntax is \$RegisterName \(=\) expression.

If a syntax error has been detected, a message box is displayed:
Incorrect Condition. Do you want to correct it?.

If you click OK, correct the error in the condition edit box.
If you click Cancel, the Condition: edit box is cleared.
Saving Breakpoints

The Debugger provides a way to store all defined breakpoints of the currently loaded application (.ABS file) into the matching breakpoints file. The matching file has the same name as the loaded .ABS file but its extension is .BPT (for example, the FIBO.ABS file has a breakpoint file called FIBO.BPT). This file is generated in the same directory as the .ABS file. This is a text file, in which a sequence of commands is stored. This file contains the following information.

- The Save & Restore on load flag (Save & Restore on load checkbox in the Controlpoints Configuration Window (Breakpoints Tab)): the SAVEBP command is used: SAVEBP on when checked, SAVEBP off when unchecked.

**NOTE** For more information about this, see the SAVEBP command.

- List of defined breakpoints: the BS command is used, as shown in Listing 4.1.

**Listing 4.1  Breakpoint (.BPT) File Syntax**

```
BS address [P|T[ state]] [;cond="condition" [ state]]
[;cmd="command" [ state]] [;cur=current [ inter=interval]]
[;cdSz=codeSize[ srSz=sourceSize]]
```

In the code above:

- **address** is the address where the breakpoint is to be set. This address is specified in ANSI C format. address can also be replaced by an expression as shown in the example below.
- **P**, specifies the breakpoint as a permanent breakpoint.
- **T**, specifies the breakpoint as a temporary breakpoint. A temporary breakpoint is deleted once it is reached.
- **state** is **E**, **D** or **C** where **E** is for enabled (state is set by default to **E** if nothing is specified), **D** is for disabled and **C** for Continue.
- **condition** is an expression. It matches the Condition field in the Controlpoints Configuration Window (Breakpoints Tab) for conditional breakpoint.
- **command** is any debugger command. It matches the Command field in the Controlpoints Configuration Window (Breakpoints Tab), for associated commands.
- **current** is an expression. It matches the Current field (Counter) in the Controlpoints Configuration Window (Breakpoints Tab), for counting breakpoints.
- **interval** is an expression. It matches the Interval field (Counter) in the Controlpoints Configuration Window (Breakpoints Tab), for counting breakpoints.
- **codeSize** is an expression. It is usually a constant number to specify (for security) the code size of a function where a breakpoint is set. If the size specified does not match the
size of the function currently loaded in the .ABS file, the breakpoint is set but it is disabled.

**sourceSize** is an expression. It is usually a constant number to specify (for security) the source (text) size of a function where a breakpoint is set. If the size specified does not match the size of the function in the source file, the breakpoint is set but it is disabled.

- If **Save & Restore on load** is checked and the user quits the Debugger or loads another .ABS file, all breakpoints will be saved.
- If **Save & Restore on load** is unchecked (default), only this flag (SAVEBP off) is saved.

**Breakpoint File (.BPT) Example**

**Case 1:** if FIBO.ABS is loaded, and **Save & Restore on load** was checked in a previous session of the same .ABS file, and breakpoints have been defined, the FIBO.BPT looks as shown in Listing 4.2.

Listing 4.2  Breakpoint File with **Save & Restore on load** Checked.

```
savebp on
BS &fibo.c:Fibonacci+19 P E; cond = "fibo > 10" E; cdSz = 47 srSz = 0
BS &fibo.c:Fibonacci+31 P E; cdSz = 47 srSz = 0
BS &fibo.c:main+12 P E; cdSz = 42 srSz = 0
BS &fibo.c:main+21 P E; cond = "fiboCount==5" E; cmd = "Assembly < spc 0x800" E; cdSz = 42 srSz = 0
```

**Case 2:** if FIBO.ABS is loaded, and **Save & Restore on load** was unchecked in a previous session of the same .ABS file and breakpoints have been defined, the FIBO.BPT looks as shown below:

```
savebp on
```

Only the flag has been saved and breakpoints have been removed.

**NOTE**  If only one or few functions differ after a recompilation, not all BP will be lost. To achieve that, BP are disabled only if the size of a function has changed. The size of a function is evaluated in bytes (when it is compiled) and in characters (number of characters contained in the function source text). When a .ABS file is loaded and the matching .BPT file exists, for each BS command, the Debugger checks if the code size (in bytes) and the source size (in characters) are different in the matching function (given by the symbol table). If there is a difference, the breakpoint will be set and disabled. If there is no difference, the breakpoint will be set and enabled.
Control Points
Setting Breakpoints

NOTE For more information about this syntax, see BS and SAVEBP commands.

Setting Breakpoints

The Debugger supports different types of breakpoints:

- Temporary breakpoints, which are activated next time the instruction is executed.
- Permanent breakpoints, which are activated each time the instruction is executed.
- Counting breakpoints, which are activated after the instruction has been executed a certain number of times.
- Conditional breakpoints, which are activated when a given condition is TRUE.

Breakpoints may be set in a Source or Assembly component window.

Positions Where a Breakpoint Is Definable

A compound statement is one that can be split into several base instructions. When using a high level language some compound statements can be generated, as shown in the following example.

Figure 4.3 Source and Assembly Windows

The Debugger helps you detect all positions where you can set a breakpoint.
1. Right-click in the Source component. The Source Popup Menu is displayed on the screen.

2. Choose **Marks** from the Popup Menu. All statements where a breakpoint can be set are identified by a special red inverted check mark:

To remove the breakpoint marks, right-click in the Source component and choose **Marks** again.

**Temporary Breakpoints**

Temporary breakpoints are activated next time the instruction is executed. A temporary breakpoint is recognized by the following icon:

**Setting Temporary Breakpoints**

**A. Using the Source Window Popup Menu:**

1. Point at a C statement in the Source window and right-click. The Source Popup Menu is displayed.

2. Choose **Run To Cursor** from the Popup Menu. The application continues execution and stops before executing the statement. You have executed a temporary breakpoint.

**B. Holding down the left mouse button, pressing the T key:**

1. Point at a C statement in the Source window, and holding down the left mouse button, press the T key.

2. A temporary breakpoint is defined.

3. Choose **Run To Cursor** from the Popup Menu. The application continues execution and stops before executing the statement.

Temporary breakpoints are automatically deleted once they have been activated. If you continue program execution, it will no longer stop on the statement that contained the temporary breakpoint.
Permanent Breakpoints

Permanent breakpoints are activated each time the instruction is executed. A permanent breakpoint is recognized by the following icon:

Setting Permanent Breakpoints

A. Using the Source Window Popup Menu:

1. Point at a C statement in the Source window and right-click. The Source Popup Menu is displayed.
2. Select Set BreakPoint from the Popup Menu. A permanent breakpoint mark is displayed in front of the selected statement.

B. Holding down the left mouse button, pressing the P key:

1. Point at a C statement in the Source window, and holding down the left mouse button, press the P key.
2. A permanent breakpoint mark is displayed in front of the selected statement.

Once a permanent breakpoint has been defined, you can continue program execution. The application stops before executing the statement. Permanent breakpoints remain active until they are disabled or deleted.
Control Points
Setting Breakpoints

Counting Breakpoints
Counting breakpoints are activated after the instruction has been executed a certain number of times. A Counting breakpoint is recognized by the following icon:

Setting Counting Breakpoints
Counting breakpoints can only be set using the Controlpoints Configuration Window (Breakpoints Tab). There are two ways to set a counting breakpoint:

A. Holding down the left mouse button, pressing the S key:
1. Point at a C statement in the Source window, and holding down the left mouse button, press the S key.
2. The Controlpoints Configuration window with the Breakpoints tab is opened.
3. A new breakpoint is inserted in the list of breakpoints defined in the application.
4. Select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints at the top of the tab.
5. In the Counter: group of this tab specify the interval for the breakpoint detection in the Interval: field.
6. Then close the window by clicking the OK button.

B. Using the Source Popup Menu:
1. Point at a C statement in the Source window and right-click. The Source Popup Menu is displayed.
2. Choose Set BreakPoint from the Popup Menu. A breakpoint is defined on the selected instruction.
3. Point in the Source window and right-click again.
4. Choose Show Breakpoints from the Popup Menu. The Controlpoints Configuration Window (Breakpoints Tab) is displayed.
5. Select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints at the top of the tab.
6. In the Counter: group of this tab specify the interval for the breakpoint detection in the Interval: field.
7. Then close the window by clicking the OK button.

If you continue program execution, the content of the Current: field is decremented each time the instruction containing the breakpoint is reached. When Current is equal to 0, the application stops. If the checkbox Temporary is unchecked (not a temporary breakpoint),
Current is reloaded with the value stored in Interval: in order to enable the counting breakpoint again.

Conditional Breakpoints
Conditional breakpoints are activated when a given condition is TRUE. A conditional breakpoint is recognized by the following icon:

Setting Conditional Breakpoints
Conditional breakpoints can only be set from the Controlpoint Configuration window’s Breakpoints tab. There are two ways to set a conditional breakpoint:

A. Holding down the left mouse button, pressing the S key:
1. Point at a C statement in the Source Component window, and holding down the left mouse button, press the S key.
2. The Controlpoints Configuration Window (Breakpoints Tab) is opened and a new breakpoint is inserted in the list of breakpoints defined in the application.
3. Select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints.
4. Specify the condition for breakpoint activation in the Condition: group Condition box. The condition must be specified using the ANSI C syntax (Example counter == 7). You can use register values in the breakpoint condition field with the following syntax: $RegisterName (Example $RX == 0x10)
5. Close the window by clicking OK.

B. Using the Source Window Popup Menu:
1. Point at a C statement in the Source Component window and right-click. The Source Popup Menu is displayed.
2. Select Set BreakPoint from the Popup Menu. A breakpoint is defined on the selected instruction.
3. Point in the Source Component window and right-click. The Source Popup Menu is displayed.
4. Select Show Breakpoints from the Popup Menu. The Controlpoints Configuration Window (Breakpoints Tab) is opened and a new breakpoint is inserted in the list of breakpoints defined in the application.
5. Select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints.
6. Specify the condition for breakpoint activation in the **Condition** group Condition box. The condition must be specified using the ANSI C syntax (Example `counter == 7`). You can use register values in the breakpoint condition field with the following syntax: `$RegisterName` (Example `$RX == 0x10`)

7. Close the window by clicking **OK**.

If you continue program execution, the condition is evaluated each time the instruction containing the conditional breakpoint is reached. When the condition is **TRUE**, the application stops.

### Deleting Breakpoints

The Debugger provides three ways to delete a breakpoint:

**A. Using Delete Breakpoint from Source Popup Menu**

1. In the Source component window, point at a C statement where a breakpoint has previously been defined and right-click. The Source Popup Menu is displayed.
2. Choose **Delete** Breakpoint from the Popup Menu. The breakpoint is deleted.

**B. Holding down the left mouse button, pressing the D key:**

1. Point at a C statement in the Source Component window where a breakpoint has previously been defined, and holding down the left mouse button, press the D key.
2. The breakpoint is deleted.

**C. Choosing Show Breakpoints... from Source Popup Menu**

1. Point in the Source Component window and right-click. The Source Popup Menu is displayed.
2. Choose **Show Breakpoints** from the Popup Menu. The **Breakpoints Setting** dialog is displayed.
3. In the list of defined breakpoints, select the breakpoint to delete.
4. Click **Delete**. The selected breakpoint is removed from the list of defined breakpoints.
5. Click **OK** to close the **Breakpoints Setting** dialog box.

The icon associated with the deleted breakpoint is removed from the source component.
Control Points
Setting Breakpoints

Associate a Command with a Breakpoint

Each breakpoint (temporary, permanent, counting or conditional) can be associated with a debugger command. This command can be specified in the Breakpoints tab of the Controlpoints Configuration window. To open this window:

Choose Show Breakpoints... from Source Window Popup Menu.

1. Point in the Source Component Window and right-click. The Source Popup Menu is displayed.
2. Choose Show Breakpoints from the Popup Menu. The Controlpoints Configuration window with the Breakpoints tab displayed appears.

In the Breakpoints tab of the Controlpoints Configuration window:

1. You can select the breakpoint to modify by clicking on the corresponding entry in the list of defined breakpoints.
2. You can enter the command in the Command field. The command is a single debugger command (at this level, the commands G, GO and STOP are not allowed). A command file can be associated with a breakpoint using the command CALL or CF (Example: CF breakCmd.cmd).
3. Click OK to close the window.

When the breakpoint is detected, the command is executed and the application stops. The Continue check button of the Controlpoints Configuration window allows the application to continue after the command is executed.

Demo Version Limitations

Only 2 breakpoints can be set.
Watchpoints

Watchpoints are control points associated with a memory range. Program execution stops when the memory range defined by the watchpoint has been accessed. The Debugger supports different types of watchpoints:

- Read Access Watchpoints, which are activated when a read access occurs inside the specified memory range.
- Write Access Watchpoints, which are activated when a write access occurs inside the specified memory range.
- Read/Write Access Watchpoints, activated when a read or write access occurs inside the specified memory range.
- Counting Watchpoints, activated after a specified number of accesses occur inside the memory range.
- Conditional Watchpoints, activated when an access occurs inside the memory range and a given condition is TRUE.

Watchpoints are controlled through the Controlpoints Configuration Window (Watchpoints Tab). This window can be opened through the Memory or Data component window popup menu, as described below:

To open the Controlpoints Configuration window with the Watchpoints tab exposed:
1. Position your cursor in either the Memory or Data component window.
2. Press the right mouse button.
3. Select Show Watchpoints from either menu.
4. Click the left mouse button.

The ControlPoints Configuration window appears. The Watchpoints tab of this window is shown in Figure 4.6.
Control Points

Watchpoints

Figure 4.4 Memory Popup Menu

Figure 4.5 Data Popup Menu
Watchpoints Tab

The Watchpoints tab of the Controlpoints Configuration window contains:

- List box that displays the list of currently defined watchpoints.
- "Watchpoint:" group box that displays the address of the currently selected watchpoint, size of the watchpoint, name of the procedure or variable on which the watchpoint has been set, state of the watchpoint (disabled or not), read access of the watchpoint (enabled or not) and write access of the watchpoint (enabled or not).
- "Condition:" group box that displays the condition string to evaluate and the state of the condition (disabled or not).
- "Command:" group box that displays the command string to execute and state of the command (disabled or continue after command execution).

Update button to Update all modifications in the dialog.

Figure 4.6 Controlpoints Configuration Window (Watchpoints Tab)
Control Points
Watchpoints

- **Delete** button to remove currently selected watchpoint and select the watchpoint that is below the removed watchpoint.
- **OK**: button to validate all modifications.
- **Add** button to add new watchpoints; specify the Address in hexadecimal when **Hex format** is checked or as an expression when **Hex format** is unchecked.
- **Counter**: group box that displays the current value of the counter and interval value of the counter.

**NOTE**
Current and Interval values are limited to 2,147,483,647. A beep occurs and the character is not appended, if a number greater than this value is entered.

**NOTE**
When the Interval value is changed, the Counter value is automatically set to the Interval value.

- **Cancel** button to ignore all modifications.
- **Help**: button to display help file and related help information.

**Multiple Selections**
For watchpoints, you can do multiple selections in the Watchpoints tab of the Controlpoints Configuration window using the **Shift** and **Ctrl** keys.

When multiple watchpoints in the list box are selected, the name of the group box “**Watchpoint:**” is changed to “**Selected Watchpoints:**”.

When multiple watchpoints are selected, the **Address** (hex), **Size**, **Name**, **Condition**, **Disable** for condition, **Command**, **Current**, and **Interval**: controls are disabled.

When multiple watchpoints are selected in the list box, the **Disable**, **Read** and **Write** controls in the **Selected watchpoints**: group box are enabled.

When multiple watchpoints are selected, **Disable** in the **Command**: group box is enabled.

Click **Delete** when multiple watchpoints are selected to remove watchpoints from the list box.

**Checking Syntax**
You can enter an expression in the Condition group edit box. The syntax of the expression will be checked when you select another watchpoint in the list box or by clicking **OK**.

If a syntax error has been detected, a message box is displayed:

“**Incorrect Condition. Do you want to correct it?**”

Click **OK** to correct the error in the condition edit box.
Click **Cancel** to clear the condition edit box.

### Setting Watchpoints

Watchpoints may be set in a Data or Memory window.

| NOTE | Due to hardware restrictions, the watchpoint function might not be implemented on hardware connections. |

### Setting a Read Watchpoint

A green vertical bar is displayed in front of a variable associated with a read access watchpoint.

The Debugger provides two ways to define a read access watchpoint:

**Using the Data Popup Menu:**

1. Point at a variable in the Data window and right-click. The [Data Popup Menu](#) is displayed.
2. Choose **Set Watchpoint** from the Popup Menu. A **Read/Write** Watchpoint is defined.
3. Point in the Data window and right-click. The Data Popup Menu is displayed.
4. Choose **Show WatchPoints** from the Popup Menu. The Controlpoints Configuration window Watchpoints tab is displayed.
5. Select the watchpoint you want to define as **read** access from the list.
6. Select the **Read** type in the dropdown box.
7. A read access watchpoint is defined for the selected variable.

**Using the Left Mouse Button and Pressing the R Key:**

1. Point at a variable in the Data window and holding down the left mouse button, press the R key.
2. A read access watchpoint is defined for the selected variable.

Once a read access watchpoint has been defined, you can continue program execution. The application stops after detecting the next read access on the variable. Read access watchpoints remain active until they are disabled or deleted.
Setting a Write Watchpoint
A red vertical bar is displayed in front of a variable associated with a write access watchpoint.

The Debugger provides two ways to define a write access watchpoint:

Using the Data Popup Menu:
1. Point at a variable in the Data window and right-click. The Data Popup Menu is displayed.
2. Choose Set Watchpoint from the Popup Menu. A Read/Write Watchpoint is defined.
3. Point in the Data Component Window and right-click. The Source Popup Menu is displayed.
4. Choose Show WatchPoints from the Popup Menu. The Controlpoints Configuration window Watchpoints tab is displayed.
5. Select the watchpoint you want to define as write access from the list.
6. Select the Write type in the dropdown box.
7. A write access watchpoint is defined for the selected variable.

Using the Left Mouse Button and Pressing the W Key:
1. Point at a variable in the Data window and holding down the left mouse button, press the W key.
2. A write access watchpoint is defined for the selected variable.

Once a write access watchpoint has been defined, you can continue program execution. The application stops after the next write access on the variable. Write access watchpoints remain active until they are disabled or deleted.

Defining a Read/Write Watchpoint
A yellow vertical bar is displayed in front of a variable associated with a read/write access watchpoint.

The Debugger provides two ways to define a read/write access watchpoint:

Using the Data Popup Menu:
1. Point at a variable in the Data window and right-click. The Data Popup Menu is displayed.
2. Choose Set Watchpoint from the Popup Menu. A Read/Write Watchpoint is defined.
Using the Left Mouse Button and Pressing the B Key:
1. Point at a variable in the Data window and holding down the left mouse button, press the B key.
2. A read/write access watchpoint is defined for the selected variable.

Once a read/write access watchpoint has been defined, you can continue program execution. The application stops after the next read or write access on the variable. Read/write access watchpoints remain active until they are disabled or deleted.

Defining a Counting Watchpoint
A counter can be associated with any type of watchpoint (read, write, read/write).
The Debugger provides two ways to define a counting watchpoint:

Using the Data Popup Menu:
1. Point at a variable in the Data window and right-click. The Data Popup Menu is displayed.
2. Choose Set Watchpoint from the Popup Menu. A Read/Write Watchpoint is defined.
3. Point in the Data Component Window and right-click. The Source Popup Menu is displayed.
4. Choose Show WatchPoints from the Popup Menu. The Controlpoints Configuration window Watchpoints tab is displayed.
5. Select the watchpoint you want to define as a counting watchpoint.
6. From the dropdown box, select the type of access you want to track.
7. In the interval field, specify the interval count for the watchpoint.
8. Close the window by clicking OK. A counting watchpoint is defined for the selected variable.

Using the Left Mouse Button and Pressing the S Key:
1. Point at a variable in the Data window and holding down the left mouse button, press the S key. The Watchpoints tab of the Controlpoints Configuration window is displayed.
2. Select the watchpoint you want to define as a counting watchpoint from the list.
3. From the dropdown box, select the type of access you want to track.
4. In the interval field, specify the interval count for the watchpoint. Close the window by clicking OK. A counting watchpoint is defined for the selected variable.

If you continue program execution, the Current field is decremented each time an appropriate access on the variable is detected. When Current is equal to 0, the application
Defining a Conditional Watchpoint

A condition can be associated with any type of watchpoint described previously (read, write, read/write).

The Debugger provides two ways to define a conditional watchpoint:

Using the Data Popup Menu:
1. Point at a variable in the Data window and right-click. The Data Popup Menu is displayed.
2. Choose Set Watchpoint from the Popup Menu. A Read/Write Watchpoint is defined.
3. Point in the Data window and right-click. The Source Popup Menu is displayed.
4. Choose Show WatchPoints from the Popup Menu. The Controlpoints Configuration window Watchpoints tab is displayed.
5. Select the watchpoint you want to define as a conditional watchpoint.
6. From the dropdown box, select the type of access you want to track.
7. Specify the condition for the watchpoint in the Condition field. The condition must be specified using the ANSI C syntax (Example: counter == 7).
8. Close the window by clicking OK. A conditional watchpoint is defined for the selected variable.

Using the Left Mouse Button and Pressing the S Key:
1. Point at a variable in the Data window and holding down the left mouse button, press the S key. The Watchpoints tab of the Controlpoints Configuration window is displayed.
2. Select the watchpoint you want to define as a conditional watchpoint.
3. From the dropdown box, select the type of access you want to track.
4. Specify the condition for watchpoint activation in the Condition field. The condition must be specified using the ANSI C syntax (Example: counter == 7). You can use register values in the breakpoint condition field with the following syntax: $RegisterName (Example $RX == 0x10)
5. Close the window by clicking OK. A conditional watchpoint is defined for the selected variable.

If you continue program execution, the condition is evaluated each time an appropriate access on the variable is detected. When the condition is TRUE, the application stops.
Deleting a Watchpoint

The Debugger provides three ways to delete a watchpoint:

**Use Delete Breakpoint from Popup Menu:**
1. In the Data window, point to a variable where a watchpoint has been defined and right-click. The Data Popup Menu is displayed.
2. Select **Delete Watchpoint** from the Popup Menu. The watchpoint is deleted and the vertical bar in front of the variable is removed.

**Using the Left Mouse Button and Pressing the D Key:**
1. Point at a variable in the Data window and holding down the left mouse button, press the D key. The Watchpoints tab of the Controlpoints Configuration window is displayed.
2. The watchpoint is deleted and the vertical bar in front of the variable is removed.

**Choosing Show Watchpoints from Data Popup Menu:**
1. Point in the Data window and right-click. The Data Popup Menu is displayed.
2. Choose **Show Watchpoints** from the Popup Menu. The Watchpoints tab of the Controlpoints Configuration window is displayed.
3. Select the watchpoint you want to delete.
4. Click **Delete**. The selected watchpoint is removed from the list of defined watchpoints.
5. Click **OK** to close the window. The watchpoint is deleted and the vertical bar in front of the variable is removed.

Associate a Command with a Watchpoint

Each watchpoint type (read, write, read/write, counting, or conditional) can be associated with a debugger command. This command can be specified in the Watchpoints tab of the Controlpoints Configuration window. To open this window:

**Choosing Show Watchpoints... from Data Popup Menu:**
1. Point in the Data Component Window and right-click. The Data Popup Menu is displayed.
2. Select **Show Watchpoints** from the Popup Menu. The Watchpoints tab of the Controlpoints Configuration window is displayed.
3. Click on the corresponding entry in the list of defined breakpoints to select the watchpoint you want to modify.
Control Points
Setting Watchpoints

4. You can enter the command in the Command field. The command is a single
debugger command. At this level, the commands \texttt{G}, \texttt{GO} and \texttt{STOP} are not allowed.
A command file can be associated with a watchpoint using the commands \texttt{CALL} or
\texttt{CF} (Example CF breakCmd.cmd).

5. Click OK to close the window.

6. When the watchpoint is detected, the command is executed and the application will
stop at this point. The \texttt{Continue} check button allows the application to continue after
command execution.

Demo Version Limitations

Only two watchpoints can be set.
Markpoints

Watchpoints are control points associated with a source line, memory or data range. They provide the programmer with accessible program markers.

Program execution does NOT stop when the Source line, data or memory range defined by the markpoint has been accessed.

Markpoints are controlled through the Markpoint tab of the Controlpoints Configuration Window (Markpoints Tab). This window can be opened through the Source, Memory or Data window popup menu, as described below:

To open the Controlpoints Configuration window with the Markpoints tab exposed:

1. Position your cursor in either the Source, Memory or Data window.
2. Press the right mouse button.
3. Select Show Watchpoints from the window’s popup menu.
4. Click the left mouse button.

The ControlPoints Configuration window appears with the Markpoints tab of this window exposed, as shown in Figure 4.10.

Figure 4.7 Source Window Popup Menu
Control Points
Markpoints

Figure 4.8 Memory Popup Menu

Figure 4.9 Data Popup Menu
Markpoints Tab

The Markpoints tab of the Controlpoints Configuration window contains:

- List box that displays the list of currently defined markpoints.
- “Markpoint:” group box that displays the address of the currently selected markpoint, size of the markpoint, name of the procedure or variable on which the markpoint has been set, and type of the markpoint.
- “General” group box that contains a checkbox that allows you to save and restore the markpoint selected.
- Add button to add new markpoints. Specify the Address in hexadecimal when Hex format is checked or as an expression when Hex format is unchecked.
- Delete button to remove currently selected markpoint and select the markpoint that is below the removed markpoint.
- Update button to update all modifications in the window.
Control Points
Setting Markpoints

- **OK**: button to validate all modifications.
- **Cancel**: button to ignore all modifications.
- **Help**: button to display help file and related help information.

**Setting Markpoints**

Markpoints may be set in a Source, Data or Memory window.

**Setting a Source Markpoint**

A blue letter L is displayed in front of a code line associated with a markpoint. To define a markpoint in source code:

**Use the Source Popup Menu:**

1. Point at a code line in the Source window and right-click. The Source Window Popup Menu is displayed.
2. Choose **Set Markpoint** from the popup menu. A markpoint is defined at the beginning of the line.
3. Point in the Source window and right-click. The Source Popup Menu is displayed.
4. Choose **Show WatchPoints** from the popup menu. The Controlpoints Configuration Window Markpoints Tab is displayed.
5. Make any modifications to the markpoint you have installed, or any other markpoints listed.
6. Click OK to close the window.

**Setting a Data Markpoint**

A blue letter L is displayed in front of a variable associated with a markpoint. To define a data range markpoint:

**Use the Data Popup Menu:**

1. Point at a variable in the Data window and right-click. The Data Popup Menu is displayed.
2. Choose **Set Markpoint** from the popup menu. A markpoint is defined at the beginning of the data range selected.
3. Point in the Data window and right-click. The Data Popup Menu is displayed.
4. Choose **Show WatchPoints** from the popup menu. The Controlpoints Configuration Window Markpoints Tab is displayed.
5. Make any modifications to the markpoint you have installed, or any other markpoints listed.
6. Click OK to close the window.

Setting a Memory Markpoint
A blue letter L is displayed in front of a memory range associated with a markpoint.
To define a Memory markpoint:

Use the Memory Popup Menu:
1. Point at a line in the Memory window and right-click. The Memory Popup Menu is displayed.
2. Choose Set Watchpoint from the Popup Menu. A Markpoint is defined.
3. Point in the Memory window and right-click. The Memory Popup Menu is displayed.
4. Choose Show WatchPoints from the Popup Menu. The Controlpoints Configuration Window Markpoints Tab is displayed.
5. Make any modifications to the markpoint you have installed, or any other markpoints listed
6. Click OK to close the window.

Deleting a Markpoint
To delete a markpoint:

Using the Left Mouse Button and Pressing the D Key:
1. Point at the markpointed variable in the Data window, the memory range in the Memory window, or the codeline in the Source window:
2. Holding down the left mouse button, press the D key.
3. The markpoint is deleted and the blue letter L in front of the variable, memory range or codeline is removed.

Choosing Show Markpoints from Appropriate Popup Menu:
1. Point in the Data, Memory or Source component window and right-click. That window’s popup menu is displayed.
2. Choose Show Markpoints from the Popup Menu. The Markpoints Tab of the Controlpoints Configuration Window is displayed.
3. In this tab’s List box, select the markpoint(s) you want to delete.
4. Click Delete. The selected markpoint is removed from the list of defined watchpoints.
Halting on a Control Point

Code execution is halted when the program reaches either a breakpoint or a watchpoint, if the conditions specified in the definition of the breakpoint or watchpoint have been reached. Code execution is NOT halted when the program reaches a markpoint.

Counting Control Point

If the interval property is greater than 1, a counting control point has been defined. When the Debugger is running, each time the control point is reached, its current value is decremented and the Debugger will halt when the value reaches zero (0). When the Debugger stops on the control point, a command will be executed (if defined and enabled).

Conditional Control Point

If a condition has been defined and enabled for a control point that halts the Debugger, a command will be executed (if defined and enabled).

Control Point with Command

When the Debugger halts on the control point, a specified command is executed.
Real Time Kernel Awareness

The Debugger allows you to load and control applications on the target system, or applications simulated on the host. It also allows you to inspect the state of the application, which includes global variables, processor registers and the procedure call chain including the local (automatic) variables.

This chapter describes how applications built of several tasks are handled by a generic awareness support and an OSEK awareness.

Click any of the following links to jump to the corresponding section of this chapter:

- Introduction
- Task Description Language
- Application Example
- Inspecting Kernel Data Structures
- OSEK Kernel Awareness

Introduction

Often operating systems (Real Time Kernels) are used to coordinate the different tasks in more complex systems. This chapter describes how applications built of several tasks can be handled with the Debugger. There are two main topics to be considered:

- Debugging of any task in the system (e.g., viewing the state of any task in the system). When using the original basic versions of the Debugger, only the current task can be inspected. Due to this extension, it is possible to switch the debugging context from the current task to any other task and between any tasks in the system.
- Real time kernels use data structures to describe the state of the system (scheduling information, queues, timers,...). Some of these data structures are interesting for the user of an operating system too and are described in this chapter.
Inspecting Task State

Each multitasking operating system stores the context of each task at a specific location, usually called the task descriptor. This context consists of the CPU context (CPU registers) and the content of the associated stack. There is more information in the task descriptor, depending on the specific implementation of the kernel.

The Debugger allows you to inspect the CPU registers and stack containing all procedure activation frames (return addresses, parameters, local variables). Therefore, it has to get this information for each task to be debugged. Since this information is specific to the kernel used, there is a universal way to specify the location where and how to collect this data.

This information is read from a file with the name ‘OSPARAM.PRMP’, which describes the algorithm of how to get all the needed data from the target memory (from the task descriptors). To describe this algorithm, a simple procedural language is used. The only parameter to the algorithm is an address specified by the user, which identifies the task to be inspected. The result is the CPU context (CPU registers) and status of the task, which allows the debugger to display the procedure activation stack in a symbolic way.

RTK Interface

When the application is halted, the debugger displays the state of the current task. To identify the task to be inspected, the user has to follow these steps.

Make the task descriptor or a pointer to it visible in any of the debugger’s data windows.

Press the P key while holding down the left mouse button on a variable of type “pointer to task descriptor”.

Now the current state of the selected task and procedure chain of that task is displayed in the 'Procedure Chain' window. By clicking on the procedures in the call chain list, the local data of that function is displayed in the 'Data1' window. All the usual debugging functions are also available to inspect this task now (including displaying the register contents).

Task Description Language

To perform debugging on any task, a file named "OSPARAM.PRMP" has to be created and must be stored in one of the directories specified in GENPATH: #include “File” Path

The file "OSPARAM.PRMP" describes the algorithm to collect the context information for a specific task (the PC, SP, DL, SR and registers).
The following syntax has to be used to specify the algorithm (in EBNF):

\[
\begin{align*}
\text{StatSequence} & = \left[ \text{Statement} \right] \{ ';'; \text{Statement} ; \}. \\
\text{Statement} & = \text{Assignment} \mid \text{ErrorMsg} \mid \text{If}. \\
\text{Assignment} & = \text{Ident} ':=' \text{Expression}. \\
\text{ErrorMsg} & = 'MSG' ':=' \text{String}. \\
\text{IfStatement} & = 'IF' \text{BoolExpr} 'THEN' \text{StatSequence} \{ \text{ELSIFPart} \} \{ \text{ELSEPart} \} 'END'. \\
\text{ELSIFPart} & = 'ELSIF' \text{BoolExpr} 'THEN' \text{StatSequence}. \\
\text{ELSEPart} & = 'ELSE' \text{StatSequence}. \\
\text{String} & = "\{ \text{char} \} ". \\
\text{BoolExpr} & = \text{Expression} \text{RelOp} \text{Expression}. \\
\text{Expression} & = \text{Term} \{ \text{Op} \text{Term} \}. \\
\text{Term} & = \text{Ident} \mid \text{Function} \mid \text{Number}. \\
\text{Ident} & = 'a'..'z' \mid 'R00'..'R31' \mid 'DL' \mid 'SP' \mid 'SR' \mid 'PC' \mid 'STATUS' \mid 'B'. \\
\text{Function} & = ('MB' \mid 'MW' \mid 'MD' \mid 'MA') \[ \text{Expression} \]. \\
\text{RelOp} & = '#' \mid '<' \mid '<=' \mid '=' \mid '>=' \mid '>'. \\
\text{Op} & = '+' \mid '-'.
\end{align*}
\]

The terminal symbols have the following meaning:

- **B** is the given reference to the task descriptor (initialized upon start).
- **a..z** are variables for intermediate storage.
- **MB** gets value of memory BYTE at given address.
- **MW** gets value of memory WORD at given address.
- **MD** gets value of DOUBLE WORD at given address.
- **MA** gets value at given address interpreted as DOUBLE WORD.
- **PC** is the program counter to be set.
- **SP** is the stack pointer to be set.
- **SR** is the status register value to be set.
- **DL** is the dynamic link (data base) to be set (if not available, same as SP).
- **STATUS** is the error number to be set (refer to manual).
- **Rnn** processor registers to be set (mapping to CPU registers see manual).
- **MSG** is the error message (has to be specified if N >= 1000).

On activation of the task debugging command, the file "OSPARAM.PRM" is opened and the selected address is stored in variable 'B'. Then the commands in the file are interpreted. The CPU context of the task is then expected in the variables PC, SP, SR, DL,
Real Time Kernel Awareness

Application Example

Rnn and EN. EN describes the status of the task. If `EN` is bigger than 1000 the status is expected in the string MSG.

Application Example

Listing 5.1 shows an example of "OSPARAM.PRM" file for SOOM System/REM.

Listing 5.1 OSPARAM.PRM File

```plaintext
{ File OSPParam.PRM, implementation for SOOM System/REM }
{ R0..R7 = D0..D7, R8..R15 = A0..A7 }
{ MSG = message displayed in Procedure Chain window }

DL := MD(B+8);  { A6 in PD, dynamic link }  
SP := MD(B+4);  { A7 in PD, stack pointer }  
PC := MD(B+14); { PC in PD, program counter }  
SR := MW(B+12); { SR in PD, status register }  
STATUS := 1000; { Initialized with 1000 }  
IF MW(B+18) = 1 THEN
{ IF (registers are saved in task Control Block) THEN }  
R0 := MD(B+22); R1 := MD(B+26); R2 := MD(B+30);  
R3 := MD(B+34); R4 := MD(B+38); R5 := MD(B+42);  
R6 := MD(B+46); R7 := MD(B+50); R8 := MD(B+54);  
R9 := MD(B+58); R10 := MD(B+62); R11 := MD(B+66);  
R12 := MD(B+70)  
END;  
R13 := B;  
R14 := DL;  
R15 := SP;  
i := MB(B+112); { i contains the current state of the selected task. }
IF i = 0 THEN MSG := "ReadyInCQSc"  
ELSIF i = 1 THEN MSG := "BlockedByAccept"  
ELSIF i = 2 THEN MSG := "WaitForDReply"  
ELSIF i = 3 THEN MSG := "WaitForMail"  
ELSIF i = 4 THEN MSG := "DelayQueue"  
ELSIF i = 5 THEN MSG := "BlockedByReceive"  
ELSIF i = 6 THEN MSG := "WaitForSemaphore"  
ELSIF i = 7 THEN MSG := "Dummy"  
ELSIF i = 8 THEN MSG := "SysBlocked"  
ELSE MSG := "invalid"  
END;
```

HC(S)12(X) Debugger Manual
Inspecting Kernel Data Structures

To allow the debugger to display the data structures of the operating system, the corresponding symbol information has to be available. This is the case when using SOOM System/REM. When another kernel is used its source code would have to be available and would have to be compiled. However, if only the object code is available, the needed symbol information can be generated in the following way:

- The kernel data structures of interest have to be described using ANSI-C language, as shown in Listing 5.2.

Listing 5.2 Kernel Data Structure Description

```c
typedef struct PD {
    int status;
    struct PD *next;
    long regs[6];
} PD;
```

This is an example of the definition of a simple task descriptor.

- Variables can be collected in a structure and have to be assigned to a segment (for example, 'OS_DATA' shown in Listing 5.3).

Listing 5.3 OS_DATA Structure

```c
#pragma DATA_SEG OS_DATA
struct {
    PD *readyList;    /* list of tasks ready to be executed */
    char filler[6];   /* unimportant variables */
    int processes;    /* total number of tasks */
    PD processes[10]; /* the 10 possible tasks */
} OS_DATA;
```

This structure should be defined so as to fit the same layout as the operating system used. It might be necessary to introduce filler variables to get the correct alignment.

This segment has to be placed by the linker to the correct address by using the PRM file shown in Listing 5.4:

Listing 5.4 Linker PRM File

```c
NAMES  ... rtk.o+ ... END
SECTIONS
...
    RTK_SEC = NO_INIT 0x1040 TO 0x1F80;
...
```
The source file (for example: ‘rtk.c’) has to be compiled and listed in the NAMES section of the linker parameter file. To force linking, the name of the object file has to be immediately followed by a ‘+’. In this example the variable is linked to the address 0x1040.

If an application is prepared in this way, all declared variables may be inspected in the data windows of the Debugger. There is no restriction in the complexity of the structures to describe the global data of the kernel.

NOTE You should not open the terminal window during testing. Errors detected during reading of a PRM file are written to this window.

### RTK Awareness Register Assignments

Table 5.1 show the register assignments for the RTK awareness for the HC12 processor.

**Table 5.1 HC12 RTK Awareness Register Assignments**

<table>
<thead>
<tr>
<th>Register</th>
<th>Register Name</th>
<th>Size (bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>A</td>
<td>8 (high byte of D)</td>
</tr>
<tr>
<td>R1</td>
<td>B</td>
<td>8 (low byte of D)</td>
</tr>
<tr>
<td>R2</td>
<td>CCR</td>
<td>8</td>
</tr>
<tr>
<td>R6</td>
<td>D</td>
<td>16 (concatenation of A:B)</td>
</tr>
<tr>
<td>R7</td>
<td>X</td>
<td>16</td>
</tr>
<tr>
<td>R8</td>
<td>Y</td>
<td>16</td>
</tr>
<tr>
<td>R9</td>
<td>SP</td>
<td>24 (concatenation of xPAGE:SP if in banked area)</td>
</tr>
<tr>
<td>R10</td>
<td>PC</td>
<td>16</td>
</tr>
<tr>
<td>R11</td>
<td>PPAGE</td>
<td>8</td>
</tr>
</tbody>
</table>
OSEK Kernel Awareness

OSEK Kernel provides a framework for building real-time applications. OSEK Kernel awareness within the debugger allows you to debug your application from the operating system perspective.

The CodeWarrior Debugger supports OSEK ORTI compliant real-time operating systems and offers dedicated kernel awareness, using the information stored in your application's ORTI file.

With CodeWarrior OSEK kernel awareness, you can monitor kernel task information, semaphores, messages, queues, resources allocations, synchronization, communicating between tasks, etc.

ORTI describes the applications in any OSEK implementation:

- A set of attributes for system objects.
- A method for interpreting the data obtained.

OSEK ORTI

The OSEK Run Time Interface (ORTI) is an interface for development tools to the OSEK Operating System. It is a part of the OSEK standard (refer to www.osek-vdx.org).

The ORTI enables the attached tool to evaluate and display information about the operating system, its state, its performance, the different task states, the different operating system objects etc.

ORTI File and Filename

The ORTI file name has the same name as the application file name, but with the extension .ort. For instance, if the application file name is winLift_demo.abs, the ORTI file name is winLift_demo.ort. Otherwise the debugger cannot use the correct ORTI file.

---

Table 5.1 HC12 RTK Awareness Register Assignments

<table>
<thead>
<tr>
<th>Register</th>
<th>Register Name</th>
<th>Size (bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R12</td>
<td>EPAGE</td>
<td>8</td>
</tr>
<tr>
<td>R13</td>
<td>DPAGE</td>
<td>8</td>
</tr>
<tr>
<td>R14</td>
<td>IP</td>
<td>24 (concatenation of PPAGE:PC if in banked area)</td>
</tr>
</tbody>
</table>

---

HC(S)12(X) Debugger Manual 179
The ORTI file contains dynamic information as a set of attributes that are represented by formulas to access corresponding dynamic values. Formulas for dynamic data access are comprised of constants, operations, and symbolic names within the target file. The given formula can then be evaluated by the debug tool to obtain internal values of the required OS objects.

### Figure 5.1 ORTI Aware Debugging System

Two types of data are made available to the CodeWarrior debug tool. One type describes static configuration data that remains unchanged during program execution. The second type of data is dynamic and this data is re-evaluated each time by CodeWarrior. The static information is useful for display of general information and in combination with the dynamic data. The dynamic data gives information about the current status of the system.

The information given to CodeWarrior is represented in a text (ORTI-File). The file describes the different objects configured in the OS and their properties. The information is represented in direct text, enumerated values, Symbolic names, or an equation that may be used for evaluating the attribute.

The ORTI File is generated when building the project through the OSEK System Generator. The generated file has the same name and the same location as executable file but its extension is .ort.

### ORTI File Structure

The ORTI file structure builds on top of the structure of the OSEK OIL file. It consists of the following parts:
Real Time Kernel Awareness
OSEK Kernel Awareness

- Version Section - This section describes the version of the ORTI standard used for the current ORTI file.
- Implementation Definition Section - This section describes the method that should be used to interpret the data obtained for the value. This section may also detail the suggested display name for a given attribute.
- Application Definition Section - This section contains information on all objects that are currently available for a given system. This section also describes the method that shall be used to reference or calculate each required attribute. This information shall either be supplied as a static value or else a formula that shall be used to calculate the required value.

OSEK RTK Inspector Component

OSEK awareness is described through the Code Warrior RTK Inspector component as shown in Figure 5.2.

Inspector window is displayed by clicking on Component>Open... menu entry and then by clicking on Inspect icon in the “Open Window Component” window.

When the RTK components icon is selected in the hierarchical content of the items, the right side displays various information about OSEK Awareness.
Figure 5.2 CodeWarrior RTK Inspect Window

The OSEK RTK Inspect Window provides access to all this information. As defined in the ORTI file, objects of the same type are grouped and can be viewed together.

- Task
- Stack
- SystemTimer
- Alarm
- Message.

The following sections offer a description of typical objects along with their attributes and how they are presented.

**NOTE** Objects and their attributes depend on the OSEK implementation and OSEK configuration, and therefore may differ from this description.
Inspector Task

The Task shown in Figure 5.3 displays the current state of the OSEK task trace.

**Figure 5.3 Inspector Task**

When selecting Task in the hierarchical tree on the left side of the Inspect window, additional information concerning tasks is displayed on the right side of the window under the following headings:

- **Name**: displays the name of the task
- **Task Priority**: displays the priority of the task.
- **Task State**: describes the current state of the task. Possible values are READY, SUSPENDED, WAITING, RUNNING or INVALID_TASK. The ORTI file defines the different states.
- **Events State**: the event is represented by its mask. The event mask is the number which range is from 1 to 0xFFFFFFFF. When the event mask value is set to 1, the event is activated. When it is set to 0, the event is disabled.
- **Waited Events**: when the bit is set to 0, the event is not expected. When the bit is set to 1, the event is expected.
- **Task Event Masks**: describes the current task event mask.
- **Current Task Stack**: displays the name of the current stack used by the task.
- **Task Properties**: describes task properties. Possible value are BASIC/EXTENDED, NONPREMPT/FULLPREMPT, Priority value, AUTO. The ORTI file defines the possible values.
Inspector Stack

The Stack shown in Figure 5.4 displays the current state of OSEK stack trace.

Figure 5.4 Inspector Stack

When selecting Stack in the hierarchical tree on the left side, additional information concerning the stack are displayed on the right side of the window under the following headings:

- **Name**: displays the name of the stack.
- **Stack Start Address**: displays the start address of the stack.
- **Stack End Address**: displays the end address of the stack.
- **Stack Size**: displays the size of the stack.

Inspector SystemTimer

The SystemTimer shown in Figure 5.5 displays the current state of OSEK SystemTimer trace.

Figure 5.5 Inspector SystemTimer

When selecting SystemTimer in the hierarchical tree on the left side, additional information concerning the timer are displayed on the right side of the window under the following headings:

- **Name**: displays name of the system timer.
- **MAXALLOWEDVALUE**: displays the maximum allowed counter value. When the counter reaches this value it rolls over and starts count again from zero.
- **TICKSPERBASE**: displays the number of ticks required to reach a counter-specific value.
• **MINCYCLE**: displays the minimum allowed number of counter ticks for a cyclic alarm linked to the counter.

• **Current Value**: displays the current value of the system timer.

• **Activated Alarm**: displays associated alarms.

**Inspector Alarm**

The Alarm shown in Figure 5.6 displays the current state of OSEK alarm trace.

**Figure 5.6 Inspector Alarm**

![Inspector Alarm](image)

When selecting Alarm in the hierarchical tree on the left side, additional information concerning the alarm are displayed on the right side of the window under the following headings:

• **Name**: displays the name of the alarm.

• **Alarm State**: displays the current state of the alarm. Possible values are ALARMRUN and ALARMSTOP.

• **Assigned Counter**: based on counters, the OSEK OS offers alarm mechanism to the application software. Assigned Counter is the name of the counter used by alarm.

• **Notified Task**: alarm management allows the user to link task activation to a certain counter value, the assignment of an alarm to a counter, as well as the action to be performed when an alarm expires. Notified Task defines the task to be notified (by activation or event setting) when the alarm expires.

• **Event to Set**: alarm management allows the user to link event setting to a certain counter value, the assignment of an alarm to a counter, as well as the action to be performed when an alarm expires. Event to set specifies the event mask to be set when the alarm expires.

• **Time to expire**: displays time remaining before the time expires and the event is set.

• **Cycle period**: displays period of a tick.
Real Time Kernel Awareness
OSEK Kernel Awareness

Inspector Message

The Message shown in Figure 5.7 displays the current state of OSEK message trace.

Figure 5.7 Inspector Message

When selecting Message in the hierarchical tree on the left side, additional information concerning task are displayed on the right side:

- **Name**: displays the name of the message.
- **Message Type**: displays message type. Possible values are: UNQUEUED/QUEUED.
- **Notified Task**: displays the task that shall be activated when the message is sent.
- **Event to be set**: displays the event which is to be set when the message is sent.
How To ...

This chapter provides answers to frequently asked questions. Click any of the following links to jump to the corresponding set of instructions:

- How To Configure the Debugger
- How To Start the Debugger
- Automating Debugger Startup
- How To Load an Application
- How To Start an Application
- How To Stop an Application
- How To Step in the Application
- How To Work on Variables
- How To Work on the Register
- Modify Content of Memory Address
- How to Consult Assembler Instructions Generated by a Source Statement
- How To View Code
- How to Communicate with the Application
- About startup.cmd, reset.cmd, preload.cmd, postload.cmd
How To Configure the Debugger

If you have installed the Debugger under Windows 95, 98, NT 4.0 and Windows 2000 or higher, the Debugger can be started from the CodeWarrior IDE, from the desktop, from the Start menu, or from an external editor (WinEdit, CodeWright, etc.). In order to work efficiently (find all requested configuration and component files), the Debugger must be associated with a working directory.

For Use from Desktop (Win 95, Win 98, Win NT4.0 or Win2000)

When starting the Debugger from Windows 95 or Windows NT V4.0 (for example, without WinEdit), the working directory can be defined in the file MCUTOOLS.INI, located in the Windows directory.

Defining the Default Directory in the MCUTOOLS.INI

When starting from the desktop or Start menu, the working directory can be set in the configuration file MCUTOOLS.INI.

The working directory including the path is defined in the environment variable DefaultDir in the [Options] group or WorkDir [WorkingDirectory].
How To Start the Debugger

This section describes various ways to start the Debugger.

**From WinEdit**

The Debugger can be started by selecting `Project>Debug` or clicking the Debugger icon (bug) in WinEdit tool bar (when configured). The Window looks like Figure 6.1.

**Figure 6.1** Debugger After Startup

![Debugger Window](image)

`READY` displayed in the status bar indicates that the simulator is ready.
Automating Debugger Startup

Often the same tasks have to be performed after starting the Debugger. These tasks can be automated by writing a command file that contains all commands to be executed after startup of the Debugger, as shown in Listing 6.1.

Listing 6.1 Example of a Command File to Automate Tasks

load fibo.abs
bs &main t

g

This file will first load an application, then set a temporary breakpoint at the start of the function main and start the application. The application will then stop on entering main (after executing the startup and initialization code).

There are several ways to execute this command file:

• specify the command file on the command line using the command line option -c:
  This is done in the application that starts the Debugger (for example, Editor, Explorer, Make utility, ...).

Example:
\Freescale\PROG\HIWAVE.EXE -c init.cmd

When the Debugger is started with this command line, it will execute the command specified in the file init.cmd after loading the layout (or project file).

• Calling the command file from the project file (Listing 6.2). The project file where the layout and connection component can be saved (File >Save...) is a normal text file that contains command line commands to restore the context of a project. This file, once created by the save command, can be extended by a call to the command file (CALL INIT.CMD). When this project is loaded by the File >Open... command or by the corresponding entry in the Project file, commands in this file are executed.

Listing 6.2 Calling a Command File from the Project File:

set Sim
CLOSE *
call \Freescale\DEMO\test.hwl
call init.cmd

• Calling the command file when the Connection Component is loaded. Most connection components will execute the command file STARTUP.CMD once the connection component is loaded and initialized. By adding the call command file in
How To Load an Application

1. Choose Simulator > Load .... The LoadObjectFile dialog box is opened.
2. Select an application (for example FIBO.ABS).
3. Click OK. The dialog box is closed and the application is loaded in the Debugger (Listing 6.2).

Figure 6.2 Load an Application in the Debugger.

The Source component contains source from the module containing the entry point for the application (usually the startup module). The highlighted statement is the entry point.

The Assembly component contains the corresponding disassembled code. The highlighted statement is the entry point.

The Global Data component contains the list of global variables defined in the module containing the application entry point.
How To ...

How To Start an Application

The Local Data component is empty.

The PC in the Register component is initialized with the PC value from the application entry point.

How To Start an Application

There are two different ways to start an application:

1. Choose Run>Start/Continue

2. Click the Start>Continue icon in the debugger toolbar.

RUNNING in the status line indicates that the application is running.

The application will continue execution until:

- you decide to stop the execution (See How To Stop an Application).
- a breakpoint or watchpoint has been reached.
- an exception has been detected (watchpoints or breakpoints).

How To Stop an Application

There are two different ways to stop program execution:

1. Choose Run>Halt

2. Click on the Halt icon in the debugger toolbar.

HALTED in the status line indicates that execution has been stopped.

The blue highlighted line in the source component is the source statement at which the program was stopped (next statement to be executed).

The blue highlighted line in the Assembly component is the assembler statement at which the program was stopped (next assembler instruction to be executed).

Data window with attribute Global displays the name and values of the global variables defined in the module where the currently executed procedure is implemented. The name of the module is specified in the Data info bar.

Data window with attribute Local displays the name and values of the local variables defined in the current procedure. The name of the procedure is specified in the Data info bar.
How To Step in the Application

The Debugger provides stepping functions at the application source level and assembler level (Figure 6.3).

On Source Level

Figure 6.3 Stepping on Source Level.

On the Next Source Instruction

The Debugger provides two ways of stepping to the next source instruction:

1. Choose Run>Single Step

2. Click the Single Step icon from the Debugger tool bar

3. STEPPED in the status line indicates that the application is stopped by a step function.

If the application was previously stopped on a subroutine call instruction, a Single Step stops the application at the beginning of the invoked function.

The display in the Assembly component is always synchronized with the display in the Source component. The highlighted instruction in the Assembly component is the first assembler instruction generated by the highlighted instruction in the Source component.
How To ... 
**How To Step in the Application**

Elements from Register, Memory or Data components that are displayed in red are the register, memory position, local or global variables, and which values have changed during execution of the source statement.

**Step Over a Function Call (Flat Step)**

The Debugger provides two ways of stepping over a function call:

1. Choose **Run >Step Over**

2. Click the **Step Over** icon from the Debugger tool bar 🔄

STEPPED OVER (STEPOVER) or STOPPED (STOP) in the status line indicates that the application is stopped by a step over function.

If the application was previously stopped on a function invocation, a **Step Over** stops the application on the source instruction following the function invocation.

The display in the Assembly component is always synchronized with the display in the Source component. The highlighted instruction in the Assembly component is the first assembler instruction generated by the highlighted instruction in the Source component.

Elements from Register, Memory or Data components that are displayed in red are the register, memory position, local or global variables, and which values have changed during execution of the invoked function.

**Step Out from a Function Call**

The Debugger provides two ways of stepping out from a function call:

1. Choose **Run >Step Out**

2. Click the **Step Out** icon from the debugger tool bar 🎵

STOPPED (STOP) in the status line indicates that the application is stopped by a step out function.

If the application was previously stopped in a function, a **Step Out** stops the application on the source instruction following the function invocation.

The display in the Assembly component is always synchronized with the display in the Source component. The highlighted instruction in the Assembly component is the first assembler instruction generated by the highlighted instruction in the Source component.

Elements from Register, Memory or Data components that are displayed in red are the register, memory position, local or global variables, and which values have changed since the **Step Out** was executed.
Step on Assembly Level

The Debugger provides two ways of stepping to the next assembler instruction:

1. Choose Run>Assembly Step

2. Click the Assembly Step icon from the debugger tool bar

TRACED in the status line indicates that the application is stopped by an assembly step function.

The application stops at the next assembler instruction.

The display in the Source component is always synchronized with the display in the Assembly component. The highlighted instruction in the Source Component is the source instruction that has generated the highlighted instruction in the Assembly component.

Elements from Register, Memory or Data components that are displayed in red are the register, memory position, local or global variables, and which values have changed during execution of the assembler instruction.

How To Work on Variables

This section shows the different methods to work on variables.

Display Local Variable from a Function

The Debugger provides two different ways to see the list of local variables defined in a function:

• Using Drag and Drop

  1. Drag a function name from the Procedure component to a Data component with attribute local.

• Using Double-click

  1. Double-click a function name in the Procedure component.

The Data component (with attribute local that is neither frozen or locked) displays the list of variables defined in the selected function with their values and type.

Display Global Variable from a Module

The Debugger provides two ways to see a list of global variables defined in a module:

• Opening Module Component
How To Work on Variables

1. Choose Component>Open. The list of all available components is displayed on the screen.
2. Double-click the entry Module. A module component is opened, which contains the list of all modules building the application.
3. Drag a module name from the Module component to a Data component with attribute Global.
   - Using Popup Menu
   1. Right-click in a Data component with attribute Global.
   2. Choose Open Module in Popup Menu. A dialog box is opened, which contains the list of all modules building the application.
      - Double-click on a module name. The Data component with attribute global, which is neither frozen nor locked is the destination component.

The destination Data component displays the list of variables defined in the selected module with their values.

Change Format for Variable Value Display

The Debugger allows you to see the value of variables in different formats. This is set by entries in Format menu (Table 6.1).

Table 6.1 Debugger Display Format

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>Variable values are displayed in hexadecimal format.</td>
</tr>
<tr>
<td>Oct</td>
<td>Variable values are displayed in octal format.</td>
</tr>
<tr>
<td>Dec</td>
<td>Variable values are displayed in signed decimal format.</td>
</tr>
<tr>
<td>UDec</td>
<td>Variable values are displayed in unsigned decimal format.</td>
</tr>
<tr>
<td>Bin</td>
<td>Variable values are displayed in binary format.</td>
</tr>
<tr>
<td>Symbolic</td>
<td>Displayed format depends on variable type.</td>
</tr>
</tbody>
</table>

1. Values for pointer variables are displayed in hexadecimal format.
2. Values for function pointer variables are displayed as function name.
3. Values for character variables are displayed in ASCII character and decimal format.
4. Values for other variables are displayed in signed or unsigned decimal format depending on the variable being signed or not.
How To Work on Variables

**Format** menu is activated as follows:

1. Right-click in the Data component. The Data Popup Menu is displayed on the screen.
2. Choose **Format** from Popup Menu. The list of all formats is displayed on the screen.

The format selected is valid for the whole Data component. Values from all variables in the data component are displayed according to the selected format.

**Modify a Variable Value**

The Debugger allows you to change the value of a variable, as shown in Figure 6.4.

**Figure 6.4 Modifying a Variable Value**

The Debugger allows you to change the value of a variable.

Double-click on a variable. The current variable value is highlighted and can be edited.

1. Formats for the input value follow the rule from ANSI C constant values (prefixed by 0x for hexadecimal value, prefixed by 0 for octal values, otherwise considered as decimal value). For example, if the data component is in decimal format and if a variable input value is 0x20, the variable is initialized with 32. If a variable input value is 020, the variable is initialized with 16.

2. To validate the input value you can either press the **Enter** or **Tab** key.

3. If an input value has been validated by the **Tab** key, the value of the next variable in the component is automatically highlighted (this value can also be edited).

4. To restore the previous variable value, press the **Esc** key or select another variable.

A local variable can be modified when the application is stopped. Since these variables are located on the stack, they do not exist as long as the function where they are defined is not active.
Get the Address Where a Variable is Allocated

The Debugger provides you with the start address and size of a variable if you do the following:

1. Point to a variable name in a Data Component
2. Click the variable name

The start address and size of the selected variable is displayed in the Data info bar.

Inspect Memory Starting at a Variable Location Address

The Debugger provides two ways to dump the memory starting at a variable allocation address.

- Using Drag and Drop
  1. Drag a variable name from the Data Component to Memory component.
  2. Holding down the left mouse button and pressing the A key

The memory component scrolls until it reaches the address where the selected variable is allocated. The memory range corresponding to the selected variable is highlighted in the memory component.

Load an Address Register with the Address of a Variable

The Debugger allows you to load a register with the address where a variable is allocated.

1. Drag a variable name from the Data Component to Register component.

The destination register is updated with the start address of the selected variable.
How To Work on the Register

This section describes how to work with the Register component.

Change Format of Register Display
The Debugger allows you to display the register content in hexadecimal or binary format.
1. Right-click in the Register component. The Register Popup Menu is displayed on the screen.
2. Choose Options .. from the Popup Menu. The pull down menu containing the possible formats is displayed.
3. Select either binary or hexadecimal format.
The format selected is valid for the Register component. The contents from all registers are displayed according to the selected format.

Modify a Register Content
The Debugger allows you to change the content of indexes, accumulators or bit registers.

Modify Index or Accumulator Register Content
Double-click a register. The current register content is highlighted and may be edited.

Figure 6.5 Modifying Index or Accumulator Register Content

1. The format of the input value depends on the format selected for the data component. If the format of the component is Hex, the input value is treated as a Hex value. If the input value is 10 the variable will be set to 0x10 = 16.
2. To validate the input value you can either press the Enter or Tab key, or select another register.
How To Work on the Register

3. If an input value has been validated by the Tab key, the content of the next register in the component is automatically highlighted. This register can also be edited.

4. To restore the previous register content, press the Esc key.

Modify Bit Register Content

In a bit register, each bit has a specific meaning (a Status Register (SR) or Condition Code Register (CCR)).

Mnemonic characters for bits that are set to 1 are displayed in black, whereas mnemonic characters for bits that are reset to 0 are displayed in grey.

Single bits inside the bit register can be toggled by double-clicking the corresponding mnemonic character.

Start Memory Dump at Address Where Register Is Pointing

The Debugger provides two ways to dump memory starting at the address a register is pointing to.

- Using Drag and Drop

1. Drag a register from the Register component to Memory component.

- Choose Address ..

Figure 6.6 Memory menu Display Address

1. Right-click in the Memory component. The Memory Popup Menu is displayed.

2. Choose Address .. from the Popup Menu. The Memory ... dialog box shown in Figure 6.6 is opened.

3. Enter the register content in the Edit Box and choose OK to close the dialog box.
Modify Content of Memory Address

The Debugger allows you to change the content of a memory address. Double-click the memory address you want to modify. Content from the current memory location is highlighted and can be edited.

1. The format for the input value depends on the format selected for the Memory component. If the format for the component is Hex, the input value is treated as a Hex value. If input value is 10 the memory address will be set to 0x10 = 16.
2. Once a value has been allocated to a memory word, it is validated and the next memory address is automatically selected and can be edited.
3. To stop editing and validate the last input value, you can either press the Enter or Tab key, or select another variable.
4. To stop editing and restore the previous memory value, press the Esc key.

How to Consult Assembler Instructions Generated by a Source Statement

The Debugger provides an on-line disassembly facility, which allows you to disassemble the hexadecimal code directly from the Debugger code area. Online disassembly can be performed in one of the following ways:

**Using Drag and Drop**
1. In the Source component, select the section you want to disassemble.
2. Drag the highlighted block to the Assembly component.

**Holding down the left mouse button and pressing the R key**
1. In the Source component window, point to the instruction you want to disassemble.
2. Hold down the left mouse button and press the R key
The disassembled code associated with the selected source instruction is greyed in the Assembly component.

How To View Code

The Debugger allows you to view the code associated with each assembler instruction.

Figure 6.7 Viewing Code Associated with Assembler instruction.

Online disassembly can be performed in one of the following ways:

- Using Popup Menu
  1. Point in the Assembly component and right-click. The Assembly Popup Menu is displayed.
  2. Choose Display Code (Figure 6.7).
- Using Assembly Menu
  1. Click the title bar of the Assembly component. The Assembly menu appears in the debugger menu bar.
  2. Choose Assembly > Display Code

The Assembly component displays the corresponding code on the left of each assembler instruction.
How to Communicate with the Application

The Debugger has a pseudo-terminal facility. Use the TestTerm or Terminal component window to communicate with the application using specific functions defined in the TERMINAL.H file and used in the calculator demo file.

1. Start the Debugger and choose Open... from the Component menu.
2. Open the TestTerm or Terminal Component.
3. Choose Load... from the Simulator menu.
4. Load the program CALC.ABS.

Data entered in the TestTerm or Terminal component window through the keyboard will be fetched by the target application with the ‘Read’ function. The target application can send data to the Terminal component window of the host with the ‘Write’ function.

About startup.cmd, reset.cmd, preload.cmd, postload.cmd

The command files startup.cmd, reset.cmd, preload.cmd, and postload.cmd are Debugger system command files. All these command files do not exist automatically. They could be installed when installing a new connection.

However, the Debugger is able to recognize these command files and execute them.

- startup.cmd is executed when a connection is loaded (the target defined in the project.ini file or loaded when you select Component>Set Connection).
- reset.cmd is executed when you select “Connection Name” > Reset in the menu (Connection Name is the real name of the connection, such as MMDS0508, SDI, etc.).
- preload.cmd is executed before loading a .ABS application file or Srecords file (when you select “Connection Name”>Load... in the menu).
- postload.cmd is executed after loading a .ABS application file or Srecords file (when you select “Connection Name”>Load... in the menu).

Depending on the connection used, other command files can be recognized by the Debugger. Refer to the appropriate connection manual for information and properties of these command files.
How To ...

About startup.cmd, reset.cmd, preload.cmd, postload.cmd
This chapter provides information on how to use and configure the Simulator/Debugger within CodeWarrior using the following software:

- CodeWarrior IDE - HC12 version 4.5 or later

Click the following link to jump to the corresponding section of this chapter:
- Debugger Configuration

**Debugger Configuration**

To configure the Real Time Debugger and True Time Simulator, in the CodeWarrior IDE open the **Target Settings Panel** by clicking on the Targets panel of the IDE main window, then double clicking on the name of your target in the list displayed in this panel. Then, select **Build Extras** as shown in (Figure 7.1).

In the **BuildExtras** pane check the **Use External Debugger** checkbox.

In the Application field, type the Debugger path, (or select from the Open window by clicking the Browse button) for example: `{Compiler}prog\hiwave.exe`.

In the Arguments field, type the arguments, for example, `%targetFilePath -Target=sim` in the Argument field.

Click on **Apply** to validate these changes.
Figure 7.1 IDE Target Window - Build Extras Panel
Debugger DDE Capabilities

Introduction
The DDE is a form of interprocess communication that uses shared memory to exchange data between applications. Applications can use DDE for one-time data transfers and for ongoing exchanges in applications that send updates to one another as new data becomes available.

NOTE  The DDE capabilities of the Debugger are deprecated. Future versions of the Debugger will have no DDE capabilities. You can use the Component Object Model (COM) Interface.

DDE Implementation
The Debugger integrates a DDE server and DDE client implementation in the KERNEL. The DDE application name of the IDF server is "HI-WAVE".

The Debugger DDE support allows you to execute almost any command that would be available from within the debugger (from Command line). There are also special DDE items for more commonly performed tasks.

This section describes topics and DDE items available to CodeWright clients. In addition to the required System topic, CurrentBuffer and the names of all CodeWright non-system buffers (documents) are available as topics.

Driving Debugger through DDE
The DDE implementation in the Debugger allows you to drive it easily by using the DDE command. To do this, you have to use a program that can send a DDE message (a DDE client application) like DDECLient.exe from Microsoft.

The service name of the Debugger DDE Server is "HI-WAVE" and the Topic name for the Debugger DDE Server is "Command".

The following example is done with DDECLient.exe from Microsoft.
1. Run the Debugger and in the "Service" field in the DDECLient type: "HI-WAVE"
2. In the "Topic" field type "Command"
Debugger DDE Capabilities

3. Push the "Connect" button of the DDEClient. The following message will appear in DDEClient: "Connected to HI-WAVE|Command".

4. In the "Exec" field of DDEClient type a Debugger command, for example "open recorder" and click the "Exec" button. The command is executed by way of DDE and you'll see a new recorder component in the Debugger.

**NOTE** You can disconnect the DDE in the Debugger. The Debugger can be started without DDE (this is saved in the project file). To view the current state, open a command line component and type the following command: "DDEPROTOCOL STATUS". The state must be: "DDEPROTOCOL ON" to ensure the DDE works properly.
Synchronized Debugging Through DA-C IDE

This chapter provides information on how to use and configure Freescale tools within the Development Assistant for C (DA-C) IDE. For more information on DA-C, refer to the "Development Assistant for C" documentation v 3.5.

You must be running:
DA-C - version 3.5 build 555 or later - (Development Assistant for C - RistanCASE).

Click any of the following links to jump to the corresponding section of this chapter:
- Configuring DA-C IDE for Freescale Tool Kit
- Debugger Interface
- Synchronized Debugging
- Troubleshooting

Configuring DA-C IDE for Freescale Tool Kit

Install the DA-C software. The Freescale CD contains a demo version located in \Addons\DA-C. Run Setup to install the Typical installation.

A few configurations are required in order to make efficient use of Freescale Tools within DA-C IDE.
- Create a new project
- Configure the working directories
- Configure the file types
- Configure the Freescale library path
- Adding files to project
- Building the Database
- Configure the tools
In the following sections, we assume that the Freescale tool kit is installed in "C:\Freescale" directory. You may have to adapt the paths to your current installation. An example configuration for the M68k CPU is provided, which can be adapted to each CPU supported by Freescale.

Create New Project

Start DA-C.exe and choose Project> New Project... from the main menu. Browse to the directory and enter a project file name, for example: "C:\Freescale\work\processor\c\myproject"
Change the <processor> field to your CPU. A specific project file is created with " .dcp" extension (for example "myproject.dcp").

Configure Working Directories

Choose Options>Project from the main menu of DA-C. The window shown in Figure 9.1 contains options, which establish directories for the project.

Figure 9.1 DA-C Project Options Window - Directories Tab
Synchronized Debugging Through DA-C IDE
Configuring DA-C IDE for Freescale Tool Kit

Project Root Directory
This text box determines the project root directory. The full path is expected, or a single
dot can be entered, which stands for the same directory where the project file resides. All
files that belong to the project are considered relative to the Project root directory, if the
full path of the file is not given. In our case, keep the single dot for the project root
directory.

Referential Project Root Directory
If not empty, this text box specifies an alternate Project Root Path for searching files not
found in the original project path. Filenames in the original path with referential
extensions are tried before those in the referential path. Specified path may be either full
or relative to project root, and it may not specify a subdirectory in the project root
directory tree. Leave this field empty.

Database Directory
This text box determines the directory where the symbols and software metrics database
will be saved. This directory can be absolute or relative to the Project Root Directory.
Leave this field empty.

User Help File
This text box determines the user help file, for example compiler help file. The hot key for
User Help File can be defined in the Keyboard definition file (default Ctrl-Shift-F1).
Browse in the "prog" directory of your Freescale installation and select the help file
matching your CPU.
Configure File Types

From the main menu of DA-C choose "File Types" to configure the basic file types. The File Types Tab of the Project Options Window contains options, which determine file types of the project. For an efficient use of Freescale tools, Figure 9.2 shows file extension types that can be defined.

Figure 9.2 DA-C Project Options Window - File Types Tab

Configure Library Path

An additional configuration path must be defined to specify the location of library header files (needed for DA-C symbol analysis). This can be done by choosing Options>Analysis for Symbols … >C Source in the main menu of DA-C.

The window shown in Figure 9.3 contains options that determine parameters of the C source code analysis.
Source

The supported C dialects of the C language used in the current project can be selected in this text field. In our example we chose the Freescale M68k language (adapt it to your needs).

Preprocessor - Header Directories

This text box determines the list of directories that are to be searched for files named within the "#include" directive. A semicolon separates directories. Only listed directories are searched for files, named between "<" and ">". Searching for files, named between quotation marks (""), starts in the directory of the source file containing "#include" directive.
The list of header directories can be assigned in a file. In that case, this field contains the file name (absolute or relative in relation to the project root) with prefix @. Directories are separated with a semi-colon or new line.

Define the library path matching your CPU (assuming Freescale tools are installed on "C:\Freescale"):

C:\Freescale\lib\<processor>c\include.

Preprocessor - Preinclude File

This text box determines the name of the file that will be included automatically at the beginning of every source module during analysis, in the same way as if #include "string" were present in the first line. The preinclude file can be used to specify predefined macros and variable and function declarations for a particular compiler, which are not set by default in DA-C analysis. We have selected the one corresponding to our example: M68k preinclude file (adapt it to your needs).

Adding Files to Project

In the Project Window the Explorer View Tab replaces the Window's Explorer and supplies you with additional information on directories containing project files. It also gives you the option to add files into the project. For example, we will now set all files needed to run the "fibo" example.

In the Explorer View, browse to the "\Freescale\WORK\<processor>c" directory of your Freescale installation and select "fibo.c" file. Then right-click mouse button and choose "Add to Project". The file is now added in the current project and a green mark appears in front of it (Figure 9.4).

Figure 9.4 Adding Files to Project Using Explorer Tab

In the same way, select "fibo.prm" file and add it to this project.
Synchronized Debugging Through DA-C IDE

Configuring DA-C IDE for Freescale Tool Kit

You can also add a directory to the project in the following way:

- Select Explorer View Tab in Project Window.
- In the left section, select the directory with files to be added to the project (files from subdirectories may also be added to the project).
- From popup menu choose "Add to project".

This operation may also be performed from Folder view, if the directory is in the left section.

NOTE When adding an entire directory to the project only files with extensions defined in Options>Project>File types (as described in section "Configure the file type") will be added to the project.

Building The Database

Development Assistant for C provides the static code analysis of C source files, as well as generating various data based on the results.

Analysis of the project source files and generation of the database are divided into two phases: the analysis of individual program modules and generation of data about global symbols usage. Results of the analysis are saved in database files on the disk, which enables their later use in DA-C. You can choose between the unconditional analysis of all project files and the analysis of changed source files only, using Start> Build database and Start>Update database commands. The latter one will optionally check if the include files used in program modules are changed as well.

To build the database in our example use Start>Build database command, which makes the unconditional analysis of all project files and creates a database containing information about analyzed source code. Errors and Warnings detected during this operation are displayed in the Messages window as illustrated in Figure 9.5 (for Fiboc.c sample file):

Figure 9.5 DA-C Message Window
After the analysis of all project files, the new database file containing information about global symbols is constructed. Refer to the DA-C manual for more information on how symbol information can be used.

In the Project Manager’s window of DA-C, select the **Logical View** Tab shown in **Figure 9.6** and unfold all fields, you will now have the overview of your project.

**Figure 9.6 Logical View Tab**

Double-click on "Fibo.c" file to open it.
Configuring The Tools

We will now configure the compiler and maker into DA-C IDE. Procedures are defined in Project>User Defined Actions... from the main menu of DA-C.

Compiler Configuration

In Menu "Start" Actions, click on new and fill in the New Action box with "C&ompile", then press ENTER (Figure 9.7). In the Toolbar field, you can associate a bitmap with each tool, for example click on the Picture radio button and browse to the "\Bitmap" directory of your current DA-C installation and choose Compiler.bmp. This is a default bitmap delivered with DA-C IDE. Here you are able to add your own bitmap.

Figure 9.7 DA-C Compiler Settings
Synchronized Debugging Through DA-C IDE
Configuring DA-C IDE for Freescale Tool Kit

Now fill in the **Action Script** field in order to associate related compiler actions. Copy the following lines shown in **Listing 9.1** in the Action Script field and change the directory to where the compiler is located.

**Listing 9.1 Script for Compiler Action Association**

```plaintext
.%If(%HasModuleExt(%CurrFile),,%Message(Not a module file!)%Cancel)
.%SaveAll
c:\Freescale\prog\cm68k.exe %CurrFile
.%if(%Exist(edout),,%Message(No Messages found!)%Cancel)
.%ErrClr(Compiler)
.%ErrGet(edout,Compiler)
.%Reset(%CurrFile)
```

Click on **OK** to validate these settings. Select "Fibo.c" file. Click on the "Compiler" button (or from the main menu of DA-C select **Start>Compile**). This file is now compiled and the corresponding object file ("Fibo.o") is generated.

**Linker Configuration**

In the same way, you can now configure the linker as illustrated in **Figure 9.8**. In the Menu "Start" Actions, click on new and fill in the created **New Action** box with "&Link", then validate with ENTER. After setting the corresponding bitmap, copy the following lines shown in **Listing 9.2** in the **Action Script** field and change the directory to where the linker is located.

**Listing 9.2 Script for Linker Action Association**

```plaintext
+c:\Freescale\prog\linker.exe fibo.prm
.%if(%Exist(edout),,%Message(No Messages found!)%Cancel)
.%ErrClr()
.%ErrGet(edout)
```
Figure 9.8 DA-C Linker Settings

![DA-C Linker Settings diagram]

**Maker Configuration**

In the same way, you can now configure the maker as illustrated in Figure 9.9. In the Menu "Start" Actions, click on new and fill in the created New Action box with "&Make", then press ENTER. After setting the corresponding bitmap, copy the lines from Listing 9.3 in the Action Script field and change the directory to where the maker is located.

**Listing 9.3 Script for Maker Action Association**

```plaintext
+c:\Freescale\prog\maker.exe fibo.mak .%if(%Exist(edout),,%Message(No Messages found!)%Cancel) .%ErrClr() .%ErrGet(edout)
```

---

*HC(S)12(X) Debugger Manual* 219
Synchronized Debugging Through DA-C IDE

Debugger Interface

DA-C v3.5 is currently integrating a DAPI interface (Debugging support Application Programming Interface). Through this interface DA-C is enabled to exchange messages with the Debugger. The advantages of such connection show that it is possible to set or delete break points from within DA-C (in an editor, flow chart, graph, browser) and to execute other debugger operations. DA-C is following Debugger in its operation, since it is always in the same file and on the same line as the debugger. Thus, usability of both the DA-C and Debugger is increased. Some configurations are required in order to make an efficient use of this Debugger Interface:

- Installation of communication DLL
- Configuration of Debugger properties
- Configuration of the Debugger project file
DA-C IDE and Debugger Communication

DA-C and the Debugger are both Microsoft Windows applications and communication is based on the DDE protocol, as shown in Figure 9.10. The whole system contains:

- DA-C
- Debugger
- cDAPI interface implementation DLL - which is used by DA-C (Cdgen32.dll)
- nDAPI communication DLL (provided by DA-C), which is used by Debugger
- Debugger specific DLL for bridging its interface to debugging environment and DA-C’s nDAPI (DAC.wnd)

Figure 9.10 Communication between DA-C IDE and Debugger

Communication DLL Installation

As described previously, the Debugger needs the nDAPI communication DLL (provided by DA-C IDE). This dll (called Ndapi.dll) is automatically installed during the Freescale Tool Kit installation. However, if you install a new release of DA-C you have to follow this procedure:
In the "Program" directory of your DA-C installation, copy the "Ndapi32.dll" (Ndapi32.dll version 1.1 or later) and paste it in your current "Freescale\PROG" directory (where Debugger is located). Then rename it to "Ndapi.dll".

Debugger Properties Configuration

In the DA-C main menu, choose Options>Debugger, the dialog box shown in Figure 9.11 is opened.

Figure 9.11  DA-C Debugger Options Dialog Box

In the "Debugger" combo-box, select the corresponding debugger: "HI-WAVE 6.0". Now specify the binary file to be opened: in our example we want to debug the "fibo.abs" file.

Then click on the Setup... button. The dialog box shown in Figure 9.12 is opened.
Specify the path to the "hiwave.exe" file or use the Browse... button then click on OK.

Debugger Project File Configuration

Before configuring the project file, close DA-C. Open Debugger (for example, from a shell) and select File>Open Project... from the main menu bar. Select the "Project.ini" file from the currently defined working directory (in our case "C:\Freescale\WORK\<processor>c\project.ini"). We will now add in the layout of the project the Debugger DAC component ("dac.wnd"). In the Debugger select Component >Open from the main menu bar and choose "Dac", as shown in Figure 9.13.
The Debugger DAC window, which is needed for communication with DA-C IDE is now opened (Figure 9.14).

You have to save this configuration by selecting File>Save Project from the main menu of the Debugger. This component will be automatically loaded the next time this project is called. Close the Debugger.
Synchronized Debugging

We can now test the synchronization between DA-C IDE and Debugger. Run DA-C.exe and open the project previously created. Open "Fibo.c" if it's not already open. Right-click mouse button on "Fibo.c" source window and select "main" in the popup menu. The cursor points to the "void main(void) {" statement. In the main menu from DA-C, select Debug>Set Breakpoint (or click on the corresponding button on the debug toolbar), the selected line is highlighted in red, indicating that a breakpoint has been set. Then select Debug>Run, the Debugger is now started and after a while stops on the specified breakpoint. Up to now, you can debug from DA-C IDE with the toolbar, as shown in Figure 9.15 or from the Debugger.

Figure 9.15 DA-C toolbar

NOTE In case of changes to your source code, don't forget to rebuild the Database when generating new binary files to avoid misalignment between the Debugger and DA-C source positions.

Troubleshooting

This section describes possible trouble when trying to connect the Debugger with the DA-C IDE.

1. When loading DAC component into the Debugger, if the message box shown in Figure 9.16 is displayed:

Figure 9.16 DA-C Component Loading Error Message

check if the Ndapi.dll is located in the "prog" directory of your current Freescale installation. If not, copy the specified DLL into this directory.
Synchronized Debugging Through DA-C IDE

Troubleshooting

2. If the message box shown in Figure 9.17 is displayed in DA-C IDE:

**Figure 9.17 DA-C Debugger Support Message**

![DA-C Debugger Support Message](image)

This means that the current name specified in the **Options>Debugger** main menu of DA-C doesn't match the debugger name specified in the Debugger. Open the setup dialog in the Debugger by clicking on the DA-C Link component and choose **DA-C Link>Setup…** from the main menu. The "Connection Specification" dialog box is opened (Figure 9.18).

**Figure 9.18 DA-C Connection Specification Dialog Box**

![Connection Specification Dialog Box](image)

Compare the "**Debugger Name**" from this dialog box with the selected Debugger in DA-C IDE (Options>Debugger), as shown in Figure 9.19.
Both must be the same. If it’s not the case, change it in the Debugger “Connection Specification” and click OK. This implies a new connection to be established and the “Connection Specification” to be saved in the current “Project.ini” file in the section shown in Listing 9.4.

Listing 9.4 DA-C Section in Project File.

```
[DA-C]
DEBUGGER_NAME=HI-WAVE 6.0
SHOWPROT=1
```
Synchronized Debugging Through DA-C IDE
Troubleshooting
Book II - HC(S)12(X) Debug Connections

Book II Contents

Each section of the Debugger manual includes information to help you become more familiar with the Debugger, to use all its functions and help you understand how to use the environment.

Book 2: HC(S)12(X) Debugger Connections - defines the connections available for debugging code written for HC12 CPUs.

- Chapter 2.1 ""
- Chapter 2.2 “ICD-12 Connection”
- Chapter 2.3 “Softec inDart HCS12 Connection”
- Chapter 2.4 “HCS12 Serial Monitor Connection”
- Chapter 2.5 “BDIK Connection”
The Full Chip Simulation (FCS) connection runs a complete simulation of all processor peripherals and I/O on the user's Personal Computer. No development board is required. Each derivative has a totally different simulation engine to accurately simulate the memory ranges, I/O, and peripherals for a given derivative (for more information on selecting a specific derivative, please see the "Select Device" section below. This section guides you through the first steps toward debugging with CodeWarrior and the HC(S)12X Full Chip Simulation connection. It does not replace all the additional documentation provided in this manual, but gives you a good starting point.

**Full Chip Simulation Menu**

This menu, shown in Figure 10.1, is associated with the Full Chip Simulation connection, and allows you to load an application in the Full Chip Simulation. Table 10.1 describes the Full Chip Simulation menu entries.

**Figure 10.1 Simulator Menu**

<table>
<thead>
<tr>
<th>Simulator</th>
<th>Component</th>
<th>Common</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load...</td>
<td>ChH-L</td>
<td></td>
</tr>
<tr>
<td>Reset</td>
<td>ChH-R</td>
<td></td>
</tr>
<tr>
<td>Set Derivative...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Configure...</td>
<td>Reset RAM</td>
<td></td>
</tr>
<tr>
<td>Reset MEM</td>
<td>Reset Static</td>
<td></td>
</tr>
<tr>
<td>Load ICs...</td>
<td>Close ICs...</td>
<td></td>
</tr>
<tr>
<td>Clock Frequency...</td>
<td>Command Files...</td>
<td></td>
</tr>
<tr>
<td>Bus Trace</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 10.1  Simulator Menu Entry Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Opens the Load Executable Window menu.</td>
</tr>
<tr>
<td>Reset</td>
<td>Resets the Full Chip Simulation.</td>
</tr>
<tr>
<td>Set Derivative</td>
<td>Selects the current simulated derivative.</td>
</tr>
<tr>
<td>Configure</td>
<td>Opens the Memory Configuration Window.</td>
</tr>
<tr>
<td>Reset Ram</td>
<td>Resets the RAM to <code>undefined</code>.</td>
</tr>
<tr>
<td>Reset Mem</td>
<td>Resets all configured memory to <code>undefined</code>.</td>
</tr>
<tr>
<td>Reset Statistic</td>
<td>Resets the statistical data.</td>
</tr>
<tr>
<td>Load I/Os</td>
<td>Opens I/O components</td>
</tr>
<tr>
<td>Close I/Os</td>
<td>Closes I/O components</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>Opens the Clock Frequency Setup to set the Real Time clock of the Full Chip Simulation.</td>
</tr>
<tr>
<td>Command Files</td>
<td>Opens the Command File Window.</td>
</tr>
<tr>
<td>Bus Trace</td>
<td>Opens the Bus Trace to enable instructions and memory accesses recording and display recording captures.</td>
</tr>
</tbody>
</table>

Debugger Status Bar with Full Chip Simulation

The status bar (Figure 10.2 and Figure 10.3) shows status and other information. As well as execution status, it includes a context-sensitive menu help line, and connection specific information like the number of CPU cycles (64 bits) or the elapsed time in hours:minutes'seconds"milliseconds(float) format since the application started.

Figure 10.2  Debugger Status Bar with CPU Cycles

Figure 10.3  Debugger Status Bar with Elapsed Time
The selected simulated derivative or simulated “CORE” or core “SAMPLE” is shown, and also the current derivative CPU frequency in MHz.

**NOTE** Clicking on the CPU frequency opens the Clock Frequency Setup.

**NOTE** Double-clicking on the CPU cycles or true time resets the value.

**NOTE** Clicking on the displayed derivative or “CORE” or core “SAMPLE” opens the Set Derivative dialog box.

**NOTE** The first left hand side CPU information in the Status Bar, like “HC12” might be alternatively displayed with “XGATE”, when an HCS12X core device is simulated. The debugger indicates on which core thread the debugger is currently halted or currently stepping.

### Open I/O Component Dialog Box

From the Simulator menu, choose Load I/Os to open the Open I/O Component dialog box. This dialog box, shown in Figure 10.4 allows you to open an I/O device (peripheral) simulation. The **Browse** button allows you to specify a location for the I/O.

**Figure 10.4** Open IO Component Dialog Box

![Open IO Component Dialog Box](image-url)
NOTE  I/O simulation components are either designed by Freescale and delivered with the tool-kit installation or designed by the user with the Peripheral Builder (separate product).

Demo Version Limitations
Only 2 I/O components can be loaded at a time.

Command File Window

Figure 10.5 Simulator HC12 Target Interface Command Files Window

Setcpu Command File
The Setcpu command file is a specific command file to the Full Chip Simulation and is executed by the Debugger after a CPU has been set or modified in the Full Chip Simulation (this occurs when the setcpu command is used or when an application is loaded in the Full Chip Simulation and the corresponding cpu is not set).

The Setcpu command file full name and status (enable/disable) can be specified either with the CMDFILE SETCPU Command Line command or using the Setcpu property tab of the connection Command Files dialog.

The default Setcpu command file is SETCPU.CMD. By default the SETCPU.CMD file located in the current project directory is enabled as the current Setcpu command file.
Other Commands Files are described in the Debugger Interface section, in the Debugger Engine book.

Memory Configuration

The memory configuration interface is a Full Chip Simulation advanced configuration feature. The emulated memory is divided into blocks. A memory manager handles the list of memory blocks. The memory configuration facility offers you some degree of automation, but does not restrict the flexibility of manual adjustment. The memory configuration facility lets you specify types and properties of memory blocks, such as RAM, ROM, and so forth.

The memory configuration facility uses a binary file format to read and set the Full Chip Simulation configuration. The extension for binary files is .mem; the default memory file is default.mem. (The subsection “Format of the Default Memory Configuration File” includes Listing 10.2, the EBNF-syntax definition of the file format.)

Memory Configuration Dialog Box Features

The memory configuration dialog box (Figure 10.6) lets you perform these memory-block operations interactively:

- Select the configuration mode for simulation
- Define a memory block name
- Define how the Full Chip Simulation verifies the memory
- Set the type of the memory: RAM, ROM, FLASH, EEPROM or I/O
- Define start and end addresses
- Define the wait state (the time for each read or write access)
- Set the width of the bus that accesses the memory
- Set access details like:
  - auto configure: automatically computing read and write access
  - misaligned access: allowing misaligned access on words and longs
- Open and save memory configuration
- Add, delete, or update memory blocks
Memory Configuration Modes

Use the Memory Configuration dialog box to select the memory configuration mode: auto configuration on access, auto configuration on load, or user defined. Depending on your settings, the Full Chip Simulation component initializes the Full Chip Simulation memory as Table 10.2 explains.

Table 10.2 Memory Configuration Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Configuration on Access (Standard Configuration)</td>
<td>Defines the Full Chip Simulation memory as RAM of unlimited size. The Mode combo box displays auto on access.</td>
</tr>
</tbody>
</table>
Depending on the configuration mode, the Memory Configuration dialog box lets you redefine memory settings within certain limits. You always must set I/O devices manually.

**Standard Configuration: Auto on Access:** The Memory Configuration dialog box contains a single RAM entry with unspecified (*) starting and ending addresses. You cannot modify these addresses. You can adjust wait states, and other such settings, only for the whole RAM block.

**Auto Configuration on Load:** Initially, the dialog box lists a single RAM and a single ROM block, with unspecified (*) starting and ending addresses. You can adjust wait states, and other such settings, separately for RAM and ROM blocks.

For the ELF/DWARF Object file format, the Memory Configuration dialog box lists separate RAM and ROM blocks for each data and code segment in the absolute file, once an application has been loaded. The segment addresses and lengths determine the starting and ending addresses of each block; you cannot modify these addresses. Initial attributes of each code and data block come from the corresponding initial RAM and ROM blocks; you can modify these attributes independently.

**Manual Configuration:** The Memory Configuration dialog box lists an entry for each memory block. You can modify such entries without restriction.

**NOTE** To simulate an absolute file generated in HIWARE object file format, you must open the Memory Configuration dialog box, set the “auto on load” mode, then add a new RAM segment. The start and end addresses of this segment

---

**Table 10.2 Memory Configuration Modes (continued)**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Auto Configuration on Load</em> (default)</td>
<td>Defines the Full Chip Simulation memory as RAM and ROM, according to the code and data area defined in a loaded absolute file. Defines code segments as ROM. Defines data segments as RAM. (Memory outside these segments is <em>not implemented</em>; access to not-implemented locations result in error messages.) The <em>Mode</em> combo box displays <em>auto on load</em>.</td>
</tr>
<tr>
<td><em>Manual Configuration: (User Defined)</em></td>
<td>Defines the Full Chip Simulation memory as RAM, ROM, non-volatile RAM, ..., depending on your configuration. You construct this definition interactively with the Memory Configuration dialog box, or read it in from a file. The <em>Mode</em> combo box displays <em>user defined</em>.</td>
</tr>
</tbody>
</table>
must match the associated .prm file. Once you close the dialog box, you can load your application and start a simulation.

Open Memory Block
Click the Open button to load a memory blocks file. The Open Memory blocks standard dialog box appears. Select a memory file, then click the OK button. The dialog box closes, and the system loads the memory blocks file.

The Mode combo box changes to indicate the mode contained in the memory map file.

The list box lists the memory blocks loaded from the file, selecting the first memory block. Appropriate data appears in the fields Name, Type, Start, End, Wait state, Bus width and Access Details.

Save Memory Block
Click the Save button to store the current memory blocks configuration. The Save Memory blocks standard dialog box appears. Enter a file name, then click the OK button. The dialog box closes, and the system stores the memory block configuration into the file.

Memory Check Options
The Memory Check group box consists of three checkboxes, all checked when you bring up the Memory Configuration dialog box:

- Stop if no memory — Check this box to have the Full Chip Simulation stop upon an access to non-existent memory. (If you do not want the Full Chip Simulation to stop, clear this checkbox.)

- Stop on read undefined — Check this box to have the Full Chip Simulation stop upon a read of undefined memory. (If you do not want the Full Chip Simulation to stop, clear this checkbox.)

- Stop on write protected — Check this box to have the Full Chip Simulation stop upon a write to read-only (write-protected) memory. (If you do not want the Full Chip Simulation to stop, clear this checkbox.)

Memory Configuration Module Startup
Memory configuration is a dynamically loaded facility. That is, the new entry Configure... appears in the Simulator menu upon loading of the Full Chip Simulation (the Full Chip Simulation dll). Selecting Configure..., opens the Memory Configuration dialog box, so that you can configure memory.

Memory Block Setting
You must set memory blocks within the available memory; each block must cover a certain range. The start address and end address define each memory block.
Memory Block Properties

Table 10.3 lists the properties you may specify for a memory block:

Table 10.3 Memory Block Properties

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Name of the memory block.</td>
</tr>
<tr>
<td>type</td>
<td>RAM, ROM, FLASH, EEPROM or I/O</td>
</tr>
<tr>
<td>start</td>
<td>Start address of the memory block</td>
</tr>
<tr>
<td>end</td>
<td>End address of the memory block</td>
</tr>
<tr>
<td>wait state</td>
<td>Time used for reading or writing a specific number of bytes</td>
</tr>
<tr>
<td>bus width</td>
<td>Width of the bus that accesses the memory</td>
</tr>
<tr>
<td>read access</td>
<td>Table that defines read-access details on Byte, Word, Word misaligned, Long, and Long misaligned</td>
</tr>
<tr>
<td>write access</td>
<td>Table that defines write-access details on Byte, Word, Word misaligned, Long, and Long misaligned</td>
</tr>
<tr>
<td>auto configure</td>
<td>Flag that directs automatic computation of read and write accesses</td>
</tr>
<tr>
<td>allow misaligned access</td>
<td>Flag that allows Word misaligned and Long misaligned</td>
</tr>
<tr>
<td>block type</td>
<td>USER_DEF (block you define), AUTO_GEN (block automatically generated), AUTO_MEM (master block for standard configuration), AUTO_RAM (RAM master block for auto configuration), or AUTO_ROM (ROM master block for auto configuration)</td>
</tr>
</tbody>
</table>

Memory Configuration Dialog Box Command Buttons

The command buttons of the Memory Configuration dialog box are:

- **Add** — Fills a new memory block according to the current data of the **Name**, **Type**, **Start**, **End**, **Bus width**, and **Access Details** controls. This new memory block appears at the end of the list box. If there are any errors in this new block (such as an improper field value), the system generates a message box that informs you of the problem.

- **Update** — Updates the current memory block according to the current data of the **Name**, **Type**, **Start**, **End**, **Bus width**, and **Access Details** controls.
Delete — Removes the currently selected memory block from the list box. The list box contents adjust, to reflect this deletion.

OK — Closes the dialog box and validates the list of modified memory blocks. The parent class can access this list, updating its own list.

Cancel — Closes the dialog box, canceling your modifications.

Help — Opens the dialog-box help file.

Access Details Dialog Box

Figure 10.7 shows the Access Details dialog box, which lets you change read and write access values for seven types.

Follow this guidance to use the Access Details dialog box:

- A check box indicates if an access kind is allowed or not.
- To modify the value of each read or write type, change the value of the associated spin box.
- The lowest possible value is 0.
- The highest possible value is 127.
- To store changes into the currently selected memory block, click the OK button. The Access Details dialog box disappears, and the system clears the Auto Configure checkbox.
Output

You can save the current memory configuration into the file you defined at the outset.

Clock Frequency Setup

The Full Chip Simulation provides a **true time information**. It is possible to provide an oscillator clock frequency to the debugger. The debugger CPU awareness and io modules provide the "clock factor" to apply to this input frequency to derive the CPU cycle frequency.

Figure 10.8 Clock Frequency Setup Dialog Box

Derivative specific IO simulations caring of bus speed change (multiply or divide) through PLL modules dynamically update the clock factor, i.e. while the application is simulation is running.

Accumulated elapsed time will not be affected and a new cycle time is applied to next simulated instructions in real time.

The Clock Frequency Setup dialog (Simulator->Clock Frequency menu entry) can be opened to set enter/edit either the oscillator frequency or the CPU frequency. However, the project saved frequency is the oscillator one. Two radio buttons allow choosing between cycles or true time display in debugger status bar.

Unchecking "reset cycles/time..." will simply make the debugger cumulate cycles/time other CPU reset. The true time unit is the microsecond. The "TRUETIME" debugger command line command gives the time as a number in microseconds. The "OSCFREQUENCY" variable displays/sets the oscillator frequency.
Bus Trace

The Full Chip Simulation is able to record all executed instructions and memory accesses in the Trace component, this up to one million of frames. Enabling recording can be done in the Trace menu/context sensitive popup menu, after opening the Trace component.

NOTE Please refer to the “HCS12 (or HCS12X) Onchip DBG Module” manual for the Trace window common functionality and common menu entries.

Figure 10.9 Trace Window Popup Menu

By default, the Full Chip Simulation records instructions only (faster). Check “Record memory accesses” and choose “Textual” mode in the Trace menu/context sensitive popup menu to record memory accesses.

Many information can be retrieved from the Trace window, like:

- instructions and instruction addresses,
- data address, data value and read/write access type,
- true time, cycles and total simulation cycles for each instruction,
- function name and module name for each instruction,
- variable name and module name for each global variable data access.
By default, the Full Chip Simulation generates warning messages when the application accesses onchip registers that are not implemented for the selected derivative. These warnings are displayed in the Command window.

For example, the following messages can be indefinitely repeated in the Command window:

```
...  
  ...  
  ...  
FCS Warning (ID 12): reading from unimplemented register at pc = 0x400a'L. 
Value: 0x0, Memory Address: 0x106. FLASH CONTROL module not implemented
```

Warning message “IDs” belong usually to a group of registers from the same simulated block, here above the “FLASH CONTROL” registers block. Therefore, any access to an unimplemented “FLASH CONTROL” register generates the same kind of message.
The debugger provides a set of (volatile, not saved in current project) commands to hide specific ID messages, or to stop the Full Chip Simulation automatically or pop up a warning message box. The commands can be executed from a POSTLOAD command file.

**WARNING_SETUP Command**

The **WARNING_SETUP** command sets the level of debugger warning to inform the user about the usage of a not simulated register.

**Components**

Debugger engine.

**Usage**

```plaintext
WARNING_SETUP <HALT|CLMSG|MSGBOX|NONE|STATUS>
```

- **WARNING_SETUP STATUS**: displays the current warning setup status.

**Example:**

```plaintext
in>warning_setup status
WARNING_SETUP STATUS: CLMSG
```

- **WARNING_SETUP HALT**: The FCS simply stops/halts the debugger when a warning message occurs.

**Example:**

```plaintext
in>warning_setup none
in>warning_setup halt
in>warning_setup status
WARNING_SETUP STATUS: HALT
```

- **WARNING_SETUP CLMSG**: Warning messages are displayed in the Command window (debugger default).

**Example:**

```plaintext
in>warning_setup none
in>warning_setup clmsg
in>warning_setup status
```
WARNING_SETUP STATUS: CLMSG

- **WARNING_SETUP MSGBOX**: A message box pops up on warning. Pressing Cancel stops the FCS. Pressing OK resumes the FCS.

**Figure 10.11** FCS Warning Message Box

![FCS Warning Message Box](image)

**Example:**

```
in>warning_setup none
in>warning_setup msgbox
in>warning_setup status
WARNING_SETUP STATUS: MSGBOX
```

- **WARNING_SETUP NONE**: clears all kind of warning messages.

```
in>warning_setup none
in>warning_setup status
WARNING_SETUP STATUS: No warning messages
```

**NOTE**  With HALT, CLMSG and MSGBOX options, executing several times the command toggles on and off the setup.

**MESSAGE_HIDE_ID Command**

The MESSAGE_HIDE_ID command hides a message of a specific ID.

**Components**

Debugger engine.

**Usage**
MESSAGE_HIDE_ID  <message number(ID)>

Example:

in>MESSAGE_HIDE_ID 1
in>warning_setup status
  WARNING_SETUP STATUS: CLMSG
  Hidden message ID: 1

MESSAGE_SHOW_ID Command

The MESSAGE_SHOW_ID shows back the hidden message of a specific ID.

Components
Debugger engine.

Usage
MESSAGE_SHOW_ID  <message number(ID)>

Example:

in>MESSAGE_SHOW_ID 1

MESSAGE_HIDE_RESET Command

The MESSAGE_HIDE_RESET commands resets all hidden messages to display them again.

Components
Debugger engine.

Usage
MESSAGE_HIDE_RESET

Example:

in>MESSAGE_HIDE_RESET

All previously hidden messages are displayed again now.
FCS and Silicon On-chip Peripherals Simulation

Full Chip Simulation means not only to simulate the core instruction set but also the on-chip i/o devices such as (CRG, PWM, ECT, ...). In the section Supported Derivatives the supported i/o devices are listed for each supported derivative.

By generating a new project with the ‘New HC(12) Project Wizard’ and the connection ‘Full Chip Simulation’ everything is setup to run the project with FCS support.

With the menu option ‘Simulator > Set Derivative’ you can change the derivative to simulate. Additional to the derivatives you will find special entries: HC(S)12(X) CORE and HC(S)12(X) SAMPLE. The CORE entries allow to simulate the chip without FCS support (plain instructions only) and the SAMPLE entries allow to simulate a chip with a minimal set of commonly available onchip peripherals, like Register Block, Memory Expansion Registers, Clock and Reset Generator, Serial Communication Interface “0” and PortB.

Figure 10.12 ‘Set Derivative Dialog Box

The current mode of Simulation (SAMPLE, CORE or MCU derivative) can be seen in the statusbar. To access the ‘Set Derivative’ dialog it is also possible to double click on the FCS support entry in the statusbar. Please see Debugger Status Bar with Full Chip Simulation.
## Supported Derivatives

Supported derivatives are listed here below. To simulate unlisted derivatives, either use a derivative with similar on-chip peripherals, or use the “SAMPLE” or “CORE” mode of the FCS.

<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC9S12A32</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td></td>
<td>Clock and Reset Generator (CRG)</td>
</tr>
<tr>
<td></td>
<td>Enhanced Capture Timer (ECT)</td>
</tr>
<tr>
<td></td>
<td>EEPROM (EETS)</td>
</tr>
<tr>
<td></td>
<td>Flash (FTS)</td>
</tr>
<tr>
<td></td>
<td>Multiplexed External Bus Interface (MEBI)</td>
</tr>
<tr>
<td></td>
<td>Motorola Scalable CAN (MSCAN)</td>
</tr>
<tr>
<td></td>
<td>Port Integration Module (PIM)</td>
</tr>
<tr>
<td></td>
<td>Pulse Width Modulator (PWM)</td>
</tr>
<tr>
<td></td>
<td>Serial Communication Interface</td>
</tr>
<tr>
<td></td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td></td>
<td>Voltage Regulator (VREG)</td>
</tr>
<tr>
<td>MC9S12A64</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td></td>
<td>J1850 Bus (BLCD)</td>
</tr>
<tr>
<td></td>
<td>Clock and Reset Generator (CRG)</td>
</tr>
<tr>
<td></td>
<td>Enhanced Capture Timer (ECT)</td>
</tr>
<tr>
<td></td>
<td>EEPROM (EETS)</td>
</tr>
<tr>
<td></td>
<td>Flash (FTS)</td>
</tr>
<tr>
<td></td>
<td>Inter-IC Bus (IIC)</td>
</tr>
<tr>
<td></td>
<td>Multiplexed External Bus Interface (MEBI)</td>
</tr>
<tr>
<td></td>
<td>Port Integration Module (PIM)</td>
</tr>
<tr>
<td></td>
<td>Pulse Width Modulator (PWM)</td>
</tr>
<tr>
<td></td>
<td>Serial Communication Interface</td>
</tr>
<tr>
<td></td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td></td>
<td>Voltage Regulator (VREG)</td>
</tr>
</tbody>
</table>
Table 10.4  Supported Derivatives (continued)

<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
</table>
| MC9S12C32       | Analog to Digital Converter  
|                 | Clock and Reset Generator (CRG)  
|                 | Flash (FTS)  
|                 | Multiplexed External Bus Interface (MEBI)  
|                 | Motorola Scalable CAN (MSCAN)  
|                 | Port Integration Module (PIM)  
|                 | Pulse Width Modulator (PWM)  
|                 | Serial Communication Interface  
|                 | Serial Peripheral Interface  
|                 | Timer Module (TIM)  
|                 | Voltage Regulator (VREG) |
| MC9S12D32       | Analog to Digital Converter  
|                 | Clock and Reset Generator (CRG)  
|                 | Enhanced Capture Timer (ECT)  
|                 | EEPROM (EETS)  
|                 | Flash (FTS)  
|                 | Multiplexed External Bus Interface (MEBI)  
|                 | Motorola Scalable CAN (MSCAN)  
|                 | Port Integration Module (PIM)  
|                 | Pulse Width Modulator (PWM)  
|                 | Serial Communication Interface  
|                 | Serial Peripheral Interface  
|                 | Voltage Regulator (VREG) |
### Table 10.4 Supported Derivatives (continued)

<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC9S12D64</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td></td>
<td>Clock and Reset Generator (CRG)</td>
</tr>
<tr>
<td></td>
<td>Enhanced Capture Timer (ECT)</td>
</tr>
<tr>
<td></td>
<td>EEPROM (EETS)</td>
</tr>
<tr>
<td></td>
<td>Flash (FTS)</td>
</tr>
<tr>
<td></td>
<td>Inter-IC Bus (IIC)</td>
</tr>
<tr>
<td></td>
<td>Multiplexed External Bus Interface (MEBI)</td>
</tr>
<tr>
<td></td>
<td>Motorola Scalable CAN (MSCAN)</td>
</tr>
<tr>
<td></td>
<td>Port Integration Module (PIM)</td>
</tr>
<tr>
<td></td>
<td>Pulse Width Modulator (PWM)</td>
</tr>
<tr>
<td></td>
<td>Serial Communication Interface</td>
</tr>
<tr>
<td></td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td></td>
<td>Voltage Regulator (VREG)</td>
</tr>
<tr>
<td>MC9S12DB128A</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td></td>
<td>Byteflight (BF)</td>
</tr>
<tr>
<td></td>
<td>Clock and Reset Generator (CRG)</td>
</tr>
<tr>
<td></td>
<td>Enhanced Capture Timer (ECT)</td>
</tr>
<tr>
<td></td>
<td>EEPROM (EETS)</td>
</tr>
<tr>
<td></td>
<td>Flash (FTS)</td>
</tr>
<tr>
<td></td>
<td>Multiplexed External Bus Interface (MEBI)</td>
</tr>
<tr>
<td></td>
<td>Motorola Scalable CAN (MSCAN)</td>
</tr>
<tr>
<td></td>
<td>Port Integration Module (PIM)</td>
</tr>
<tr>
<td></td>
<td>Pulse Width Modulator (PWM)</td>
</tr>
<tr>
<td></td>
<td>Serial Communication Interface</td>
</tr>
<tr>
<td></td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td></td>
<td>Voltage Regulator (VREG)</td>
</tr>
</tbody>
</table>
Table 10.4  Supported Derivatives (continued)

<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
</table>
| MC9S12DB128B    | Analog to Digital Converter  
|                 | Byteflight (BF)  
|                 | Clock and Reset Generator (CRG)  
|                 | Enhanced Capture Timer (ECT)  
|                 | EEPROM (EETS)  
|                 | Flash (FTS)  
|                 | Multiplexed External Bus Interface (MEBI)  
|                 | Motorola Scalable CAN (MSCAN)  
|                 | Port Integration Module (PIM)  
|                 | Pulse Width Modulator (PWM)  
|                 | Serial Communication Interface  
|                 | Serial Peripheral Interface  
|                 | Voltage Regulator (VREG)  |
| MC9S12DG128B    | Analog to Digital Converter  
|                 | Clock and Reset Generator (CRG)  
|                 | Enhanced Capture Timer (ECT)  
|                 | EEPROM (EETS)  
|                 | Flash (FTS)  
|                 | Inter-IC Bus (I2C)  
|                 | Multiplexed External Bus Interface (MEBI)  
|                 | Motorola Scalable CAN (MSCAN)  
|                 | Port Integration Module (PIM)  
|                 | Pulse Width Modulator (PWM)  
|                 | Serial Communication Interface  
|                 | Serial Peripheral Interface  
|                 | Voltage Regulator (VREG)  |
Table 10.4 Supported Derivatives (continued)

<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
</table>
| MC9S12DG256B    | Analog to Digital Converter  
|                  | Clock and Reset Generator (CRG)  
|                  | Enhanced Capture Timer (ECT)  
|                  | EEPROM (EETS)  
|                  | Flash (FTS)  
|                  | Inter-IC Bus (IIC)  
|                  | Multiplexed External Bus Interface (MEBI)  
|                  | Motorola Scalable CAN (MSCAN)  
|                  | Port Integration Module (PIM)  
|                  | Pulse Width Modulator (PWM)  
|                  | Serial Communication Interface  
|                  | Serial Peripheral Interface  
|                  | Voltage Regulator (VREG) |
| MC9S12DJ128B    | Analog to Digital Converter  
|                  | J1850 Bus (BLCD)  
|                  | Clock and Reset Generator (CRG)  
|                  | Enhanced Capture Timer (ECT)  
|                  | EEPROM (EETS)  
|                  | Flash (FTS)  
|                  | Inter-IC Bus (IIC)  
|                  | Multiplexed External Bus Interface (MEBI)  
|                  | Motorola Scalable CAN (MSCAN)  
|                  | Port Integration Module (PIM)  
|                  | Pulse Width Modulator (PWM)  
|                  | Serial Communication Interface  
|                  | Serial Peripheral Interface  
<p>|                  | Voltage Regulator (VREG) |</p>
<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC9S12DJ256B</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td></td>
<td>J1850 Bus (BLCD)</td>
</tr>
<tr>
<td></td>
<td>Clock and Reset Generator (CRG)</td>
</tr>
<tr>
<td></td>
<td>Enhanced Capture Timer (ECT)</td>
</tr>
<tr>
<td></td>
<td>EEPROM (EETS)</td>
</tr>
<tr>
<td></td>
<td>Flash (FTS)</td>
</tr>
<tr>
<td></td>
<td>Inter-IC Bus (IIC)</td>
</tr>
<tr>
<td></td>
<td>Multiplexed External Bus Interface (MEBI)</td>
</tr>
<tr>
<td></td>
<td>Motorola Scalable CAN (MSCAN)</td>
</tr>
<tr>
<td></td>
<td>Port Integration Module (PIM)</td>
</tr>
<tr>
<td></td>
<td>Pulse Width Modulator (PWM)</td>
</tr>
<tr>
<td></td>
<td>Serial Communication Interface</td>
</tr>
<tr>
<td></td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td></td>
<td>Voltage Regulator (VREG)</td>
</tr>
<tr>
<td>MC9S12DJ64</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td></td>
<td>J1850 Bus (BLCD)</td>
</tr>
<tr>
<td></td>
<td>Clock and Reset Generator (CRG)</td>
</tr>
<tr>
<td></td>
<td>Enhanced Capture Timer (ECT)</td>
</tr>
<tr>
<td></td>
<td>EEPROM (EETS)</td>
</tr>
<tr>
<td></td>
<td>Flash (FTS)</td>
</tr>
<tr>
<td></td>
<td>Inter-IC Bus (IIC)</td>
</tr>
<tr>
<td></td>
<td>Multiplexed External Bus Interface (MEBI)</td>
</tr>
<tr>
<td></td>
<td>Motorola Scalable CAN (MSCAN)</td>
</tr>
<tr>
<td></td>
<td>Port Integration Module (PIM)</td>
</tr>
<tr>
<td></td>
<td>Pulse Width Modulator (PWM)</td>
</tr>
<tr>
<td></td>
<td>Serial Communication Interface</td>
</tr>
<tr>
<td></td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td></td>
<td>Voltage Regulator (VREG)</td>
</tr>
</tbody>
</table>
### Table 10.4 Supported Derivatives (continued)

<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
</table>
| MC9S12DP256B    | Analog to Digital Converter  
|                 | J1850 Bus (BLCD)  
|                 | Clock and Reset Generator (CRG)  
|                 | Enhanced Capture Timer (ECT)  
|                 | EEPROM (EETS)  
|                 | Flash (FTS)  
|                 | Inter-IC Bus (IIC)  
|                 | Multiplexed External Bus Interface (MEBI)  
|                 | Motorola Scalable CAN (MSCAN)  
|                 | Port Integration Module (PIM)  
|                 | Pulse Width Modulator (PWM)  
|                 | Serial Communication Interface  
|                 | Serial Peripheral Interface  
|                 | Voltage Regulator (VREG)  |
| MC9S12DP512     | Analog to Digital Converter  
|                 | J1850 Bus (BLCD)  
|                 | Clock and Reset Generator (CRG)  
|                 | Enhanced Capture Timer (ECT)  
|                 | EEPROM (EETS)  
|                 | Flash (FTS)  
|                 | Inter-IC Bus (IIC)  
|                 | Multiplexed External Bus Interface (MEBI)  
|                 | Motorola Scalable CAN (MSCAN)  
|                 | Port Integration Module (PIM)  
|                 | Pulse Width Modulator (PWM)  
|                 | Serial Communication Interface  
|                 | Serial Peripheral Interface  
|                 | Voltage Regulator (VREG)  |
### Table 10.4 Supported Derivatives (continued)

<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
</table>
| **MC9S12DT128B** | Analog to Digital Converter  
Clock and Reset Generator (CRG)  
Enhanced Capture Timer (ECT)  
EEPROM (EETS)  
Flash (FTS)  
Inter-IC Bus (IIC)  
Multiplexed External Bus Interface (MEBI)  
Motorola Scalable CAN (MSCAN)  
Port Integration Module (PIM)  
Pulse Width Modulator (PWM)  
Serial Communication Interface  
Serial Peripheral Interface  
Voltage Regulator (VREG) |
| **MC9S12DT256B** | Analog to Digital Converter  
Clock and Reset Generator (CRG)  
Enhanced Capture Timer (ECT)  
EEPROM (EETS)  
Flash (FTS)  
Inter-IC Bus (IIC)  
Multiplexed External Bus Interface (MEBI)  
Motorola Scalable CAN (MSCAN)  
Port Integration Module (PIM)  
Pulse Width Modulator (PWM)  
Serial Communication Interface  
Serial Peripheral Interface  
Voltage Regulator (VREG) |
<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
</table>
| MC9S12XDP512   | Analog to Digital Converter  
|                | Clock and Reset Generator (CRG) |
|                | Debug Module (DBG) |
|                | External Bus Interface (EBI) |
|                | Enhanced Capture Timer (ECT) |
|                | EEPROM (EETS) |
|                | Flash (FTS) |
|                | Inter-IC Bus (IIC) |
|                | Module Mapping Control (MMC) |
|                | Motorola Scalable CAN (MSCAN) |
|                | Port Integration Module (PIM) |
|                | Periodic Interrupt Timer (PIT) |
|                | Pulse Width Modulator (PWM) |
|                | S12X_INT |
|                | Serial Communication Interface |
|                | Serial Peripheral Interface |
|                | Voltage Regulator (VREG) |
|                | XGATE |
Communication Modules

Byteflight (BF)
The I/O device Byteflight (BF) is not simulated.

J1850 Bus (BLCD)
The I/O device J1850 Bus (BLCD) is not simulated.

Motorola Scalable CAN (MSCAN)
The I/O device Motorola Scalable CAN (MSCAN) is not simulated.

Table 10.4  Supported Derivatives (continued)

<table>
<thead>
<tr>
<th>Derivative Name</th>
<th>Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC9S12XD T512</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td></td>
<td>Clock and Reset Generator (CRG)</td>
</tr>
<tr>
<td></td>
<td>Debug Module (DBG)</td>
</tr>
<tr>
<td></td>
<td>External Bus Interface (EBI)</td>
</tr>
<tr>
<td></td>
<td>Enhanced Capture Timer (ECT)</td>
</tr>
<tr>
<td></td>
<td>EEPROM (EETS)</td>
</tr>
<tr>
<td></td>
<td>Flash (FTS)</td>
</tr>
<tr>
<td></td>
<td>Inter-IC Bus (IIC)</td>
</tr>
<tr>
<td></td>
<td>Module Mapping Control (MMC)</td>
</tr>
<tr>
<td></td>
<td>Motorola Scalable CAN (MSCAN)</td>
</tr>
<tr>
<td></td>
<td>Port Integration Module (PIM)</td>
</tr>
<tr>
<td></td>
<td>Periodic Interrupt Timer (PIT)</td>
</tr>
<tr>
<td></td>
<td>Pulse Width Modulator (PWM)</td>
</tr>
<tr>
<td></td>
<td>S12X_INT</td>
</tr>
<tr>
<td></td>
<td>Serial Communication Interface</td>
</tr>
<tr>
<td></td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td></td>
<td>Voltage Regulator (VREG)</td>
</tr>
<tr>
<td></td>
<td>XGATE</td>
</tr>
</tbody>
</table>
Inter-IC Bus (IIC)

The I/O device Inter-IC Bus (IIC) is not simulated.

Serial Communication Interface

This I/O device simulates the Serial Communication Interface (SCI). The not memory mapped registers ‘SCIInput/SCIInputH’ and ‘SerialInput’ serve to send characters to the SCI Module. The not memory mapped registers ‘SCIOutput/SCIOutputH’ and ‘SerialOutput’ contain the characters sent from to the SCI Module.

Registers:

SC0BDH (SCI Baud Rate Register High)
SBR12, SBR11, SBR10, SBR9 and SBR8 are simulated.

SC0BDL (SCI Baud Rate Register Low)
SBR7, SBR6, SBR5, SBR4, SBR3, SBR2, SBR1 and SBR0 are simulated.

SC0CR1 (SCI Control Register 1)
M and ILT are simulated.

SC0CR2 (SCI Control Register 2)
TIE, TCIE, RIE, ILIE, TE, RE and SBK are simulated.

SC0SR1 (SCI Status Register 1)
TDRE, TC, RDRF, IDLE and OR are simulated.

SC0SR2 (SCI Status Register 2)
RAF is simulated.

SC0DRH (SCI Data Register High)
R8 and T8 are simulated.

SC0DRL (SCI Data Register Low)
R7/T7, R6/T6, R5/T5, R4/T4, R3/T3, R2/T2, R1/T1 and R0/T0 are simulated.
Not-memory-mapped Registers

SCInput

This is a not memory mapped register and will serve to send a character to the SCI. This value will be received from the SCI and can be read through a read access to the SCDR. The ninth bit is taken from the SCInputH register. A read access to SCInput has no specified meaning.

Bit 7..0 character send to the SCI.

SCInputH

This is a not memory mapped register and will serve to send a character to the SCI. It contains the ninth bit. This register must be written before the SCInput register. A read access to SCInputH has no specified meaning.

Bit 0 ninth bit send to the SCI.

SCIOutput

This is a not memory mapped register and will serve to receive a character which is sent from the SCI. The value received in the SCIOutput is triggered through a write access to the SCDR. The ninth bit is written to the SCIOutputH register. A write access to SCIOutput has no specified meaning.

Bit 7..0 character send from the SCI.

SCIOutputH

This is a not memory mapped register and will serve to receive a character which is sent from the SCI. It contains the ninth bit. A write access to SCIOutput has no specified meaning.

Bit 0 ninth bit send from the SCI

SerialInput

This not memory mapped register is an alias for the SCInput register and serve to connect the SCI to the terminal window. The ninth bit is not supported. A read access to SerialInput has no specified meaning.

Bit 7..0 data from terminal window to SCI

SerialOutput

This not memory mapped register is an alias for the SCIOutput register and serve to connect the SCI to the terminal window. The ninth bit is not supported. A write access to SerialOutput has no specified meaning.

Bit 7..0 data sent from SCI to terminal window
Serial Peripheral Interface
This I/O device simulates the Serial Peripheral Interface (SPI).

Registers:

**SPI CR1 (SPI Control Register 1)**
SPIE, SPE, MSTR, CPOL, CPHA and LSBFE are simulated.

**SPI CR2 (SPI Control Register 2)**
SPISWAI and SPC0 are simulated.

**SPI BR (SPI Baud Rate Register)**
SPPR2, SPPR1, SPPR0, SPR2, SPR1 and SPR0 are simulated.

**SPI SR (SPI Status Register)**
SPIF, SPTEF and MODF are simulated.

**SPI DR (SPI Data Register)**
Bit 7..0 are simulated.

Not-memory-mapped Registers

**SPI Value**
This is a not-memory-mapped register and will serve to sent and receive (swap) a character from and to the SPI.
Bit 7..0 data sent from/to SPI

Converter Modules

Analog to Digital Converter
This I/O device simulates the Analog to Digital Converter (ATD). 8 channels and 16 channel versions of the ATD module are supported. The analog inputs are reachable separately through the object pool. They are called PAD0 to PAD7/PAD15. For the ATD module 1, PAD0 input corresponds to the PAD8/PAD16 pin of the microcontroller.
Conversion Results

The analog inputs of ATD module are simulated as 8-bit logic values. Therefore, the simulation of the conversion itself only has a limited interest. The conversion result will be an image of the simulated input.

For the unsigned, right justified 8-bit conversion, the result displayed in the corresponding data register will be the exact image of the input.

Still, the simulation is accurate on the conversion delays, the modification that affect the input (8-10 bits, left/right justified, signed/unsigned), the data registers in which the conversion results should be transferred and gives a precise image on how the ATD modules should be configured for proper conversion process.

Registers:

**ATDCTL2 (ATD Control Register 2)**
ADPU, AFFC, AWAI, ETRIGLE, ETRIGP, ETRIGE, ASCIE and ASCIF are simulated.

**ATDCTL3 (ATD Control Register 3)**
S8C, S4C, S2C and S1C are simulated.

**ATDCTL4 (ATD Control Register 4)**
SRES8, SMP1, SMP0, PRS4, PRS3, PRS2, PRS1 and PRS0 are simulated.

**ATDCTL5 (ATD Control Register 5)**
DJM, DSGN, SCAN, MULT, CC, CB and CA are simulated.

**ATDSTAT0 (ATD Status Register 0)**
SCF, ETORF, FIFOR, CC2, CC1 and CC0 are simulated.

**ATDSTAT1 (ATD Status Register 1)**
CCF7, CCF6, CCF5, CCF4, CCF3, CCF2, CCF1 and CCF0 are simulated.

**ATDDIEN (ATD Input Enable Register) (8 Channel)**
IEN7, IEN6, IEN5, IEN4, IEN3, IEN2, IEN1 and IEN0 are simulated.

**ATDDIEN0 (ATD Input Enable Register) (16 Channel)**
IEN15, IEN14, IEN13, IEN12, IEN11, IEN10, IEN9 and IEN8 are simulated.

**ATDDIEN1 (ATD Input Enable Register) (16 Channel)**
IEN7, IEN6, IEN5, IEN4, IEN3, IEN2, IEN1 and IEN0 are simulated.
PORTAD (Port Data Register) (8 Channel)
PTAD7, PTAD6, PTAD5, PTAD4, PTAD3, PTAD2, PTAD1, PTAD0 are simulated.

PORTAD0 (Port Data Register) (16 Channel)
PTAD15, PTAD14, PTAD13, PTAD12, PTAD11, PTAD10, PTAD9, PTAD8 are simulated.

PORTAD1 (Port Data Register) (16 Channel)
PTAD7, PTAD6, PTAD5, PTAD4, PTAD3, PTAD2, PTAD1, PTAD0 are simulated.

ATDDRx (ATD Conversion Result Registers)
Is simulated.

Not-memory-mapped Registers
PADx
This are eight not-memory-mapped registers that will serve to be the ‘measured’ values for the ATD. The format of each 4 bytes big PAD is IEEE32. To setup a PAD easier the following command can be used:
ATDx_SETPAD <CHANNEL> <VOLTAGE AS FLOAT>

Memory Modules

EEPROM (EETS)
The I/O device EEPPROM (EETS) is not simulated.

Flash (FTS)
The I/O device Flash (FTS) is not simulated.

Miscellaneous Modules

Voltage Regulator (VREG)
The I/O device Voltage Regulator (VREG) is not simulated.
Debug Module (DBG)
The I/O device Debug Module (DBG) is not simulated.

S12X_INT

Registers:
IVBR (Interrupt Vector Base Register)
Is simulated

INT_XGPRIO (XGATE Interrupt Priority Config. Reg.)
Is simulated

INT_CFDADDR (Interrupt Req. Config. Address Reg.)
Are simulated

INT_CFDATA0...7 (Interrupt Req. Config. Data Reg.)
Are simulated

XGATE

Registers:
XGMCTL (XGATE Module Control Register)
Is simulated

XGCHID (XGATE Channel ID Register)
Is simulated

XGVBR (XGATE Vector Base Address)
Is simulated

XGIF (XGATE Interrupt Flag Vector)
Is simulated

XGSWT (XGATE Software Trigger Register)
Is simulated
XGSEM (XGATE Semaphore Register)
Is simulated

XGCCR (XGATE Condition Code Register)
Is simulated

XGPC (XGATE Program Counter)
Is simulated

XGR1 (XGATE Register 1)
Is simulated

XGR2 (XGATE Register 2)
Is simulated

XGR3 (XGATE Register 3)
Is simulated

XGR4 (XGATE Register 4)
Is simulated

XGR5 (XGATE Register 5)
Is simulated

XGR6 (XGATE Register 6)
Is simulated

XGR7 (XGATE Register 7)
Is simulated

Port I/O Modules

External Bus Interface (EBI)
The I/O device External Bus Interface (EBI) is not simulated.
Module Mapping Control (MMC)

Registers:

**GPAGE (Global Page Index Register)**
Is simulated.

**DIRECT (Direct Page Register)**
Is simulated.

**Miscellaneous System Control Register**
Is not simulated.

**MTSTO (Reserved Test Register Zero)**
Is not simulated.

**RPAGE (RAM Page Index Register)**
Is simulated.

**EPAGE (EEPROM Page Index Register)**
Is simulated.

**PPAGE (Program Page Index Register)**
Is simulated.

**Multiplexed External Bus Interface (MEBI)**

This I/O device simulates the Multiplexed External Bus Interface (MEBI). The MEBI block is part of the Core and its description can be found in the CORE manual. This block controls the behavior of the ports A, B, E and K, the IRQ and XIRQ signals and the operating mode of the Core (normal/extended/special…).

In the Full Chip Simulation, only the single chip mode is simulated. Therefore ports A and B cannot be used as external bus lines.

Only the I/O behavior of the ports is simulated, except for port E. The IRQ and XIRQ functionality going through port E pins 0 and 1 are simulated together with the various I/O enabling conditions of the port E pins described in the PEAR register. When a port E pin is not selected as an I/O pin, it stays at 0, other functionality are not simulated.
Port Integration Module (PIM)

This I/O device simulates the Port Integration Module (PIM). The PIM module controls all the ports that are not directly associated to the CORE. All registers present in the PIM module are port specific apart from the MODRR register that affects ports S, P, M, J and H. All port specific registers have been implemented together with the interrupt logic associated.

Timer Modules

Clock and Reset Generator (CRG)

This I/O device simulates the Clock and Reset Generator (CRG). The simulated parts of the CRG are the PLL, RTI and COP. Additional features of the CRG such as hardware failures of the oscillator system are not simulated.

The PLL output clock frequency \( \text{PLLCLK} = 2 \times \text{OSCCLK} \times (\text{SYNR} + 1)/(\text{REFDV} + 1) \).

The PLL block is considered as a frequency converter, other functionality of the PLL in the hardware have been ignored.

Reference Clock

The reference clock of the CRG module is CLK24 given at the output. The CLK3 and CLK23 clocks are not simulated.

When PLLSEL is set to 0, the oscillator clock frequency (used by the RTI and COP) is the same as the reference clock frequency.

When PLLSET is set to 1, OSCCLK frequency = CLK24 * (REFDV + 1) / (2 * (SYNR + 1)).

As some systems might not work for a CLK24 frequency less than the OSCCLK frequency on the hardware, the simulation does not accept it and a warning message is generated.

Any OSCCLK frequency set to be greater than the CLK24 frequency will have the same frequency as the CLK24.

Blocks:

PLL (Phase lock Loop)

The clock divider functionality of the PLL are simulated, this includes the REFDV and the SYNR registers and the PLLSEL bit in the CLKSEL register.
Changing the value of the PLLSEL bit will automatically update the COP and the RTI events, this may cause cycle irregularities as described in the manual. For proper use of the COP and RTI, these modules should be enabled after changing PLLSEL.

A stabilization time is simulated for the PLL, it ranges from 100 to 1500 clock cycles after REFDV or SYNR registers have been modified.

Setting PLLSEL to ‘1’ before this stabilization time elapses will generate a warning message. The Full Chip Simulation will operate properly but the corresponding program might not work on the hardware.

**RTI (Real Time Interrupt) and COP**

The reference clock for these events is CLK24, if OSCCLK is different from CLK24, the RTI and COP period will be adapted to the clock difference.

**Registers:**

**SYNR (CRG Synthesizer Register)**
SYN5, SYN4, SYN3, SYN2, SYN1 and SYN0 are simulated.

**REFDV (CRG Reference Divider Register)**
REFDV3, REFDV2, REFDV1 and REFDV0 are simulated.

**CRGFLG (CRG Flags Register)**
RTIF is simulated.

**CRGINT (CRG Interrupt Enable Register)**
RTIE is simulated.

**CLKSEL (CRG Clock Select Register)**
PLLSEL is simulated.

**PLLCTL (CRG PLL Control Register)**
Is not simulated.

**RTICTL (CRG RTI Control Register)**
RTR6, RTR5, RTR4, RTR3, RTR2, RTR1 and RTR0 are simulated. The RTDEC is also simulated if the derivative supports it.

**COPCTL (CRG COP Control Register)**
WCOP, RSBCK, CR2, CR1 and CR0 are simulated.
ARMCOP (CRG COP Timer Arm/Reset Register)
Is simulated.

Enhanced Capture Timer (ECT)
This I/O device simulates the Enhanced Capture Timer (ECT). The various functionality are cycle accurate up to 99%. The simulation might differ from the hardware concerning the pipelining of the instructions; some interruptions might be raised with a delay of one instruction.
The function with error detected in the hardware are not simulated, one mode of operation being used as default, further information are given in the case of not implemented features.

Modes of Operation
NORMAL and STOP mode are implemented, when entering the FREEZE or WAIT mode, the system behaves like in STOP mode.

Registers:

TIOS (Timer Input Capture/Output Compare Select Register)
IOS7, IOS6, IOS5, IOS4, IOS3, IOS2, IOS1 and IOS0 are simulated.

CFORC (Timer Compare Force Register)
FOC7, FOC6, FOC5, FOC4, FOC3, FOC2, FOC1 and FOC0 are simulated.

OC7M (Output Compare 7 Mask Register)
OC7M7, OC7M6, OC7M5, OC7M4, OC7M3, OC7M2, OC7M1 and OC7M are simulated.

OC7D (Output Compare 7 Data Register)
OC7D7, OC7D6, OC7D5, OC7D4, OC7D3, OC7D2, OC7D1 and OC7D0 are simulated.

TCNT (Timer Count Register)
Partly simulated: In the test mode TCNT is not writable.

TSCR1 (Timer System Control Register 1)
TEN and TFFCA are simulated.

TTOV (Timer Toggle On Overflow Register 1)
TOV7, TOV6, TOV5, TOV4, TOV3, TOV2, TOV1 and TOV0 are simulated.
TCTL1/TCTL2 (Timer Control Register 1-2)
OM7, OL7, OM6, OL6, OM5, OL5, OM4, OL4,
OM3, OL3, OM2, OL2, OM1, OL1, OM0 and OL0 are simulated.

TCTL3/TCTL4 (Timer Control Register 3-4)
EDG7B, EDG7A, EDG6B, EDG6A, EDG5B, EDG5A, EDG4B, EDG4A,
EDG3B, EDG3A, EDG2B, EDG2A, EDG1B, EDG1A, EDG0B and EDG0 are simulated.

TIE (Timer Interrupt Enable Register)
C7I, C6I, C5I, C4I, C3I, C2I, C1I and C0I are simulated.

TSCR2 (Timer System Control Register 2)
TOI, TCRE, PR2, PR1 and PR0 are simulated.

TFLG1 (Main Timer Interrupt Flag 1)
C7F, C6F, C5F, C4F, C3F, C2F, C1F and C0F are simulated.

TFLG2 (Main Timer Interrupt Flag 2)
TOF is simulated.

TCx (Timer Input Capture/Output Compare Registers 0-7)
Is simulated

PACTL (16-Bit Pulse Accumulator A Control Register)
PAEN, PEDGE and PAOVI are simulated.

PAFLG (Pulse Accumulator A Flag Register)
PAOVF is simulated.

PACN3, PACN2 (Pulse Accumulators Count Registers 2-3)
Is simulated.

PACN1, PACN0 (Pulse Accumulators Count Registers 0-1)
Is simulated.

MCCTL (16-Bit Modulus Down-Counter Control Register)
MCZI, MODMC, RDMCL, ICLAT, FLMC, MCEI, MCPR1 and MCPR0 are simulated.
MCFLG (16-Bit Modulus Down-Counter FLAG Register)
MCZF, POLF3, POLF2, POLF1 and POLF0 are simulated.

ICPAR (Input Control Pulse Accumulators Register)
PA3EN, PA2EN, PA1EN and PA0EN are simulated.

DLYCT (Delay Counter Control Register)
Not simulated.

ICOVW (Input Control Overwrite Register)
NOVW7, NOVW6, NOVW5, NOVW4, NOVW3, NOVW2, NOVW1 and NOVW0 are simulated.

ICSYS (Input Control System Control Register)
SH37, SH26, SH15, SH04, TFMOD, PACMX, BUFEN and LATQ are simulated.

PTPSR (Precision Timer Prescaler Select Register)
Is simulated if the derivative supports it.

PTMCPSR (Precision Timer Mod. Cnt Prescaler Select Reg.)
Is simulated if the derivative supports it.

PBCTL (16-Bit Pulse Accumulator B Control Register)
PBEN and PBOVI are simulated.

PBFLG (Pulse Accumulator B Flag Register)
PBOVF is simulated.

PA3H–PA0H (8-Bit Pulse Accumulators Holding Registers 0-3)
Is simulated.

MCCNT (Modulus Down-Counter Count Register)
Is simulated

TC0H-TC3H (Timer Input Capture Holding Registers 0-3)
Is simulated.
Not-memory-mapped Registers

PORTT (Port T)
The functionality linking the PWM module and the port T have been simulated; the register involved is PTT (Port T I/O Register).

PORTTBitx
The pins are simulated as ‘not memory mapped’ and can be accessed one by one through the object pool (PORTTBit0 to PORTTBit7).

Periodic Interrupt Timer (PIT)
The I/O device Periodic Interrupt Timer (PIT) is not simulated.

Pulse Width Modulator (PWM)
This I/O device simulates the Pulse Width Modulator (PWM). PWM with 8 and 6 channels are supported. The PWM with 6 channel is a subset of the other one and has fewer registers and in some registers less bits are used. The simulation is accurate up to one instruction; this limitation is due to the different pipelining of instruction in the hardware and in the simulation.

However, the simulation strictly respects the period and the duty time of the generated pulses.

Changing control registers while the counters are running causes irregularities on the hardware outputs and cycle duration. Irregularities are present in the simulation as well but these irregularities might differ from the one encountered in the hardware. For proper use of the module, channels should be disabled (PWME register) and the counter reset (PWMCNTx registers) before modifying the corresponding control register (clock selection, period settings etc.) as described in the manual.

Clock Select
Scalers and prescalers are simulated for the clock selection. Changing clock control bits while channels are operating can cause irregularities, that affects the time until the next end of a period (and duty) and the value displayed in the PWN counter registers.

Polarity, Duty and Period
It is important to note the information given in the inspector component concerning the various events. The two types of event used in the PWM module are the “Duty” and “Period” events.

In left aligned mode:
The “End of Period Time” represents the number of bus clock cycles to come before the counter is reset.

The “End of Duty Time” represents the number of bus clock cycles to come before the output changes state.

In center aligned mode:

- The “End of Period Time” represents the number of bus clock cycles to come before the counter changes state. This means that the “event period” is half the effective period of the centered output waveform.
- The “End of Duty Time” represents the number of bus clock cycles to come before the output changes state. A “End of Duty Time” is set after the end of each “Period Event”.

**Registers:**

**PWME (PWM Enable Register)**

PWME7, PWME6, PWME5, PWME4, PWME3, PWME2, PWME1 and PWME0 are simulated.

**PWMPOL (PWM Polarity Register)**

PPOL7, PPOL6, PPOL5, PPOL4, PPOL3, PPOL2, PPOL1 and PPOL0 are simulated.

**PWMCLK (PWM Clock Select Register)**

PCLK7, PCLKL6, PCLK5, PCLK4, PCLK3, PCLK2, PCLK1 and PCLK0 are simulated.

**PWMPRCLK (PWM Prescale Clock Select Register)**

PCKB2, PCKB1, PCKB0, PCKA2, PCKA1 and PCKA0 are simulated.

**PWMCAE (PWM Center Align Enable Register)**

CAE7, CAE6, CAE5, CAE4, CAE3, CAE2, CAE1, CAE0 are simulated.

**PWMCTL (PWM Control Register)**

CON45, CON23 and CON01 are simulated. PFRZ is not simulated but the system will act as if PFRZ is always set to 1.

**PWMSCLA (PWM Scale A Register)**

Is simulated.

**PWMSCLB (PWM Scale B Register)**

Is simulated.
PWMCNTx (PWM Channel Counter Registers 0-5/7)
Is simulated.

PWMPERx (PWM Channel Period Registers 0-5/7)
Is simulated.

PWMDTYx (PWM Channel Duty Registers 0-5/7)
Is simulated.

PWMSDN (PWM Shutdown Register)
PWMIF, PWMIE, PWMRSTRT, PWMLVL, PWM7IN, PWM7INL and PWM7EN are simulated.
Not memory mapped registers

PORTP (Port P)
The functionality linking the PWM module and the port P have been simulated; the register involved is PTP (Port P I/O Register).

PWMoutx
As in the hardware, writing to PTP has no effect. The input pins are simulated as ‘not memory mapped’ and can be accessed one by one through the object pool (PWMout0 to PWMout7). Only the PWMout7 pin can be configured as an input. Writing to the other pins has no effect.

Timer Module (TIM)
This I/O device simulates the Timer Module (TIM). This module can be viewed as a subset of the ECT module. The TIM for example has only two Pulse Accumulator Count Registers and they are called PACNT_H and PACNT_L. Both registers are simulated. For more information see Enhanced Capture Timer (ECT).

Legacy HC12 (CPU12) Derivatives Simulation

MC68HC812A4
This section explains the simulated features of the MC68HC812A4 derivative. The Full Chip Simulation implements onchip peripherals listed here below.
Register Block

Table 10.5 shows the register block functionality. You can move all I/O registers, according to the INITRG (Register Block Mapping) at offset $11 inside of the register block.

Table 10.5 MC68HC12A4 Register Block

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Register Address</th>
<th>Initial Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITRG</td>
<td>0x0011</td>
<td>0x00</td>
<td></td>
</tr>
</tbody>
</table>

Lite Integration Module

The Full Chip Simulation simulates many functions of the Lite Integration Module (LIM), including:

- Interrupt handling
- Watchdog
- Periodic Interrupt

General restrictions:

- The Full Chip Simulation does not distinguish normal from special mode. Accordingly, it allows all write accesses, as if the chip were in special mode.
- Table 10.6 includes restrictions relative to special registers and single bits of registers.

LIM Simulated Registers

Table 10.6 shows the LIM Simulated Registers.
Table 10.6 LIM Simulated Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Register Address</th>
<th>Initial Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLKCTL</td>
<td>0x0047</td>
<td>0x00</td>
<td>LCKF, PLLON, PLLS, BCSC, BCSB, BCSA: These CLKCTL bits control settings of the PLL. But the Full Chip Simulation does not simulate the PLL, so values of these bits have no effect.</td>
</tr>
<tr>
<td>RTICTL</td>
<td>0x0014</td>
<td>0x00</td>
<td>RSWAI: The Full Chip Simulation does not support the CPU Clock stop, so this bit of the RTICTL register has no effect. RSBCK: The Full Chip Simulation does not simulate background mode, so this bit of the RTICTL register has no effect.</td>
</tr>
<tr>
<td>RTIFLG</td>
<td>0x0015</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>COPCTL</td>
<td>0x0016</td>
<td>0x0F</td>
<td>CME, FCME, FCM: The Full Chip Simulation does not support these COPCTL bits; writing to these bits has no effect.</td>
</tr>
<tr>
<td>COPRST</td>
<td>0x0017</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>INTCR</td>
<td>0x001E</td>
<td>0x60</td>
<td>The Full Chip Simulation does not distinguish normal from special mode. IRQE: The implementation allows any write access. In normal mode, there should be only one write to this register. In special mode, the system should ignore the first write access.</td>
</tr>
<tr>
<td>HPRIO</td>
<td>0x001F</td>
<td>0xF2</td>
<td>The system may write to the HPRIO register if the I mask in the CPU condition code register CCR is set. The Full Chip Simulation does not simulate this fact.</td>
</tr>
</tbody>
</table>

Standard Timer Module (TIM)
All functions of the timer module TIM are simulated.

General restrictions:
The HPRIO register [S001F] may be written to if the I mask in the CPU condition code register CCR is set. This fact is not simulated.

The external timer output occurs at the PORTT register. This is done for testing purposes only and will be disabled in future versions.

Restrictions considering special registers and single bits of registers are mentioned in Table 10.7.

**TIM Simulated Registers**

Table 10.7 shows all TIM Simulated Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Register Address</th>
<th>Initial Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIOS</td>
<td>0x0080</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>CFORC</td>
<td>0x0081</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>OC7M</td>
<td>0x0082</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>OC7D</td>
<td>0x0083</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TCNT_H</td>
<td>0x0084</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TCNT_L</td>
<td>0x0085</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TSCR</td>
<td>0x0086</td>
<td>0x00</td>
<td>TSWAI: The Full Chip Simulation does not support the CPU Clock stop, so setting this bit has no effect. TSBCK: The Full Chip Simulation does not simulate background mode, so this bit of the TSCR register has no effect.</td>
</tr>
<tr>
<td>TQCR</td>
<td>0x0087</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TCTL1</td>
<td>0x0088</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TCTL2</td>
<td>0x0089</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TCTL3</td>
<td>0x008A</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TCTL4</td>
<td>0x008B</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TMSK1</td>
<td>0x008C</td>
<td>0x00</td>
<td></td>
</tr>
</tbody>
</table>
### Table 10.7 TIM Simulated Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Register Address</th>
<th>Initial Value</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMSK2</td>
<td>0x008D</td>
<td>0x30</td>
<td>TPU: This bit controls a pull-up resistor or a pin. But the Full Chip Simulation does not have real pins, so setting this bit has no effect. TDRB: This bit controls the output drive of a pin. But the Full Chip Simulation does not have real pins, so setting this bit has no effect.</td>
</tr>
<tr>
<td>TFLG1</td>
<td>0x008E</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TFLG2</td>
<td>0x008F</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC0_H</td>
<td>0x0090</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC0_L</td>
<td>0x0091</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC1_H</td>
<td>0x0092</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC1_L</td>
<td>0x0093</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC2_H</td>
<td>0x0094</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC2_L</td>
<td>0x0095</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC3_H</td>
<td>0x0096</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC3_L</td>
<td>0x0097</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC4_H</td>
<td>0x0098</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC4_L</td>
<td>0x0099</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC5_H</td>
<td>0x009A</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC5_L</td>
<td>0x009B</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC6_H</td>
<td>0x009C</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC6_L</td>
<td>0x009D</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC7_H</td>
<td>0x009E</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>TC7_L</td>
<td>0x009F</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>PACTL</td>
<td>0x00A0</td>
<td>0x00</td>
<td></td>
</tr>
</tbody>
</table>
Serial Communication Interface (SCI)

You should implement the SCI module as a separate class, because there are several almost-identical instances of this class.

Supported Features

Table 10.8 shows the SCI supported features.

Table 10.8 SCI Supported Features

<table>
<thead>
<tr>
<th>Abbr</th>
<th>Implemented Meaning</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reserved for internal tests</td>
<td>SBRx: Baud Rate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BTST: Reserved for internal tests</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BSPL: Reserved for internal tests</td>
</tr>
</tbody>
</table>
The LOOP mode determines SCI connection to the outer world. As this SCI is simulated, there is no connection to simulate.

<table>
<thead>
<tr>
<th>Abbr</th>
<th>Full Name</th>
<th>Implemented Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRLD</td>
<td>Reserved for internal tests</td>
<td>Ignored</td>
</tr>
<tr>
<td></td>
<td>Control Register</td>
<td></td>
</tr>
<tr>
<td>LOOP</td>
<td>LOOP Mode</td>
<td>The LOOP mode determines SCI connection to the outer world. As this SCI is simulated, there is no connection to simulate.</td>
</tr>
<tr>
<td>WOMS</td>
<td>Wired Or Mode</td>
<td>Special feature of LOOP mode, not simulated</td>
</tr>
<tr>
<td>RSRC</td>
<td>Receiver Source</td>
<td>Special feature of LOOP mode, not simulated</td>
</tr>
<tr>
<td>M Mode</td>
<td>8 or 9 data bits</td>
<td>Supported (different timing, 9th bit)</td>
</tr>
<tr>
<td>WAKE</td>
<td>Wakeup by Address Mark/Idle</td>
<td>Not supported</td>
</tr>
<tr>
<td>ILT</td>
<td>Idle Line Type</td>
<td>Considered in the Idle Line Detection</td>
</tr>
<tr>
<td>PE</td>
<td>Parity Enabled</td>
<td>Not simulated</td>
</tr>
<tr>
<td>PT</td>
<td>Parity Type</td>
<td>Not simulated</td>
</tr>
<tr>
<td>TIE</td>
<td>Transmit Interrupt Enable</td>
<td>Supported</td>
</tr>
<tr>
<td>TCIE</td>
<td>Transmit Complete Interrupt Enable</td>
<td>Supported</td>
</tr>
<tr>
<td>RIE</td>
<td>Receive Interrupt Enable</td>
<td>Supported</td>
</tr>
</tbody>
</table>
### Table 10.8 SCI Supported Features

<table>
<thead>
<tr>
<th>Abbr</th>
<th>Full Name</th>
<th>Implemented Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILIE</td>
<td>Idle Line Interrupt Enable</td>
<td>Supported</td>
</tr>
<tr>
<td>TE</td>
<td>Transmitter Enable</td>
<td>Transmission process stops if this bit is clear</td>
</tr>
<tr>
<td>RE</td>
<td>Receiver Enable</td>
<td>Receive process stops if this bit is clear. As the input register is not part of the simulation, it still receives stimuli.</td>
</tr>
<tr>
<td>RWU</td>
<td>Receiver Wake Up Control</td>
<td>Not supported</td>
</tr>
<tr>
<td>SBK</td>
<td>Send Break</td>
<td>Upon the first set of the SBK Flag, the transmitter starts sending 10 (11 if M bit is set) 0 values. The counter will be set only if the flag was cleared previously. After the counter sends the required number of 0 bits, it continues send 0 bits as long as the SBK flag remains set.</td>
</tr>
<tr>
<td></td>
<td>Status Registers</td>
<td></td>
</tr>
<tr>
<td>TDRE</td>
<td>Transmit Data Register Empty Flag</td>
<td>The system sets this flag upon the move of the value to be transmitted from the transmit data register to the serial shift register.</td>
</tr>
<tr>
<td>TC</td>
<td>Transmit Complete Flag</td>
<td>The system sets this flag if the transmission of one value ends, but no other value is yet in the transmit data register.</td>
</tr>
<tr>
<td>RDRF</td>
<td>Receive Data Register Full Flag</td>
<td>The system sets this flag upon the complete read of a value and the clearing of RDRF.</td>
</tr>
<tr>
<td>IDLE</td>
<td>Idle Line Detection Flag</td>
<td>The system sets this flag after a period without any input as stated in [3]. The system considers the ILT flag.</td>
</tr>
<tr>
<td>OR</td>
<td>Overrun Error Flag</td>
<td>The system sets this flag if the receipt of value ends, but the processor has not yet read the value.</td>
</tr>
</tbody>
</table>
The Full Chip Simulation use non-memory-mapped registers to simulate SCI connection to the outer world. The Full Chip Simulation buffers all values sent to the input registers, then simulates receipt from another SCI (with maximum speed and no transmission errors). If the buffer contains no values, the Full Chip Simulation simulates an empty input line. All these sent values are available in the output registers, which Table 10.9 lists. Other modules can subscribe to these registers to receive the sent values.

### Table 10.8 SCI Supported Features

<table>
<thead>
<tr>
<th>Abbr</th>
<th>Full Name</th>
<th>Implemented Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NF</td>
<td>Noise Error Flag</td>
<td>Not supported, as no physical transmission takes place.</td>
</tr>
<tr>
<td>FE</td>
<td>Framing Error Flag</td>
<td>Not supported, as no physical transmission takes place.</td>
</tr>
<tr>
<td>PF</td>
<td>Parity Error Flag</td>
<td>Not supported, as no physical transmission takes place.</td>
</tr>
<tr>
<td>RAF</td>
<td>Receiver Active Flag</td>
<td>Supported and cleared only when going into idle mode. Detection of a false start bit does not clear this flag, as no physical transmission takes place.</td>
</tr>
<tr>
<td>Data Register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td>Receive Bit 8</td>
<td>Supported</td>
</tr>
<tr>
<td>T8</td>
<td>Transmit Bit 8</td>
<td>Supported</td>
</tr>
<tr>
<td>Rx/Tx</td>
<td>Receive/Transmit Bit x</td>
<td>Supported, with autoclear feature</td>
</tr>
</tbody>
</table>

---

HC(S)12(X) Full Chip Simulation Connection
FCS and Silicon On-chip Peripherals Simulation
Table 10.9 Input, Output, Serial Output Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Adds a value to be received. The system takes the 9th bit from the last value written to InputH. Read has no specified meaning</td>
<td></td>
</tr>
<tr>
<td>InputH</td>
<td>9th Input bit; must be written before Input. Read has no specified meaning</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>Contains the last value sent. A notification is sent every time a new value is written. Write has no specified meaning</td>
<td></td>
</tr>
<tr>
<td>OutputH</td>
<td>9th Output bit. Must be read immediately after Output. Write has no specified meaning</td>
<td></td>
</tr>
<tr>
<td>SerialInput</td>
<td>Alias for Input for SCI 0; connects SCI 0 to terminal window. Only supports 8 bits.</td>
<td>Only available in SCI 0.</td>
</tr>
<tr>
<td>SerialOutput</td>
<td>Alias for Output for SCI 0; connects SCI 0 to terminal window. Only support 8 bits.</td>
<td>Only available in SCI 0</td>
</tr>
</tbody>
</table>

Serial Peripheral Interface (SPI)

Table 10.10 describes the SPI interface.

Table 10.10 SPI interface

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Full Name</th>
<th>Implemented Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Control Register 1</td>
<td></td>
</tr>
<tr>
<td>SPIE</td>
<td>Interrupt Enable</td>
<td>Implemented</td>
</tr>
<tr>
<td>SPE</td>
<td>System Enable</td>
<td>If set, the Full Chip Simulation supports SPI functions</td>
</tr>
</tbody>
</table>
### Table 10.10 SPI interface

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Full Name</th>
<th>Implemented Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWOM</td>
<td>Port S Wired-OR Mode</td>
<td>Not simulated, as no physical transmission takes place.</td>
</tr>
<tr>
<td>MSTR</td>
<td>Master Slave Mode Select</td>
<td>Master or Slave mode select</td>
</tr>
<tr>
<td>CPOL</td>
<td>Clock Polarity</td>
<td>Not simulated, as no physical transmission takes place.</td>
</tr>
<tr>
<td>CPHA</td>
<td>Clock Phase</td>
<td>Not simulated, as no physical transmission takes place.</td>
</tr>
<tr>
<td>SSOE</td>
<td>Slave Select Output Enable</td>
<td>Not simulated, as no physical transmission takes place.</td>
</tr>
<tr>
<td>LSBF</td>
<td>LSB First Enable</td>
<td>Not simulated, as no physical transmission takes place.</td>
</tr>
<tr>
<td></td>
<td>Control Register 2</td>
<td></td>
</tr>
<tr>
<td>PUPS</td>
<td>Pull Up Port S Enable</td>
<td>Not simulated, as no physical transmission takes place.</td>
</tr>
<tr>
<td>RDS</td>
<td>Reduce Drive of Port S</td>
<td>Not simulated, as no physical transmission takes place.</td>
</tr>
<tr>
<td>SPC0</td>
<td>Serial Pin Control 0</td>
<td>Selects Normal or Bidirectional transmission mode</td>
</tr>
<tr>
<td></td>
<td>Baud Rate Register</td>
<td></td>
</tr>
<tr>
<td>SPRx</td>
<td>Baud Rate Register</td>
<td>Baud rate of the SPI transmission</td>
</tr>
<tr>
<td></td>
<td>Status Register</td>
<td></td>
</tr>
<tr>
<td>SPIF</td>
<td>Interrupt Request</td>
<td>System sets SPIF after the eighth SCK cycle in a data transfer and clearing by reading the Status Register, followed by a read or write access to the SPI data register.</td>
</tr>
<tr>
<td>WCOL</td>
<td>Write Collision Status Register</td>
<td>System sets this flag upon the writing of new data to the Data Register, during a serial data transfer.</td>
</tr>
<tr>
<td>MODF</td>
<td>Mode Error Interrupt Status Flag</td>
<td>Not simulated, as no physical transmission takes place.</td>
</tr>
<tr>
<td></td>
<td>Data Register</td>
<td></td>
</tr>
</tbody>
</table>
Virtual register Value simulates the data register of a second SPI device. This permits simulate communication with a second SPI device. The transmission can be in Normal or a Bidirectional Mode; the device can be set as Master or Slave. See also “Technical Summary MC68HC812A4” page 84, figure 24.

**Key Wakeups**

Table 10.11 defines the Key Wakeups.

**Table 10.10 SPI interface**

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Full Name</th>
<th>Implemented Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP0DR</td>
<td>8-bit Data Register for SPI data.</td>
<td></td>
</tr>
<tr>
<td>Port S</td>
<td>Port S Data Register</td>
<td>Not simulated, as no physical transmission takes place.</td>
</tr>
<tr>
<td>PORTS</td>
<td>Data Direction Register</td>
<td></td>
</tr>
<tr>
<td>DDRSx</td>
<td>Data Direction for Port S Bit x</td>
<td>Direction of Data. Only bits 4 and 5 have any effect.</td>
</tr>
</tbody>
</table>

**Table 10.11 Key Wakeups**

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Full Name</th>
<th>Implemented Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTD</td>
<td>Port D Register</td>
<td>Implemented</td>
</tr>
<tr>
<td>DDRD</td>
<td>Port D Data Direction Register</td>
<td>Implemented</td>
</tr>
<tr>
<td>KWIED</td>
<td>Port D Interrupt Enable Register</td>
<td>Implemented</td>
</tr>
<tr>
<td>KWIFD</td>
<td>Port D Flag Register</td>
<td>A falling edge on the associated pin sets each flag, provided that the corresponding DDRD Register bit is reset. To clear the flag, write one to the corresponding bit of the KWIFD register.</td>
</tr>
<tr>
<td>PORTH</td>
<td>Port H Register</td>
<td>Implemented</td>
</tr>
</tbody>
</table>
The Full Chip Simulation does not implement Port-D register mapping in wide expanded modes. The Full Chip Simulation does not implement this mapping in special expanded narrow mode with MODE Register bit EMD set.

**Memory-Mapped Page Registers**

Table 10.12 describes the Memory-Mapped Page Registers.
## Table 10.12 Memory Mapped Page Registers

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Full Name</th>
<th>Implemented Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Port F Register</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>Chip Select / General Purpose IO (Bit 0-6)</td>
<td>Not implemented, as there are no physical outputs.</td>
</tr>
<tr>
<td></td>
<td>Port G Register</td>
<td></td>
</tr>
<tr>
<td>ADDR</td>
<td>Memory Expansion / General Purpose IO (Bit 0-5)</td>
<td>Not implemented, as there are no physical outputs.</td>
</tr>
<tr>
<td></td>
<td>Port F Data Direction Register</td>
<td></td>
</tr>
<tr>
<td>DDRF</td>
<td>Data Direction Register Port F (Bit 0-6)</td>
<td>Not implemented, as there are no physical outputs.</td>
</tr>
<tr>
<td></td>
<td>Port G Data Direction Register</td>
<td></td>
</tr>
<tr>
<td>DDRG</td>
<td>Data Direction Register Port G (Bit 0-5)</td>
<td>Not implemented, as there are no physical outputs.</td>
</tr>
<tr>
<td></td>
<td>Data Page Register</td>
<td></td>
</tr>
<tr>
<td>PDA</td>
<td>Data Page</td>
<td>Selects the data page</td>
</tr>
<tr>
<td></td>
<td>Program Page Register</td>
<td></td>
</tr>
<tr>
<td>PPA</td>
<td>Program Page</td>
<td>Selects the program page</td>
</tr>
<tr>
<td></td>
<td>Extra Page Register</td>
<td></td>
</tr>
<tr>
<td>PEA</td>
<td>Extra Page</td>
<td>Selects the extra page</td>
</tr>
<tr>
<td></td>
<td>Window Definition Register</td>
<td></td>
</tr>
<tr>
<td>DWEN</td>
<td>Data Window Enable</td>
<td>Enables paging of data space</td>
</tr>
</tbody>
</table>
### Current Non-Supported Modules

**Non-Supported Modules**

- A/D Converter Device

### Register Block Address Map

*Table 10.13* shows the mapping of the Register Block Address.

#### Table 10.13 Register Block Address Map

<table>
<thead>
<tr>
<th>Register Block Address</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0000-$000D</td>
<td>Port access</td>
<td>Not simulated: memory configuration controls correct timing of memory accesses</td>
</tr>
<tr>
<td>$000E-$000F</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>$0010</td>
<td>Internal RAM mapping</td>
<td>Register not simulated. Use the memory configuration dialog box to specify simulated memory configuration.</td>
</tr>
<tr>
<td>0x0011</td>
<td>Register Block mapping</td>
<td>Completely simulated</td>
</tr>
</tbody>
</table>
Table 10.13  Register Block Address Map

<table>
<thead>
<tr>
<th>Register Block Address</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0012-$0013</td>
<td>ROM/EEPROM mapping</td>
<td>Registers not simulated. Use the memory configuration dialog box to specify simulated memory configuration.</td>
</tr>
<tr>
<td>$0014-$0017</td>
<td>Clock Function Control</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$001E-$001F</td>
<td>Interrupt Control &amp; Highest Priority I Interrupt</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$0020-$002E</td>
<td>Key Wakeup Control</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$002F</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>$0030-$0033</td>
<td>Port Registers</td>
<td>Currently not simulated</td>
</tr>
<tr>
<td>$0034-$0038</td>
<td>PAGE &amp; memory configuration Registers</td>
<td>Page Registers are simulated</td>
</tr>
<tr>
<td>$0039-$003B</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>$003C-$003F</td>
<td>Chip select control registers</td>
<td>Currently not simulated</td>
</tr>
<tr>
<td>$0040-$0043</td>
<td>PLL divider registers</td>
<td>Currently not simulated</td>
</tr>
<tr>
<td>$0044-$0046</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>$0047</td>
<td>Clock Control Register</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$0048-$005F</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>$0060-$0069</td>
<td>Analog to Digital Converter</td>
<td>Currently not simulated</td>
</tr>
</tbody>
</table>
Table 10.13 Register Block Address Map

<table>
<thead>
<tr>
<th>Register Block Address</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$006A-$006E</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>$006F</td>
<td>PORTAD</td>
<td>Currently not simulated</td>
</tr>
<tr>
<td>$0070-$007F</td>
<td>ADRxH/ reserved</td>
<td>Currently not simulated</td>
</tr>
<tr>
<td>$0080-$009F</td>
<td>Timer Registers</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$00A0-$00A3</td>
<td>Pulse Accumulator Control Registers</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$00A4-$00AC</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>$00AD-$00AF</td>
<td>Timer Test, Timer Port</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$00B0-$00BF</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>$00C0-$00C7</td>
<td>SCI0</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$00C8-$00CF</td>
<td>SCI1</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$00D0-$00D3</td>
<td>SPI</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$00D4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>$00D5-$00D7</td>
<td>SPI, PORTS</td>
<td>Completely simulated</td>
</tr>
<tr>
<td>$00D8-$00EF</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>$00F0-$00F3</td>
<td>EEPROM Control</td>
<td>Currently not simulated</td>
</tr>
<tr>
<td>$00F3-$00FF</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

Related Documentation

The following documents are available from Motorola:

- MOTOROLA SEMICONDUCTOR TECHNICAL DATA, MC68HC812A4, Technical Summary 16-Bit Microcontroller 1996
HC912DG128x, HC912DT128x

This section explains derivative simulated mechanisms and implemented features that match the real HC12 derivatives. It also explains simulation limitations. (For technical specifications of all I/O mechanisms, please see MOTOROLA MC68HC912DA128/MC68HC912DG128 16-Bit Microcontroller Technical Summary from MOTOROLA INC., 1997, 27 August 1997, rev1.0.)

Register Block
You can reassign the 1-kilobyte register block to any 2-kilobyte boundary within the standard 64-kilobyte address space.

Related Register:
INITRG Initialization of Internal Register Position Register, simulated.

Memory Expansion Register
The system fully simulates this mechanism within CALL and RTC instructions for banked memory model.

NOTE Also see the Programming in Bank Windows section of this manual for application programs creation/adaptation.

Related Register:
Program Page Register PPAGE: PIX2/PIX1/PIX0 bits memory defined but NOT updated.

Enhanced Capture Timer
16-Bit Modulus Down-Counter Simulated.

8 Input Capture/Output Compare channels: all channels are NON-BUFFERED and identical, except channel 7 with TCRE (Timer Counter Reset Enable) also implemented.

You may configure PORTT pins individually as standard, parallel-port I/O pins, or as timer pins. For standard parallel I/O pins, reading and writing are transparent, behaving like reading/writing in typical RAM. For this configuration, assign the value 1 to the channel x bit IOSx, in the TIOS register (for compare mode). Assign the value 0 to the OMx and OLx bits of the TCL1 or TCTL2 register for Timer disconnected from output pin logic mode/output action.
Capture Stimulation on PORTT. You can simulate rising- and falling-edge input signals on PPORT with the Stimulat component (I/O Stimulation). In this case, PORTT is bit accessible via non-memory-mapped I/O registers PORTTBit0 through PORTTBit7.

The stimulation example below periodically stimulates the PORTT bit 5 to simulate an input capture.

```python
def a = TIMER.PORTTBit5;
PERIODICAL 4000, 500:
    1000 a = 1;
    3000 a = 0;
END
```

Other user-designed I/O components also can set the PORTT bit value. Use `OP_SetValue("RegisterBlock.PORTTBit5", &parameter, NO_UPDATE);` function (with `parameter.n = 0 | 1`).

### 16-Bit Modulus Down-Counter

**Related Registers:**

- **MCCTL:** (16-bit modulus down counter control register) All bits simulated except ICLAT bit.
- **MCCNT:** (modulus down-counter count register) Fully simulated.
- **TIOS:** (timer input capture/output compare select) Simulated.
- **CFORC:** (timer compare force register) Simulated.
- **TCNT:** (timer count register) Simulated.
- **TCTL1** and **TCTL2:** (timer control register - output) Simulated.
- **TCTL3** and **TCTL4:** (timer control register - input) Simulated.
- **TMSK1:** (timer interrupt mask) Simulated.
- **TMSK2:** (timer interrupt mask) Simulated bits: TOI (overflow interrupt), TCRE (timer counter reset enable), PR2, PR1, PR0 (prescaler)
- **TFLG1:** (main timer interrupt flag) Simulated.
- **TFLG2:** (main timer interrupt flag) Simulated.
- **TC0** to **TC7:** (timer input capture/output compare registers) Simulated.
Serial Communication Interface (SCI)

This I/O Device simulates the two SCI signals SCI0 and SCI1. The non-memory-mapped registers SCIInput/SCIInputH and SerialInput send characters to the SCI Module. The non-memory-mapped registers SCIOutput/SCIOutputH and SerialOutput contain the characters sent from the SCI Module.

Related registers:

SC0BDH/SC1BDH: SCI Baud Rate Register High
- bit 7 BTST  Reserved for test functions, Not simulated
- bit 6 BSPL  Reserved for test functions, Not simulated
- bit 5 BRLD  Reserved for test functions, Not simulated
- bit 4..0 SBR  (SCI Baud Rate) Simulated

SC0BDL/SC1BDL: SCI Baud Rate Register Low
- bit 7..0 SBR  SCI Baud Rate, Simulated

SC0CR1/SC1CR1: SCI Control Register 1
- bit 7 LOOPS LOOP Mode, Not simulated
- bit 6 WOMS Wired Or Mode, Not simulated
- bit 5 RSRC Receiver Source, Not simulated
- bit 4 M Mode, Simulated
- bit 3 WAKE Wakeup by Address Mark/Idle, Not simulated
- bit 2 ILT  Idle Line Type, Simulated
- bit 1 PE Parity Enabled, Not simulated
- bit 0 PT Parity Type, Not simulated

SC0CR2/SC1CR2: SCI Control Register 2
- bit 7 TIE  Transmit Interrupt Enable, Simulated
- bit 6 TCIE Transmit Complete Interrupt Enable, Simulated
- bit 5 RIE  Receive Interrupt Enable, Simulated
- bit 4 ILIE Idle Line Interrupt Enable, Simulated
- bit 3 TE Transmitter Enable, Simulated
- bit 2 RE Receiver Enable, Simulated
- bit 1 RWU  Receiver Wake Up Control, Not simulated
- bit 0 SBK  Send Break, Simulated

SC0SR1/SC1SR1: SCI Status Register 1
- bit 7 TDRE  Transmit Data Register Empty Flag, Simulated
bit 6 TC Transmit Complete Flag, Simulated
bit 5 RDRF Receive Data Register Full Flag, Simulated
bit 4 IDLE Idle Line Detection Flag, Simulated
bit 3 OR Overrun Error Flag, Simulated
bit 2 NF Noise Error Flag, Not simulated
bit 1 FE Framing Error Flag, Not simulated
bit 0 PF Parity Error Flag, Not simulated
SC0SR2/SC1SR2: SCI Status Register 2
bit 7..1 unused
bit 0 RAF Receiver Active Flag, Simulated
SC0DRH/SC1DRH: SCI Data Register High
bit 7 R8 Receive Bit 8, Simulated
bit 6 T8 Transmit Bit 8, Simulated

SC0DRL/SC1DRL: SCI Data Register Low, contains the Receive-/Transmit Data Bits 7..0.

SCIInput:
This is a non-memory-mapped register that sends a character to the SCI. A read access to the SCDR can read this value. The system takes the ninth bit from the SCIInputH register. A read access to SCIInput has no specified meaning.

bit 7..0 character send to the SCI

SCIInputH:
This is a non-memory-mapped register that sends a character, the ninth bit, to the SCI. You must write this register value before you write the SCIInput register value. A read access to SCIInputH has no specified meaning.

bit 7..1 unused
bit 0 ninth bit send to the SCI

SCIOutput:
This is a non-memory-mapped register that receives a character sent from the SCI. A write access to the SCDR triggers the value that the SCIOutput receives. The SCIOutputH register receives the ninth bit. A write access to SCIOutput has no specified meaning.

bit 7..0 character send from the SCI

SCIOutputH:
This is a non-memory-mapped register that receives a character, the ninth bit, sent from the SCI. A write access to SCIOutput has no specified meaning.
HC(S)12(X) Full Chip Simulation Connection
FCS and Silicon On-chip Peripherals Simulation

bit 7..1 unused
bit 0  ninth bit send from the SCI

SerialInput:
This non-memory-mapped register is an alias for the SCIIInput register. It connects the SCI to the terminal window, but does not support the ninth bit. A read access to SerialInput has no specified meaning.

bit 7..0  data from terminal window to SCI

SerialOutput:
This non-memory-mapped register is an alias for the SCIOOutput register. It connects the SCI to the terminal window, but does not support the ninth bit. A write access to SerialOutput has no specified meaning.

bit 7..0  data sent from SCI to terminal window
FCS Visualization Utilities

Besides components that provide the Debugger engine a well-defined service dedicated to the task of application development, the debugger component family includes utility components that extend to the productive phase of applications, such as, the host application builder components, process visualization components, etc.

Among these components, there are visualization utilities that graphically display values, registers, memory cells, etc., or provide an advanced graphical user interface to simulated I/O devices, program variables, and so forth.

The following components of the continuously growing set of visualization utilities belong to the standard Debugger installation.

WARNING! The following visualization components can only be used with the Full Chip Simulation connection.

Analog Meter Component

The Analog Meter window shown in Figure 10.13 is a component that can be used as a basis for user-provided debugger extension components. It displays several input and output controls that can be manipulated with the mouse.

NOTE For legacy reasons, the Analog Meter component is called “Template”.

Figure 10.13  Analog Meter Template Window

The Analog Meter contains four controls: an analog gauge in the middle, a vertical level bar to the left, a horizontal level bar on top, and a turning ‘knob’ in the top left corner. Click in any of these controls to adjust the value of the meter. The Analog Meter is assigned to address 0x210.

**Analog Meter Operations**

In the vertical bar, the value can be tracked vertically, in the gauge and horizontal bar, the value can be tracked horizontally, and in the knob, the value is adjusted when tracking the mouse around the center.

**Analog Meter Menu**

The Analog Meter does not have a menu.
IO_LED Component

The IO_LED window shown in Figure 10.14 contains 8 LEDs used to manipulate and display the values of memory at an address specified in the related dialog box. LED colors are set at the PORT address (red or green) and the LED states are switched on/off at the DDR address (symbolic values).

Figure 10.14 IO_LED Component Window

When you change the state of LEDs in this window, the value of the corresponding bit at the DDR address changes in the Memory component window.

IO_LED Operations

By clicking and changing the state of one led will change the value of the byte at the DDR address.

If you want to change the color of the leds, you must change the value of the byte at the PORT address in the Memory Component window.

The location is specified with a string in the form `object.value`. Possible objects and their values can be listed in the Inspector component.

IO_LED Menu

The IO LED Menu contains a single item Setup... that opens the IO_LED Setup dialog box shown in Figure 10.15, which allows you to specify the PORT and DDR addresses.
Figure 10.15  ILed Setup Dialog Box

Associated Popup Menu
Identical to menu.

Drag Out:
Nothing can be drag out.

Drop Into:
Nothing can be dropped into.

Demo Version Limitations
No limitation
LED Component

The LED window shown in Figure 10.16 is a visual utility that displays an arbitrary 8 bit value by means of an LED bar.

Figure 10.16 LED Window

The LED component displays the value in a bit-wise manner with the most significant bit to the left, and the least significant bit to the right. Bits with value 0 are shown using a green LED, and bits with value 1 use a red LED. The user can click a LED to toggle its state. The underlying value is changed accordingly.

LED Operations

If you click an LED, its state toggles between green (0) and red (1). The corresponding bit in the underlying value is changed as well.

When you right-click within the component window, a popup menu appears with the same menu entries as listed in the LED menu in the main menu bar.

LED Menu

The LED menu contains a single item Setup... that opens the dialog box shown in Figure 10.17.

Figure 10.17 LED Setup Dialog Box

In the text field, the user can specify which value should be displayed by the LED bar. The location is specified with a string in the form object.value. Possible objects and their values can be listed in the Inspector component.
Click **OK** to accept the specified location. Click **Cancel** to discard changes and retain the previous location.

**Example:**
If the specified location is **TargetObject.#210** the LED bar displays the memory byte at address 0x210.

**Drag Out:**
Nothing can be dragged out.

**Drop Into:**
Nothing can be dropped in.

**Demo Version Limitations**
No limitation
Phone Component

The Phone window shown in Figure 10.18 is an input utility that provides a graphical keyboard pad that allows you to interactively modify the value of a memory cell.

Figure 10.18 Phone Window

The phone component displays the front panel of a cellular phone with a numeric keypad and LCD display. Keys on the keypad can be clicked to store the corresponding value into the configured memory location. If the mouse is on top of an active key, the arrow shape of the cursor changes to a pointing hand. Currently, the LCD component is not operational.
Phone Operations
Click a phone key and the matching ASCII character of the label on the key is stored at the configured memory cell.

Right-click within the component to display a popup menu with the same menu entries as the Phone menu in the main debugger menu.

Phone Menu
The Phone menu contains the Address... command, which opens the Phone Address dialog shown in Figure 10.19. In the text field of this dialog box, the user can specify the address of the memory cell where keypad characters will be stored. The location is specified with a hexadecimal number.

![Figure 10.19 Phone Address Dialog Box](image)

Click OK to accept the specified address. Click Cancel to discard changes and retain the previous address.

Example:
If the specified address is 210, the Phone component keypad is associated with the memory byte at address 0x210.

Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dropped in.

Demo Version Limitations
No limitation
ADC/DAC Component

The ADC/DAC window, shown in Figure 10.20 consists of a Digital to Analog and an Analog to Digital converter.

Figure 10.20  ADC/DAC Window

Description

The ADC/DAC component is made of 4 units as shown in Figure 10.21:

- A signal generator
- An analog to digital converter (ADC)
- A digital to analog converter (DAC)
- A visualization unit
The 4th unit shows the value of the initial analog signal and value of the DAC output analog signal.

Communication with the mainframe is done through 3 parallel ports of 8 bits:

- A port with 1 significant bit to indicate the conversion state.
- An input port to recover the ADC values
- An output port to send values to the DAC in order to visualize them

**Signal Generator**

The signal generator only generates a sinus signal. The generator output is connected to the ADC visualization screen.

**Visualization Screen**

The visualization screen is a 200 point horizontal resolution screen. The sinus signal period is deployed by default in red, in the upper part of the screen shown in Figure 10.20, and the signal generated by the DAC is displayed in blue in the lower part.

**ADC**

The ADC is an 8 bit resolution converter generating unsigned values. As we can see in Figure 10.21, its entry is directly connected to the signal generator. On the other hand, the conversion order will be given by a timer not connected to the data bus (it can not be set by software).

At the end of a conversion, it sets the state bit. This bit is automatically reset after read.
DAC

Also an 8 bit resolution converter whose output is connected to the visualization screen. Its use is simplified, we only have to send a byte into its data port to have its conversion displayed on the visualization screen. This screen only has a 200 point resolution; it is useless to send more than 200 bytes to the converter.

ADC/DAC Menu

The ADC/DAC menu shown in Figure 10.22 contains all functions associated with the Adc-Dac component. These entries are described in Table 10.14.

Figure 10.22 ADC/DAC Menu

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Opens the ADC/DAC - Setup dialog box, allowing you to set the port addresses.</td>
</tr>
<tr>
<td>Reset</td>
<td>This function erases the visualization screen and re-initializes the display properties.</td>
</tr>
<tr>
<td>Conversion parameters</td>
<td>Opens the Conversion Parameters dialog box, allowing you to set the signal frequency</td>
</tr>
<tr>
<td>Start conversion</td>
<td>Runs the conversion process</td>
</tr>
<tr>
<td>Display properties</td>
<td>Opens the Display Properties dialog box allowing you to set the display properties</td>
</tr>
</tbody>
</table>

Table 10.14 ADC/DAC Menu Description

ADC/DAC - Setup Dialog Box

This dialog box shown in Figure 10.23 allows you to define the port and address or select one port of the five proposed. These are used when this component functions with the Programmable IO Ports Component component.
Conversion Parameters Dialog Box

This dialog box shown in Figure 10.24 allows you to choose the analog signal frequency generated by the sinus generator and the sampling frequency. The choice of these two frequencies internally initializes the timer, which gives the conversion orders.

Figure 10.24 Conversion Parameters Dialog Box

Now you can start the conversion with Start conversion menu entries.
Display Properties Dialog Box

This dialog box shown in Figure 10.25 allows you to modify the display properties form the Adc_Dac component. The Up and Down buttons allow you to define the vertical position of the input and output curves. Two control buttons allow you to change the axes scales.

Figure 10.25 Display Properties Dialog Box

ADC/DAC Operations
To convert a signal from an example application:
1. Load the application and the ADC/DAC component.
2. Choose the ports address
3. Define the input signal frequency
4. Define the sampling frequency
5. Start the application
6. Choose Start Conversion

Drag Out:
Nothing can be dragged out.

Drag Into:
Nothing can be dragged in.
IT_Keyboard Component

The IT_Keyboard window shown in Figure 10.26 is a 20 key keyboard that generates an interruption when a key is pressed.

Figure 10.26  IT_Keyboard Window

An interruption is raised when an active key (line or column activated) is pressed.

Figure 10.27  IT_Keyboard Constitution
Scanning is one method to read such keyboards. Typically, we can proceed as follows (the line being in output and the column in input):

- Put a 0 at line X4 (X3, X2, X1, X0 being at 1).
- Read the column successively, from Y3 to Y0.
- Put a 0 at line X3 (X4, X2, X1, X0 being at 1).
- Read the column again from Y3 to Y0.
- ...till the last column of the last line, and restart at the beginning

All keyboard keys are scanned until we find one that is activated. During the scanning process, it is easy to update a counter representing the number of the key pressed. Raising an interruption when a key is pressed is interesting when scanning. This one could work only when a key is activated and not continually.

**IT_Keyboard Menu**

Figure 10.28 shows the IT_Keyboard menu and its entries are described in Table 10.15.

![IT_Keyboard menu](Setup...)

**Table 10.15 IT_Keyboard Menu Description**

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Opens the Interrupt keyboard setup dialog.</td>
</tr>
</tbody>
</table>

**Interruption Keyboard Setup**

The Interruption Keyboard Setup dialog box shown in Figure 10.29 allows you to set the address of the lines port, the columns port and the number of the interruption vector.
In the Port address section, for each two ports you can insert an address (in hexadecimal) in the Lines field or select one of the five ports listed in the Columns field. These are used when the component works with the Programmable IO Ports Component.

The Vector number field allows you to specify an interruption vector number (in hexadecimal).

The Keys label buttons permit you to change the symbols displayed on the keyboard keys.

Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dropped into the IT_Keyboard Component window.

Demo Version Limitations
No limitations
**LCD Component**

The LCD Display component message box shown in Figure 10.30 is the LCD utility, which can display 1 or 2 lines of 16 characters and show or hide the position cursor.

**Figure 10.30 LCD Display Message Box**

The display module consists of 2 eight-bit-width parallel couplers: a data port and a control port, as shown in Figure 10.31. These ports communicate with the mainframe.

**Figure 10.31 The LCD Display Module Ports**

The bits d7-d0 represent an ASCII code to display characters or an instruction code. The RS bit defines the status of bits d7-d0.

**LCD Operation**

The LCD Display device can display 1 or 2 lines of 16 characters and show or hide the position cursor.

To manage the display, this device contains a controller: the DDRAM (Display Data RAM). The DDRAM stores the ASCII codes of characters written during a write operation. Only two lines of 16 characters each can be displayed but up to 64 characters can be stored.

This RAM can be seen as organized in 2 lines: the first one starting at the address 00h, ending at 1Fh and the second one starting at 40h, ending at 5Fh. Figure 10.32 illustrates this arrangement.
The Address Counter (AC) is an internal register of the display controller pointing at the current address. In the default configuration, AC is initialized at 00h and is increased when an ASCII character is stored at the address AC is pointing to. When AC is equal to 1Fh, the next increased value will not be 20h but 40h.

For example, if we send a 48 character string after initialization, the bytes will be stored at addresses 00h to 1Fh and 40h to 4Fh.

**NOTE** Only characters having their ASCII codes in the visible interval of the 16 characters (positions 1 to 16) of RAM are displayed.

### Sending Information to the Display

Two steps are necessary to send a character to the display:

1. Put the bits E and RS at 1 and the bit R/W at 0 (control word 00000100b)
2. Write the character ASCII code on the data port. Put bit E at 0 (this validates bits d7-d0)

For an instruction, only step 2 is different: the Byte to write on the data port is the instruction code the display controller should execute.

### Instruction Listing

Figure 10.33 lists the instructions available for the LCD component.
Figure 10.33 LCD DisplayComponent Instruction Listing

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Display</td>
<td>$0\ 0\ 0\ 0\ 0\ 0\ 0\ 1$</td>
<td>Erases the display and puts AC at 0.</td>
</tr>
<tr>
<td>Return Home</td>
<td>$0\ 0\ 0\ 0\ 0\ 0\ 0\ 1$</td>
<td>Puts the address 00h into AC and re-updates the display.</td>
</tr>
<tr>
<td>Entry Mode Set</td>
<td>$0\ 0\ 0\ 0\ 0\ 0\ 1\ I/D$</td>
<td>Re-initializes the display if shifts occurred.</td>
</tr>
<tr>
<td>Display On/Off Control</td>
<td>$0\ 0\ 0\ 0\ 0\ 1\ D\ C$</td>
<td>Lights on/off the display and shows or hides the cursor.</td>
</tr>
<tr>
<td>Cursor or Display Shift</td>
<td>$0\ 0\ 1\ B\ D\ R/L\ N$</td>
<td>Moves the cursor and shifts the display.</td>
</tr>
<tr>
<td>Set DDRAM Address</td>
<td>$1\ A_1\ A_0\ A_4\ A_3\ A_2\ A_1\ N$</td>
<td>Fixes the AC value.</td>
</tr>
<tr>
<td>Function Set</td>
<td>$0\ 0\ 1\ D_L^*\ N$</td>
<td>Fixes the data exchange width and the line number to display.</td>
</tr>
</tbody>
</table>

**Clear Display**
- Completely fills the DDRAM with the code 20h (space character)
- Puts the address 00h into AC (address counter)
- Re-initializes the display if shifts occurred.
- Puts the cursor in position 1 on the display first line.

**Return Home**
- AC = 00h and re-initializes the display.
- Puts the cursor in position 1 on the display first line.
- The DDRAM is unchanged.

**Entry Mode Set**
- Increases AC (if I/D = 1) or decreases AC (if I/D = 0) after an ASCII code is written into RAM
- Moves the cursor to the right if ID = 1 or to the left if I/D = 0

**Display On/Off Control**
- The display is on if D = 1 and off if D = 0 (data still stay in RAM)
- If C = 1 the cursor will be shown.

**Cursor or Display Shift**
- Doesn't change the DDRAM content.
HC(S)12(X) Full Chip Simulation Connection
FCS Visualization Utilities

- AC is unchanged in case of a screen shift.
- Moves and/or shifts the cursor to the right or left. The cursor goes to the second line if it exceeds the 32nd position of the first line. It also goes to the first line when it exceeds the 32nd position of the second line.
- During a screen shift the two lines only move horizontally, the first line will never pass to the second one.

Figure 10.34 describes how to choose the moving direction.

**Figure 10.34 Left Right Choice**

<table>
<thead>
<tr>
<th>S/C</th>
<th>R/L</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Moves the cursor to the left (decreases AC)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Moves the cursor to the right (increases AC)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Moves the full screen to the left. The cursor follows this move</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Moves the full screen to the right. The cursor follows this move</td>
</tr>
</tbody>
</table>

**Set DDRAM Address**
- Puts the address indicated by a6a5a4a3a2a1a0 into AC.
- When the number of lines is 2, the address goes from 00h to 1Fh for the 1st line, and from 40h to 5Fh for the 2nd line.
- The a6 bit indicates the line: a6=0 to indicate the 1st line and 1 to indicate the 2nd one.

**Function Set**
- If DL = 1, the data exchange is 8 bits wide.
- If N = 0, the display will take place on one line. If N = 1, the display will take place on two lines.

**The Initialization Step**

Initialization needs essentially 7 steps. The Function Set instruction must be sent 3 times successively to fix the exchange data width, and a 4th time to fix the number of lines used.

The example shown in Figure 10.35 configures the display module in 8 bit mode, 2 lines, with the cursor visible and an increase of AC (the cursor moves to the right).
Figure 10.35 The LCD Display Initialization

<table>
<thead>
<tr>
<th>RS</th>
<th>RW</th>
<th>d7</th>
<th>d6</th>
<th>d5</th>
<th>d4</th>
<th>d3</th>
<th>d2</th>
<th>d1</th>
<th>d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Functions Set:
- 8 bits Mode.
- 2 lines display.
- Screen on and cursor visible.
- Screen erased.
- Entry Mode Set: AC will be increased.

End of initialization

LCD Menus

Figure 10.36 shows the LCD menu, which is identical to the popup menu. Its entry is described in Table 10.16.

Figure 10.36 LCD Menu

The LCD menu contains the Setup function to launch the LCD Display dialog box.

Table 10.16 LCD Display Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Opens the LCD Display dialog box (Setup)</td>
</tr>
</tbody>
</table>
LCD Display

The LCD Display dialog box shown in Figure 10.37 allows you to set the address of the lines port and columns port.

Figure 10.37  LCD Display Dialog Box (Setup)

In the Ports address section, for each two ports you can insert an address (in hexadecimal) in the Lines field or select one of the five ports listed in the Columns field. These are used when the component works with Programmable IO Ports Component.

Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dropped into the Lcd display Component.

Demo Version Limitations
No limitations
Monitor Component

The Monitor window shown in Figure 10.38 is a basis oscilloscope that can display the result of debugger objects.

![Monitor Window](image)

The purpose of this component is to display in a graphical format (similar to an oscilloscope) the results of debugger objects observation. The monitor component can save the list of state modifications and associated time in a file.

Monitor Menu

Figure 10.39 shows the Monitor menu and its entries are described in Table 10.17.

![Monitor Menu](image)

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Channel</td>
<td>Opens the dialog box to create a new Channel in the Monitor.</td>
</tr>
<tr>
<td>Delete Channel</td>
<td>Deletes the Selected Monitor Channel (click on it in the monitor view)</td>
</tr>
<tr>
<td>Show Control</td>
<td>Opens the Settings dialog box to change the time base.</td>
</tr>
<tr>
<td>Change Colors</td>
<td>Changes colors from the selected Channel.</td>
</tr>
</tbody>
</table>
Add Channel

The Add Channel dialog box shown in Figure 10.40 allows you to create a new Channel in the monitor.

Figure 10.40  Add Channel Dialog Box

In the text area Object to monitor, enter the object name and bit e.g TIM12.PORTT bit 0 and click OK to validate or Cancel to exit.

Monitor Settings

The Monitor Settings dialog box shown in Figure 10.41 allows you to change the time base.

Select the object name in the list, enter in the Ticks field a CPU timer proportional value and a number of pixels in the Pixels field to define the horizontal scale. Click OK to validate or Cancel to exit.
Figure 10.41 Settings Dialog Box

Change Colors

The Change Colors dialog box shown in Figure 10.42 allows you to change the colors from the selected Channel.

Figure 10.42 Change Colors Dialog Box

Select the intended element in the categories field and click Change to open the standard color selection dialog, click on OK to validate or Cancel to exit.
HC(S)12(X) Full Chip Simulation Connection
FCS Visualization Utilities

Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dropped in.

Demo Version Limitations
No limitations
Push Buttons Component

The Push Buttons window shown in Figure 10.43 is a basis input device.

Figure 10.43 Push Buttons Window

Push Buttons Menu

Figure 10.44 shows the Push Buttons menu and its entry is described in Table 10.18.

Figure 10.44 Push Buttons Menu

Table 10.18 Push Buttons Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Opens the Push Buttons Setup dialog box.</td>
</tr>
</tbody>
</table>

Push Buttons Setup

The Setup dialog box shown in Figure 10.45 allows you to specify (in hexadecimal format) the port address or select the port in the list.

Figure 10.45 Setup Dialog Box
NOTE  The port should be an output port for the LEDs component.

Use with IO_Ports

The address defined in the Push Buttons Setup dialog box is used when the component works with the Programmable IO_Ports Component.

Use with LEDs Component

The Bytes sent to the LEDs component coming from the Push Button component are described in Figure 10.46.

Figure 10.46  Push Buttons Input port

<table>
<thead>
<tr>
<th>Push Buttons Input Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>b7</td>
</tr>
<tr>
<td>PB7</td>
</tr>
</tbody>
</table>

Value 1 for a bit, lights on the corresponding led on the LEDs device. For example, if button 3 is pressed, a read access at the address of the component port will return the value 00001000b (08h).

Drag Out:

Nothing can be dragged out.

Drop Into:

Nothing can be dropped in.

Demo Version Limitations

No limitations
Programmable IO_Ports Component

The Programmable IO_Ports window shown in Figure 10.47 consists of 5 IO_Ports with 8 configurable bits in input or output. In the default configuration all couplers are in input. The graphical interface suggests the state of each one.

Figure 10.47 Programmable IO_Ports Window

The data exchange between the processor and peripherals are done by the intermediary of some circuits called «input / output couplers». The peripherals are connected to the data bus and are in parallel in an electrical point of view. A concerned output circuit will catch information on the data bus and save it (in a latch) until the next data reception.

The input/output couplers are perceived by the processor as memory cases with a wired fixed address. The capability exists to do input/output actions at a known address. In the C language, access is done by forced pointers to these addresses.

A read operation where the coupler is in input mode, activates this input during all the read steps. A write operation where the coupler is in output mode activates the output latch during all write steps.

The programmable IO_Ports allows you to define the coupler in input and output. This configuration can be modified during program execution. The first step in the test program is to configure the used couplers.

Programmable IO_Ports Menu

Figure 10.48 shows the Programmable IO_Ports menu and its entry is described in Table 10.19.

Figure 10.48 The Programmable IO_Ports Menu

Table 10.19 Programmable IO_Ports Menu Description

<table>
<thead>
<tr>
<th>Menu entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Opens the Programmable IO_Ports Port Address dialog.</td>
</tr>
</tbody>
</table>
Port Address

The Port Address dialog box shown in Figure 10.49 allows you to set the port address and control port address.

![Port Address Dialog Box (Setup)](image)

You can enter the address for the 5 ports A,B,C,D,E and the address for the Control port. Click OK to validate.

The coupler Control register allows you to configure the port type: for each port, set a bit to 1 to configure the port as output and set to 0 to configure the port as input, as shown in Figure 10.50.

![Coupler Control Register](image)

**Drag Out:**
Nothing can be dragged out.

**Drop Into:**
Nothing can be dropped in.

**Demo Version Limitations**
No limitations
7-Segments Display Component

The 7-Segments Display window shown in Figure 10.51 consists of 8 "7-segment" display systems.

Figure 10.51 7-Segments Display Window

Operation of the Seven segments display component is based on the display scanning principle. Only one display can be activated simultaneously for the purpose of limiting consumption of the set.

Common connection of the segments is the power of the component, the other connections serve as code input, so the same code is applied to all seven, as shown in Figure 10.52.

Scanning consists of selecting a display and activating its segments with adequate code to the input terminals and then attend to the next display.

Figure 10.52 7-Segments Display Component Constitution
7-Segments Display Menu

Figure 10.53 shows the Seven segments display component menu and the menu entry is described in Table 10.20.

Figure 10.53 7-Segments Display Menu

Table 10.20 7-Segments Display Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Opens the Seven segments display component setup dialog.</td>
</tr>
</tbody>
</table>

7-Segments Display Setup

The 7-Sements Display dialog box shown in Figure 10.54 allows you to select the display and related value.

Figure 10.54 7-Segments Display Dialog Box (Setup)

In the Select a display section, you can insert an address (in hexadecimal) to select the display. In the Segment Activation field, you can set the value of this display. The predefined port is the one used when this component works with the Programmable IO Ports Component.
Control Bits Configuration

The 2 bytes sent to the 7 segments must be composed as shown in Figure 10.55.

Figure 10.55 Seven Segments Display Control Bits

<table>
<thead>
<tr>
<th>SELAFF</th>
<th>SELSEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>b7  b6  b5  b4  b3  b2  b1  b0</td>
<td>b7  b6  b5  b4  b3  b2  b1  b0</td>
</tr>
<tr>
<td>A    A    A    A    A    A    A    A</td>
<td>-    g    F    s    d    c    b    a</td>
</tr>
</tbody>
</table>

NOTE The Seven segments display component is much slower than its real equivalent. So in simulation you don’t need to insert delays between each display scan (for segments light on and observer eye perception).

Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dropped in.

Demo Version Limitations
No limitations
Stimulation Component

The Debugger also supports I/O Stimulation. Using this feature you can generate (stimulate) interrupts or memory access generated by an external I/O device.

NOTE the True Time I/O Stimulation section describes in detail and with example how to take advantage of this component.

The Stimulation window shown in Figure 10.56 is a window component that provides the basic functionality of the Full Chip Simulation. It serves to execute timed action and raise exception events. The Stimulation component displays and executes I/O stimulation described in a text file.

Figure 10.56 Stimulation Window

```
def a = TargetObject.#1FA0;
def b = TargetObject.#1FB0;
def c = TargetObject.#1FC0.B[7:3];
10000
```

Stimulation Popup Menu

Figure 10.57 shows functions associated with the Source component. Table 10.21 describes these functions.

Figure 10.57 Stimulation Popup Menu

Table 10.21 Stimulation Popup Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open File</td>
<td>Opens a dialog box to load a stimulation file.</td>
</tr>
<tr>
<td>Execute</td>
<td>Starts execution of the input file.</td>
</tr>
</tbody>
</table>
Cache Size

The Size of the Cache dialog box shown in Figure 10.58, allows you to define the number of lines displayed in the Stimulation component. If the 'Limited Size of Cache' checkbox is unchecked, the number of lines is unlimited. If the 'Limited Size of Cache' check box is checked, the number of lines is limited to the value displayed in the edit box. This value should be between 10 and 1000000. By default, the number of lines is 1000.

NOTE  The bigger the cache size, the slower new lines are logged.

Table 10.21  Stimulation Popup Menu (continued)

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>Switches display of stimulated file on or off.</td>
</tr>
<tr>
<td>Cache size</td>
<td>Opens the 'Size of the Cache' dialog box.</td>
</tr>
</tbody>
</table>

Figure 10.58  Size of the Cache Dialog Box

Example of a Stimulation File

Using an editor, open the file named IO_VAR.TXT located in the project directory. Listing 10.1 is an example file.

Listing 10.1  Stimulation File Example

```plaintext
def a = TargetObject.#210.B;

PERIODICAL 200000, 50:
  50000 a = 128;
  150000 a = 4;
END
10000000 a = 0;
```
In the first line, the stimulated object is defined. This object is located at address 0x210 and is 1 byte wide.

Once 200000 cycles have been executed, the memory location 0x210 is accessed periodically 50 times (line 3). First the memory location is set to 128 and then 100000 cycles latter, it is set to 4.

NOTE the True Time I/O Stimulation section describes in detail and with example how to take advantage of this component.

NOTE

Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dragged into.

Demo Version Limitations
Only 15 interrupts and memory access will be generated.
TestTerm Component

The TestTerm window shown in Figure 10.59 is a user-friendly terminal input/output. It provides a simple SCI (Serial Communication Interface) interface, which is “Full Chip Simulation independent”.

Figure 10.59 TestTerm Window

The TestTerm component emulates a serial communication interface based at the address 200 hex, therefore providing 5 simulated memory mapped registers described in Table 10.22.

Table 10.22 TestTerm Simulated Memory Mapped Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Function</th>
<th>Register Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAUD</td>
<td>Baud Rate Control</td>
<td>0x0200</td>
</tr>
<tr>
<td>SCCR1</td>
<td>Serial Communication Control Register</td>
<td>0x0201</td>
</tr>
<tr>
<td>SCCR2</td>
<td>Serial Communication Control Register</td>
<td>0x0202</td>
</tr>
<tr>
<td>SCSR</td>
<td>Serial Communication Status Register</td>
<td>0x0203</td>
</tr>
<tr>
<td>SCDR</td>
<td>Serial Communication Data Register</td>
<td>0x0204</td>
</tr>
</tbody>
</table>

In the Serial Communication Status Register, the bits used are described in Table 10.23.

Table 10.23 TestTerm Serial Communication Status Register

<table>
<thead>
<tr>
<th>Bit Name (flag)</th>
<th>Function</th>
<th>Bit Mask Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDRE</td>
<td>Transmit Data Register Empty</td>
<td>0x80</td>
</tr>
<tr>
<td>RDRF</td>
<td>Receive Data Register Full</td>
<td>0x20</td>
</tr>
</tbody>
</table>
However, reading and writing in the BAUD, SCCR1, SCCR2 or SCSR registers has no effect in the TestTerm component, but are required to make the component compatible with specific SCI interfaces.

Simulated I/Os of the TestTerm component do not need initialization. In the terminal interface file `termio.c`, BAUD and SCSR registers are initialized to be compatible with real SCI interfaces.

The SCDR register is valid for reading or writing data. When reading a value from the SCDR register, the RDRF flag is cleared in the SCSR register. Also when the user enters a character on the keyboard while TestTerm is active, the RDRF flag is set in the SCSR register and the ASCII code of the typed key is put into the SCDR register.

Conceptually when a new value is written in the SCDR register by the target application, the TDRE flag is cleared in SCSR. When the transmission is finished, the TDRE flag is set again. As TestTerm is only an I/O emulation, no delay is simulated and writing into SCDR sets the TDRE flag in the SCSR register.

### Output Redirection

Outputs can be redirected to a TestTerm window, a file, or to both at the same time. File output is monitored by the target system and cannot be specified interactively.

Redirection is handled through “Escape” sequences of the output data stream. Table 10.24 illustrates the different possible redirections and associated escape sequences where `filename` is a sequence of characters terminated by a control character (e.g., CR) and is a valid filename.

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC “h” “1”</td>
<td>Output to Terminal window only.</td>
</tr>
<tr>
<td>ESC “h” “2” filename</td>
<td>Output to both Terminal window and file.</td>
</tr>
<tr>
<td>ESC “h” “3” filename</td>
<td>Output to file only.</td>
</tr>
<tr>
<td>ESC “h” “4”</td>
<td>Read from keyboard</td>
</tr>
<tr>
<td>ESC “h” “5” filename</td>
<td>Read input from file ‘fileName’</td>
</tr>
<tr>
<td>ESC “h” “6” filename</td>
<td>Output to Terminal window and append to file</td>
</tr>
<tr>
<td>ESC “h” “7” filename</td>
<td>Append to file only</td>
</tr>
</tbody>
</table>

ESC is the ESC character (ASCII code 27 decimal).

These commands can be used anywhere in the output stream.
How to Redirect

By default, an output redirection is set to the TestTerm component window.

The Term_Direct function declared in terminal.h is used to redirect an output. The source code in terminal.c is given in Listing 10.2.

Listing 10.2 Term_Direct Source Code

```c
void Term_Direct(int what, char *fileName)
{
    if (what < 1 && what > FROM_FILE) return;
    Write(ESC); Write('h');
    Write(what + '0');
    if (what != TO_WINDOW && what != FROM_KEYS) {
        PutString(fileName); Write(CR);
    }
}
```

where “what” is one of the following items:

- TERM_TO_WINDOW (sends output to terminal window),
- TERM_TO_BOTH (send output to file and window),
- TERM_TO_FILE (send output to file 'fileName'),
- TERM_FROM_KEYS (read from keyboard),
- TERM_FROM_FILE (read input from file 'fileName'),
- TERM_APPEND_BOTH (append output to file and window),
- TERM_APPEND_FILE (append output to file 'fileName').

See also terminal.h for more information.

Using TestTerm

Listing 10.3 shows the functions defined in termport.h that can be called to access the TestTerm component:

Listing 10.3 Functions to Access the TestTerm Component

```c
char GetChar(void);
void PutChar(char ch);
void PutString(char *str);
void InitTermIO(void);
```

Source code for the functions in termport.c is given in Listing 10.4.
Listing 10.4 Functions to Access the TestTerm Component in termport.c

typedef struct {
    unsigned char BAUD;
    unsigned char SCCR1;
    unsigned char SCCR2;
    unsigned char SCSR;
    unsigned char SCDR;
} SCIstruct;

#define SCI (*((SCIStruct*)(0x0200))
char GetChar(void)
{
    while (!(SCI.SCSR & 0x20)); /* wait for input */
    return SCI.SCDR;
}

void PutChar(char ch)
{
    while (!(SCI.SCSR & 0x80)); /* wait for output buffer empty */
    SCI.SCDR = ch;
}

void PutString(char *str)
{
    while (*str) {
        PutChar(*str);
        str++;  
    }
}

void InitTermIO(void)
{
    SCI.BAUD  = 0x30;     /* baud rate 9600 at 8 MHz */
    SCI.SCCR2 = 0x0C;     /* 8 bit, TE and RE set */
}

TestTerm Menu

The TestTerm component menu shown in Figure 10.60 lets you set the Cache Size in lines of the TestTerm window in the dialog box shown in Figure 10.61.
Figure 10.60  TestTerm Menu

Select **Cache Size** in the menu.

Figure 10.61  TestTerm Cache Size Dialog Box

Drag Out:
Nothing can be dragged out.

Drop Into:
Nothing can be dropped in.

Demo Version Limitations
No limitation
Terminal Component

The Terminal window shown in Figure 10.62 can be used to simulate input and output. It can receive characters from several input devices and send them to other devices.

Figure 10.62 Terminal Window

![Terminal Window](image)

You can use a virtual SCI (Serial Communication Interface) port provided by the framework for communication with the target, but it is also possible to use the keyboard, the display, some files or even the serial port of your computer as I/O-devices.

To control and configure a terminal component use the Terminal menu of the terminal shown in Figure 10.63.

Figure 10.63 Terminal Menu and Popup Menu

![Terminal Menu and Popup Menu](image)

To open the popup menu just right click in the terminal window.
Configure Terminal Connections

The terminal window is very flexible and can redirect characters received from any available input device to any available output device. You can specify these connections by choosing Configure Connections... in the context menu of the terminal component. This opens the dialog box shown in Figure 10.64.

Figure 10.64 Configure Terminal Connections Dialog Box

You can simply choose one of the default configurations in the “Default Configuration” combo box. In the “Connections” section all active connections are listed in a list box. There you can customize which input devices will be redirected to which output devices by adding and removing connections.

To add a connection specify the source and target devices using the “From” and “To” combo boxes and then press the “Add” button. The new connection will then appear in the list below, which shows all active connections.

To remove connections, select them in the list of active connections and press the “Remove” button.

In the “Serial Port” section you can specify which serial port to use and its properties. This is only possible if there is at least one connection from or to the serial port.
If a connection from or to the virtual SCI port has been chosen it is also possible to specify in the “Virtual SCI” section which ports will be taken as virtual SCI ports. This enables you to make a connection to any port in the Full Chip Simulation framework.

Input and Output File

It is also possible to take a file as an input stream for the terminal component or redirect the output to a file.

If you want to use a file as an input stream, make sure that there exists at least one connection from the input file to any output device. Then you can open an input file by simply choosing Input File... from the context menu. As soon as you press the “OK” button in the “File Open” dialog, input from the file will start. The file will be closed as soon as the end of file is reached or you choose Close Input File from the context menu.

When the input file has reached its end a CTRL-Z character (ASCII code 26 decimal) will be sent to all output devices receiving characters from the input file to notify them that the file transfer has been finished.

If you want to redirect some input devices to an output file, you have to proceed similarly. Make sure that you have chosen your connections from input devices to the output file. Then you can open or create your output file by choosing Output File... from the context menu. If the file does not exist it will be created. Otherwise you can choose to overwrite or append the existing file. To stop writing to the output file you can choose Close Output File from the context menu.

File Control Commands

It is also possible to open and close input and output files through special “Escape” sequences in the data stream from serial port or virtual SCI. Table 10.25 illustrates the different possible commands and associated Escape sequences where filename is a sequence of characters terminated by a control character (e.g. CR) and is a valid filename.

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC “h” “1”</td>
<td>Close output file.</td>
</tr>
<tr>
<td>ESC “h” “2” filename</td>
<td>Open output file.</td>
</tr>
<tr>
<td>ESC “h” “3” filename</td>
<td>Open output file and suppress output to terminal display.</td>
</tr>
<tr>
<td>ESC “h” “4”</td>
<td>Close input file</td>
</tr>
<tr>
<td>ESC “h” “5” filename</td>
<td>Open input file.</td>
</tr>
</tbody>
</table>
ESC is the ESC Character (ASCII code 27 decimal).

These commands can be given in the data stream sent from the serial port or virtual SCI port, but not from the input file or the keyboard. They only have an effect if there are any connections reading from the input file or writing to the output file.

The TERM_Direct function declared in terminal.h is used to send such commands from a target via SCI to the terminal. The source code in terminal.c is given in Listing 10.5.

### Listing 10.5 TERM_Direct Source Code

```c
void TERM_Direct(TERM_DirectKind what, const char* fileName) {
    /* sets direction of the terminal */
    if (what < TERM_TO_WINDOW || what > TERM_APPEND_FILE) return;
    TERM_Write(ESC); TERM_Write('h');
    TERM_Write((char)(what + '0'));
    if (what != TERM_TO_WINDOW && what != TERM_FROM_KEYS) {
        TERM_WriteString(fileName); TERM_Write(CR);
    }
}
```

In the example, the parameter what is one of the following constants:

- TERM_TO_WINDOW: send output to terminal window
- TERM_TO_BOTH: send output to file and window
- TERM_TO_FILE: send output to file 'fileName'
- TERM_FROM_KEYS: read from keyboard (close input file)
- TERM_FROM_FILE: read input from file 'fileName'
- TERM_APPEND_BOTH: append output to file and window
- TERM_APPEND_FILE: append output to file 'fileName'

See also terminal.h for further details.
How to Use Virtual SCI

In its default “Virtual SCI” configuration the terminal component accesses the target through the Object Pool interface.

To make the terminal component work in this default configuration, the target must provide an object with the name "Sci0". If no Sci0 object is available, no input or output happens. It is possible to check, through the Inspector component, if the environment currently provides an Sci0 object.

NOTE
Only some specific Full Chip Simulation components currently have a Sci0 object. For all other Full Chip Simulation components the default virtual SCI port does not work unless a user defined Sci0 object with the specified register name is loaded.

Write access to the target application is done with the Object Pool function "OPSetValue" at the address "Sci0.SerialInput".

Input from the target application is handled with a subscription to an Object Pool register with the name Sci0.SerialOutput. When this register changes (sends a notification), a new value is received.

For implementations of this register with help of the "IOBase" class, the flag "IOB_NotifyAnyChanges" should be used. Otherwise only the first of two identical characters are received.

It is also possible to configure the terminal to use another object in the Object Pool instead of Sci0 with which to communicate. Please refer to Configure Terminal Connections for informations about where you can do this.

Cache Size

The item Cache Size... in the context menu allows you to set the number of lines in the terminal window with the dialog shown in Figure 10.65.

Figure 10.65 Size of the Cache Dialog Box
Wagon Component

The Wagon window shown in Figure 10.66 simulates a tool machine wagon functionality.

Figure 10.66  Wagon Window

At first, the wagon is at the left border position. When you click the RUN button, the wagon goes to the right side. Upon arriving at the right border, the wagon returns to the left side. The RESET button also positions the wagon at the left border. The STOP button stops the wagon at the current position.

Wagon Menu

Figure 10.67 shows the Wagon menu and is described in Table 10.26.

Figure 10.67  Wagon Menu

Table 10.26  Wagon Menu Description

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup</td>
<td>Opens the Wagon setup dialog box shown in Figure 10.68.</td>
</tr>
</tbody>
</table>

Wagon Setup

When you select Setup from the Wagon Menu, the Ports Address Selection dialog box appears. This is the Wagon component Setup window.
In the **Motor Port** section, you can insert an address (in hexadecimal) to select the Wagon direction, in the **Sensor Port** field you can insert an address (in hexadecimal) to select the Wagon position. Predefined ports are fixed when the component operates with the **Programmable IO Ports Component**.

**Control Bits Configuration**

The 2 bytes sent to the 7 segments must be composed as shown in Figure 10.69.

**Figure 10.69  Wagon Control Bits Description**

To move the wagon to the right, set bit \( r \) and to move the wagon to the left, set bit \( l \). The sensor port sets the \( bl \) bit when the wagon is at the left border, sets bit \( br \) when the wagon is at the right border; sets bit \( st \) when **START** button is clicked with left mouse button, and sets \( stp \) when **STOP** button is clicked.
True Time I/O Stimulation

The Full Chip Simulation I/O Stimulation component is a facility to trigger I/O events. With the Stimulation component loaded, interrupts and register modifications invoked by the hardware can be simulated. In this tutorial, examples of stimulation files are introduced and explained.

Click any of the following links to jump to the corresponding section of this chapter:

- Stimulation Program Examples
- Stimulation Input File Syntax

Stimulation Program Examples

Running an Example Program Without Stimulation

1. Run the debugger with the Full Chip Simulation connection.

The Main Window is shown in Figure 10.70.

Figure 10.70  FCS I/O-Simulation Main Window
2. Choose Simulator > Set > Sim.
3. Choose Component > Open > Io_led.
The IO_Led component is shown in Figure 10.71.

**Figure 10.71 IO_Led Component Window**

4. Choose Component > Open > Template.
The Template component is shown in Figure 10.72.

**Figure 10.72 Template Component Window**

5. Choose Simulator > Load io_demo.abs.
6. Choose Run > Start/Continue or click the ‘green arrow’ icon.
7. If the program halts in startup, click the Start/Continue command again.
8. Choose Run > Halt to stop execution after a few seconds.
The Template component is a view linked to a specific memory location in TargetObject. In the source code of the test program, you can find a variable associated with it:

```c
#define PORT_DATA (*((volatile unsigned char *)0x0210)) /* Value with range 0..255 */
```

The Template component polls this value and displays it in a speedometer like outlook.
In the procedure “IO_Show” in “io_demo.c” shown in Hypertextparamum, this value is incremented or decremented, depending on the raise direction. The raise direction depends on a global variable “dir”, that is turned back, when the top or bottom value is reached.
Listing 10.6   IO_Show Procedure in io_demo.c

static void IO_Show(void) {
for (;;) {                       // endless loop
    dir = 1;
    do {
        Delay();
        PORT_DATA++;
    } while ((dir == 1) && (PORT_DATA != 255));
    dir = -1;
    do {
        Delay();
        PORT_DATA--;
    } while ((dir == -1) && (PORT_DATA != 0));
}
}

Example Program with Periodical Stimulation of a Variable

1. Choose Simulator >Reset.
2. Choose Simulator | Load  Io_demo.abs.
3. Choose Component | Open | Stimulation
   The Stimulation component is shown in Figure 10.73.

Figure 10.73  The Stimulation Component Window

4. Activate Stimulation Window by clicking on it.
5. Choose Stimulation > Open File  io_var.txt.
6. Choose Stimulation > Execute.
7. Choose Run > Start/Continue.
   The Stimulation component executing  io_var.txt  accesses TargetObject at the address 0x210 associated with PORT_DATA in the source. You can observe this by
watching the Template component. The arrow is not raising with continuity, but jumping around. The value of PORT_DATA is now handled from “outside”, from our Stimulation component.

Using an editor, open the file named io_var.txt in the Full Chip Simulation demo directory. This file looks like Hypertextparanum.

**Listing 10.7 io_var.txt**

```c
/* Define an identifier a, which is located at address 0x210*/
/* This identifier is 1 Byte wide.*/
def a = TargetObject.#210.B;

/* After 200 000 cycles have expired, repeat 50 time */
/* the code sequence specified between the keywords */
/* PERIODICAL and END. */
PERIODICAL 200000, 50:
  50000  a = 128; /* After 50 000 cycles, write 128 at address 0x210. */
  150000  a = 4;  /* After 150 000 cycles, write 4 at address 0x210. */
END

10000000  a = 0;/*! After 10 000 000 cycles, write 0 at address 0x210. */
```

First, the simulated object is defined. This object is located at address 0x210 and is 1 byte wide. Once 200,000 cycles have been executed, the memory location 0x210 is accessed periodically 50 times. First the memory location is set to 128 and then 100,000 cycles later, it is set to 4.

**Example Program with Stimulated Interrupt**

1. Choose Simulator>Reset.
2. Activate Stimulation Window by clicking on it.
3. Choose Stimulation>Open File io_int.txt.
4. Select the Source component window.
5. Choose Source>Open Module io_demo.c.
6. Scroll into the procedure Interrupt_Routine.
7. Set a breakpoint in the Interrupt_Routine as shown below.

The Source component window is shown in Figure 10.74.
8. Activate Stimulation Window by clicking on it.


10. Choose Run>Start/Continue.

After about 300,000 cycles the Full Chip Simulation stops on the breakpoint in the interrupt routine and the corresponding source line is highlighted. The interrupt has been called. Start the Full Chip Simulation. It stops approximately each 100,000 cycles on the same breakpoint. Restart and repeat these actions until 1,200,000 cycles. Start again, the Full Chip Simulation runs until 10,000,000 cycles and stops on the breakpoint. Start the Full Chip Simulation. It continues to run. The stimulation is finished.

The interrupts have been invoked by the Stimulation component source `io_int.txt`. The listing of the Stimulation file is given in Hypertextparanum.

Listing 10.8 `io_int.txt`

```python
def a = TargetObject.#210.B;

PERIODICAL 200000, 10:
  100000 RAISE 7, 3, "test_interrupt";
END

1000000 RAISE 7, 3, "test_interrupt";
```

In the first line, the stimulated object is defined. The interrupt is raised periodically 10 times. The RAISE command takes the number of the interrupt in the interrupt vector map as the first argument. This number, "7" in our example is arbitrarily chosen. To export this example to a different target processor, take a look at the interrupt vector map in the technical data manual of the matching MCU. Using an editor, open the `io_demo.prm` file in the same demo directory. You can see at the end of this file how to set the interrupt vector (adapt it to your needs).
VECTOR 7 Interrupt_Function  /* set vector on Interrupt 7 */

If the interrupt vector address is not specified in the prm file, the Full Chip Simulation will stop when interruption is generated. The exception mnemonic (matching the interrupt vector number) is displayed in the status bar of the Full Chip Simulation.

The second argument specifies the interrupt priority and the third argument is a free chosen name of the interrupt.

The file io_int.txt is used to generate 11 interrupts. 10 periodical interrupts are generated every 100'000 CPU cycles from 200'000 + 100'000 = 300'000 to 1'200'000 CPU cycles. A last one is generated when the number of CPU cycles reaches 10'000'000.

Example of a Larger Stimulation File

Hypertextparanum contains this example and is commented below. This example file may not work as expected if the variables defined here do not refer to a port in TargetObject. In our example, we have only defined the objects TargetObject.#210 and #212 over the Io_led component. Definitions of b, c and pbits are only here for illustration. Remove these definition lines and the lines that refer to them, if the example presented here is not executable.

Listing 10.9 Example File io_ex.txt.

def a = TargetObject.#210.B;
def x = TargetObject.#212;
def b = TargetObject.#216.W;
def c = TargetObject.#220.L;
def pbits = Leds.Port_Register.B[7:3];

#10000 pbits = 3;
20000 a = 0;
+20000 b = pbits + 1;

PERIODICAL 100000, 10:
    10000 a = 128;
30000 RAISE 7, 3, "test_interrupt";
END

1000000 RAISE 7, 3, "test_interrupt";
Detailed Explanation

```python
def a = TargetObject.#210.B;
```
defines `a` as byte mapped at address 0x210 in TargetObject.

```python
def x = TargetObject.#212;
```
defines `x` as byte mapped at address 0x212 in TargetObject. Size can be omitted. .B for byte is default.

```python
def b = TargetObject.#216.W;
```
defines `b` as word (.W) mapped at address 0x216 in TargetObject.

```python
def c = TargetObject.#220.L;
```
defines `c` as long (.L) mapped at address 0x220 in TargetObject.

```python
def pbits = Leds.Port_Register.B[7:3];
```
defines `pbits` as bits 5,6 and 7 in the byte (.B) register named `Port_Register` in `Leds`. (In the Full Chip Simulation, names of target objects can be specified. In our example, it is the name of the port register defined by the IO-Led component).

```python
#10000 pbits = 3;
```
sets the 3 bits of `Leds.Port_Register` accessed with `pbits` to binary 011. Other bits are unaffected. The new value of `Port_Register` will be 0x75, if the initial value was 0x55. Values outside the valid BitRange of pbits are truncated (in this example only values from 0 to 7 are allowed for pbits). The # means that the time of execution of the instruction is 10000 cycles after the start of the simulation.

```python
20000 a = 0;
```
sets `a` to 0. Without # or + in front of the time marker, the time refers to the absolute time after starting execution of the Stimulation file.

**NOTE** In a periodical loop, the time marker without operator is interpreted as +.
reads pb\textit{bits} (3 \textit{bits in Leds, Port_Register}), increments this value and writes it to \textit{b}. The + in front of the time marker refers to the time relative to the last encountered time value in the Stimulation file.

\begin{verbatim}
+20000 \textit{b} = \textit{pbits} + 1;
\end{verbatim}

PERIODICAL 100000, 10:

executes the following block

\begin{verbatim}
10000 \textit{a} = 128;
30000 \textit{RAISE 7, 3, "test\_interrupt"};
\end{verbatim}

10 times. Starts execution 100000 cycles after the start of the simulation.

\begin{verbatim}
10000 \textit{a} = 128;
30000 \textit{RAISE 7, 3, "test\_interrupt"};
\end{verbatim}

assigns 128 to \textit{a}, 10000 cycles after each start of the periodical event.

\begin{verbatim}
30000 \textit{RAISE 7, 3, "test\_interrupt"};
\end{verbatim}

raises an interrupt with priority 3 with vector number 7, 40000 cycles (!) after each start of the periodical event. The time specification in the \texttt{PERIODICAL} loop is always relative. So 30000 means +30000. The raised interrupt has the name "\texttt{test\_interrupt}". This name is not important for the interrupt functionality.

END

end of the periodical block. The block is looped again after finishing. So the loop restarts after 10000 + 30000 = 40000 cycles.

\begin{verbatim}
1000000 \textit{RAISE 7, 3, "test\_interrupt"};
\end{verbatim}

raises the interrupt for the last time. This instruction marks the terminating point of the Stimulation, if there are no pending periodical events left.
Stimulation Input File Syntax

**EBNF**

StimulationFile={ IdDeclaration | TimedEvent | PeriodicEvent }.
IdDeclaration=“def” ObjectId “=” ObjectField “;”.
ObjectField=ObjectSpec [ “[” BitRange “]” ].
BitRange=StartBit “;” NoOfBits.

AssignmentList={ Assignment | Exception }.

PeriodicEvent=“PERIODICAL” Start “,” NbTimes “:” { PerTimedEvent } “END”.
PerTimedEvent= [“+”] Time AssignmentList.

Exception=“RAISE” Vector “,” Priority [“,” ArbPrio] [“,” Name “;”].
Assignment={ ObjectId | ObjectField } “=” Expression “;”.
Name= “” {character} “”.

Expression=a standard ANSI-C expression. The expression accepts object identifiers previously defined (ObjectSpec and ObjectField).
Time=a number which represents a number of cycle.
ObjectSpec=the name of an object as defined in Requirement specification for Object Pool.
Vector= the exception vector number.
Priority= the exception priority number.
ArbPrio= the arbitration priority of the exception.
Start= the number of cycle when the periodical event must be called for the first time after the initial time.
NbTimes= the number of time the periodical event has to be called (0 = infinity).

Remarks

- If bitRange is omitted, all bits of the object register are affected. If bitRange is specified, the mask defined by this bitRange applies to the value calculated with the Expression. Only the bits of the object register defined in the bitRange are affected by this value.
HC(S)12(X) Full Chip Simulation Connection
True Time I/O Stimulation

- Bits are numbered from right to left (in a byte, bit 7 is the most left bit). So in bitRange, noOfBits is always less or equal than StartBit +1.
- ObjectSpec is defined in Requirement specification for Object Pool as below:

```
ObjectSpec::=ObjectName ["." FieldName].
  ObjectName::=Ident [":" Index].
  FieldName::=IdentNum ( [".." IdentNum] | ["." Size] ).
  IdentNum::=Ident | "#" HexNumber.
  Size::="B" | "W" | "L".
```

- The identifiers declared in IdDeclaration are stored in a table of identifiers and can be also used in Expression.
- If "#" is specified, the time is absolute: it is the number of cycles since the Full Chip Simulation was started.
- If "+" is specified, the time is relative to the previous time specification.
- If nothing is specified, time is the number of cycles since execution of the Stimulation file.
- If size is omitted, the default size is byte (B).
- The periodical event is sent for the first time at initial time + start + time specified in periodical timed event.
- In the PerTimedEvent declaration, the "+" is optional. If specified or not, the following time is interpreted exactly the same way.
- The periodical events are not displayed in the stimulation screen.
Electrical Signal Generators and Signals Application to Device Pins

Signal IO Component

This "Signal" io is the first implementation of a Signal Generator reading - in real debugger time - a file describing (electrical) levels. Levels are applied/available at a virtual io pin called "SignalPin" as "float" value.

Levels are programmed one after the other in duration in the debugger internal Event queue of the Full Chip Simulation.

If levels duration are smaller then cycle time or smaller than cycles, undersampling is performed in the signal file.

Up to 16 Signal Generators can be run at the same time.

Signal Description File EBNF

Signal File Format

FILELOOP=<INF| nbr of file loops to perform> {signal block}* EOF

Signal Block Description

{signal header}
{signal data}

Signal Header Description

LOOP=<INF| nbr of file loops to perform>
TIMEUNIT=<NONE|CYCLES|SECONDS>
TIMEFACTOR=<double value>
GAIN=<double value>
DCOFFSET=<double value>
OPTION=NORMAL|ONLYPOSITIVE|ONLYNEGATIVE|ABSOLUTE

Signal Data Description

(<level double value> [<time double value (duration in seconds or cycles)>])*
File Example 1

FILELOOP=INF
LOOP=4
TIMEUNIT=SECONDS
TIMEFACTOR=0.5
GAIN=1
DCOFFSET=0
OPTION=NORMAL
0.000000e+000 3.051758e-005
3.051758e-005 3.051758e-005
6.103516e-005 3.051758e-005
9.155273e-005 3.051758e-005
1.220703e-004 3.051758e-005
1.525879e-004 3.051758e-005
1.831055e-004 3.051758e-005
LOOP=16
TIMEUNIT=SECONDS
TIMEFACTOR=3.6
GAIN=-4.2
DCOFFSET=2.5
OPTION=NORMAL
2.136230e-004 3.051758e-005
2.441406e-004 3.051758e-005
2.746582e-004 3.051758e-005
3.051758e-004 3.051758e-005
3.356934e-004 3.051758e-005
3.662109e-004 3.051758e-005
EOF
File Example 2

FILELOOP=INF
LOOP=INF
TIMEUNIT=NONE
TIMEFACTOR=0.5
GAIN=1
DCOFFSET=0
OPTION=NORMAL
-5
5
2
8
-0.4e-3
300
123
EOF

File Parameters

LOOP/FILELOOP
INF means infinite loop. If a block is "INF", scanning stays in this block till the io is closed or "CLOSESIGNALFILE" command is executed. If a number is specified, loops the number of time.

TIMEUNIT
- CYCLES means that the second data field are cycles.
- SECONDS means that the second data field are seconds.
-NONE means that the second data field does not exist and the data time unit is forced to 1 second. Then data time unit can then be adjust by the TIMEFACTOR parameter.

TIMEFACTOR
At event programation, multiplies the number of cycles or time duration by this factor.
GAIN
At Pin level setup, multiply the level by this gain.

DCOFFSET
At Pin level setup, level offset applied after gain is applied.

OPTION
- NORMAL: do nothing.
- ONLYPOSITIVE: at Pin level setup, after gain and offset, set 0 if signal level < 0.
- ONLYNEGATIVE: at Pin level setup, after gain and offset, set 0 if signal level > 0.
- ABSOLUTE: at Pin level setup, after gain and offset, set |signal level|.

Signal IO Usage
The Signal io can handle 16 signals at the same time. Each signal block is independent in parameters and options from other blocks. The Signal component can be opened within Open I/O Component Dialog Box or with the ‘openio signal’ command. It can be released within the same dialog or with the ‘close signal’ command.

Signal Commands

SETSIGNALFILE Command
SETSIGNALFILE specifies the signal file to use.
Syntax: SETSIGNALFILE <value (0-15)> <"file name">
The SETSIGNALFILE X command creates a virtual SignalGeneratorX module having a SignalPin.
The file name can include the path of the file. If no path is given, the Signal component will first search in the current project folder, then in the "prog\FCSsignals" folder of the debugger installation path.
For example, creating 3 generators:
setsignalfile 0 "sinus_11bit_0_5v_1Hz.txt"
setsignalfile 1 "saw_11bit_0_5v_1Hz.txt"
setsignalfile 2 "square_1_5v_1Hz.txt"
Then virtual pins connections with the Pinconn IO CONNECT command:
connect "SignalGenerator0.SignalPin","Atd0.PAD0"
connect "SignalGenerator1.SignalPin","Atd0.PAD1"
connect "SignalGenerator2.SignalPin","Atd0.PAD2"

**TIP** Commands to create a signal generators can be placed in a command file like a “Postload” command file.

**CLOSESIGNALFILE Command**

**CLOSESIGNALFILE** closes the current signal file and generator.

Syntax: **CLOSESIGNALFILE <value (0-15)>**

Example: **CLOSESIGNALFILE 1**

**Remarks**

A messagebox is displayed with line error in case of signal file scripting error.

The Signal component is compatible with cycle time duration modification (bus speed change via PLL) and True Time feature, and reprograms automatically level duration (when duration in seconds is provided or no duration info provided).

Currently, all header parameters are mandatory, also EOF, in the same order as described in EBNF here above, **without spacing between words**.

**Base Signal Files Provided**

The following files can be reused to create more complex signal description. There are current stored in the “prog\FCSsignals” folder of the debugger installation path.

- “saw_11bit_0_5v_1Hz.txt”: a “saw” tooth signal, with a 11-bit sampling definition, scaled on a 1 Herz frequency, in a 0 to 5 Volts voltage range.
- “saw_8bit_0_5v_1kHz.txt”: a “saw” tooth signal, with a 8-bit sampling definition, scaled on a 1000 Herz frequency, in a 0 to 5 Volts voltage range.
- “sinus_11bit_0_5v_1Hz.txt”: a “sinus” signal, with a 11-bit sampling definition, scaled on a 1 Herz frequency, in a 0 to 5 Volts voltage range.
- “sinus_8bit_0_5v_1kHz.txt”: a “sinus” signal, with a 8-bit sampling definition, scaled on a 1000 Herz frequency, in a 0 to 5 Volts voltage range.
- “square_1_5v_1Hz.txt”: a pure “square” signal, scaled on a 1 Herz frequency, with 1 volt at low level and 5 volts at high level.
- “square_1_5v_1kHz.txt”: a pure “square” signal, scaled on a 1000 Herz frequency, with 1 volt at low level and 5 volts at high level.
Virtual Wire Connections with the Pinconn IO Component

Pinconn IO

The Pinconn IO component is used to create virtual links/shortcuts between processor device pins, like a simple wire. The Pinconn component can be opened within Open I/O Component Dialog Box or with the “openio pinconn” command. It can be released within the same dialog or with the “close pinconn” command.

WARNING!  It is up to the user to properly connected input pins to output pins without bus/level conflicts.

Pinconn Commands

- CONNECT: Connects output pin to input.
  Syntax: CONNECT "<OutputPin>", "<InputPin>"
  Example: CONNECT "Pim.PORTHPin0", "Pim.PORTPPin3"
- DISCONNECT: Removes connection between pins.
  Syntax: DISCONNECT "<OutputPin>", "<InputPin>"
  Example: DISCONNECT "Pim.PORTHPin0", "Pim.PORTPPin3"
- CONNECT_STATE: Displays the list of active connections.

NOTE  There is no limitation of connections.

NOTE  This list of simulated pins for a derivative Full Chip Simulation is provided in the Inspect component, under the “Object Pool” key.
Command Set to Apply Signal on ATD Pin

The following example loads the Pinconn and Signal IO components, create a signal generator generating the signal described in "square_1_5v_1kHz.txt". The generator output signal pin is connected to the onchip ATD via the PAD0 pin.

openio Pinconn
openio Signal
setsignalfile 0 "square_1_5v_1kHz.txt"
connect "SignalGenerator0.SignalPin","Atd0.PAD0"
This chapter contains a tutorial on how to use the Full Chip Simulation. The tutorial is split up into small steps. After completing the last step a full functional example should exist.

This chapter contains the following sections:
- Supported Derivatives
- PWM Channel 0

Guess the Number
We are going to create step by step the demo run in the executive tutorial. The application makes use of the SCI (Serial Communication Interface) and a terminal window from the debugger. At the end the user can guess a number between 0 and 9. This guessing is done via terminal window. The produced application will run on real hardware as well.

Step 1 - Environment Setup
- In order to run the produced example on real hardware, you will need a serial cable. This cable must connect COM1 (PC) with the SCI0 (Hardware Board).

Step 2 - Creating the project
- Launch the ‘CodeWarrior IDE’
- In the CodeWarrior menu, Select File > New
- Make sure the ‘Project’ tab is active, Select HC(S)12 New Project Wizard
- Enter a project name like ‘MyGuessTheNumber’
- Change the directory if you want (Location, Set…)
- Click OK. The project wizard opens to let you select the device, language, etc.
- Select a derivative like ‘MC9S12DP256B’ and click Next.
- Select ‘C’ for the language and click Next.
- Select ‘Yes’ for Processor Expert support and click Next.
- Select ‘No’ for PCLint support and click Next.
- Select ‘Float is IEEE 32 and double is IEEE 32’ and click Next.
- Select ‘Full Chip Simulation’ and click Finish.
A new project is created using the wizard and the Processor Expert is available. Several windows should be visible:

Figure 10.75 Created Project

Step 3 - ‘Target CPU’ Window

The ‘Target CPU’ window in the center shows a footprint of the processor selected for the development. In the device, we see the different on-chip modules such as CPU, Timer, A/D converter. Modules with an icon attached to them are modules used by the application. The pins that are used to connect external functions are indicated by a line and an icon, symbol of the function attached (CPU and Port A).
Optional:

- Place the cursor of the mouse on the pins to see a description of their functions.
- Enlarge the ‘Target CPU’ window and you will see different on-chip modules.

**Step 4 - ‘Bean Selector’ Window**

The ‘Bean Selector’ window offers the developer a list of beans to add to the project. Some of the beans may not be usable with the version of CodeWarrior installed. The Standard and Professional Editions offer a wider range of hardware and software beans than the Special Edition.
• Select ‘Bean Categories’ > ‘CPU internal peripherals’ > ‘Communication’ > ‘AsynchroSerial’

Figure 10.77  Bean Selector Window - Selection of ‘AsynchroSerial’ Bean

Step 5 - ‘Project Panel’ Window

The ‘Project Panel’ window shows and keeps track of the beans that have been created for this application. This Panel is a tab of the Project Manager window. A click on the [+ ] next to a bean shows a list of methods and/or events related to the bean. A green tick indicate if the named methods or event is selected and a red cross that code has not been generated.
Under ‘Beans’ you should find the previously created bean with the name ‘AS1:AsynchrSerial’.

**Step 6 - ‘Bean Inspector AS1:AsynchrSerial’ Window**

In this window you can modify the behavior of the bean to your needs. In the tab ‘Properties’ you will find general settings. Software drivers are found under the tab ‘Methods’ and ‘Events’

- Select ‘Properties’ tab
- Enter a proper **baud rate**. If you want to run it on real hardware check your board manual for the right value. If you want to run it on the Full Chip Simulation only you can enter ‘9600’.
Step 7 - Generation of Driver Code

We are going to generate the code for the I/O drivers and the files for the user code.

- Select the ‘Make’ icon in the Project Manager window (or the menu bar Project > Make or [F7]).

Processor Expert shows several messages. One message indicates that we have started the code generation. The second message shows the progress with the information processed and the code generated. Another window shows compiling and linking progress.

Step 8 - Verification of Files Created

We can verify the folders created by Processor Expert:

‘User Modules’

A file “MyGuessTheNumber.C” that is the placeholder for the main procedure and any other procedure desired by the user. These other procedures can of course be placed in additional files.
'Generated Code'

The .C files for the code associated with the beans added to the project. This includes initialization, input, output and the declarations necessary for the use of the functions.

Step 9 - Entering User Code

- Open the user module "MyGuessTheNumber.C"
- Insert the following code before the main routine.

```c
#include <stdlib.h>

void PutChar(unsigned char c) {
    while (AS1_SendChar(c) == ERR_TXFULL) {
        // could wait a bit here
    }
}

void PutString(const char* str) {
    while (str[0] != '\0') {
        PutChar(str[0]);
        str++;
    }
}

void GuessTheNumber(void) {
    int ran = rand() / (RAND_MAX / 9);
    AS1_Init();

    PutString("Guess a Number between 0 and 9\n");
    PutString("Number: ");
    for (;;) {
        unsigned char c;
        if (AS1_RecvChar(&c) == ERR_OK) {
            PutChar(c); PutChar(' ');
            if(c < '0' || c > '9') {
                PutString("not a number, try again\n");
            } else {
            }
        }
    }
}
```
} else if (c == ran + '0') {
    PutString("\nCongratulations! You have found the number!\n");
    PutString("\nGuess a new number\n");
    ran = rand() / (RAND_MAX / 9);
} else if (c > ran + '0') {
    PutString("lower\n");
} else {
    PutString("greater\n");
}
PutString("Number: ");
} else {
    // could wait a bit here
}
} // for

• Call the function GuessTheNumber in the main routine.

void main(void) {
    /**< Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! */
    PE_low_level_init();
    /**< End of Processor Expert internal initialization. */

    /**< Write your code here*/
    GuessTheNumber();

    /**< Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! */
    for(;;);
    /**< Processor Expert end of main routine. DON'T WRITE CODE BELOW!!! */
} /**< End of main routine. DO NOT MODIFY THIS TEXT!!! */
Step 10 - Run

The application is now finished and we can launch it. Make sure you have chosen the Full Chip Simulation connection.

- Select the ‘Debug’ icon in the Project Manager window (or the menu bar Project > Debug or [F5]).
- Select ‘Component > Open’ in the debugger and open the ‘Terminal’ component.
- Select the ‘Save’ icon in debugger (or the menu bar File > Save Configuration) to save the window layout.
- Select the ‘Debug’ icon in debugger (or the menu bar Run > Start/Continue or [F5]).

Figure 10.80 Debugger Main Window - Final Application
PWM Channel 0

We are going to create step by step the demo run in the executive tutorial. The application makes use of the PWM (Pulse Width Accumulator). With the final application you will be able to change the period and duty time of the PWM and you will see the changes displayed in a chart.

Step 1 - Environment Setup


Step 2 - Creating Project

- Launch the ‘CodeWarrior IDE’
- In the CodeWarrior menu, Select File > New
- Make sure the ‘Project’ tab is active, Select HC(S)12 New Project Wizard
- Enter a project name like ‘MyPWMChannel0’
- Change the directory if you want (Location, Set…)
- Click OK. The project wizard opens to let you select the device, language, etc.
- Select a derivative like ‘MC9S12DP256B’ and click Next.
- Select ‘C’ for the language and click Next.
- Select ‘Yes’ for Processor Expert support and click Next.
- Select ‘No’ for PCLint support and click Next.
- Select ‘none’ for floating point support and click Next.
- Select ‘Full Chip Simulation’ and click Finish.

A new project is created using the wizard and the Processor Expert is available. Several windows should be visible:

Step 3 - ‘Target CPU’ Window

The ‘Target CPU’ window in the center shows a footprint of the processor selected for the development. In the device, we see the different on-chip modules such as CPU, Timer, A/D converter. Modules with an icon attached to them are modules used by the application. The pins that are used to connect external functions are indicated by a line and an icon, symbol of the function attached (CPU and Port A).

Optional:

- Place the cursor of the mouse on the pins to see a description of their functions.
• Enlarge the ‘Target CPU’ window and you will see different on-chip modules.

**Step 4 - Creating PWM Bean**

• Select ‘Bean Categories’ > ‘CPU internal peripherals’ > ‘Timer’ > ‘PWM’

**Step 5 - ‘Project Panel’ Window**

The ‘Project Panel’ window shows and keeps track of the beans that have been created for this application. **This Panel is a tab of the Project Manager window.** A click on the [+] next to a bean shows a list of methods and/or events related to the bean. A green tick indicate if the named methods or event is selected and a red cross that code has not been generated.

Under ‘Beans’ you should find the previously created bean with the name ‘PWM8:PWM’.

**Step 6 - ‘Bean Inspector PWM8.PWM’**

In this window you can modify the behavior of the bean to your needs. In the tab ‘Properties’ you will find general settings. Software drivers are found under the tab ‘Methods’ and ‘Events’

• Select ‘Properties’ tab
• Select ‘Period’ and enter ‘100’ms
• Select ‘Starting pulse width’ and enter ‘10’ms

**Step 7 - Generate Driver Code**

We are going to generate the code for the I/O drivers and the files for the user code.

• Select the ‘Make’ icon in the Project Manager window (or the menu bar Project > Make or [F7]).

Processor Expert shows several messages. One message indicates that we have started the code generation. The second message shows the progress with the information processed and the code generated. Another window shows compiling and linking progress.

**Step 8 - Verification of Files Created**

We can verify the folders created by Processor Expert.
‘User Modules’
A file “MyPWMChannel0.C” that is the placeholder for the main procedure and any other procedure desired by the user. These other procedures can of course be placed in additional files.

‘Generated Code’
The .C files for the code associated with the beans added to the project. This includes initialization, input, output and the declarations necessary for the use of the functions.

Step 9 - Entering User Code
- Open the user module “MyPWMChannel0.C”
- Replace the main routine with the following code:

```c
volatile static byte pwmChannel[1];
volatile static unsigned int pwmRatio= 6939;
void main(void) {
    /**< Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! **/
    PE_low_level_init();
    /**< End of Processor Expert internal initialization. */
    for(;;) {
        pwmChannel[0]= PTP_PTP0;
        void)PWM8_SetRatio16(pwmRatio);
    }
    /**< Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! */
    for(;;);
    /**< Processor Expert end of main routine. DON'T WRITE CODE BELOW!!! */
} /**< End of main routine. DO NOT MODIFY THIS TEXT!!! */
```
Step 10 - Run

The application is now finished and we can launch it. Make sure you have chosen the Full Chip Simulation connection.

- Select the ‘Debug’ icon in the Project Manager window (or the menu bar Project > Debug or [F5]).
- Select ‘Component > Open’ in the debugger and open the ‘VisualizationTool’ component.

In the following text we will create a nice visualization for our propose. All has to be done in the VisualizationTool window. Make sure that you are in the ‘Edit mode’ (switch with ‘Right mouse click’ > ‘Edit Mode’ or [Ctrl-E])

- ‘Right mouse click’ > ‘Properties’

VisualizationTool Properties

- Select for ‘Refresh Mode’ ‘CPU Cycles’
- Select for ‘Cycle Refresh Count’ ‘10000’

Now lets add a nice chart, where we can see the changing value of the channel in a graphic.

- ‘Right mouse click’ > ‘Add New Instrument’ > ‘Chart’
- ‘Double click’ on the ‘Chart’ to see the ‘Chart Properties’.

‘Chart’ Properties

- Select for ‘Kind of Port’ ‘Expression’
- Select for ‘Port to Display’ ‘pwmChannel[0]’
- Select for ‘High Display Value’ ‘2’
- Select for ‘Type of Unit’ ‘Target Periodical’
- Select for ‘Unit Size’ ‘1000’
- Select for ‘Numbers of Units’ ‘1000’
- Leave all others on default.

With the following bar we can change the period value of the PWM channel 0.

- ‘Right mouse click’ > ‘Add New Instrument’ > ‘Bar’
- ‘Double click’ on the ‘Bar’ to see the ‘Bar Properties’.
Period ‘Bar Properties’

- Select for ‘Kind of Port’ ‘Variable’
- Select for ‘Port to Display’ `_PWMPER01.Overlap_STR.PWMPER0STR.Byte`
- Leave all others on default.

You might add labels with ‘Right mouse click’ > ‘Add New Instrument’ > ‘Static Text’. Now lets add a bar to change the duty time.

- ‘Right mouse click’ > ‘Add New Instrument’ > ‘Bar’
- ‘Double click’ on the ‘Bar’ to see the ‘Bar Properties’.

Duty Time ‘Bar Properties’

- Select for ‘Kind of Port’ ‘Variable’
- Select for ‘Port to Display’ ‘pwmRatio’
- Select for ‘High Display Value’ ‘65535’
- Leave all others on default.

Now lets leave the Edit mode and run the final application. First we might save the window layout.

- ‘Right mouse click’ > ‘Edit Mode’ (or [Ctrl-E])
- Select the ‘Save’ icon in debugger (or the menu bar File > Save Configuration) to save the window layout.
- Select the ‘Debug’ icon in debugger (or the menu bar Run > Start/Continue or [F5] ).
Figure 10.81 Debugger Main Window - Final Application
ICD-12 Connection

Introduction

An advanced feature of HI-WAVE for the embedded system development world is the ability to load different framework targets. This document introduces the ICD-12 Background Debug Interface.

The ICD-12 is an interface developed by P&E Microcomputer Systems. HI-WAVE uses the ICD-12 to communicate with an external system (also called a connection).

With this interface, you can download an executable program from the HI-WAVE environment. The destination of this program is an external connection (target), based on a Freescale MCU, that executes the program. HI-WAVE receives feedback of real target-system behavior.

HI-WAVE fully supervises and monitors the connection (target-system) MCU. That is, HI-WAVE controls the CPU execution. You can read and write in internal or external memory when the MCU is in background mode. You have full control over the CPU state. You can stop execution, proceed in single-step mode, and set breakpoints in the code.

NOTE HI-WAVE base installation does not include installation of the ICD-12 connection (target) component.

Installation Notice

For access to the parallel port of a host computer running Windows NT, you must install a special driver.

To do so, after you install the Freescale Development Kit on the Host, run the setup program icd_dr.bat. This program is in the directory prog\drivers\Nt\ICD; Freescale installation on the host computer includes installation of this directory.

Alternatively, you can run the setup program from the ICD driver setup folder in your Freescale installation group, or choose this command sequence from the Windows Taskbar: Start > Programs > Freescale > ICD driversetup > Win NT driver.
Interfacing Your System and ICD-12

You use a parallel-to-serial interface for communication between the ICD-12 connection and the host computer. Communication to the target is serial. The ICD-12 connection driver fully handles the communication protocol between the ICD-12 and the host computer; loading the ICD-12 connection component automatically includes loading the ICD-12 connection driver.

To prevent communication problems, use the shortest possible Centronics (printer) cable between the host and the ICD-12 interface. (You may even be able to remove this cable completely.)
Loading the ICD-12 Connection Component

Usually, the PROJECT.INI file specifies the connection: Target=Icd (please see the next section). The ICD-12 driver automatically detects the ICD-12 connection to your system. However, if the driver detects nothing, an error message informs you that the target is not connected or that the target is connected to a different port.

If the PROJECT.INI file connection setting is incorrect, load the ICD-12 driver. From the main menu, select Component | Set Target..., as shown below. Then choose Icd from the list of possible targets.
ICD-12 Connection

Default Connection Setup

Figure 11.3 Component Menu

After successful connection loading, the ICD menu replaces the Target menu.

Figure 11.4 ICD Menu

The ICD menu selection Icd | Force BDM triggers a reset of the target, forces the CPU into background mode, and executes command files “forcebdm.cmd” and “reset.cmd”. (See the Command Files section).

The menu selection Icd | Reset triggers a reset of the connection and executes command file “reset.cmd”.

The menu selection Icd | Load.. loads an executable (“.abs”) file into target memory. The file’s program counter will point to the first instruction of the startup section.

Default Connection Setup

As with any connection, you can use the Target menu to load the ICD-12 connection component, or you can set the ICD-12 connection as a default in the PROJECT.INI file. This file should be in the working directory.
Listing 11.1  Example of PROJECT.INI file:

<table>
<thead>
<tr>
<th>Window</th>
<th>Type</th>
<th>X</th>
<th>Y</th>
<th>Width</th>
<th>Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window0</td>
<td>Source</td>
<td>0</td>
<td>0</td>
<td>60</td>
<td>30</td>
</tr>
<tr>
<td>Window1</td>
<td>Assembly</td>
<td>60</td>
<td>0</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>Window2</td>
<td>Procedure</td>
<td>0</td>
<td>30</td>
<td>60</td>
<td>25</td>
</tr>
<tr>
<td>Window3</td>
<td>Register</td>
<td>60</td>
<td>30</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>Window4</td>
<td>Memory</td>
<td>60</td>
<td>60</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Window5</td>
<td>Data</td>
<td>0</td>
<td>55</td>
<td>60</td>
<td>23</td>
</tr>
<tr>
<td>Window6</td>
<td>Command</td>
<td>0</td>
<td>78</td>
<td>60</td>
<td>22</td>
</tr>
<tr>
<td>Target</td>
<td>ICD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ICD-12 Default Environment

As with any HI-WAVE program, you can set ICD-12 connection component parameters in the DEFAULT.ENV file. This file should be in the working directory.

In normal use, you set these parameters in the DEFAULT.ENV file once, interactively, during installation. You use these parameter values in subsequent debugging sessions.

Text below introduces the environment variables associated with the ICD-12 connection component.

ICDPORT Variable

This variable specifies (to the host computer) the parallel communication port to which the ICD-12 connects.

Syntax

ICDPORT=LPTn
ICDPORT=LPTn:
ICDPORT=portAddr

where

n: number of the printer port (1,2)
portAddr: address of the printer port (1,2). Specifying the printer-port address is only possible with Windows 95, or with Windows 3.1x with Win32s. Under Windows NT, a driver that evaluates the port address must handle access to a port, so you cannot specify a port address. First try to define the Icd-Port by name (LPT1 or LPT2). If that does not work, define the communication port by address.
ICD-12 Connection
Default Connection Setup

Examples
ICDPORT=LPT2
   //Name of the port.
ICDPORT=0x378
   //Address of the port.

Default
ICDPORT=LPT1

NOTE  ICDPORT=0x378 is the MS-DOS first parallel printer port address;
      ICDPORT=0x278 is the MS-DOS second parallel printer port address. Under
      some Win 3.x installations, it could be necessary to specify the ICD-12 port by
      address.

BMDELAY Variable
This variable slows down the communication speed of the serial link (the ICD-12 cable).
The MCU clock speed is the maximum speed available, but the PC also affects the
communication speed. So if your target MCU clock speed is slower than 1 MHz, you may
need a delay that is greater than 0.

Syntax
BMDELAY=x
where
x: communication delay. The x value 0 yields the fastest communication speed.

Example
BMDELAY=9
You may have to work down from a high x value, such as 150 or 100, until you find the
optimal value for your system.

Default
The default x value is 0.
Command Files and ICD-12 Commands

Consider these hints for configuring the ICD-12 connection for a specific MCU and memory configuration. Connection initialization includes execution of startup file “startup.cmd,” during ICD-12 driver loading. Triggering a reset through the Icd | Reset menu selection executes another command file, “reset.cmd.” Choosing the Icd | Force BDM menu selection executes command files “forcebdm.cmd” and “reset.cmd.”

[CMDL] describes the general syntax of command files.

Memory Configuration

After a reset, the system configures only the boot chip select (CSBOOT). To access to other memories, you also must configure the MCU chip select logic. Normally, your application startup code does this configuration. But for debugging, you must repeat this configuration after each reset, so that you can load your application. To configure chip selects appropriately for access to the external devices, write your own startup and reset command files (“startup.cmd” and “reset.cmd”). Use the commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>WB</td>
<td>Write byte</td>
<td>WB &lt;address&gt; &lt;value&gt; //write byte</td>
</tr>
<tr>
<td>WW</td>
<td>Write word (16 bit)</td>
<td>WW &lt;address&gt; &lt;value&gt; //write word (16 bit)</td>
</tr>
<tr>
<td>WL</td>
<td>Write long (32 bit)</td>
<td>WL &lt;address&gt; &lt;value&gt; //write long (32 bit)</td>
</tr>
</tbody>
</table>

**NOTE**
Your ICD-12 connection includes command files that match the Motorola board default settings: an MPFB1632 board with 2 pseudo-ROM (RAMs) of 32k, no SRAM, and no FLASH. These files are appropriate as well for MPB332AB boards that support the MC68332 (which has 2kB internal RAM), and for an MPB16Y1B board that supports an HC16Y1 (which also has 2kB internal RAM).

Examine the command files in your installation directory. Text below is a short explanation of the most important system registers. Refer the your MCU manual for a detailed description.

**NOTE**
Each MCU version has its own memory map. Software and hardware chip selects must correspond to the MCU hardware and to external hardware, that is, ROM (pseudo-ROM), RAM (such as FLASH or SRAM), and so forth.
Softec inDart HCS12 Connection

This section guides you through the first steps toward debugging with CodeWarrior and the SofTec inDart HCS12 connection. It does not replace all the additional documentation provided in this manual, but gives you a good starting point.

Technical Considerations

The 8/16 bits debugger (and then the CodeWarrior IDE) might be connected to HCS12 hardware using the SofTec HCS12.

When the debugger runs the SofTec HCS12 connection, it can communicate and debug CPU12 (HCS12) core-based hardware connected through the SofTec in-circuit debugger/programmer units, i.e:

SofTec Microsystems HCS12 ISP Debuggers/Programmers (inDART Series) and Starter Kits (AK/SK/PK/ZK and newer Series).

Please refer to the “inDART®-HCS12 In-Circuit Debugger/Programmer for Freescale HCS12 Family FLASH Devices User’s Manual” from SofTec for communication hardware requirements and SofTec product installation.

CodeWarrior and SofTec HCS12 Connection

There are two separate paths that may be followed to take the first steps toward debugging with Codewarrior and the SofTec inDART-HCS12 connection. The differences between the two paths hinge on the starting point for the steps:

- Using the Stationary Wizard at the start of the project
- From within an existing project
First Steps Using the Stationery Wizard

To take the first steps toward debugging with CodeWarrior and the SofTec inDART-HCS12 using the stationery Wizard:

1. Run the CodeWarrior IDE with the shortcut created in the program group.
2. Choose the menu File > New to create a new project from a stationery - the HCS12 New Project Wizard first screen appears.

Figure 12.1 New Window - Project Tab

3. In the list box on the left of the screen, select HCS12 New Project Wizard.
4. In the Project Name textbox, type the name of your new project.
5. Click the OK button to proceed.
6. In the list box on the screen, select the HCS12 MCU you are targeting.

7. Click the Next button to proceed.

8. Select the Language format by checking its checkbox and click the Next button.
9. In the Connections list box, select **SofTec Microsystems Hardware Debugging** as the connection.

10. Finish the Wizard steps - the IDE opens.
Figure 12.5 Wizard Connection Selection

11. In the IDE main window toolbar Project menu, choose Project > Make.

12. Now choose Project > Debug to start the debugger.
To take the first steps toward debugging with CodeWarrior and setting the SofTec HCS12 connection from within an existing debugging project:

1. Run the CodeWarrior IDE with the shortcut created in the program group.
2. Open the project.
3. Choose the menu Project > Debug to start the debugger.
4. In the Debugger window menubar, display the Component menu.
5. Choose Component > Set Target.. from this menu to select another target interface in the Set Connection dialog box.

6. Select **HC12** as Processor.

7. Select **SofTec HCS12 Target Interface** as connection.
8. In the MCU Configuration dialog box, choose the correct target processor.

9. Press the OK button to start debugging.

**inDart-HCS12 Menu Options**

Once the SofTec HCS12 connection is set, the connection menu entry in the debugger main toolbar is “inDART-HCS12”.

![inDART-HCS12 Menu Options](image)
MCU Configuration Option

Select the inDART-HCS12 > MCU Configuration option to display the MCU Configuration Dialog Box.

User’s Manual Option

Select the inDART-HCS12 > User’s Manual option to open the “inDART®-HCS12 In-Circuit Debugger/Programmer for Freescale HC12 Family FLASH Devices User’s Manual” from SofTec.

About Option

Select the inDART-HCS12 > About... option to display the About Dialog Box.

MCU Configuration Dialog Box

The Hardware Model drop down list can be expanded to select another type of debug interface than the SofTec inDART-HCS12. The HW Code drop down list can be expanded to select another HCS12 derivative. Note that at this document release time, only the SofTec inDART-HCS12 is available.

Figure 12.11  MCU Configuration Dialog Box

Pressing the Communication Settings button opens the Communication Settings Dialog Box.
Communication Settings Dialog Box

Pressing the “Communication Settings” button in the MCU Configuration dialog box opens the Communication Settings dialog box, which allows you to fine-tune critical parameters needed for proper operation with the chosen target microcontroller.

The dialog box is divided into four sections: “BDM Clock Frequency Detection”, “Miscellaneous”, “3DM Clock Source”, and “Fast Programming”. All of the parameters must be carefully set, otherwise successful operations cannot result.

Refer to the “inDART®-HCS12 In-Circuit Debugger/Programmer for Freescale HCS12 Family FLASH Devices User’s Manual” from SofTec for further details.

Figure 12.12 Communication Settings Dialog Box

NOTE
If your hardware supports stopping the application while running, an additional interrupt service routine is required for the IRQ vector. Please see Stop Command Handling section in “inDART®-HCS12 In-Circuit Debugger/Programmer for Freescale HCS12 Family FLASH Devices User’s Manual” from SofTec for further details.
About Dialog Box

This dialog box belongs to the SofTec GDI DLL and provides information about the indART_HCS12.dll release and version.

Figure 12.13  About Dialog Box
HCS12 Serial Monitor Connection

This section guides you through the first steps toward debugging with CodeWarrior and the HCS12 Serial Monitor connection. It does not replace all the additional documentation provided in this manual, but gives you a good start.

Serial Monitor Technical Considerations

The 8/16 bit debugger (and then the CodeWarrior IDE) might be connected to HCS12 hardware using the HCS12 Serial Monitor connection. This connection supports communication specifications described in the application note from Freescale.

When the debugger runs the HCS12 Serial Monitor connection, it can communicate and debug hardware running the HCS12 Serial Monitor in full compliance with the Freescale Application Note specifications. Please refer to this Application Note for communication hardware requirements.

CodeWarrior and Serial Monitor Connection

There are two separate paths that may be followed to take the first steps toward debugging with CodeWarrior and the HCS12 Serial Monitor connection. The differences between the two paths hinge on the starting point for the steps:

- Using the Stationary Wizard at the start of the project
- From within an existing project
First Steps Using the Stationery Wizard

To take the first steps toward debugging with CodeWarrior and the HCS12 Serial Monitor connection using the stationery Wizard:

To take the first steps toward debugging with CodeWarrior and the SofTec inDART-HCS12 using the stationery Wizard:

1. Run the CodeWarrior IDE with the shortcut created in the program group.
2. Choose the menu File > New to create a new project from a stationery - the HCS12 New Project Wizard first screen appears.

3. In the list box on the left of the screen, select **HC(S)12 New Project Wizard**.
4. In the Project Name textbox, type the name of your new project.
5. Click the OK button to proceed.
6. In the list box on the screen, select the HCS12 MCU you are targeting.

7. Click the Next button to proceed.

8. Select the Language format by checking its checkbox and click the Next button.
9. In the Connections list box, select *Freescale Serial Monitor Hardware Debugging* as the connection.

10. Finish the Wizard steps - the IDE opens.
Figure 13.5 IDE Project Window

11. Click the Finish button - IDE opens as shown in Figure 13.6. “IDE Main Window - Project Menu”
12. Choose the menu option **Project > Make**.

13. Choose the menu option **Project > Debug** to start the debugger.

First Steps From Within an Existing Project

To take the first steps toward debugging with CodeWarrior and setting the HCS12 Serial Monitor connection from within an existing debugging project:

1. Run the CodeWarrior IDE with the shortcut created in the program group.
2. Open the existing project.
3. Choose the menu Project > Debug to start the debugger - debugger main window opens.
4. In the debugger main window, from the Component menu, choose Component > Set Connection... to select another connection.

Figure 13.7 Debugger Main Window - Component Menu

Figure 13.8 Set Connection Dialog Box - HCS12 Serial Monitor Selection
5. Select HC12 as Processor then HCS12 Serial Monitor Target Interface as connection in the Set Connection dialog box and click the OK button.

6. Now in the Monitor Setup window, Monitor Communication tab, choose the correct Host serial communication port if necessary.

Figure 13.9 Monitor Setup Window - Monitor Communication Tab

7. Press the OK button. The HCS12 Serial Monitor connection reads the device silicon ID. This ID can match several derivatives.

8. Set the correct derivative matching with your hardware in the Derivative Selection dialog box.

Figure 13.10 Derivative Selection Dialog Box

9. Press the OK button. The Monitor Setup window is opened again, to propose to use the “mirrored vector table” feature. We recommend that you use the Vector Table
Mirroring feature. Otherwise, vectors cannot be programmed as captured and protected from erasing or overwriting by the HCS12 Serial Monitor.

Figure 13.11 Monitor Setup Window - Vector Table Mirroring Tab

10. To enable this specific feature, check the “Enable Vector Table Mirroring” checkbox.
11. Press the “Auto Detect” button to make the debugger search for the vector table address and vectors reserved by the HCS12 Serial Monitor.

12. Once the autodetection succeeded, press the OK button to start debugging.
MONITOR-HCS12 Menu Options

Once the HCS12 Serial Monitor connection is set, the “MONITOR-HCS12” menu entry is set in the Debugger menu, as shown below.

Figure 13.13  MONITOR-HCS12 Menu Entries

<table>
<thead>
<tr>
<th>Option</th>
<th>Hotkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load…</td>
<td>Ctrl+L</td>
</tr>
<tr>
<td>Reset</td>
<td>Ctrl+R</td>
</tr>
<tr>
<td>Setup…</td>
<td></td>
</tr>
<tr>
<td>Connect…</td>
<td></td>
</tr>
<tr>
<td>Command Files…</td>
<td></td>
</tr>
</tbody>
</table>

Monitor Communication...

Select the MONITOR-HCS12> Monitor Communications... option to display the Monitor Setup Window - Monitor Communication Tab.

Vector Mirroring Setup...

Select the MONITOR-HCS12> Vector Mirroring Setup... option to display the Monitor Setup Window - Vector Table Mirroring Tab.

Erase Flash

Select the MONITOR-HCS12> Erase Flash option to force immediate mass erasure of the target processor flash.

Trigger Module Settings...

Select the MONITOR-HCS12> Trigger Module Settings... option to open the Trigger Module Settings dialog. Refer to the “Debugger HCS12 Onchip DBG Module User Interface” manual for all related information.
Bus Trace

Select the MONITOR-HCS12 > Bus Trace option to open the Trace component window within the debugger main window. Refer to the “Debugger HCS12 Onchip DBG Module User Interface” manual for all related information.

Select Derivative

Select the MONITOR-HCS12 > Select Derivative option to open the Derivative Selection Dialog Box.

Monitor Setup Window

The Monitor Setup window has two tabs, as shown in Figure 22.10 Monitor Setup Window - Monitor Communication Tab and Figure 22.11 Monitor Setup Window - Vector Table Mirroring Tab.

Figure 13.14 Monitor Setup Window - Monitor Communication Tab

Monitor Communication Tab

Using the Monitor Communication tab, it is possible to set or modify the current serial communication port when opening the “HOST Serial Communication Port” list box’s drop down list.
Checking the “Show Monitor TX/RX” checkbox, reports in the debugger Command Line window all low level communication frames between the host computer and the HCS12 Serial Monitor.

**Figure 13.15 Monitor Setup Window - Vector Table Mirroring Tab**

![Image of Monitor Setup Window - Vector Table Mirroring Tab]

**Vector Table Mirroring Tab**

Using the Vector Table Mirroring tab, it is possible to set the “Vector Table Mirroring” feature. See the Vector Redirection section of Freescale Serial Monitor for MC9S08GB/GT Application Note AN2140/D for all details.

The HCS12 Monitor start address is given in the **Monitor Start Address** edit box.

The real vector table address is given in the **Vector Table Address** edit box.

The list of vectors reserved by the HCS12 Serial Monitor is given in the **Vectors reserved by Monitor** edit box.

**NOTE** In the Vectors reserved by Monitor list box above, the number “1” matches the **RESET** vector, “2” is the **SWI** vector, “5” is the **ICG** vector, etc.

Vector table mirroring allows you to access chip vectors transparently. Indeed, the HCS12 Serial Monitor also uses some vectors, and the vector area is protected from erasing and overwriting. We recommend that you use this feature. Otherwise, user application vectors cannot be programmed as captured and are not protected from erasing/overwriting by the HCS12 Serial Monitor.
To enable this feature, check the “Enable Vector Table Mirroring” checkbox, then press the “Auto Detect” button to make the debugger search for the vector table address and vectors reserved by the HCS12 Serial Monitor. Once autodetection has succeeded, you can press the OK button to save and quit this window.

**Derivative Selection Dialog Box**

Within this dialog box, it is possible to select a specific derivative according to the *System Device Identification Register* (SDIDH, SDIDL) (also sometimes called PARTID) returned by the silicon device.

![Derivative Selection Dialog Box](image)

As several silicon devices might return the same value, a selection list is available to select the debugged derivative according to text reference written on the top of the silicon.
This document includes information on the BDIK Connection and helps you understand how to use this debugger connection. This document is divided into following sections:

- This BDIK Connection chapter provides answers to common questions and describes how to use the advanced features of the BDIK Connection.
- The BDIK Connection Introduction section introduces the BDIK Connection.
- The Interfacing BDIK and Your System section contains information about the connection between the BDI interface box and the debugger.
- The BDI Interface Software Setup section describes how to setup the BDI interface box using the ABATRON configuration tool. The discussion focuses on the firmware and the initialization list (startup init list).
- The BDIK Connection Menu Entries section provides a description of the BDIK Connection specific menu entries.
- The BDIK Connection Windows, Edit and Dialog Boxes section provides a description of the BDIK Connection specific dialog boxes.
- The BDIK Status Bar Information section describes the status bar messages for the BDIK Connection.
- The Terminal Emulation section describes how to emulate a text terminal between CPU12, CPU16 and CPU32 derivatives and the debugger.
- The Flash Programming section describes how to proceed to program on-chip non-volatile memory area.
- The BDIK Connection Environment section lists all the variables used by this connection to store the configuration.
- The BDIK Connection Commands section lists all the debugger commands specific to this connection.
- The Banked Memory Location Window section describes how to use the banked memory model manager with related CPU12 derivatives.
- The BDIK Connection Command Files section describes the BDIK Connection command files.
BDIK Connection

BDIK Highlights

- The *BDIK* Connection currently supports the BDI - Background Debug Interfaces - designed by *ABATRON AG*, including the *BDI-HS* on CPU12, and the *BDI1000* on the CPU12.
- For CPU12 and HCS12 supports banked memory handling (e.g. *M68HC812A4*, *M68HC912DG128*, *MC9S12DP256* ...). Refer to the *Banked Memory Location Window* section.

BDIK Requirements

- Ensure that your hardware target board incorporates a Background Debug Mode - BDM - port for CPU background interfacing with the BDI interface and the debugger. Please check the technical specifications provided by the *ABATRON User Manuals and Freescale*.
- One free serial communication port of your computer is required to communicate with the BDI interface. You may need to set it up even if you will be using an Ethernet communication instead of an RS-232 serial communication.

BDIK Connection Introduction

Another advanced feature of the debugger for the embedded systems development world is the ability to load different connections, which implements the interface with target systems. The *BDIK* Connection is introduced in this document.

This document describes the specific features of the BDIK Connection.

With this interface, you can download an executable program from the debugger environment to an external target system based on a Motorola MCU which will execute it. You also have the feedback of the real target system behaviour to the debugger.

The debugger supervises and monitors the MCU of the target system (i.e. control the CPU execution). You can read and write in internal/external memory (even while the CPU is running), single-step/run/stop the CPU, set breakpoints and watchpoints (not all CPUs) in the code.

NOTE  Unconcerned Components  As the code is executed by an external processor, memory statistics are not available with the BDIK Connection. Profiling, Coverage analysing and I/O simulation are not available with the BDIK Connection.
Interfacing BDIK and Your System

NOTE  BDI Structure, Configuration, Connection to the Host, Connection to the Target, Configuration, and Working Modes are described in ABATRON User Manuals.

The BDI interface is connected to the host computer either by a serial communication link or by a Ethernet connection. Any available communication port of your host system can be used. The communication protocol between the BDI and your target system is fully handled by the BDI Target driver automatically loaded with the BDIK Target Component. However you can adapt your target system to the BDI interface.

The BDI-to-target system communication uses a single wire serial connection. The target system has to be equipped with a BDM connector/port (see the BDI User Manual from ABATRON).

- Make sure that your hardware target board is/has been designed with a Background Debug Mode - BDM - port for CPU background interfacing with the BDI interface and the debugger. Please check the technical specifications provided by ABATRON User Manuals and MOTOROLA.
- One free serial communication port of your computer is required to communicate with the BDI interface. You may need to set it up even if later even if you use Ethernet communication instead of an RS-232 serial communication medium.

Figure 14.1 Your System/BDIK Interface
BDI Interface Software Setup

The BDIK Target component is delivered during installation and contains all required files, demo projects to use the BDIK debugger, and some BDI setup (init list) files. The drivers delivered with the BDI interface must be installed in order to make sure you use the latest drivers for the BDI interface. These files are delivered on a disk from ABATRON.

You must set up the target MCU through the BDI interface, according to your hardware configuration. Copy all files from the ABATRON disk to a new directory on your computer.

ABATRON provides a .EXE configuration application and a set of configuration files for specific evaluation boards and processors. These files contain microprocessor/microcontroller initialization data, vectors, chip selects for internal/external ROMs/ RAMs, running modes, etc. They contain information bound to the MCU and MCU version used, and information bound to the MCU environment on the board (RAM, ROM, PIA, ACIA, etc.). Each of these files is very specific.

Running the ABATRON Configuration Tool

The configuration program (e.g., B10C12.EXE for CPU12 processor with BDI1000, B20MCORE.EXE for the M-CORE with BDI2000, BDISHSCL.EXE for CPU16/CPU32 with BDI-HS, etc.) can also be run within the debugger on the condition that you browse for it choosing the menu entry BDI | Configure BDI Box... or specify the tool path in the BDI | Setup...dialog (Setup dialog). Otherwise, run the configuration tool directly from the File Manager or the Explorer.

Example with B10C12.EXE Configuration Tool

NOTE Please refer first to the ABATRON User Manual for further details about the BDI interface and BDI setup.
Firmware Loading

In the dialog box shown above, select **Setup | Firmware** to open the firmware dialog.

Figure 14.2 BDI1000 Setup for CPU12 Dialog Box

Figure 14.3 BDI Update/Setup Dialog Box
In the dialog box shown above, set the communication port and the baud rate according to your installation and press the Connect button. If the connection is passed, the current BDI firmware/logic is displayed. If unknown is displayed for the Current firmware/logic, you must load new firmware by pressing the Update button.

If you plan to use a Ethernet communication between your computer and the BDI interface, set the IP address reserved for the BDI then press the Transmit button. Quit the dialog by pressing the Ok button.

Initialization List (Startup Init List) Loading

Select File | Open ... from the File menu to load a configuration file (e.g. HC912DA128.BDI).

Select Setup | Init List... from the Setup menu to see and edit (if necessary) the content if this configuration file.

The Startup Init List/configuration file is displayed in the Startup Init List dialog box. You can edit, add, remove (etc.) “memory write” instructions in this dialog box to configure your MCU and MCU environment.
Figure 14.6 Startup Init List Dialog Box

<table>
<thead>
<tr>
<th>Type</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM8</td>
<td>0x0000012</td>
<td>0x01</td>
<td>INITEE: Map EEPROM to 0x0800</td>
</tr>
<tr>
<td>WM8</td>
<td>0x00000F1</td>
<td>0x00</td>
<td>EEPROM: Unprotect all EEPROM blocks</td>
</tr>
<tr>
<td>WM8</td>
<td>0x00000FF</td>
<td>0x00</td>
<td>FPAGE: Select Flash Array 0</td>
</tr>
<tr>
<td>WM8</td>
<td>0x00000F5</td>
<td>0x00</td>
<td>FEEMCR: Enable program of boot block</td>
</tr>
<tr>
<td>WM8</td>
<td>0x00000FF</td>
<td>0x02</td>
<td>FPAGE: Select Flash Array 1</td>
</tr>
<tr>
<td>WM8</td>
<td>0x00000F5</td>
<td>0x00</td>
<td>FEEMCR: Enable program of boot block</td>
</tr>
<tr>
<td>WM8</td>
<td>0x00000FF</td>
<td>0x04</td>
<td>FPAGE: Select Flash Array 2</td>
</tr>
<tr>
<td>WM8</td>
<td>0x00000F5</td>
<td>0x00</td>
<td>FEEMCR: Enable program of boot block</td>
</tr>
<tr>
<td>WM8</td>
<td>0x00000FF</td>
<td>0x06</td>
<td>FPAGE: Select Flash Array 3</td>
</tr>
<tr>
<td>WM8</td>
<td>0x00000F5</td>
<td>0x00</td>
<td>FEEMCR: Enable program of boot block</td>
</tr>
<tr>
<td>WM8</td>
<td>0x0000010</td>
<td>0x20</td>
<td>INITRM: Map RAM to 0x2000</td>
</tr>
</tbody>
</table>

Quit this dialog box by clicking OK and save the settings if necessary.

Communication with the Debugger Setup

Select Setup | Communication ... from the Setup menu to open the Communication Setup dialog box.

Figure 14.7 Communication Setup Dialog Box
BDIK Connection
BDI Interface Software Setup

In this dialog box, set the communication for your future use of the BDI with the debugger. Settings made here should be identical to communication settings made in the debugger within the Communication Setup Dialog Box. Press the Test button to check the setup then click OK to quit this dialog and save the settings if necessary.

BDI Working Mode and Setup/List Transmission

Select Setup | Mode... to open the dialog below and download the configuration to the target board by clicking Transmit, after setting the required parameters.

Figure 14.8 BDI Working Mode Dialog Box

![BDI Working Mode Dialog Box]

Loading the BDIK Connection

The target is set in the [Environment Variables] section of the project file, through the statement Target = BDIK.

The BDIK Connection automatically detects that the target is connected to your system. If nothing is detected, the Communication Device Specification Edit Box pops up. The target is not connected or is connected to a different port.

If no target is set or if a different target is set, load the BDIK Connection as described below.

In the debugger, select Component | Set Target... in the component menu.
The Set Connection dialog box is displayed. Select *BDIK Connection* in the list of proposed targets and click **OK**.

After a successful target loading, the Debugger Main Window Target menu item is replaced by *BDIK*. 

---

**BDIK Connection**

*BDI Interface Software Setup*

**Figure 14.9 Debugger Component Menu**

![Debugger Component Menu](image)

**Figure 14.10 Set Connection Dialog Box**

![Set Connection Dialog Box](image)
You can change the communication parameter (baud rate and port) by selecting the menu entry **BDIK | Connect**...

If communication with the BDI Interface could not be established, an error message is displayed followed by the Communication Device Specification dialog box.

In this dialog box, you can modify the device specification (e.g. Communication Port and baud rate). These settings are saved in the current project and will be used again in future sessions.
BDIK Connection

BDIK Connection Menu Entries

After loading the BDIK Connection, the Target menu item is replaced by BDIK.

Figure 14.13 Debugger BDIK Menu

The Set Bank... menu entry is available if the connected target processor is a CPU12/HC12 derivative.

If the connection to the target has failed, the entry Communication... of menu BDIK is replaced with Connect....

The different entries of the BDIK menu are described below:

Load...

Select BDIK | Load... to load the application to debug, i.e. a .ABS file.

Reset

The menu entry BDIK | Reset executes the Reset Command File and resets the hardware target. The BDI interface automatically processes the initialization list (startup init list) stored in the interface.

Communication... or Connect...

Select entry BDIK | Communication... or BDIK | Connect... to display the Communication Device Specification Edit Box. If the connection to the target has failed, the entry Communication... of menu BDIK is replaced with Connect....
Setup...
Select **BDI K | Setup...** to open the Setup Dialog Box to set the link to the ABATRON configuration tool, to set the download mode, or to set the *Continue on illegal break* (*banked hardware breakpoint*) option (only available for HC12/CPU12 derivative).

Configure BDI Box...
Select **BDI K | Configure BDI Box...** to open the configuration tool delivered by ABATRON that you copied on your computer. If no application tool path is currently set in the Setup Dialog Box, a browser dialog, "Select BDI Box Configuration Tool", is automatically opened to create a link to the configuration tool application. The link is then saved in the Setup Dialog Box.

Set Bank...
This dialog is only available if the connected processor is a *Freescale CPU12/HC12* derivative. Select the entry **BDI K | Set Bank...** to display the Banked Memory Location Window.

Command Files
Select the entry **BDI K | Command Files** to display the Target Interface Command Files Window.

Help
Select the entry **BDI K | Help** to open the BDIK Connection Help File.

**BDIK Connection Windows, Edit and Dialog Boxes**

This section describes the dialog boxes which are specific to the **BDIK** Connection.

Those dialogs are:

- The Communication Device Specification Edit Box.
- The Setup Dialog Box.
- The Banked Memory Location Window (available only if the connected derivative is a Freescale HC12 that supports banking).
- The Target Interface Command Files Window.
Communication Device Specification Edit Box

The Communication Device Specification edit box pops up automatically if the BDIK Connection could not establish the communication with the BDI (box) interface. However, this dialog box can be opened by selecting the menu entry BDIK | Communication ... or BDIK | Connect ...

If the connection to the target has been successfully achieved (dialog opened using menu entry BDIK | Communication ...), it is not possible to modify the Communication Device edit box. Only the Show Protocol check box can be modified.

If the connection to the BDI box has failed (dialog box automatically opened or using menu entry BDIK | Connect ...), it is possible to modify the Communication Device Specification edit box.

The Communication Device specification edit box should contain the communication settings to connect to the BDI box. The syntax of the initialisation string is:

"COMn baudrate"

where n is the COM port number like 1, 2, 3, etc., and where baudrate is 9600, 19200, 38400, 57600, 115200, according to the setup done with the ABATRON configuration application, e.g. "COM1 57600".

For the communication via an Ethernet and bdiNet, use the following initialisation string:

"NETWORK ip_address port"

where ip_address is the IP address of the BDI box or bdiNet in the form xxx.xxx.xxx.xxx and port is the bdiNet port, usually "1" for BDI1000 and BDI2000, e.g. "NETWORK 151.120.25.101 1".
The Show Protocol check box allows you to switch on/off the displays of the messages sent between the debugger and the BDI interface. If the Show Protocol box is checked, all commands and responses sent and received are reported in the Command Line window.

**NOTE** The Show Protocol checkbox is a useful debugging feature if there is a communication problem. The settings performed in the Communication Device Specification edit box are stored for a later debugging session in the [BDIK] section of the project file.

**Setup Dialog Box**

The Setup dialog box is opened selecting the BDIK menu entry BDIK | Setup...

**Figure 14.15 Setup Dialog Box**

The BDI Box Configuration Tool Path edit box is set up with the path and application name of the configuration tool from ABATRON. The application tool is automatically browsed when selecting the BDIK | Configure BDI Box... menu entry and browsing for the application. Otherwise, press the Browse... button to look for the tool. The edit box contains (for example):
"C:\tmp\B10c12.exe"

In the Download Mode and Data Transfer Verification, you can set different options to transfer data from the computer to the BDI box. By default, use the Verify only first... option. If necessary, you can set a different option to improve transfer speed or security. By default, data compression is enabled for asynchronous communication channels. With older computers, it is possible that download speed is faster without data compression.

The Continue on illegal break (banked hardware breakpoint) option check box is only available for the HC12/CPU12 derivative. You can check this check box to overcome the 2-byte address size on-chip break module, which does not handle the PPAGE (e.g. HC912DG128). Note that internally, the target is halted by the hardware breakpoint (in Flash memory), compared with the breakpoint that you set, then relaunched if not (bank) matching. This feature is available as an optional. Code execution breaks are not handled when this option is set and illegal code execution is not detected. Please use this option carefully.

BDIK Status Bar Information

When the BDIK connection has been loaded, specific information is given in the debugger status bar. From left to right, the name of the target CPU and the debugger status (target status) are displayed.

Figure 14.16 BDIK Connection - Debugger Status Bar

| For Help, press F1 | MC68HC12 (CPU12) | HALTED |

Status Messages

Status messages are described in the following sections.

BDI ready V x.xx

The debugger is ready and waits until a new target or application is loaded. This message is generated once the debugger has been started and the connection to the hardware target has been established by the BDI. "V x.xx" is the current BDI firmware version.

No Link To Target

Connection to the target system has failed.

RUNNING

The application is currently executing in the debugger.
HALTED
Execution of the application has been stopped on user request. The menu entry Run | Halt or the Halt icon in the tool bar has been selected.

RESET
This message is generated when the debugger has been reseted on user request. The menu entry BDIK | Reset or the Reset icon in the tool bar has been selected, or the command Reset has been used.

Stepping and Breakpoint Messages
Stepping and breakpoint messages are described in the following sections.

STEPPED
Execution of the application has been stopped after a single step on source level. The menu entry Run | Single Step or the Single Step icon in the tool bar has been selected.

STEPPED OVER
Execution of the application has been stopped after a step over a function call. The menu entry Run | Step Over or the Step Over icon in the tool bar has been selected.

STOPPED
Execution of the application has been stopped after a step out from function call. The menu entry Run | Step Out or the Step Out icon in the tool bar has been selected.

TRACED
Execution of the application has been stopped after an single step on assembler level. The menu entry Run | Assembly Step or the Assembly Step icon in the tool bar has been selected.

BREAKPOINT
Execution of the application has been stopped because a breakpoint has been reached.
WATCHPOINT

Execution of the application has been stopped because a watchpoint has been reached.

Terminal Emulation

The BDIK Connection supports the emulation of a terminal. The BDI interface supports this emulation for CPU12. This allows the target application to write into the debugger Terminal component. Also, characters typed on the host’s keyboard can be directed to the target application. In order to use the terminal emulation, the Terminal component has to be opened in the debugger:

Choose Component | Open | Terminal to open the Terminal component.

In order to simulate the terminal I/O, a work space of 4 bytes is needed. The address of this work space has to be configured with the setup program from ABATRON.

For more information, see the section “Terminal” in the User Manual from ABATRON and check the “termbgnd.c” source file for communication primitives on the installation disk for BDI from ABATRON.

Refer to the section Terminal Component in the debugger core manual.

Example for CPU12 Targets:

The following structure is located in unpaged data memory on the target:

0x00 RX - Flag (Byte)
0x01 RX - Char (Byte)
0x02-0x03 TX - String Pointer (Word)

The address of this structure is defined during BDI box setup. The TermData structure address (0x0800) must match with the software setup of the BDI, and exactly match the Terminal Address in the BDI Working Mode dialog of the ABATRON tool. Refer to the BDI Interface Software Setup section.

While the target is running, the BDI periodically checks if the TX - String Pointer is not zero. Received characters from the host are written to RX - Char, and the RX - Flag is set.
The following is a possible target implementation:

**Listing 14.1 CPU-12 Target Implementation**

typedef struct {
    unsigned char rxFlag;
    unsigned char rxChar;
    char* txBuffer;
} TermDataT;

#define TermData (*((TermDataT*)(0x0800)))

static char txBuffer[2];

char GetChar(void)
{
    char rxChar;
    while (TermData.rxFlag == 0); /* wait for input */
    rxChar = TermData.rxChar;
    TermData.rxFlag = 0;
    return rxChar;
}

void PutChar(char ch)
{
    txBuffer[0] = ch;
    txBuffer[1] = 0;
    TermData.txBuffer = txBuffer;
    while (TermData.txBuffer != 0); /*wait for output buffer empty*/
}

void PutString(char *str)
{
    TermData.txBuffer = str;
    while (TermData.txBuffer != 0); /*wait for output buffer empty*/
}
Flash Programming

The BDI supports downloading and debugging code that runs in the internal Flash memory of the target CPU. Breakpoints are automatically mapped to the hardware breakpoint registers. To erase the internal flash and to enable writing to flash, Direct Commands to BDI are used. Direct commands to BDI can be executed from a .CMD command file like in the Preload Command File and the Postload Command File or the Command Line component with the BDI command.

This flash programming support is not available for all CPUs. Please check for availability in the User Manual for your CPU from ABATRON.

Use the following sequence to load code into the internal flash:

1. Flash.Erase
2. Flash.Load
3. Download the code using the normal debugger BDI | Load... browser/menu entry or the debugger LOAD command. Every write to the flash range including WB, WW, WL commands uses the programming algorithm.
4. Flash.Idle

Examples of HC12/CPU12 Direct Commands

For the Direct Commands, the following default values are used:

**HC912B32:**

FLASH.ERASE [addr=8000] [size=8000] [sram=BDI-Workspace]
FLASH.LOAD [addr=8000] [size=8000] [sram=BDI-Workspace]

**HC912D60:**

FLASH.ERASE [addr=8000] [size=8000] [sram=BDI-Workspace]
FLASH.LOAD [addr=1000] [size=F000] [sram=BDI-Workspace]

**HC912DA/G128:**

FLASH.ERASE [addr=8000] [size=8000] [sram=BDI-Workspace]
FLASH.LOAD [addr=4000] [size=C000] [sram=BDI-Workspace]

Finally, set your Preload Command File and Postload Command File of your project directory with the BDI command as shown:

**HC912B32:**

Before downloading (in Preload Command File):
BDIK Connection
Flash Programming

BDI flash.erase
BDI flash.load

After downloading (in Postload Command File):
BDI flash.idle

HC912D60:
Before downloading (in Preload Command File):
BDI flash.erase addr=8000 size=8000
BDI flash.erase addr=1000 size=7000
BDI flash.load

After download (in Postload Command File):
BDI flash.idle

HC912DA128 / HC912DG128:
Before downloading (in Preload Command File):
BDI flash.erase addr=08000 size=8000
BDI flash.erase addr=28000 size=8000
BDI flash.erase addr=48000 size=8000
BDI flash.erase addr=68000 size=8000
BDI flash.load

After downloading (in Postload Command File):
BDI flash.idle
BDIK Connection Environment

Default Target Setup

As any other debugger connection, the BDIK Connection can be loaded from the Target menu or can be set as a default target in the project file.

The target is set in the [Environment Variables] section from your project file as shown above. However, if the target is not defined, load the BDIK Connection interactively. Please refer to the Loading the BDIK Connection section.

Example of the [Environment Variables] section from your project file:

```
[Environment Variables]
...
Target=BDIK
...
```

**NOTE** Please see the True-Time Simulator and Real-Time Debugger core manual for further information about the project file.

BDIK Connection Environment Variables

This section describes the environment variables which are used by the BDIK Connection. The BDIK Connection specific environment variables are:

- `BDICONF`
- `COMDEV`
- `COMPRESS`
- `SHOW PROT`
- `SKIPILLEGALBREAK`
- `VERIFY`

These variables are stored in the [BDIK] section from the project file.

**Listing 14.2 Example of the [BDIK] section from the project file:**

```
[BDIK]
CMDFILE0=CMDFILE STARTUP ON "startup.cmd"
CMDFILE1=CMDFILE RESET ON "reset.cmd"
CMDFILE2=CMDFILE PRELOAD ON "preload.cmd"
CMDFILE3=CMDFILE POSTLOAD ON "postload.cmd"
```
The remainder of this section describes each of the variables available for the *BDIK* Connection. The variables are listed in alphabetical order and are divided into several topics.

### Table 14.1 Variable Description Parameters

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Provides a short description of the variable.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the variable in a EBNF format.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the default setting for the variable.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the variable and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Small example of how to use the variable.</td>
</tr>
</tbody>
</table>

### BDICONF

**Short Description**

Defines the ABATRON configuration tool file and path

**Syntax**

`BDICONF=ConfigurationToolFileNameandPath`

where `ConfigurationToolFileNameandPath` is the ABATRON configuration tool file name and path.

**Default**

The default value does not exist. The string "Enter here the path to the ABATRON configuration tool." is displayed in the edit box.
Description
This variable defines the communication device between the computer and the BDI. It is set according to the BDIBox Configuration ToolPath edit box of the Setup Dialog Box. The BDIBox Configuration ToolPath edit box can be set up with the path and application name of the configuration tool from ABATRON. The application tool is automatically browsed when selecting the BDIK | Configure BDIBox... menu entry and browsing for the application. Otherwise, press the Browse... button to look for the tool.

Example
BDICONF=C:\tmp\B10c12.exe

COMDEV

Short Description
Defines the communication device between the computer and the BDI.

Syntax
COMDEV=COMn baudrate
where n is the COM port number like 1, 2, 3, etc. and where baudrate is 9600, 19200, 38400, 57600, 115200, according to the setup done in the ABATRON configuration application.

For the communication via an Ethernet:
COMDEV=NETWORK ip_address port
where ip_address is the IP address of the BDI box or bdiNet in the form xxx.xxx.xxx.xxx and port is the bdiNet port, usually "1" for BDI1000 and BDI2000.

Default
The default value is COM1 57600.

Description
This variable defines the communication device between the computer and the BDI. It is set according to the Communication Device Specification Edit Box.

Example
COMDEV=COM1 57600
COMPRESS

**Short Description**
Sets data transfer compression

**Syntax**
```
COMPRESS=1 | 0
```

**Default**
The default value is 1.

**Description**
This variable sets the BDI download mode with data compression. By default, data compression is enabled for asynchronous communication channels. With older computers, it is possible that download speed is faster without data compression. It is set according to the Use Data Compression check box of the Setup Dialog Box.

**Example**
```
COMPRESS=1
```

SHOWPROT

**Short Description**
Set Show Protocol On/Off

**Syntax**
```
SHOWPROT=1 | 0
```

**Default**
The default value is 0.

**Description**
If the Show Protocol is used, all the commands and responses sent and received are reported in the Command Line component of the debugger.
If the variable is set to 1, Show Protocol is activated.
This variable is set according to the Show Protocol check box of the Communication Device Specification Edit Box.

**Example**

```
SHOWPROT=1
```

**NOTE** The Show Protocol is a useful debugging feature if there is a communication problem.

---

**SKIPILLEGALBREAK**

**Short Description**

Enables skipping illegal breakpoints

**Syntax**

```
SKIPILLEGALBREAK=1 | 0
```

**Default**

The default value is 0.

**Description**

This variable is set according to the Continue on illegal break (banked hardware breakpoint) option check box of the Setup Dialog Box.

The Continue on illegal break (banked hardware breakpoint) option check box is only available for the HC12/CPU12 derivative. You can check this check box to overcome the 2-byte address size on-chip break module which does not handle the PPAGE (e.g. HC912DG128). Note that internally, the target will be halted by the hardware breakpoint (in Flash memory), compared with the breakpoint that you set, then relaunched if not (bank) matching.

**Example**

```
SKIPILLEGALBREAK=1
```
VERIFY

**Short Description**
Sets data transfer verification

**Syntax**

```
VERIFY=0|1|2|3
```

with 0 for no verification at all (fastest mode), 1 for first byte verification only, 2 for all data read back verification, and 3 for only verification (no write).

**Default**
The default value is 1.

**Description**
This variable sets the BDI download mode with data verification. By default, use `VERIFY` only first...option. If necessary, you can set a different option to improve transfer speed or security. It is set according to the Data Transfer Verification radio buttons of the Setup Dialog Box.

**Example**

```
VERIFY=1
```
BDIK Connection

BDIK Connection Commands

This section describes the BDIK Connection-specific commands that are used when the BDIK Connection is set.

The BDIK Connection specific commands are:

- BANKREG
- BDI
- PROTOCOL
- RESET

Those commands are entered in the BDIK Connection Command Files or in the Command Line component of the debugger.

This section describes each of the commands available for the BDIK Connection. The commands are listed in alphabetical order. Each is divided into several topics.

Table 14.2 Command Description  Parameters

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Provides a short description of the command.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the command in a EBNF format.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the command and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Small example of how to use the command.</td>
</tr>
</tbody>
</table>

BDI

Short Description
Executes any direct BDI command

Syntax

BDI <ABATRON_direct_command>

where ABATRON_direct_command has the following syntax:

<Object>.<Action> [ <parName>=<parameterValue> ] ...

HC(S)12(X) Debugger Manual 435
BDIK Connection

**BDIK Connection Commands**

**Description**

The BDI command executes any ABATRON direct command. ABATRON direct commands are described in the *User Manual* for your CPU from ABATRON. They are commonly used to download to non-volatile memory areas (please see also the Flash Programming section).

**Example**

BDI FLASH.ERASE addr=8000 size=8000 sram=0800

---

**BANKREG**

**Short Description**

Sets banked memory handling for HC12/CPU12 derivatives

**Syntax**

BANKREG [PPAGE=\langle PPAGE\_register\_adrs\rangle]
   [DPAGE=\langle DPAGE\_register\_adrs\rangle]
   [EPAGE=\langle EPAGE\_register\_adrs\rangle]

**Description**

**CAUTION**

This command is still available, but for compatibility only. However, it should not be used. The *Banked Memory Location Window* handles the banked memory handling when debugging on HC12/CPU12 derivatives.

The BANKREG command lets you define if paging is used, like PPAGE (HC912DG128, HC812A4), DPAGE (HC812A4) or EPAGE (HC812A4). This command must be inserted in the *Startup Command File* of your project directory. As soon as the command is executed, the specified registers are displayed in the Register component window.

**Example**

for HC812A4:

BANKREG PPAGE=0x35 DPAGE=0x34 EPAGE=0x36

for HC912DG128:

BANKREG PPAGE=0xFF
PROTOCOL

Short Description
Switch on/off the Show Protocol functionality

Syntax
PROTOCOL ON|OFF

Description
If this command is used, all the messages sent to and received from the debugger are reported in the Command Line window of the debugger.

The Show Protocol facility can also be switched on/off using the corresponding check box in the Communication Device Specification Dialog Box.

The state of the Show Protocol is stored in the [BDIK] section of the project file using variable SHOWPROT.

Example
PROTOCOL ON

NOTE The Show Protocol is a useful debugging feature if there is a communication problem.

RESET

Short Description
Reset of the target board

Syntax
RESET

Description
Use this command to reset the target from the Command Line component of the debugger. The Reset Command File is also executed and the BDI interface automatically processes the initialization list (startup init list) stored in the interface.
You can define which banked memory format you want to use and its location in the memory of the Motorola HC12 or HCS12 derivative you are using. The PPAGE, DPAGE and the EPAGE formats are supported, if available on the target HC12 or HCS12 derivative.

Banked Memory Location Window

This Banked Memory Location window is available only if the connected derivative is a Freescale HC12 (CPU12) or HCS12.

The Banked Memory Location window can be opened by selecting the menu entry "TargetName" | Set Bank.... (In this section, Target Name is the name of the target, like SDI, Hitex, BDIK, ICD-12, Noral-BDM, etc.) Using some connections, the Banked Memory Location window automatically pops up when the connection is used with a Freescale HC12 or HCS12 derivative that supports banking. In this case, it also pops up when the banked memory area locations are not defined in the project file of the current project directory.

In this window you can define which banked memory you want to use and its location. The PPAGE, DPAGE and the EPAGE indexes are supported, if they are available on the currently connected HC12 or HCS12 derivative.

Figure 14.17 Banked Memory Location Window - PPage Tab
PPAGE Tab

The PPAGE tab of the Banked Memory Location window lets you set up the PPAGE banked memory area. Once you have enabled PPAGE memory banking by checking the Enable Banked Memory Area check box, you must set the start address and the end address of this memory range.

The PPAGE register address must be specified in hexadecimal (e.g. 0x35 for HC812A4, 0xFF for HC912DG128, 0x30 for MC9S12DP256B).

The number of pages must be specified in decimal (e.g. 0 to 256 for HC812A4, 8 for HC912DG128, 64 for the MC9S12DP256B).

NOTE  For the Hitex Connection, the PPAGE index tab does not appear in this dialog box if the PPAGE register is not available on the currently connected Motorola HC12 derivative. For this connection it is not needed to enter the PPAGE register address.

![Banked Memory Location Window - DPage Tab](image)

Figure 14.18 Banked Memory Location Window - DPage Tab

DPAGE Tab

The DPAGE index tab of this window lets you set up the DPAGE banked memory area. Once you have enabled DPAGE memory banking by checking the Enable Banked Memory Area check box, you must set the start address and the end address of this memory range.

The number of pages must be specified in decimal (e.g. 0 to 256 for HC812A4). The DPAGE register address must be specified in hexadecimal (e.g. 0x34 for HC812A4).
NOTE For the Hitex Connection, the DPAGE index tab does not appear in this dialog box if the DPAGE register is not available on the currently connected Motorola HC12 derivative. For this connection it is not needed to enter the DPAGE register address.

Figure 14.19 Banked Memory Location Window - EPage Tab

EPAGE Tab

The EPAGE index tab of this dialog box lets you set up the EPAGE banked memory area. Once you have enabled EPAGE memory banking by checking the Enable Banked Memory Area check box, you must set the start address and the end address of this memory range.

For some connections the number of pages must be specified in decimal (e.g. 0 to 256 for HC812A4).

For some other connections, the EPAGE register address must be specified in hexadecimal (e.g. 0x36 for HC812A4).

NOTE For the Hitex Connection, the EPAGE index tab does not appear in this dialog box if the EPAGE register is not available on the currently connected Motorola HC12 derivative. For this connection it is not needed to enter the DPAGE register address.
Various Tab (Not For All Connections)

If you are using an HC12 derivative which supports banking and you don’t want to enable this mechanism, or if you want to use only one bank out of three, you can suppress the automatic display of the Banked Memory Location dialog by checking the Display dialog at connection if banked memory locations not defined check box.

**NOTE** The settings entered in this dialog box are stored for a later debugging session in the ["targetName"] section of the project file.

**NOTE** When using the HITEK Connection and the M68HC12DG128 DProbeHC12-DG, at least one page must be defined from 0x8000 to 0xBFFF. Otherwise some display problems might be encountered in the Memory component of HI-WAVE.

Banked Memory Location-associated Commands

The following sections describe the Banked Memory Location Command Line commands which are used by the BDIK Connection. These variables are:

BANKWINDOW

Those commands can be entered in the BDIK Connection Command Files or in the Command Line component of HI-WAVE.
The Banked Memory Location commands which are used by the connection are described as shown in the following table.

### Table 14.3 Command Description Parameters

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Provides a short description of the command.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the command in a EBNF format.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the command and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Small example of how to use the command.</td>
</tr>
</tbody>
</table>

The following sections describe each command related to the Banked Memory Location available for the connection. The variables are listed in alphabetical order.

## BANKWINDOW

### Short Description
Specify a banked memory area and its status (enable/disable).

### Syntax

```
BANKWINDOW <bank> [OFF|ON] [<range> <reg> <numofpages>]
```

with

- `bank = (PPAGE | DPAGE | EPAGE)`
- or
- `BANKWINDOW VARIOUS [DLGATCONNECT | NODLGATCONNECT]`

### Description

The command `BANKWINDOW` allows to set up the debugger to work in banked memory model.

Three different Banked Memory Area can be defined: DPAGE, EPAGE and PPAGE. Each banked memory area has an associated bank register, which is displayed in the Register component.

Using `BANWINDOW PPAGE ...` command will have the same effect than using the PPAGE index tab in the Banked Memory Location Window.
Using BANWINDOW DPAGE ... command will have the same effect than using the DPAGE index tab in the Banked Memory Location Window.

Using BANWINDOW EPAGE ... command will have the same effect than using the EPAGE index tab in the Banked Memory Location Window.

Using BANWINDOW VARIOUS ... command will have the same effect than using the Various index tab in the Banked Memory Location Window.

A banked memory area is defined by its start address, end address and the address of the Bank register.

The maximum number of pages parameter allows to see in the memory component only the available pages.

The status of the banking mechanism in the debugger is also monitored through this command: a command can be defined, but the debugger banking mechanism can be disabled.

Consider the command:

```
BANKWINDOW PPAGE ON 0x8000..0xBFFF 0x30 64
```

This command allows to use the banked memory model in the debugger using the MC9S12DP256B.

This commands means the PPAGE register located at address 0x30 must be used to build the PC address when the code is located in banked memory area, from 0x8000 to 0xBFFF. The 64 first page in the memory map are visible (page 0x3F is the last one).

The PPAGE register (located at address 0x30) will be displayed in the register component.

The bank settings are stored in the ["targetName"] section of the PROJECT file using variable BANKWINDOWn.

**BANKWINDOW Examples**

The Banking status can be gotten by typing BANKWINDOW without any parameters in the Command Line component.

**Listing 14.3 BANKWINDOW > Banking Status**

```bash
in>in>bankwindow
PPAGE Settings:
Status: enabled
Reg. Adr: 0x30
Range: 0x8000 to 0xBFFF
Number of Pages: 64

DPAGE Settings:
Status: disabled
```
BDIK Connection
HC12 and HCS12 Banked Memory support

Reg. Adr: 0x34
Range: 0x7000 to 0x7fff
Number of Pages: 0

EPAGE Settings:
Status: disabled
Reg. Adr: 0x36
Range: 0x400 to 0x7ff
Number of Pages: 0

The status of the PPAGE Banked Memory area can be changed:

Listing 14.4 BANKWINDOW > PPage Status Change

in>BANKWINDOW PPAGE OFF
in>BANKWINDOW
PPAGE Settings:
Status: disabled
Reg. Adr: 0x30
Range: 0x8000 to 0xbfff
Number of Pages: 64

DPAGE Settings:
Status: disabled
Reg. Adr: 0x34
Range: 0x7000 to 0x7fff
Number of Pages: 0

EPAGE Settings:
Status: disabled
Reg. Adr: 0x36
Range: 0x400 to 0x7ff
Number of Pages: 0

Banked Memory Location-associated Environment Variables

The following sections describe the Banked Memory Location environment variables which are used by the BDIK connection. These variables are:
BANKWINDOWn

These variables are stored in the ["targetName"] section from the project file.

Example of the [BDIK] target section from a project file:

```
[BDIK]
BANKWINDOW0=BANKWINDOW PPAGE ON 0x8000..0xBFFF 0x30 64
BANKWINDOW1=BANKWINDOW DPAGE OFF 0x7000..0x7FFF 0x34 256
BANKWINDOW2=BANKWINDOW EPAGE OFF 0x400..0x7FF 0x36 256
```

The Banked Memory Location environment variables which are used by the connection are described as shown in the following table.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Provides a short description of the variable.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the variable in a EBNF format.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the default setting for the variable.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the variable and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Small example of how to use the variable.</td>
</tr>
</tbody>
</table>

The following sections describe each variable available for the connection. The variables are listed in alphabetical order.

BANKWINDOWn

**Short Description**

Contains a BANKWINDOW Command Line command to be used to set up the Banked Memory support.

**Syntax**

```
BANKWINDOWn=<one BANKWINDOW Command Line command>
```
Default
All available banked memory area are disabled by default.
The default PPAGE memory banked area is 0x8000 to 0xBFFF, 8 pages allowed, with PPAGE register at address 0x35.
The default DPAGE memory banked area is 0x7000 to 0x7FFF, 256 pages allowed, with PPAGE register at address 0x34.
The default EPAGE memory banked area is 0x400 to 0x7FF, 256 pages allowed, with PPAGE register at address 0x36.
The default settings for the VARIOUS page is that the bank window dialog is displayed automatically when connecting when settings are not done (do only apply to the Hitex Connection).

Description
The BANKWINDOWn variable specifies a command file definition using BANKWINDOW Command Line command. Three or four of those entries should be present in the project file, depending on the connection.
Those variables are used to store the Banked Memory Location definition (range, address, number of pages) and status (enable/disable) specified either with the BANKWINDOW Command Line command the Banked Memory Location Window.

Example
BANKWINDOW0=BANKWINDOW PPAGE OFF 0x8000..0xBFFF 0x30 64
BANKWINDOW1=BANKWINDOW DPAGE OFF 0x7000..0x7FFF 0x34 256
BANKWINDOW2=BANKWINDOW EPAGE OFF 0x400..0x7FF 0x36 256
BANKWINDOW3=BANKWINDOW VARIOUS DLGATCONNECT

BDIK Connection Command Files

The BDIK Connection offers the possibility to play a specific command file on different events:
- at connection: Startup Command File.
- at reset: Reset Command File.
- right before a file is loaded: Preload Command File.
- right after a file has been loaded: Postload Command File.
- right before a "Non Volatile Memory" is erased or right before a file is programmed in "Non Volatile Memory": Vppon Command File. This command file can be used for example to enable a programming voltage by software. This command file is not available for all connections.
• right after a "Non Volatile Memory" has been erased or right after a file has been programmed in "Non Volatile Memory": Vppoff Command File. This command file can be used for example to disable a programming voltage by software. This command file is not available for all connections.

The command files full name and status (enable/disable) can be specified either with the CMDFILE Command Line command or using the Target Interface Command Files Window.

You can use any HI-WAVE command in those files and take advantage of the wide set of commands introduced in the HI-WAVE manual to setup the target hardware on one of those events.

Example of a command file content:

```
WB 0x0035 0x00
WB 0x0012 0x11
PROTOCOL OFF
```

- The WB 0x0035 0x00 command sets memory location 0x35 to 0.
- The WB 0x0012 0x11 command sets memory location 0x12 to 0x11.
- The command PROTOCOL OFF switch of the Show Protocol

### Startup Command File

The Startup command file is executed by HI-WAVE straight after the connection has been loaded.

The Startup command file full name and status (enable/disable) can be specified either with the CMDFILE STARTUP Command Line command or using the Startup tab of the Target Interface Command Files Window.

By default, the STARTUP.CMD file located in the current project directory is enabled as the current Startup command file.

### Reset Command File

The Reset command file is executed by HI-WAVE straight after the reset button, menu entry or Command Line command has been selected.

The Reset command file full name and status (enable/disable) can be specified either with the CMDFILE RESET Command Line command or using the Reset tab of the Target Interface Command Files Window.

By default, the RESET.CMD file located in the current project directory is enabled as the current Reset command file.
Preload Command File

The Preload command file is executed by HI-WAVE right before an application is loaded to the target system through the connection.

The Preload command file full name and status (enable/disable) can be specified either with the CMDFILE PRELOAD Command Line command or using the Preload index of the Target Interface Command Files Window.

By default, the PRELOAD.CMD file located in the current project directory is enabled as the current Preload command file.

Postload Command File

The Postload command file is executed by HI-WAVE right after an application has been loaded to the target system through the connection.

The Postload command file full name and status (enable/disable) can be specified either with the CMDFILE POSTLOAD Command Line command or using the Postload index of the Target Interface Command Files Window.

By default, the POSTLOAD.CMD file located in the current project directory is enabled as the current Postload command file.

Vppon Command File

The Vppon command file is executed by HI-WAVE right before a "Non Volatile Memory" is erased or right before a file is programmed in "Non Volatile Memory" to the target system through the connection Non Volatile Memory Control dialog (Flash... menu entry) or FLASH PROGRAM/ERASE commands from Flash Programming utilities.

The Vppon command file full name and status (enable/disable) can be specified either with the CMDFILE VPPON Command Line command or using the Vppon index of the Target Interface Command Files Window.

By default, the VPPON.CMD file located in the current project directory is enabled as the current Vppon command file.

This command file can be used for example to enable a programming voltage by software.

NOTE This command file is not available for all connections.

Vppoff Command File

The Vppoff command file is executed by HI-WAVE right after a "Non Volatile Memory" has been erased or right after a file has been programmed in "Non Volatile Memory" to the target system through the connection Non Volatile Memory Control dialog (Flash...
menu entry) or FLASH PROGRAM/ERASE commands from *Flash Programming*
utilities.

The Vppoff command file full name and status (enable/disable) can be specified either
with the CMDFILE VPPOFF Command Line command or using the Vppoff
index of the Target Interface Command Files Window.

By default, the VPPOFF.CMD file located in the current project directory is enabled as
the current Vppoff command file.

This command file can be used for example to disable a programming voltage by
software.

**NOTE** This command file is not available for all connections.

## Associated Commands

This section describes the Command Files command which can be used when the
connection is set.

The Connection-specific command is:

**CMDFILE**

Those commands can be entered in the BDIK Connection Command Files or in the
Command Line component of HI-WAVE.

This section describes each command available for the connection. The commands are
listed in alphabetical order.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Provides a short description of the command.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the command in a EBNF format.</td>
</tr>
</tbody>
</table>
| Description        | Provides a detailed description of the command and how
to use it. |
| Example            | Small example of how to use the command. |

### CMDFILE

**Short Description**

Defines a command file path, name and status (enable/disable).
BDIK Connection

BDIK Connection Command Files

Syntax
CMDFILE <file kind> ON|OFF ["<file name and path>"]
and
file kind = STARTUP|RESET|PRELOAD|POSTLOAD|VPPON|VPPOFF

Description
The CMDFILE command is to be used set up a command file full name and status (disabled/enabled).

This command allows you to perform the same settings than using the Command Files window through the Command Line component.

The settings of a command file are stored in the ["targetName"] section of the PROJECT file using variable CMDFILEn.

Example
The list of available command files (and their status) can be get typing CMDFILE without any parameters in the Command Line component.

in>CMDFILE
Hitex Target Interface Command Files:
STARTUP ON startup.cmd
RESET ON reset.cmd
PRELOAD ON preload.cmd
POSTLOAD ON postload.cmd

The status of the Startup command file can be changed:

in>CMDFILE STARTUP OFF "my own startup.cmd"
in>CMDFILE
Hitex Target Interface Command Files:
STARTUP OFF my own startup.cmd
RESET ON reset.cmd
PRELOAD ON preload.cmd
POSTLOAD ON postload.cmd
Target Interface Command Files Window

The Target Interface Command Files window can be opened by selecting menu entry "Target Name | Command Files. (In this section, TargetName is the name of the target, like SDI, Hitex, BDIK, ICD-12, Noral-BDM, etc.)

Figure 14.21 Target Interface Command Files Window

Each tab of this window corresponds to an event on which a command file can be automatically run from HI-WAVE: Startup Command File, Reset Command File, Preload Command File, Postload Command File, Vppon Command File (not available for all targets), or Vppoff Command File (not available for all targets).

The command file in the edit box is executed when the corresponding event occurred.

Using the Browse button, you can set up the path and name of the command file.

The Enable Command File check box allows to enable/disable a command file on an event. By default, all command files are enabled:

- the default Startup command file is STARTUP.CMD,
- the default Reset command file is RESET.CMD,
- the default Preload command file is PRELOAD.CMD,
- the default Postload command file is POSTLOAD.CMD,
- the default Vppon command file is VPPON.CMD,
- the default Vppoff command file is VPPOFF.CMD.
NOTE The settings performed in this window are stored for a later debugging session in the "targetName" section of the PROJECT file using variables CMDFILE0, CMDFILE1, ... CMDFILEn.

Associated Environment Variables
This section describes the Command Files window environment variables which are used by the connection.

CMDFILEn
These variables are stored in the "targetName" section from the project file.

Example of the [NORAL FLEX BDM] target section from the project file:

[NORAL FLEX BDM]
CMDFILE0=CMDFILE STARTUP ON "startup.cmd"
CMDFILE1=CMDFILE RESET ON "reset.cmd"
CMDFILE2=CMDFILE PRELOAD ON "preload.cmd"
CMDFILE3=CMDFILE POSTLOAD ON "postload.cmd"
CMDFILE4=CMDFILE VPPON ON "vppon.cmd"
CMDFILE5=CMDFILE VPPOFF ON "vpoff.cmd"

The following section describes each variable available for the connection. The variables are listed in alphabetical order.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Provides a short description of the variable.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the variable in a EBNF format.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the default setting for the variable.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the variable and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Small example of how to use the variable.</td>
</tr>
</tbody>
</table>
**CMDFILEn**

**Short Description**
Contains a CMDFILE Command Line command to be used to define a command file on an event.

**Syntax**
CMDFILEn=<command file specified using CMDFILE Command Line command>

**Default**
All command files are enabled by default.
The default Startup command file is STARTUP.CMD.
The default Reset command file is RESET.CMD.
The default Preload command file is PRELOAD.CMD.
The default Postload command file is POSTLOAD.CMD.
The default Vppon command file is VPPON.CMD.
The default Vppoff command file is VPPOFF.CMD.

**Description**
The CMDFILEn variable specifies a command file definition using CMDFILE Command Line command. As there are four HI-WAVE command files for the connection, four of those entries should be present.

Those variables are used to store the command files status (enable/disable) and full name specified either with the CMDFILE Command Line command or using the Target Interface Command Files Window.

**Example**
CMDFILE0=CMDFILE STARTUP ON "startup.cmd"
CMDFILE1=CMDFILE RESET ON "reset.cmd"
CMDFILE2=CMDFILE PRELOAD ON "preload.cmd"
CMDFILE3=CMDFILE POSTLOAD ON "postload.cmd"
CMDFILE4=CMDFILE VPPON OFF "vpon.cmd"
CMDFILE5=CMDFILE VPPOFF OFF "vppoff.cmd"
BDIK Connection Demo

Debugging with the BDIK Connection

This section provides an overview of debugging with the BDIK Connection.

With this interface, you can download an executable program from the debugger environment to an external target system based on a Motorola MCU which will execute it. You will also have the feedback of the real target system behaviour to the debugger.

The debugger will fully supervise and monitor the MCU of the target system i.e. control the CPU execution. You can read and write in internal/external memory (even while the CPU is running), single-step/run/stop the CPU, set breakpoints and watchpoints (not all CPUs) in the code.

NOTE  
Unconcerned Components  As the code is executed by an external processor, memory statistics are not available with the BDIK Connection. Profiling, Coverage analysis and I/O simulation are not available with the BDIK Connection.

Starting with the BDIK Connection

1. Link the BDI box to the hardware target with the BDM cable provided by ABATRON.
2. Link the BDI box with a serial cable provided by ABATRON to your computer.
3. Make sure to connect the power supply correctly to the BDI interface and the hardware target, as specified by ABATRON.
4. Use the ABATRON configuration tool to setup the BDI box according to your hardware. Later, map this tool in the debugger itself and call the tool from the debugger. It is more secure to have the BDI interface correctly set according to your target system, as communication settings for the ABATRON configuration tool are similar for the debugger. The latest firmware and the initialization list (startup init list) matching your CPU and hardware must be loaded in the BDI. Refer to the ABATRON User Manual for your CPU derivative procedure. Refer to the BDI Interface Software Setup section for a quick overview of this procedure.
5. Once you have configured the BDI interface with the tool delivered by ABATRON, start the debugger from a demo project for your CPU derivative.
6. The debugger will get the CPU derivative type from the BDI interface and will sets itself automatically to this CPU family. All debugger windows are updated. BDI Ready (then HALTED, sometimes) is displayed in the status bar.
Debugging an Application in RAM

1. Select BDIK | Load... – The Load Executable File dialog is opened.
2. Select the file BDI_*_FIBORAM.ABS and click Open – The dialog is closed and the program is loaded.
3. Select Run | Start/Continue or click – The application is started.
4. Select Run | Halt or click – The execution of the program is stopped.
Book III - HC(S)12(X) Debug Connections - Common Features

Book III Contents

Each section of the Debugger manual includes information to help you become more familiar with the Debugger, to use all its functions and help you understand how to use the environment.

Book 3: HC(S)12(X) Debugger Connections - Common Features

- Chapter 3.1 “HCS12 On-chip DBG Module”
- Chapter 3.2 “HCS12X On-chip DBG Module”
- Chapter 3.3 “Debugging Memory Map”
- Chapter 3.4 “HC(S)12(X) Flash Programming”
HCS12 On-chip DBG Module

The HCS12 derivatives featuring an on-chip DBG module require a debugger graphical user interface to setup this module and take full advantage of this enhanced debugging feature. This manual describes the debugger DBG module user interface.

Within several HCS12 debugger connections (e.g. P&E (PEDebug), HCS12 Serial Monitor and inDART-HCS12, a complete graphical user interface is provided, through a trigger setup dialog box combined with context sensitive popup menus (mouse right-click) in Source, Assembly, Data and Memory component windows to set the on-chip DBG module and triggers.

This DBG module support is automatically enabled or disabled, according to user selected derivative (if the device is user configurable) or automatically through device Part Id.

DBG Features

The debugger covers all features available within the on-chip DBG module:

- Regular hardware breakpoints and watchpoints,
- Predefined preset “Instruction” Triggers, “Memory Access” Triggers or “Capture” Triggers, a wide set of complex hardware breakpoints (triggers on program code instructions) and watchpoints (triggers on device memory access) and data bus recording,
- Expert Triggers, as powerful as predefined preset triggers, “Do It Yourself” way,
- Code program flow rebuild from DBG data capture within the Trace window component (the Trace component should be opened to display the code program flow rebuild),
- Real time program code profiling and coverage within the Profiler and Coverage window components (the Profiler and/or the Coverage components should be opened to display code profiling and code coverage)
Specific Connection Menu Options

Specific DBG support menu options are added to the Connection menu as soon as the debugger target processor is acknowledged by the DBG module. Two additional popup menu entries are displayed: “Trigger Module Settings” and “Bus Trace” in the Connection menu. Shown below is an example with the P&E (PEDebug) connection.

Figure 15.1 Connection Menu - Added DBG Options

Choose “Trigger Module Settings” to open the Trigger Module Settings Window.
Choose “Bus Trace” to open the Trace Component Window.

Popup Menu Entries in Source, Data, Assembly and Memory Windows

Specific DBG support menu options are added to the Connection menu as soon as the debugger target processor is acknowledged by the DBG module.

Source and Assembly Windows

Source and Assembly windows have menu entries to set/delete “Instruction” Triggers A and/or B, a Trigger Settings to set the DBG module Triggers Settings and the Trigger Module Usage to set the DBG module functionality globally.

Setting a trigger, which can be assimilated as a complex breakpoint or watchpoint, is as simple as setting a breakpoint.
Instead of setting a breakpoint, a trigger can be set. Note that only 2 triggers can be set: Trigger A and Trigger B. In a general way, the on-chip DBG module provides combinations of trigger A and trigger B conditions, and according to the number of triggers defined (one or two), different triggers DBG Module Mode Setup can be chosen.

To set a trigger, choosing a Set TriggerAddress entry sets a trigger at the selected source location/address.
The trigger is displayed in the Source window and at the corresponding address in the Assembly window, just like a breakpoint icon. To be distinguishable from breakpoints, the trigger A is marked with a red “A” icon and trigger B with a red “B” icon.
Once a trigger is set, it can be deleted by opening any context sensitive popup menu that contains the *Delete Trigger Address* options.

**Trigger Storing as Markpoints**

Triggers are stored in the debugger as special markpoints. Like breakpoints, markpoints can be viewed on choosing *Show Markpoints...* in the menu.

Triggers are stored as “*Trigger A*” and “*Trigger B*” markpoints. These markpoint names are therefore reserved by the debugger. The markpoint type “*INSTRUCTION*” is automatically selected when the trigger was set from the Source or the Assembly window.
Figure 15.5 Show Markpoints Option

Selecting the Show Markpoints option from the Source window causes the Controlpoints Configuration window to open with its Markpoints tab displayed.
Figure 15.6 Controlpoints Configuration Window - Markpoints Tab

Editing triggers through the Markpoints tab in the Controlpoints Configuration window below is not user friendly. However, the “Save and Restore on load” option (also available with breakpoints and watchpoints) can be very useful to automatically save the application with the DBG module setup and trigger positions saved as they are set in the current application.
Data and Memory Windows

Data and Memory windows have popup menu options to set/delete "Memory Access" Triggers A and/or B, a Trigger Settings option to set the DBG module triggers settings and the Trigger Module Usage option to set globally the DBG module functionality.

Setting a trigger, which can be assimilated as a complex breakpoint or watchpoint, is as simple as setting a watchpoint.

Figure 15.7 Data Window Popup Menu - Set Trigger A Option

In the Data window, instead of setting a watchpoint, a trigger can be set. Note that only 2 triggers can be set: Trigger A and trigger B. In a general way, the on-chip DBG module provides combinations of trigger A and trigger B conditions, and according to the number of triggers defined (one or two), a different trigger DBG Module Mode Setup can be chosen.

To set a trigger, choosing a Set Trigger Address entry and the kind of access - Read, Write, Read/Write - sets a trigger at the selected place.
The trigger is displayed in the Data window and at the corresponding address in the Memory window like a watchpoint icon. To be distinguished from watchpoints, the trigger A is marked with a red dotted vertical line and trigger B with a blue dotted vertical line.

**Expert Triggers**

The Expert Mode has a different set of triggers and trigger designs. Indeed, to completely separate the Expert mode from the Automatic mode, the debugger provides a second set of triggers for the Expert mode. Expert Triggers are independent from the regular triggers described previously.

Context sensitive popup menu entries are slightly different, basically replacing the “Set Trigger Address A” entry by a “Set DBGCA” entry and the “Set Trigger Address B” entry by a “Set DBGCB” entry. The renaming is due to a more physical DBG registers approach in Expert mode and in the Expert Mode Tab.
As shown in the next picture, Expert triggers are displayed in Source and Assembly windows with a small additional “e” character and different colors in the Memory component.

**NOTE** When the Expert mode is set, preset “Instruction” Triggers, “Memory Access” Triggers or “Capture” Triggers designs are grayed out. When the automatic mode is set or a predefined preset trigger is set, the Expert mode trigger designs are grayed out.
As shown above, expert triggers are stored in the Markpoints tab of the Controlpoints Configuration window as “DBGCA” and “DBGCB” markpoints. These markpoint names are therefore reserved by the debugger.

The markpoint type “INSTRUCTION” is automatically selected when the trigger was set from the Source or the Assembly window.

As for regular triggers, the markpoint types “READACCESS”, “WRITEACCESS” or “READWRITEACCESS” are automatically selected when the trigger was set from the Data or the Memory window.
Just as with regular triggers, editing expert triggers through the *Markpoints* tab in the *Controlpoints Configuration* window is not user friendly. However, the “Save and Restore on load” option (also available with breakpoints and watchpoints) can be very useful to automatically save the application with the current DBG module setup and trigger positions just as they were used in the application.

**Trigger Settings**

The Trigger Settings option of a popup menu can be chosen to set all kinds of triggers without opening the Trigger Module Settings Window. However, the amount of trigger types is dynamic, depending on if no triggers are defined, if only Trigger A is defined, if only trigger B is defined if both triggers are defined, and also depending on the trigger type (Instruction, Read Access, Read/Write Access, Write Access. Only possible combinations are displayed.

Also DBG Module Options can be directly edited.

**Figure 15.11  Triggers Setting Menu Option - Extended Menu**
Trigger Module Usage

This menu entry can be used to set globally the DBG module functionality without opening the Trigger Module Settings Window to do the DBG Module Mode Setup.

Figure 15.12 Source Window Extended Menu

DBG Support Status Bar Item

A specific DBG support debugger status bar item is present as soon as the debugger target processor features the DBG module. Clicking on this item opens immediately the Trigger Module Settings Window (future debugger revision only).

Figure 15.13 Status Bar Item

The status bar displays the current DBG Module Mode Setup (as shown above) or the current preset “Instruction” Triggers, “Memory Access” Triggers or “Capture” Triggers used or the current DBG Module Mode Setup.
Trigger Module Settings Window

This window can be opened from context sensitive popup menus in the Source, Data, Memory and Assembly component windows, from the Connection menu and also when clicking on a Status Bar item (future debugger revision only).

The on-chip DBG module can be fully controlled from within this window.

Figure 15.15 Trigger Module Settings Window - Trigger Settings Tab

![Trigger Module Settings Window - Trigger Settings Tab](image-url)
DBG Module Mode Setup

First of all, the on-chip DBG module provides some exclusive debugging features. Open the top drop down list to display all modes and complex breakpoints/watchpoints i.e. kind of triggers available.

Figure 15.16 Trigger Settings Tab Listbox

Automatic Mode (Default)

The DBG Module is used to set up three hardware breakpoints or one watchpoint or to set up triggers selected by the user from the list or from a context sensitive popup menu. This mode is simply the default selection when no triggers have been set yet.

The trigger condition and trigger addresses can be set from the debugger Source, Assembly, Memory and Data component using Set Trigger A or Set Trigger B context sensitive pop up menu entry, or within this dialog.

The DBG module is setup to record the executed change of flows. As no triggers are set, the debugger is stopped on the user request or typical breakpoints/watchpoint. To summerize, in this mode, the DBG module is used to set regular hardware breakpoints and watchpoints.
Expert Mode

The User needs to know the on-chip DBG module really well to use this mode. It can be seen as a “Do It Yourself” way to set the DBG module. The HCS12 core manual is needed, to understand the meaning of the registers and flags.

The triggers comparator addresses can be set from the debugger Source, Assembly, Memory and Data windows using Set DBGCA or Set DBGCB. The DBG module is set by the debugger. DBG module enabling and arming depend on the selected flags set within the DBG register control registers (through the Expert triggers tab property page). The settings are written to the hardware right before the application is run. The DBG module is reset when the application stops.

Figure 15.17 Trigger Settings Tab - Expert Mode Information

To set Expert triggers, the Trigger Module Settings window “Expert Triggers” tab must be used. Select the Expert mode in the drop down list to enable the Expert Mode Tab.
Expert Mode Tab

The expert mode tab gives you an access to most of the on-chip DBG module registers. Trigger types can be directly set from the "DBGT - Debugger Trigger Register" drop down list.

Code program flow rebuild and data recording are also synchronized with the Expert mode and results are displayed in the Trace Component Window.

Profiling and Coverage Mode

Choosing this mode, the DBG module is setup to source code execution profiling and source code execution coverage. The Profiler and/or Coverage components should be opened to display results.

Neither triggers nor DBG based controlpoints can be set in this mode, and the debugger must be stopped on the user request (software breakpoints can still be used).

Profiling and Coverage features are based on a periodical debugger program counter real time fetch from the debugger to the on-chip DBG module. Also this fetch is statistical and cannot cover all program counters and longer is the program running and tests period, more precise would get resulted statistics.
Please see also Limitations section for this mode.

Please refer to the debugger engine manual for Coverage and Profiler component features.

Figure 15.19 Debugger Main Window - Coverage and Profiler Windows

**Disabled Mode**

The User needs to know the on-chip DBG module to use this mode. It can be seen as a “Do It Yourself” way to set hardware breakpoints, watchpoints, and triggers. Please consult the DBG Features section and documents to get all information about the HCS12 on-chip DBG module before attempting to use this mode.

There is no dedicated graphical user interface to access DBG module register. The triggers comparator addresses and DBG control registers are handled by the user through the debugger Memory component or using command line commands. The DBG module is NOT set by the debugger. DBG module enabling and arming depend on the selected flags set within the DBG register control registers. The DBG module is NOT reset when the application stops. By default, the FIFO content is protected from unexpected reads, the DBG module is automatically disarmed and the FIFO is analyzed when the debugger stops. This can be optionally disabled by the user.
“Memory Access” Triggers

Memory Access at Address A
This mode is used to trigger on a program instruction read and/or write at Address A memory location.
The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Memory Access at Address A or Address B
This mode is used to trigger on a program instruction read and/or write at Address A or at Address B memory location.
The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Memory Access Inside Address A - Address B Range
This mode is used to trigger on a program instruction read and/or write inside the Address A - Address B memory range locations.
The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Memory Access at Address A then Memory Access at Address B
This mode is used to trigger on a program instruction sequence first reading and/or writing at Address A memory location then reading and/or writing at Address B memory location.
The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Memory Access at Address A and Value on Data Bus Match
This mode is used to trigger on a program instruction read and/or write of a specific matching byte value at Address A memory location.
When choosing this trigger type, the **trigger B** address is used as a **match value** rather than an address. Also when setting this trigger via a context sensitive popup menu, the following message is displayed if the match value was never set.

**Figure 15.20 Memory Access at Address A and Value on Data Bus Match Dialog Box**

The **Trigger Editing** dialog is not available for the trigger B. Special “**Match value**” edit boxes are displayed instead of **Address B** edit box.

The code program flow rebuild is displayed in the **Trace Component Window** automatically switched to **Instructions Display** mode.

**Memory Access at Address A and Value on Data Bus Mismatch**

This mode is used to trigger on a program instruction read and/or write of a NOT matching byte value at Address A memory location.

When choosing this trigger type, the **trigger B** address is used as a **mismatch value** rather than an address. Also when setting this trigger via a context sensitive popup menu, the following message is displayed if the match value was never set.
Figure 15.21  Memory Access at Address A and Value on Data Bus Mismatch Dialog Box

The Trigger Editing dialog is not available for the trigger B. Special “Match value” edit boxes are displayed instead of Address B edit box.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

“Instruction” Triggers

Instruction at Address A Is Executed

This mode is used to trigger on a program instruction execution (program counter) at Address A.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Instruction at Address A or Address B Is Executed

This mode is used to trigger on a program instruction execution (program counter) at Address A or at Address B.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.
Instruction Execution Inside Address A - Address B Range"

This mode is used to trigger on a program instruction execution (program counter) inside the Address A - Address B range.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Instruction Execution Outside Address A - Address B Range"

This mode is used to trigger on a program instruction execution (program counter) outside the Address A - Address B range.

NOTE IMPORTANT: With the HCS12 Serial Monitor via GDI connection, this trigger type might be interfered with by the monitor code itself and therefore the debugger might break for executed code not belonging to the user application.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Instruction at Address A then at Address B Were Executed

This mode is used to trigger on a program instruction execution (program counter) sequence first at Address A then at Address B.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Instruction at Address A and Value on Data Bus Match

This mode is used to trigger on a program instruction execution (program counter) at Address A, this instruction opcode matching a specific byte value.

When choosing this trigger type, the trigger B address is used as a match value rather than an address. Also when setting this trigger via a context sensitive popup menu, the following message is displayed if the match value was never set.
The Trigger Editing dialog is not available for the trigger B. Special “Match value” edit boxes are displayed instead of Address B edit box.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

**Instruction at Address A and Value on Data Bus Mismatch**

This mode is used to trigger on a program instruction execution (program counter) at Address A, this instruction opcode NOT matching a specific byte value.

When choosing this trigger type, the trigger B address is used as a mismatch value rather than an address. Also when setting this trigger via a context sensitive popup menu, the following message is displayed if the match value was never set.
The Trigger Editing dialog is not available for the trigger B. Special “Match value” edit boxes are displayed instead of Address B edit box.

<table>
<thead>
<tr>
<th>Triggers Address Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address A: 0x0100 Memory Read Access.</td>
</tr>
<tr>
<td>Match value: 00 (hex)</td>
</tr>
</tbody>
</table>

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

“Capture” Triggers

Capture Read/Write Values at Address B

This mode is used to capture the data involved in a read and/or write access to the address specified by the trigger B, such as the address of a particular control register or program variable.

Captured byte data are displayed in the Trace Component Window automatically switched to Recorded Data Display mode.

The trigger address is typically not a program code address (program counter), but rather a data/memory address.

Capture Read/Write Values at Address B After Access at Address A

This mode is used to capture the data involved in a read and/or write access to the addresses specified by the trigger A and the trigger B, such as the address of a particular control register or program variable. Triggering/capture will start only after the trigger A address was accessed.

The trigger addresses is typically not a program code address (program counter), but rather data/memory addresses.

Captured byte data are displayed in the Trace Component Window automatically switched to Recorded Data Display mode.
DBG Module Options

Program Code Change of Flow Recording

The program code change of flow options are available for “Instruction” Triggers and “Memory Access” Triggers and controlled through the Trigger Module Settings window’s Trigger Settings tab list box.

Figure 15.24 Change of Flow Recording Control

- **Record continuously and halt on trigger hit**: The DBG module starts recording program flow information immediately after run. The DBG module halts the processor/debugger on trigger condition match.
- **Record continuously and DO NOT halt on trigger hit**: The DBG module starts recording program flow information immediately after run. The DBG module does not halt the processor/debugger on trigger condition match.
- **Start recording after trigger hit and halt when the fifo is full**: The DBG module starts recording program flow information on trigger condition match and halts the processor/debugger when the capture buffer is full.
**HCS12 On-chip DBG Module**

*Trigger Module Settings Window*

- **Start recording after trigger hit and halt when the fifo is full**: The DBG module starts recording program flow information on trigger condition match. The DBG module does not halt the processor/debugger on trigger condition match.

**When Data Recording**

The data recording options are available for “Capture” Triggers only and are selected from the list box in the Trigger Settings tab of the Trigger Module Settings window.

**Figure 15.25 Data Recording Control**

- **Halt when the fifo is full**: The DBG module records continuously data accesses and halts the processor/debugger when the capture buffer is full.
- **Do not halt when the fifo is full**: The DBG module records continuously data accesses and but does not halt the processor/debugger when the capture buffer is full.
Trigger Editing

Typically trigger addresses and/or type can be set using context sensitive popup menus. It is also possible to modify trigger addresses and type within the Trigger Module Settings Window. Pressing “Modify Trigger” buttons opens a trigger editor dialog box.

Figure 15.26 Browse for Trigger A Dialog Box

In the trigger editor dialog box:

The “Address” edit box contains the initial and final trigger address value. This value can be directly set by typing in the edit box.

The “Type” drop down list should be used to select/change the type of trigger. “Instruction” type should be used for “Instruction” Triggers and “Read”, “Write” and “R/W Access” should be used for “Memory Access” Triggers and “Capture” Triggers.

Pressing “Modify Trigger” in this sub dialog will modify and record the trigger in the trigger database (Trigger Storing as Markpoints).

NOTE Pressing the OK button does NOT update the trigger database. The “Modify Trigger” button in the Trigger Module Settings window must be explicitly pressed before closing the dialog box to update the trigger database.

Pressing “Delete Trigger” in the dialog box removes the trigger in the trigger database (Trigger Storing as Markpoints). This trigger address is then considered as “undefined”.

HCS12 On-chip DBG Module
Trigger Module Settings Window
The “Show Location” button shows the location of the trigger (as program code location or program data) in the Source, Data, Assembly and Memory windows.

The left hand side tree is a user friendly way to find a trigger address in the debugger symbol database by selecting a variable (the address of the variable will be taken and copied in the Address edit box) or a function (the entry point of the function will be taken and copied in the Address edit box), and also regular markpoints (the address of the markpoint will be taken and copied in the Address edit box) from the markpoint list.

Figure 15.27 Finding Trigger Address in Editor Dialog Box
**Trigger Module Settings Window - Display Information**

A large grayed edit box dynamically provides information about the current triggers and selected options.

As context sensitive popup menus will only display triggers matching the amount and the kind of triggers which are currently set, the Trigger Module Settings Window checks dynamically the validity are current triggers set vs. the trigger mode.

As shown below, if one or more triggers do not match the trigger mode selection, a warning icon and message is displayed on the bottom of the dialog.

Here below, the *Memory Write Access* type of trigger selected by the mouse cursor does not match with the *Instruction* Triggers type selected in the drop down list.

**Figure 15.28  Trigger Settings Tab Information**
HCS12 On-chip DBG Module

Trigger Module Settings Window

General Settings Tab

Most of the time, there would be no reason to change any of these settings, which are rather default settings of the DBG user interface. However, in some debug special cases, it is possible to disable some automated debugger background processes.

Figure 15.29  Trigger Module Settings Window - General Settings Tab

- **Automatically analyze the FIFO content**: When the Trace Component Window is open, after the debugger is halted by the user or a breakpoint, watchpoint or a trigger, DBG module results are automatically analysed then displayed in the Trace window. If the Trace window is closed, the DBG user interface does not perform any result analyse except trigger flags reported in the status bar. Unchecking this check box would do the same, with the Trace window open.

- **Disarm automatically the module when the debugger stops**: By default, once the debugger target processor is halted due to user break (not any trigger), the on-chip DGB module is still armed. If this option is selected (by default) the debugger will disarm it to retrieve data from the DBG Fifo. If not selected, the DBG Fifo/buffer information cannot be retrieved until the module is disarmed.

- **Protect DBG FIFO content from unexpected reads**: The DBG Fifo data are retrieved from DBGFH-DBGFL registers (address 0x1814-0x1815 in register block at reset location). Several reads are performed to retrieve the entire shifting buffer.
However, when the debugger is halted, while refreshing Data and Assembly windows, it might read also the debugger target processor memory at the same location, reading the first DBG Fifo data, shifting the buffer, and therefore corrupt the DBG user interface DBG Fifo data retrieving. This option hides to the debugger and also user (see blue “-- --” designs in the Memory window at address 0x1814-0x1815) the DBG Fifo buffer location.

- **When starting, automatically step if a trigger is set at PC address (otherwise: warn):** To run again the application, the debugger usually needs to exit the trigger current match condition and avoid being stuck/halted/locked by the trigger. A single step is usually required to “escape” from “Instruction” Triggers. When this option is disabled, the debugger prompts the following dialog to validate this choice.

![Figure 15.30 Trigger “Escape” Dialog Box](image)

**Trace Component Window**

The Trace component is a debugger generic component used to display in a Trace window a debugger internal database. The context sensitive popup menu is set up by the connection (or the GDI DLL) making usage of the component.

Any debugger connections including the DBG user interface are synchronized with the Trace component.

It is not necessary to open the Trace window/component to make usage of the DBG user interface triggers. However, several triggers are used to collect code program flow information or access data information. The Trace window can be opened from **Specific Connection Menu Options**, from **Popup Menu Entries in Source, Data, Assembly and Memory Windows**, and from the **DBG Support Status Bar Item**. The window can be saved in the debugger layout when pressing the debugger Save icon.

**NOTE** When the Trace component/window is closed, the debugger might be faster, as code program flow rebuild is discarded, this last disassembling back the assembly data from the connection CPU’s memory.
Instructions Display

This display mode is automatically set when “Instruction” Triggers and “Memory Access” Triggers are used. It is also the default display in Automatic Mode (Default).

Displayed columns:

- **Frame**: A number representing an information item stored in the Trace component database.
- **Address**: Instruction program counter.
- **Instruction**: Code program flow instruction disassembly.
- **FIFO Analyse remark**: A debugger information: “DBG FIFO data” means that this data was recorded by the on-chip DBG module. “traced” means an item/instruction obtained by debugger/user singlestep or assembly step. “Program flow rebuild gap” means that the debugger could not track completely the code program flow between two frames.

Figure 15.31 Trace Window - Popup Menu Options

Selecting “Show Location” in the Trace window context sensitive popup menu will simply display in Source and Assembly window the frame matching source and assembly code.
Graphical Display

This display mode can be select when selecting “Graphical” in the Trace window context sensitive popup menu. It provides a graphical representation of the same information.

Figure 15.32 Trace Window - Graphical Display

Textual Display

This display mode can be select when selecting “Textual” in the Trace window context sensitive popup menu, when using “Instruction” Triggers and “Memory Access” Triggers are used. This display mode is rather useless for the DBG user interface, as no read/write accesses are recorded at the same time than program change of flow information by the on-chip DBG module. By consequence, the Textual display mode simply expands instruction assembly code in the Trace window.
Column Display and Moving

Selecting “Items...” in the Trace window context sensitive popup menu opens a small dialog to setup the columns to hide/display in each display mode. The “Displaying mode” drop down list can be opened to make column display modification in Textual, Instructions or Graphical mode.

Figure 15.34 Items Configuration Dialog Box
Dumping Frames to File

Selecting “Dump...” in the Trace window context sensitive popup menu opens a small dialog to dump/save Trace component frames to a text file.

Figure 15.35 Dump Trace Frames Dialog Box

Goto Frame

Selecting “Go to Frame...” in the Trace window context sensitive popup menu opens a small dialog to go to a frame in the Trace window.

Figure 15.36 Search Frames Dialog Box
Clearing Frames

Selecting “Clear” in the Trace window context sensitive popup menu will simply flush the frames in the Trace window (flushing in background the database).

DBG Module FIFO/Buffer Display

Selecting “Display DBG FIFO data” in the Trace window context sensitive popup menu will simply display data information retrieved from the on-chip DGB module Fifo/buffer.
Selecting “Display program flow” in the Trace window context sensitive popup menu will simply turn back to code program flow display.

Displayed columns:

- **FIFO Depth**: A number representing the depth in the DBG/Fifo of the word data value. The first frame (Depth 1) is the oldest value in the time.
- **DBG FIFO Data**: the word value retrieved from the DBG Fifo/buffer from DBGFH and DBGFL DBG on-chip module registers.

Figure 15.37 Trace Window - FIFO Display
Recorded Data Display

This display mode is automatically set when "Capture" Triggers are used. Displayed columns:

- **FIFO Depth**: A number representing the depth in the DBG/Fifo of the byte data value. The first frame (Depth 1) is the oldest value in the time.

- **Data value**: The byte value retrieved from the DBG Fifo/buffer from the DBGFL DBG on-chip module register.

Figure 15.38 Trace Window - Recorded Data Display
Limitations

Demo/unregistered debugger mode

- In demo/unregistered debugger mode, code program reconstruction has a limited number of frames displayed in the Trace window.
- Also Real time code Profiling and code Coverage are disabled.
- No preset/predefined “Instruction” Triggers, “Memory Access” Triggers or “Capture” Triggers are provided. Only Expert Triggers can be set.
HCS12X On-chip DBG Module

The HCS12X derivatives featuring an on-chip DBG module require a debugger graphical user interface to setup this module and take full advantage of this enhanced debugging feature. This manual describes the debugger DBG module user interface.

Within several HC(S)12(X) debugger connections (e.g. P&E (PEdebug), HCS12 Serial Monitor and inDART-HCS12, a complete graphical user interface is provided, through a trigger setup dialog box combined with context sensitive popup menus (mouse right-click) in Source, Assembly, Data and Memory component windows to set the on-chip DBG module and triggers.

This DBG module support is automatically enabled or disabled, according to user selected derivative (if the device is user configurable) or automatically through device Part Id.

DBG Features

The debugger covers all features available within the on-chip DBG module:

- Regular hardware breakpoints and watchpoints,
- Predefined preset “Instruction” Triggers, “Memory Access” Triggers or “Capture” Triggers, a wide set of complex hardware breakpoints (triggers on program code instructions) and watchpoints (triggers on device memory access) and data bus recording,
- Expert Triggers, as powerful as predefined preset triggers, “Do It Yourself” way,
- Code program flow rebuild from DBG data capture within the Trace window component (the Trace component should be opened to display the code program flow rebuild),
- Real time program code profiling and coverage within the Profiler and Coverage window components (the Profiler and/or the Coverage components should be opened to display code profiling and code coverage)
Specific Connection Menu Options

Specific DBG support menu options are added to the Connection menu as soon as the debugger target processor is acknowledged by the DBG module. Two additional popup menu entries are displayed: “Trigger Module Settings” and “Bus Trace” in the Connection menu. Shown below is an example with the P&E (PEdebug) connection.

Figure 16.1 Connection Menu - Added DBG Options

Choose “Trigger Module Settings” to open the Trigger Module Settings Window.
Choose “Bus Trace” to open the Trace Component Window.

Popup Menu Entries in Source, Data, Assembly and Memory Windows

Specific DBG support menu options are added to the Connection menu as soon as the debugger target processor is acknowledged by the DBG module.

Source and Assembly Windows

Source and Assembly windows have menu entries to set/delete “Instruction” Triggers A and/or B, a Trigger Settings to set the DBG module Triggers Settings and the Trigger Module Usage to set the DBG module functionality globally.

Setting a trigger, which can be assimilated as a complex breakpoint or watchpoint, is as simple as setting a breakpoint.
Instead of setting a breakpoint, a trigger can be set. Note that only 2 triggers can be set: **Trigger A** and **Trigger B**. In a general way, the on-chip DBG module provides combinations of trigger A and trigger B conditions, and according to the number of triggers defined (one or two), different triggers DBG Module Mode Setup can be chosen.

To set a trigger, choosing a **Set TriggerAddress** entry sets a trigger at the selected source location/address.
The trigger is displayed in the Source window and at the corresponding address in the Assembly window, just like a breakpoint icon. To be distinguishable from breakpoints, the trigger A is marked with a red “A” icon and trigger B with a red “B” icon.
Once a trigger is set, it can be deleted by opening any context sensitive popup menu that contains the **Delete Trigger Address** options.

### Trigger Storing as Markpoints

Triggers are stored in the debugger as special markpoints. Like breakpoints, markpoints can be viewed on choosing **Show Markpoints...** in the menu.

Triggers are stored as **“Trigger A”** and **“Trigger B”** markpoints. These markpoint names are therefore reserved by the debugger. The markpoint type **“INSTRUCTION”** is automatically selected when the trigger was set from the Source or the Assembly window.
Figure 16.5 Show Markpoints Option

Selecting the Show Markpoints option from the Source window causes the Controlpoints Configuration window to open with its Markpoints tab displayed.
Editing triggers through the *Markpoints* tab in the *Controlpoints Configuration* window below is not user friendly. However, the “*Save and Restore on load*” option (also available with breakpoints and watchpoints) can be very useful to automatically save the application with the DBG module setup and trigger positions saved as they are set in the current application.
Data and Memory Windows

Data and Memory windows have popup menu options to set/delete “Memory Access” Triggers A and/or B, a Trigger Settings option to set the DBG module triggers settings and the Trigger Module Usage option to set globally the DBG module functionality.

Setting a trigger, which can be assimilated as a complex breakpoint or watchpoint, is as simple as setting a watchpoint.

Figure 16.7 Data Window Popup Menu - Set Trigger A Option

In the Data window, instead of setting a watchpoint, a trigger can be set. Note that only 2 triggers can be set: Trigger A and trigger B. In a general way, the on-chip DBG module provides combinations of trigger A and trigger B conditions, and according to the number of triggers defined (one or two), a different trigger DBG Module Mode Setup can be chosen.

To set a trigger, choosing a Set Trigger Address entry and the kind of access - Read, Write, Read/Write - sets a trigger at the selected place.
The trigger is displayed in the Data window and at the corresponding address in the Memory window like a watchpoint icon. To be distinguished from watchpoints, the trigger A is marked with a red dotted vertical line and trigger B with a blue dotted vertical line.

**Expert Triggers**

The Expert Mode has a different set of triggers and trigger designs. Indeed, to completely separate the Expert mode from the Automatic mode, the debugger provides a second set of triggers for the Expert mode. Expert Triggers are independent from the regular triggers described previously.

Context sensitive popup menu entries are slightly different, basically replacing the “Set Trigger Address A” entry by a “Set DBGCA” entry and the “Set Trigger Address B” entry by a “Set DBGCB” entry. The renaming is due to a more physical DBG registers approach in Expert mode and in the Expert Mode Tab.
As shown in the next picture, Expert triggers are displayed in Source and Assembly windows with a small additional “e” character and different colors in the Memory component.

**NOTE** When the Expert mode is set, preset “Instruction” Triggers, “Memory Access” Triggers or “Capture” Triggers designs are grayed out. When the automatic mode is set or a predefined preset trigger is set, the Expert mode trigger designs are grayed out.
As shown above, expert triggers are stored in the Markpoints tab of the Controlpoints Configuration window as "DBGCA" and "DBGCB" markpoints. These markpoint names are therefore reserved by the debugger.

The markpoint type "INSTRUCTION" is automatically selected when the trigger was set from the Source or the Assembly window.

As for regular triggers, the markpoint types "READACCESS", "WRITEACCESS" or "READWRITEACCESS" are automatically selected when the trigger was set from the Data or the Memory window.
Just as with regular triggers, editing expert triggers through the Markpoints tab in the Controlpoints Configuration window is not user friendly. However, the “Save and Restore on load” option (also available with breakpoints and watchpoints) can be very useful to automatically save the application with the current DBG module setup and trigger positions just as they were used in the application.

**Trigger Settings**

The Trigger Settings option of a popup menu can be chosen to set all kinds of triggers without opening the Trigger Module Settings Window. However, the amount of trigger types is dynamic, dependanding if no triggers are defined, if only Trigger A is defined, if only trigger B is defined if both triggers are defined, and also depending on the trigger type (Instruction, Read Access, Read/Write Access, Write Access). Only possible combinations are displayed.

Also DBG Module Options can be directly edited.

**Figure 16.11 Triggers Setting Menu Option - Extended Menu**
Trigger Module Usage

This menu entry can be used to set globally the DBG module functionality without opening the Trigger Module Settings Window to do the DBG Module Mode Setup.

Figure 16.12 Source Window Extended Menu

DBG Support Status Bar Item

A specific DBG support debugger status bar item is present as soon as the debugger target processor features the DBG module. Clicking on this item opens immediately the Trigger Module Settings Window (future debugger revision only).

Figure 16.13 Status Bar Item

The status bar displays the current DBG Module Mode Setup (as shown above) or the current preset “Instruction” Triggers, “Memory Access” Triggers or “Capture” Triggers used or the current DBG Module Mode Setup.
Trigger Module Settings Window

This window can be opened from context sensitive popup menus in the Source, Data, Memory and Assembly component windows, from the Connection menu and also when clicking on a Status Bar item (future debugger revision only).

The on-chip DBG module can be fully controlled from within this window.
DBG Module Mode Setup

First of all, the on-chip DBG module provides some exclusive debugging features. Open the top drop down list to display all modes and complex breakpoints/watchpoints i.e. kind of triggers available.

Figure 16.16 Trigger Settings Tab Listbox

Automatic Mode (Default)

The DBG Module is used to set up three hardware breakpoints or one watchpoint or to set up triggers selected by the user from the list or from a context sensitive popup menu. This mode is simply the default selection when no triggers have been set yet.

The trigger condition and trigger addresses can be set from the debugger Source, Assembly, Memory and Data component using Set Trigger A or Set Trigger B context sensitive pop up menu entry, or within this dialog.

The DBG module is setup to record the executed change of flows. As no triggers are set, the debugger is stopped on the user request or typical breakpoints/watchpoint. To summarize, in this mode, the DBG module is used to set regular hardware breakpoints and watchpoints.
Expert Mode

The User needs to know the on-chip DBG module really well to use this mode. It can be seen as a “Do It Yourself” way to set the DBG module. The HCS12X core manual is needed, to understand the meaning of the registers and flags.

The triggers comparator addresses can be set from the debugger Source, Assembly, Memory and Data windows using Set DBGCA or Set DBGCB. The DBG module is set by the debugger. DBG module enabling and arming depend on the selected flags set within the DBG register control registers (through the Expert triggers tab property page). The settings are written to the hardware right before the application is run. The DBG module is reset when the application stops.

Figure 16.17 Trigger Setings Tab - Expert Mode Information

To set Expert triggers, the Trigger Module Settings window “ExpertTriggers” tab must be used. Select the Expert mode in the drop down list to enable the Expert Mode Tab.
Expert Mode Tab

The expert mode tab gives you an access to most of the on-chip DBG module registers. Trigger types can be directly set from the “DBGT - Debugger Trigger Register” drop down list.

Code program flow rebuild and data recording are also synchronized with the Expert mode and results are displayed in the Trace Component Window.

![Trigger Module Settings Window - Expert Triggers Tab](image)

Profiling and Coverage Mode

Choosing this mode, the DBG module is setup to source code execution profiling and source code execution coverage. The Profiler and/or Coverage components should be opened to display results.

Neither triggers nor DBG based controlpoints can be set in this mode, and the debugger must be stopped on the user request (software breakpoints can still be used).

Profiling and Coverage features are based on a periodical debugger program counter real time fetch from the debugger to the on-chip DBG module. Also this fetch is statistical and cannot cover all program counters and longer is the program running and tests period, more precise would get resulted statistics.
Please see also Limitations section for this mode.

Please refer to the debugger engine manual for Coverage and Profiler component features.

**Figure 16.19 Debugger Main Window - Coverage and Profiler Windows**

### Disabled Mode

The User needs to know the on-chip DBG module to use this mode. It can be seen as a “Do It Yourself” way to set hardware breakpoints, watchpoints, and triggers. Please consult the DBG Features section and documents to get all information about the HCS12X on-chip DBG module before attempting to use this mode.

There is no dedicated graphical user interface to access DBG module register. The triggers comparator addresses and DBG control registers are handled by the user through the debugger Memory component or using command line commands. The DBG module is NOT set by the debugger. DBG module enabling and arming depend on the selected flags set within the DBG register control registers. The DBG module is NOT reset when the application stops. By default, the FIFO content is protected from unexpected reads, the DBG module is automatically disarmed and the FIFO is analyzed when the debugger stops. This can be optionally disabled by the user.
“Memory Access” Triggers

Memory Access at Address A
This mode is used to trigger on a program instruction read and/or write at Address A memory location.
The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Memory Access at Address A or Address B
This mode is used to trigger on a program instruction read and/or write at Address A or at Address B memory location.
The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Memory Access Inside Address A - Address B Range
This mode is used to trigger on a program instruction read and/or write inside the Address A - Address B memory range locations.
The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Memory Access at Address A then Memory Access at Address B
This mode is used to trigger on a program instruction sequence first reading and/or writing at Address A memory location then reading and/or writing at Address B memory location.
The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Memory Access at Address A and Value on Data Bus Match
This mode is used to trigger on a program instruction read and/or write of a specific matching byte value at Address A memory location.
When choosing this trigger type, the trigger B address is used as a match value rather than an address. Also when setting this trigger via a context sensitive popup menu, the following message is displayed if the match value was never set.

**Memory Access at Address A and Value on Data Bus Mismatch**

This mode is used to trigger on a program instruction read and/or write of a NOT matching byte value at Address A memory location.

When choosing this trigger type, the trigger B address is used as a mismatch value rather than an address. Also when setting this trigger via a context sensitive popup menu, the following message is displayed if the match value was never set.
The Trigger Editing dialog is not available for the trigger B. Special "Match value" edit boxes are displayed instead of Address B edit box.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

**“Instruction” Triggers**

**Instruction at Address A Is Executed**

This mode is used to trigger on a program instruction execution (program counter) at Address A.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

**Instruction at Address A or Address B Is Executed**

This mode is used to trigger on a program instruction execution (program counter) at Address A or at Address B.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.
Instruction Execution Inside Address A - Address B Range"

This mode is used to trigger on a program instruction execution (program counter) inside the Address A - Address B range.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Instruction Execution Outside Address A - Address B Range"

This mode is used to trigger on a program instruction execution (program counter) outside the Address A - Address B range.

NOTE IMPORTANT: With the HCS12X Serial Monitor via GDI connection, this trigger type might be interfered with by the monitor code itself and therefore the debugger might break for executed code not belonging to the user application.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Instruction at Address A then at Address B Were Executed

This mode is used to trigger on a program instruction execution (program counter) sequence first at Address A then at Address B.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Instruction at Address A and Value on Data Bus Match

This mode is used to trigger on a program instruction execution (program counter) at Address A, this instruction opcode matching a specific byte value.

When choosing this trigger type, the trigger B address is used as a match value rather than an address. Also when setting this trigger via a context sensitive popup menu, the following message is displayed if the match value was never set.
The Trigger Editing dialog is not available for the trigger B. Special "Match value" edit boxes are displayed instead of Address B edit box.

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

Instruction at Address A and Value on Data Bus Mismatch

This mode is used to trigger on a program instruction execution (program counter) at Address A, this instruction opcode NOT matching a specific byte value.

When choosing this trigger type, the trigger B address is used as a mismatch value rather than an address. Also when setting this trigger via a context sensitive popup menu, the following message is displayed if the match value was never set.
The Trigger Editing dialog is not available for the trigger B. Special “Match value” edit boxes are displayed instead of Address B edit box.

<table>
<thead>
<tr>
<th>Triggers Address Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address A: 0x0100 Memory Read Access. Modify Trigger A...</td>
</tr>
<tr>
<td>Match value: 00 (hex)</td>
</tr>
</tbody>
</table>

The code program flow rebuild is displayed in the Trace Component Window automatically switched to Instructions Display mode.

“Capture” Triggers

Capture Read/Write Values at Address B

This mode is used to capture the data involved in a read and/or write access to the address specified by the trigger B, such as the address of a particular control register or program variable.

Captured byte data are displayed in the Trace Component Window automatically switched to Recorded Data Display mode.

The trigger address is typically not a program code address (program counter), but rather a data/memory address.

Capture Read/Write Values at Address B After Access at Address A

This mode is used to capture the data involved in a read and/or write access to the addresses specified by the trigger A and the trigger B, such as the address of a particular control register or program variable. Triggering/capture will start only after the trigger A address was accessed.

The trigger addresses is typically not a program code address (program counter), but rather data/memory addresses.

Captured byte data are displayed in the Trace Component Window automatically switched to Recorded Data Display mode.
DBG Module Options

Program Code Change of Flow Recording

The program code change of flow options are available for "Instruction" Triggers and "Memory Access" Triggers and controlled through the Trigger Module Settings window’s Trigger Settings tab list box.

Figure 16.24 Change of Flow Recording Control

- **Record continuously and halt on trigger hit**: The DBG module starts recording program flow information immediately after run. The DBG module halts the processor/debugger on trigger condition match.
- **Record continuously and DO NOT halt on trigger hit**: The DBG module starts recording program flow information immediately after run. The DBG module does not halt the processor/debugger on trigger condition match.
- **Start recording after trigger hit and halt when the fifo is full**: The DBG module starts recording program flow information on trigger condition match and halts the processor/debugger when the capture buffer is full.
• **Start recording after trigger hit and halt when the fifo is full**: The DBG module starts recording program flow information on trigger condition match. The DBG module does not halt the processor/debugger on trigger condition match.

### When Data Recording

The data recording options are available for **“Capture” Triggers** only and are selected from the list box in the Trigger Settings tab of the Trigger Module Settings window.

**Figure 16.25 Data Recording Control**

- **Halt when the fifo is full**: The DBG module records continuously data accesses and halts the processor/debugger when the capture buffer is full.
- **Do not halt when the fifo is full**: The DBG module records continuously data accesses and but does not halt the processor/debugger when the capture buffer is full.
Trigger Editing

Typically trigger addresses and/or type can be set using context sensitive popup menus. It is also possible to modify trigger addresses and type within the Trigger Module Settings Window. Pressing “Modify Trigger” buttons opens a trigger editor dialog box.

Figure 16.26 Browse for Trigger A Dialog Box

In the trigger editor dialog box:

The “Address” edit box contains the initial and final trigger address value. This value can be directly set by typing in the edit box.

The “Type” drop down list should be used to select/change the type of trigger. “Instruction” type should be used for “Instruction” Triggers and “Read”, “Write” and “R/W Access” should be used for “Memory Access” Triggers and “Capture” Triggers.

Pressing “Modify Trigger” in this sub dialog will modify and record the trigger in the trigger database (Trigger Storing as Markpoints).

NOTE Pressing the OK button does NOT update the trigger database. The “Modify Trigger” button in the Trigger Module Settings window must be explicitly pressed before closing the dialog box to update the trigger database.

Pressing “Delete Trigger” in the dialog box removes the trigger in the trigger database (Trigger Storing as Markpoints). This trigger address is then considered as “undefined”.
The “Show Location” button shows the location of the trigger (as program code location or program data) in the Source, Data, Assembly and Memory windows.

The left hand side tree is a user friendly way to find a trigger address in the debugger symbol database by selecting a variable (the address of the variable will be taken and copied in the Address edit box) or a function (the entry point of the function will be taken and copied in the Address edit box), and also regular markpoints (the address of the markpoint will be taken and copied in the Address edit box) from the markpoint list.

**Figure 16.27 Finding Trigger Address in Editor Dialog Box**
Trigger Module Settings Window - Display Information

A large grayed edit box dynamically provides information about the current triggers and selected options.

As context sensitive popup menus will only display triggers matching the amount and the kind of triggers which are currently set, the Trigger Module Settings Window checks dynamically the validity are current triggers set vs. the trigger mode.

As shown below, if one or more triggers do not match the trigger mode selection, a warning icon and message is displayed on the bottom of the dialog.

Here below, the Memory Write Access type of trigger selected by the mouse cursor does not match with the Instruction Triggers type selected in the drop down list.

Figure 16.28 Trigger Settings Tab Information
General Settings Tab

Most of the time, there would be no reason to change any of these settings, which are rather default settings of the DBG user interface. However, in some debug special cases, it is possible to disable some automated debugger background processes.

Figure 16.29 Trigger Module Settings Window - General Settings Tab

- **Automatically analyze the FIFO content**: When the Trace Component Window is open, after the debugger is halted by the user or a breakpoint, watchpoint or a trigger, DBG module results are automatically analysed then displayed in the Trace window. If the Trace window is closed, the DBG user interface does not perform any result analyse except trigger flags reported in the status bar. Unchecking this check box would do the same, with the Trace window open.

- **Disarm automatically the module when the debugger stops**: By default, once the debugger target processor is halted due to user break (not any trigger), the on-chip DGB module is still armed. If this option is selected (by default) the debugger will disarm it to retrieve data from the DBG Fifo. If not selected, the DBG Fifo/buffer information cannot be retrieved until the module is disarmed.

- **Protect DBG FIFO content from unexpected reads**: The DBG Fifo data are retrieved from DBGFH-DBGFL registers (address 0x1814-0x1815 in register block at reset location). Several reads are performed to retrieve the entire shifting buffer.
However, when the debugger is halted, while refreshing Data and Assembly windows, it might read also the debugger target processor memory at the same location, reading the first DBG Fifo data, shifting the buffer, and therefore corrupt the DBG user interface DBG Fifo data retrieving. This option hides to the debugger and also user (see blue “-- --” designs in the Memory window at address 0x1814-0x1815) the DBG Fifo buffer location.

- **When starting, automatically step if a trigger is set at PC address (otherwise: warn):** To run again the application, the debugger usually needs to exit the trigger current match condition and avoid being stuck/halted/locked by the trigger. A single step is usually required to “escape” from “Instruction” Triggers. When this option is disabled, the debugger prompts the following dialog to validate this choice.

**Figure 16.30 Trigger “Escape” Dialog Box**

<table>
<thead>
<tr>
<th>HI-WAVE</th>
<th><img src="image" alt="Trigger “Escape” Dialog Box" /></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Trigger “Escape” Dialog Box" /></td>
<td></td>
</tr>
</tbody>
</table>

**Trace Component Window**

The Trace component is a debugger generic component used to display in a Trace window a debugger internal database. The context sensitive popup menu is set up by the connection (or the GDI DLL) making usage of the component.

Any debugger connections including the DBG user interface are synchronized with the Trace component.

It is not necessary to open the Trace window/component to make usage of the DBG user interface triggers. However, several triggers are used to collect code program flow information or access data information. The Trace window can be opened from **Specific Connection Menu Options**, from **Popup Menu Entries in Source, Data, Assembly and Memory Windows**, and from the **DBG Support Status Bar Item**. The window can by saved in the debugger layout when pressing the debugger Save icon.

**NOTE** When the Trace component/window is closed, the debugger might be faster, as code program flow rebuild is discarded, this last disassembling back the assembly data from the connection CPU’s memory.
Instructions Display

This display mode is automatically set when “Instruction” Triggers and “Memory Access” Triggers are used. It is also the default display in Automatic Mode (Default).

Displayed columns:

- **Frame**: A number representing an information item stored in the Trace component database.
- **Address**: Instruction program counter.
- **Instruction**: Code program flow instruction disassembly.
- **FIFO Analyse remark**: A debugger information: “DBG FIFO data” means that this data was recorded by the on-chip DBG module. “traced” means an item/instruction obtained by debugger/user singlestep or assembly step. “Program flow rebuild gap” means that the debugger could not track completely the code program flow between two frames.

Figure 16.31 Trace Window - Popup Menu Options

Selecting “Show Location” in the Trace window context sensitive popup menu will simply display in Source and Assembly window the frame matching source and assembly code.
Graphical Display

This display mode can be select when selecting “Graphical” in the Trace window context sensitive popup menu. It provides a graphical representation of the same information.

![Figure 16.32 Trace Window - Graphical Display](image)

Textual Display

This display mode can be select when selecting “Textual” in the Trace window context sensitive popup menu, when using “Instruction” Triggers and “Memory Access” Triggers are used. This display mode is rather useless for the DBG user interface, as no read/write accesses are recorded at the same time than program change of flow information by the on-chip DBG module. By consequence, the Textual display mode simply expands instruction assembly code in the Trace window.
Column Display and Moving

Selecting "Items..." in the Trace window context sensitive popup menu opens a small dialog to setup the columns to hide/display in each display mode. The "Displaying mode" drop down list can be opened to make column display modification in Textual, Instructions or Graphical mode.

Figure 16.34 Items Configuration Dialog Box
Dumping Frames to File

Selecting “Dump...” in the Trace window context sensitive popup menu opens a small dialog to dump/save Trace component frames to a text file.

Figure 16.35 Dump Trace Frames Dialog Box

Goto Frame

Selecting “Go to Frame...” in the Trace window context sensitive popup menu opens a small dialog to go to a frame in the Trace window.

Figure 16.36 Search Frames Dialog Box
Clearing Frames

Selecting “Clear” in the Trace window context sensitive popup menu will simply flush the frames in the Trace window (flushing in background the database).

DBG Module FIFO/Buffer Display

Selecting “Display DBG FIFO data” in the Trace window contest sensitive popup menu will simply display data information retrieved from the on-chip DGB module Fifo/buffer. Selecting “Display program flow” in the Trace window context sensitive popup menu will simply turn back to code program flow display.

Displayed columns:

* **FIFO Depth**: A number representing the depth in the DBG/Fifo of the word data value. The first frame (Depth 1) is the oldest value in the time.

* **DBG FIFO Data**: the word value retrieved from the DBG Fifo/buffer from DBGFH and DBGFL DBG on-chip module registers.

Figure 16.37 Trace Window - FIFO Display
Recorded Data Display

This display mode is automatically set when “Capture” Triggers are used.

Displayed columns:

- **FIFO Depth**: A number representing the depth in the DBG/Fifo of the byte data value. The first frame (Depth 1) is the oldest value in the time.
- **Data value**: the byte value retrieved from the DBG Fifo/buffer from the DBGFL DBG on-chip module register.

![Figure 16.38 Trace Window - Recorded Data Display](image)

<table>
<thead>
<tr>
<th>FIFO Depth</th>
<th>Data value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
</tr>
<tr>
<td>1</td>
<td>04</td>
</tr>
<tr>
<td>2</td>
<td>05</td>
</tr>
<tr>
<td>3</td>
<td>06</td>
</tr>
<tr>
<td>4</td>
<td>07</td>
</tr>
<tr>
<td>5</td>
<td>08</td>
</tr>
<tr>
<td>6</td>
<td>09</td>
</tr>
<tr>
<td>?</td>
<td>0A</td>
</tr>
</tbody>
</table>
Limitations

**Demo/unregistered debugger mode**

- In demo/unregistered debugger mode, code program reconstruction has a limited number of frames displayed in the Trace window.
- Also Real time code Profiling and code Coverage are disabled.
- No preset/predefined “Instruction” Triggers, “Memory Access” Triggers or “Capture” Triggers are provided. Only Expert Triggers can be set.
Debugging Memory Map

Introduction

The Debugging Memory Map (DMM) is a software “Manager” caring for all debugger accesses to device/chip memory and also caring for memory data caching.

The DMM provides a global approach for all different CPU families/cores, each family having its own method for memory access and its own memory on-chip layout and memory address range priorities.

NOTE The Debugging Memory Map Manager and DMM user interface replaces the Legacy "Banked Memory Location" dialog ("Set Bank..." menu entry). By the way, the "BANKWINDOW" command has been completely removed, as banks handling is now transparently handled by the Debugging Memory Map Manager. In case of error due to this command execution, please remove/comment this obsolete command. In case of any further debugging problem due to this command removal, please contact the support team.

The DMM gets all memory read and write calls from the debugger. On the other side, the DMM has the very low level function read/write primitives to call third party cable drivers of BDM pods, Monitors, etc.

For each CPU core, the debugger provides the DMM with core specific read/write access methods that are called "Types" within the DMM GUI (Graphical User Interface), and core specific priority rules that are called "Priority" within the DMM GUI.

Indeed, the DMM has a GUI, therefore providing to the user a way to change anytime the way to access the memory.

Debugging Memory Map GUI

The graphical user interface is flexible enough to be handled without much difficulty, and live diagnostic is displayed within the dialog. Anytime, it is possible to revert to default (factory) setup, and most of the time, the user does not even need to edit/change settings within the DMM GUI.

The DMM GUI can be opened by choosing the “Debugging Memory Map...” option in the connection menu in the debugger main window. This opens the DMM Window.

Figure 17.1  Debugging Memory Map Window

The DMM GUI shows a list of memory address ranges also called in this manual “Modules” defined to access the device/chip memory.

In the “Type” column, the type of memory for the defined memory address range given in the “Range” column. The “Active” column indicates whether the defined range is active/mapped by the DMM. If “No”, the DMM considers the range is if it was not defined at all.

NOTE  Note that all which is NOT defined is considered by the DMM as NOT accessible/not implemented. The debugger will display some “--” in the Memory window in that case, and for sure, the DMM will NEVER try/attempt to read or write unimplemented memory.

The “Comment” column contains a text information comment about the defined memory address range.
The “Information” scrollable window gives a general diagnostic of the DMM. This diagnostic has less information than the edition mode diagnostic.

Pressing the "New" button will open the edition dialog box to create a new memory address range.

Pressing the "Modify" button opens the edition dialog of the selected memory address range to modify it. More memory range information is displayed in the edition dialog, and an enhanced diagnostic is also displayed.

Pressing the "Delete" button will lead to memory range removal, after a warning dialog.

Pressing the "Revert to default" button will remove (after a warning dialog) the current setup (usually saved in the current project) and retrieve the default (factory) setup from an internal database.

**Edition Dialog Box**

*Figure 17.2 Memory Address Range - Edition Dialog Box*

The "Enable memory module" option checkbox maps the module/memory range in the debugger. Unchecking this option makes the module completely "transparent" for the DMM and the debugger.

The “Start” edit box contains the first address of a memory range and the “End” edit box contains the last address of a memory range.

Range boundaries are always limited to an overlapped range with a bigger priority. For example, if 2 bytes have been defined in a range which overlaps another range, accessing
these 2 bytes will be performed using the “type” and rules of this 2-byte range. The
memory on both sides of these 2 bytes will be accessed using the “type” and rules of the
overlapped range.

**NOTE** The "Start" "End" range is a range address for a “Type” and for a “Priority”.
Internally, ranges can overlap only if they are of the same type and of the same
priority. The debugger will always read with rules of the range with the highest
priority.

---

**Types**

The “Type” drop down list provides all kinds of memory type available for the processor
displayed in the title bar of the dialog. For some connections, the CPU core might be
displayed instead of the processor name.

Types are internally rules to read and write a kind of memory. For examples, the HCS12
“banked” type requires to set first a register called PPAGE to read the memory, then to
restore this value as it was before reading. Also this "banked" type does not physically
provide a memory access while running. "Memory access while running" is possible in
physical memory (ram, registers).

**Figure 17.3 Edition Dialog Box - Type Dropdown List**
A minor fiction has been created to simplify the GUI, "read protected", “write protected” and “r/w protected” types should be outside the Type selection, as individual options. Indeed, these are not memory types. These “protection” types are based on a “physical” type, but the goal was to provide to the user I/O Registers protections that are typically accessed as “physical” memory.

NOTE  CPU core specific memory “types” and “Priorities” are listed at the end of this manual section.

Priorities

The "Priority" drop down list provides all of the memory overlap priority available for a type of processor core. The debugger can have a higher priority ("highest debugger") to setup an upper address range that can overlap an on-chip address range, this to make a debugger display filter (for a Memory window), e.g. when creating an "no read access while running" memory address range.

A “Flat” memory architecture i.e. without memory blocks moving feature would provide the following Priority drop down list (e.g. HC12, HCS12, HCS12X cores):

![Figure 17.4 HC12, HCS12 or HCS12X Core Types](image)

The default is the way the CPU sees the different memory onchip blocks, all memory block with the same priority.

Memory Read Caching

The "Refresh memory when halting" option controls the debugger memory cache. When this option is checked, internal image/cache of memory data are always deleted and the data is always retrieved from hardware when required by the debugger. When unchecked (usually by default for Non Volatile Memory areas), the DMM keeps a copy of the data and does not read/retrieve the data from hardware until next application loading/programming.

NOTE  Each declared memory address range in the GUI has its own private code cache monitored by the DMM.

The “DMM CACHINGOFF” command can fully disable the caching feature for the entire DMM, i.e. for all defined memory ranges. The “DMM CACHINGON” command re-enables the caching feature.
Access While Running

The “no memory access while running” option can be used to discard debugger access to a memory range which could typically be accessed while running. This feature is useful to protect onchip I/O Register flags from being triggered by debugger memory reads due to display refreshes.

Remarks

It is possible to create as many memory ranges as desired, down to a single byte.
Deleting Default/Factory ranges generates warning dialogs. Indeed, some settings are required for the debugger to debug and removing ranges would lead to erroneous debugging information.
All GUI settings can be done by debugger commands.
Settings and DMM changes are saved in the current user project. The user can always restart from draft pressing the “Revert to Default” button.
Automatic DMM range remapping can be disabled by a debugger command.
The default settings are retrieved from a complete database describing each derivative, or, in some cases, describing the CPU core (when not necessary to go to derivative level).

CPU Core Types and Priorities

HC12(CPU12) Core

Priorities:

MC68HC812A4 derivative

- "highest (debugger)”: a high debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read.
- "internal register space”: please refer to MC68HC812A4 specifications.
- "RAM memory block”: please refer to MC68HC812A4 specifications.
- "EEPROM memory block”: please refer to MC68HC812A4 specifications.
- "E space (external)”: please refer to MC68HC812A4 specifications.
- "CS space (external)”: please refer to MC68HC812A4 specifications.
- "P space (external)”: please refer to MC68HC812A4 specifications.
- "D space (external)”: please refer to MC68HC812A4 specifications.
• "remaining external space": please refer to MC68HC812A4 specifications.
• "lowest (debugger)"; a low debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read. This priority is of poor usage but can still be used for display purposes on chip unimplemented memory range.

All Others
• "highest (debugger)"; a high debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read.
• "internal register space": please refer to device specifications.
• "RAM memory block": please refer to device specifications.
• "EEPROM memory block": please refer to device specifications.
• "onchip flash-EEPROM": please refer to device specifications.
• "remaining external space": please refer to device specifications.
• "lowest (debugger)"; a low debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read. This priority is of poor usage but can still be used for display purposes on chip unimplemented memory range.

Types:
• "read protected": this sets the memory as “read protected”. The DMM never tries to read this range. Writing in this range is done as “physical” memory type.
• "write protected": this sets the memory as “write protected”. The DMM stops all debugger writes to this range. Reading in this range is done as “physical” memory type.
• "r/w protected": this sets the memory as “read and write protected”. Reading and writing to this memory range is fully blocked by the DMM.
• "physical": this sets the memory range as physical, i.e. with linear 16-bit address bus access as performed by the CPU when reading and writing the onchip memory.

additionally, for MC68HC812A4 derivative:
• "extra banked": this type handles the EPAGE register when accessing the Extra page banked data, typically data in $400-$7FF window.
• "banked": this type handles the PPAGE register when accessing the Program page banked data, typically program code in $8000-$BFFF address range window.
**Debugging Memory Map**

**CPU Core Types and Priorities**

- **data banked**: this type handles the DPAGE register when accessing the Data page banked data, typically variables in $7000-$7FFF address range window.

**additionally, for MC68HC912xx128 derivatives:**
- **banked**: this type handles the PPAGE register when accessing the Program page banked data, typically program code in onchip Flash in $8000-$BFFF address range window.

**HCS12 Core**

The HCS12 core provides memory block moving, with overlap priorities. These overlap rules are handled by the DMM, and rules handle the Memory Expansion Registers (MER), i.e: INITRM, INITRG, INITEE.

On each debugger halt, the MER Registers are read, and if necessary, the DMM offsets internally range addresses.

**NOTE**  The debugger does not poll the MER registers while running. Also the remapping is performed only on factory defined memory range, not on user defined memory ranges.

The “DMM HCS12MERHANDLINGOFF” command can be executed to disable the MER Registers tracking. The “DMM HCS12MERHANDLINGON” command can be executed to re-engage this feature.

**NOTE**  By factory/default setup, HCS12 DBG12 Fifo Registers have been protected to reserve the DBG12 Fifo Reading for the debugger DBG interface. Removing this protection leads to incorrect program flow rebuild.

**Priorities:**

- **highest (debugger)**: a high debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read.
- **internal register space**: please refer to device specifications.
- **RAM memory block**: please refer to device specifications.
- **EEPROM memory block**: please refer to device specifications.
- **onchip flash-EEPROM**: please refer to device specifications.
- **remaining external space**: please refer to device specifications.
- **lowest (debugger)**: a low debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read. This priority is
of poor usage but can still be used for display purposes on chip unimplemented memory range.

Types:

- **"read protected"**: this sets the memory as “read protected”. The DMM never tries to read this range. Writing in this range is done as “physical” memory type.
- **"write protected"**: this sets the memory as “write protected”. The DMM stops all debugger writes to this range. Reading in this range is done as “physical” memory type.
- **"r/w protected"**: this sets the memory as “read and write protected”. Reading and writing to this memory range is fully blocked by the DMM.
- **"physical"**: this sets the memory range as physical, i.e. with linear 16-bit address bus access as performed by the CPU when reading and writing the onchip memory.
- **"banked"**: this type handles the PPAGE register when accessing the Program page banked data, typically program code in onchip Flash in $8000-$BFFF address range window.
- **"Registers"**: This type cares of the I/O Registers block and Memory Expansion Registers changes, including I/O Registers block moving.

**HCS12X Core**

Priorities:

- **"highest (debugger)"**: a high debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read.
- **"default (device)"**: default CPU visibility of the entire device/memory with a same priority, as no memory range can be moved to overlap another memory range.
- **"lowest (debugger)"**: a low debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read. This priority is of poor usage but can still be used for display purposes on chip unimplemented memory range.

Types:

- **"read protected"**: this sets the memory as “read protected”. The DMM never tries to read this range. Writing in this range is done as “physical” memory type.
Debugging Memory Map

CPU Core Types and Priorities

- **"write protected"**: this sets the memory as “write protected”. The DMM stops all debugger writes to this range. Reading in this range is done as “physical” memory type.
- **"r/w protected"**: this sets the memory as “read and write protected”. Reading and writing to this memory range is fully blocked by the DMM.
- **"physical"**: this sets the memory range as physical, i.e. with linear 16-bit address bus access as performed by the CPU when reading and writing the onchip memory.
- **"banked"**: this type handles the PPAGE register when accessing the Program page banked data, typically program code in onchip Flash in $8000-$BFFF address range window.
- **"ram banked"**: this type covers accessing $1000-$1FFF ram data window (the user application accesses via RPAGE) in global address space. Important: All accesses are casted by the DMM to global memory which should therefore be defined for the matching range.
- **"eeprom banked"**: this type covers accessing $800-$BFF eeprom data window (the user application accesses via EPAGE) in global address space. Important: All accesses are casted by the DMM to global memory which should therefore be defined for the matching range.
- **"global"**: this type covers accessing of the global memory space via BDM GPAGE register (Global address space). The Memory window with “Address Space” setup to “Global” displays the global space memory of the device.
- **"xgate"**: this type covers accessing of the XGATE memory space as the XGATE core would “see” it. The Memory window with “Address Space” setup to “XGATE” displays the XGATE space memory of the device. When existing, the Flash/Ram XGATE memory split is internally evaluated by the DMM.

**NOTE**

By factory/default setup, HCS12X DBG12X Fifo Registers have been protected to reserve the DBG12X Fifo reading for the debugger DBG interface. Removing this protection leads to incorrect program flow rebuild.

Except “physical” and “protected” access types, all types are routed to Global memory when reading from the device. However, for Non Volatile Memory programming reasons, “eeprom banked” and “banked” types are routed to logical paged when writing to the device.

**DMM Commands**

All DMM GUI settings can be done by debugger command line commands. these commands are explained in the Debugger Connection-specific Commands chapter of this manual.
HC(S)12(X) Flash Programming

Non-volatile Memory Control Utility
Introduction

Writing to Flash modules, eeproms, or other non-volatile memory modules in modern MCUs requires special algorithms from microprocessor designers. Before you write to Flash devices, you must erase them. Many Flash devices need initialization to become accessible; some devices may need write protection removed.

This manual explains The Non-volatile Memory Control (NVMC) utility, an extension component that lets you control the on-chip Flash devices for all Debugger connections. As it supports many MCUs and Flash modules, the NVMC utility is very flexible. This flexibility comes from a generic Debugger component, which calls a graphical user interface, then loads an MCU-specific module. The module provides the appropriate information (such as structure, access algorithms, and location) for that MCU.

The NVMC utility lists all non-volatile memory devices, indicating their structure, state, and location. You can change the state (enabled/disabled, blank, programmed, protected/encrypted) and program data into the modules.

Automated Application Programming

The debugger is able to program an application without making usage of the NVMC dialog/GUI, which remains useful for specific operations only. Currently, CodeWarrior “wizard-created” projects might be programmed/“flashed” immediately. The debugger prompts by default a warning dialog to get a user acceptance before mass erasing then programming the application.
Figure 18.1 Flash Programming Loader Warning Dialog Box

Pressing the OK button launches backgrounded flash commands described later in this section to arm programming, load/program an application file, then disarm programming.

Checking the “Do not display...” checkbox removes the Warning message definitely for the current project (saved in project under the project variable: AEFWarningDialog=FALSE). 

Setup

The Open and Load Code (Executable File) dialog box is opened when you choose the “Load...” menu entry in the debugger main window’s connection menu.

Figure 18.2 Open and Load Code Options Dialog Box

Checking the above option engages the automated device mass erasing and application programming into non volatile memory, i.e. Flash and/or Eeprom.

To set this option permanently, the setup should be performed in the debugger Preferences window’s Load tab (“File” menu, “Configuration...” entry).

Figure 18.3 Preferences Window - Load Tab
Advanced Options: Preventing Erasing

Pressing the “Advanced...” button in the “Load” tab of the debugger Preferences window opens the NVM Programming Selection list box.

Figure 18.4 NVM Programming Selection List Box

The list box lists all the Non Volatile Memory modules registered by the debugger for the current selected processor device.

Clicking once on a line selects an item (highlighted in blue) and clicking another time on a selected item unselects it.

Erasing is skipped for all selected modules. If all modules are selected, the debugger will simply program the application without erasing at all any non volatile memory the device.

CAUTION The debugger cannot care about pre-programmed modules and the user must care about reprogramming limitations, risks and impossibility.

The NVM Programming Selection list box does not give many details about the listed blocks. More information can be displayed when typing the “FLASH” command in the Command window, or when opening the “Non Volatile Memory Control” dialog box.

The “NVM Programming Selection” list box is tightly associated to the “FLASH AEFSKIPERASING” command of the debugger.

TIP When using this feature, make sure to also select modules that could cover/include all other modules listed, modules usually called PAGED, EEPROM, ALL_PPAGES, ALL_EPAGES, ALL_xxx, etc.
The NVMC utility is integrated into the Debugger, as an extension of certain debugger connections. If the NVMC utility is available, your connection menu includes a Flash... selection, as shown below.

### Modules and Module States

In following sections, the expression *available modules* means all the FLASH or EEPROM on-chip modules that the NVMC dialog box lists. The module definitions track with the CPU derivative technical summary and special technical considerations. If an onchip module consists of several independent blocks, the NVMC dialog box might list all of these blocks, however, it would typically group the entire non volatile onchip blocks under one single listed module and separate relevant and important non volatile memory blocks (like mirrored “non banked” memory range) and provide an individual/selective module for these.

<table>
<thead>
<tr>
<th>SDI</th>
<th>Component</th>
<th>Comm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication...</td>
<td>Set MCU Type...</td>
<td>Set MCU Speed...</td>
</tr>
<tr>
<td>Memory Map...</td>
<td>Flash...</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE** Please see Hardware Considerations for more information about the Flash modules of your CPU derivative.

Other important expressions are:

- **Enabled** — An *enabled module* is a module currently active on the chip. It is possible to read (as a ROM) or program an enabled module.

- **Disabled** — A *disabled module* is not active on the chip, so program and reading are not possible. The usual control for enabling or disabling a module is setting/clearing a flag in a special register. Note that a few modules always must be active; you may not disable such modules.

- **Blank** — A *blank module* is empty of code. You can program its full address range. Each blank byte contains the value 0xFF or 0x00, depending on hardware.
• **Programmed** — A *programmed module* is partially programmed (not all bytes contain 0xFF or 0x00). You must keep track of the areas still available for programming, if any.

• **Protected** — A *protected module* is partially protected from erasure or programming. The usual control for protecting a module is setting/clearing a flag in a special register. Note that a few modules always must be unprotected; you may not protect such modules.

• **Unprotected** — An *unprotected module* can be erased and programmed.

To select a module or other list item, left-click the module. To unselect a module, press the Ctrl key and left-click. For multiple selections or unselections, use the Shift key.

**NVMC Dialog Box**

The NVMC dialog box lists all the Flash or EEPROM modules of a CPU derivative. A derivative such as the HC12B32 has just one on-chip Flash module; other derivatives have multiple modules.

---

**NOTE** The dialog box does not have a Select or Unselect button, as you merely click on a module in the list to select it. But selecting and unselecting are not automatic from the command line. Before you use the command line to perform any operation on a module, you must use the `SELECT` command to select the module.
Figure 18.6 Non Volatile Memory Control Dialog Box

For each block, the dialog box has a line composed of the following fields:

- **Name** — the module name.
- **Start** — the module start address.
- **End** — the module end address.
- **State** — the module states, such as disabled, enabled, blank, programmed, protected, unprotected.

Possible state combinations are:

- **Bad Device** (the interface could not detect a correct device)
- **Disabled** (one or all modules are disabled)
- **[Enabled] / [Blank | Programmed] / [Unprotected | Protected]

The NVMC dialog box displays only meaningful states. For example, it displays Enabled only if it is possible to disable a module. It displays Unprotected only if it is possible to protect a module.

The Configuration group identifies the current .FPP parameter file. This group also includes the **Auto select according to MCUID** checkbox; the **Configuration: FPP loading** section explains this option.
The second checkbox of the Configuration group is **Save and restore workspace content**. If this checkbox is clear, Flash programming applications overwrite any data in RAM. To save the current RAM data, check this box. (Saving RAM data slows down the NVMC; checking this checkbox is equivalent to entering the **SAVECONTEXT** and **LOADCONTEXT** commands.)

### Flash Module Handling

Flash parameter files (which have the extension `.FPP`) contain MCU-specific parameters, as well as programs to handle internal flash modules. (The **FPP Files** section includes additional information about `.FPP` files.) The `.FPP` files also include code-applet descriptions of flash operations; later text of this manual includes these descriptions.

You also may use the Command Line component to handle flash operations. The **NVMC Commands** section explains the corresponding commands.

The NVMC dialog box has buttons for commands you can apply to each block. These buttons are dynamic: active if the operation is possible for at least one selected item, disabled if the operation is not possible.

- **Select All/Unselect All** — The **Select All...** button selects all modules in the list box. When the button got pressed, the button changes to “Unselect All”, that can be pressed to remove all current selections.

- **Enable/Disable** — The **Enable** button enables all selected modules currently disabled. The **Disable** button disables all selected modules currently enabled. (The possibility of enabling or disabling a flash module depends on the MCU features and context.)

- **Protect/Unprotect** — The **Protect** button protects all selected modules currently unprotected. The **Unprotect** button unprotects all selected modules currently protected. (The possibility of protecting or unprotecting a flash module depends on the MCU features and context.)

**NOTE** For some MCUs, protection is possible only for the Boot section and boot routines, not for the entire module. Please see **Hardware Considerations** for protection information about your CPU derivative.

- **Erase** — The **Erase** button removes any programming from all selected Flash modules. (That is, it assigns the value 0xFF or 0x00 to each byte.) Erasure changes the module status to Blank. If all the selected modules already are blank, the Erase button is disabled.

- **Load** — The **Load..** button arms all selected modules, executes a **LOAD** command, then disarms the modules. If you click the **Load..** button without selecting any flash modules, the NVMC utility selects and loads all the modules.
NOTE You merely click on a module in the list to select and/or use Select All/Unselect All buttons to adjust your selection. But selecting and unselecting are not automatic from the command line. Before you use the command line to perform any operation on a module, you must use the SELECT command to select the module. (Also see the FLASH SELECT and FLASH UNSELECT commands in Debugger Connection-specific Commands.)

MCU Speed Information
The displayed MCU speed is the device bus speed/clock sensed by the Flash Programmer, the same value as the one returned by the FLASH command.

CAUTION A non relevant displayed speed is symptomatic of a Flash Programmer diagnostic problem. In that case, please close the dialog, check the hardware set again the connection.

Configuration: FPP File Loading
When the dialog box is open, the NVMC utility loads the .FPP configuration file according to this algorithm:

1. The utility reads the NV_PARAMETER_FILE entry from the connection-specific section of the project.ini file. [Motorola ESL] is such a connection-specific section.
   Example:
   [Motorola ESL]
   NV_PARAMETER_FILE=C:\MYINSTALL\PROG\FPP\mcu03C4.fpp
2. If the utility retrieves a valid .FPP file name, it loads the file.
3. If the utility cannot find a valid .FPP file name, it displays an appropriate error message.
4. If the utility does not find an entry, or if it finds an empty entry, the utility automatically checks the Auto select according to MCUID: checkbox. Then the utility loads the parameter file from the \FPP subdirectory of the METROWERKS installation, according to the MCUID.
5. If the utility finds a file that has the wrong format, it displays an appropriate error message.
6. The utility always displays the MCUID, if the Id is available from the connection.
Another way to load an .FPP parameter file is clicking the **Browse...** button. This brings up a standard **Open** dialog box, which you can use to select the file. When you do so, the **Open** dialog box disappears, and the NVMC utility loads the file, automatically clearing the **Auto select according to MCUID**: checkbox. In case of any error during loading, the utility displays an appropriate message.

**Figure 18.7 Open Dialog Box**

If you check the **Auto select according to MCUID**: checkbox, the NVMC utility searches for and loads the corresponding .FPP parameter file.

Click the **OK** button to close the NVMC dialog box. If the **Auto select according to MCUID**: checkbox is clear, the NVMC utility saves the name of the selected configuration file under the NV_PARAMETER_FILE entry of the project.ini file. If you check this checkbox, the utility does not save the .FPP in the project file.

Click the **Cancel** button to close the dialog box without saving changes.

### Loading an Application in Flash

The **Load...** button and the **Load...** selection of the connection-specific menu function identically. Using either of these controls brings up the Load Executable File dialog box, which lets you select the file to be loaded. The Load Executable File dialog box lists the executable files that relate to blocks selected in the NVMC dialog box.
If a problem occurs during application loading into flash, the NVMC utility displays an error message.

If programming voltage is not available, the following error message is displayed.

This means that you tried to load a program into an unselected section. The NVMC utility’s selecting/unselecting feature reduces the risk of overwriting, erasing, or unprotecting valuable data.
Preparing, Loading an Application in Flash

Use either:

- The NVMC dialog box, which the NVMC Dialog Box section explains.
- Flash commands within a command file. Debugger Connection-specific Commands explains these commands.

If necessary, link your application with the appropriate memory model. The example below shows a .PRM file for an HC12DG128 application. The default ROM is in pages 2 and 4; the application uses the banked memory model. Make sure that your code location is within a Flash address range.

Listing 18.1 Loading an Application in Flash

```
LINK my_appli.abs

NAMES my_appli.o ansib.lib start12b.o END

SECTIONS
  MY_RAM = READ_WRITE 0x2010 TO 0x23FF;
  MY_ROM = READ_ONLY 0xC000 TO 0xFEFF;
  PAGE_2 = READ_ONLY 0x28000 TO 0x2BFFF;
  PAGE_4 = READ_ONLY 0x48000 TO 0x4BFFF;

PLACEMENT
  _PRESTART, STARTUP,
  ROM_VAR, STRINGS,
  NON_BANKED, COPY INTO MY_ROM;
  DEFAULT_RAM INTO MY_RAM;
  MyPage, DEFAULT_ROM INTO PAGE_2, PAGE_4;

END

STACKSIZE 0x50

VECTOR ADDRESS 0xFFFE _Startup /*set reset vector IN FLASH on _Startup */
```

Follow the loading command example in Debugger Connection-specific Commands or follow these instructions:

1. From the Debugger menu bar, open the connection-specific menu (such as SDI).
   Select Flash... — the NVMC dialog box appears.

2. If you are sure about the absolute location of your application, you do not need to select a module. But if you program in a protected area (boot block), make sure that the matching module is unprotected.
3. Click the **Load...** button — the NVMC utility selects all modules and opens the Load Executable File dialog box.

4. Select the .ABS file to be loaded into Flash. Loading begins; a progress bar appears. When loading is finished, the NVMC dialog box displays the new state of the modules.

5. This completes loading. You can close the NVMC dialog box and run your application. For some some hardware, however, you first must do a connection reset, by clicking the reset button of the Debugger.

### Hardware Considerations

This section consists of hardware-specific information about current .FPP files. A release note will explain any new .FPP file features.

---

**NOTE**

The Flash Programming release note, in the on-line documentation of your toolkit installation, contains the latest information about .FPP files.

---

### HC12 (CPU12) CPU devices

#### HC12B32

- **fpp file name**: mcu03c1.fpp
- **number of Flash modules**: 1
  - applet code currently not relocatable, loaded at 0x800, using 0x400 bytes.
  - module name: FLASH_B32 / module number: 0
  - 32 KBytes flash located in 0x8000-0xFFFF or 0x0000-0x7FFF (both handled, according to MAPROM bit in MISC register).
  - boot sector unprotectable/protectable (2 KBytes in range 0xF800-0xFFFF or 0x7800-0x7FFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).
  - flash enable/disable via ROMON bit in MISC register.

#### HC12D60

- **fpp file name**: mcu03c3.fpp
- **number of Flash modules**: 2
  - applet code currently not relocatable, loaded at 0x400, using 0x400 bytes.
  - module name: FEE28 / module number: 0
- 28 KBytes flash located in 0x1000-0x7FFF or 0x9000-0xFFFF (both handled, according to MAPROM bit in MISC register).
- boot sector unprotectable/protectable (8 KBytes in range 0x6000-0x7FFF or 0xE000-0xFFFF) (via BOOTP bit in FEE28MCR register and LOCK bit in FEE28LCK register).
- flash enable/disable via ROMON28 bit in MISC register.

module name: FEE32 / module number: 1

- 32 KBytes flash located in 0x8000-0xFFFF or 0x0000-0x7FFF (both handled, according to MAPROM bit in MISC register).
- boot sector unprotectable/protectable (8 KBytes in range 0xE000-0xFFFF or 0x6000-0x7FFF) (via BOOTP bit in FEE32MCR register and LOCK bit in FEE32LCK register).
- flash enable/disable via ROMON32 bit in MISC register.

HC12DG128

fpp file name: mcu03c4.fpp

number of Flash modules: 10

- applet code currently not relocatable, loaded at 0x2000, using 0x400 bytes.
- all flash modules enable/disable at same time via ROMON bit in MISC register.

module name: FLASH_4000 / module number: 0

- 16 KBytes unpaged Flash located in 0x4000-0x8000 also matches 11FEE even page (6), that is, FLASH_PAGE6.

module name: FLASH_PAGE0 / module number: 1

- 16 KBytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to (cf Technical Summary) 00FEE Flash even page (0).

module name: FLASH_C000 / module number: 2

- 16 KBytes unpaged Flash located in 0xC000-0xFFFF also matches 11FEE odd page (7), that is, FLASH_PAGE7.

- boot sector unprotectable/protectable (8 KBytes in range 0xE000-0xFFFF or paged range 0xA000-0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).

module name: FLASH_PAGE1 / module number: 3

- 16 KBytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to (cf Technical Summary) 00FEE Flash odd page (1).

module name: FLASH_PAGE2 / module number: 4
HC(S)12(X) Flash Programming

Hardware Considerations

- 16 KBytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to (cf Technical Summary) 01FEE Flash even page (2).
  module name: FLASH_PAGE3 / module number: 5

- 16 KBytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to (cf Technical Summary) 01FEE Flash odd page (3).
  - boot sector unprotectable/protectable (8 KBytes in range 0xA000-0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).
  module name: FLASH_PAGE4 / module number: 6

- 16 KBytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to (cf Technical Summary) 10FEE Flash even page (4).
  module name: FLASH_PAGE5 / module number: 7

- 16 KBytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to (cf Technical Summary) 10FEE Flash odd page (7).
  - boot sector unprotectable/protectable (8 KBytes in range 0xA000-0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).
  module name: FLASH_PAGE6 / module number: 8

- 16 KBytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to (cf Technical Summary) 11FEE Flash even page (6). Also equivalent to FLASH_4000 module.
  module name: FLASH_PAGE7 / module number: 9

- 16 KBytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to (cf Technical Summary) 11FEE Flash odd page (7). Also equivalent to FLASH_C000 module.
  - boot sector unprotectable/protectable (8 KBytes in range 0xA000-0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).

**HCS12 and HCS12X CPU devices**

All protections are fully removed when erasing and programming. The security byte at $FF0F is always reprogrammed to “unsecure” when erasing (actually due to “aligned word” programming, $FF0E-FF0F is programmed to #$FFFE). The debugger asserts “aligned word” programming as specified in FTSxxxK and FTXxxxK specifications.

HCS12 and HCS12X device fpp files having been simplified to increase programming speed, as devices may have up to 512 kBytes of onchip flash. Indeed, changing programming methods for each Program Page (32 PPAGE’s on MC9S12DP512) would slow down the programming.

By consequence, only relevant onchip flash blocks have their own listed module, and the list below gives an overall availability for all HCS12 and HCS12X devices.
• **FLASH_4000**: onchip flash in $4000-$7FFF, also a mirror of PPAGE $3E on HCS12 devices and $FD on HCS12X devices. This module is provided to design non banked code, like for ISR’s code, startup code, etc.

• **FLASH_C000**: onchip flash in $C000-$FFFF, also a mirror of PPAGE $3F on HCS12 devices and $FF on HCS12X devices. This module is provided to design non banked code, like for ISR’s code, startup code, etc. and vectors.

• **ALL_PPAGES (also previously called “PAGED”)**: The entire onchip flash memory.

**CAUTION** Erasing this module erases de facto FLASH_4000 and FLASH_C000 modules.

• **FLAT8000_Pxx or FLASH_8000 (HCS12X) and EEPROM_800 (HCS12X):** This range gives the possibility to design a linear source code to be programmed from address $4000 to $FFFF. The reset default page (“Pxx”) visible in $8000-$BFFF may vary from one HCS12 device to another, and be exactly the $3E PPAGE on HCS12X devices. This way of programming can be used to evaluate a 48 kBytes application other several devices, but remain critical due to the lack of control of the current PPAGE. Indeed, if the PPAGE changes (by program “CALL” or changed by PPAGE register accidentally writing), the program code stored in the window range $8000-$BFFF cannot be executed properly. This implies not to use the entire capacity of the flash of a device.

**NOTE** For some backward compatibility reasons, these modules cannot be erased, but only programmed. Erasing is available but does not make any operation.

• **ALL_EPAGES (also previously called “EEP_PAGED”) (HCS12X only)**: The entire onchip flash memory.

**CAUTION** Erasing this module erases de facto all other EEPROM modules.

### HCS12 EEPROM’s Relocation

HCS12 devices provide sometimes hidden EEPROM memory that can only be accessed when changing the Memory Expansion Register called “INITEE”.

#### Fully Hidden EEPROM

The EEPROM is fully blankchecked.

The FPP file will remap the EEPROM via INITEE automatically to $2000, at the condition that the user did not relocate himself the EEPROM changing INITEE. In that
case, the FPP driver will handle the EEPROM at the place it has been relocated by the user.

**Partially Visible EEPROM in $400-7FF or $400-FFF**
The EEPROM is fully blankchecked.
If the EEPROM is not at the reset location, the EEPROM size and location are automatically updated.
The EEPROM size in the NVMC dialog is automatically updated if the RAM does not overlap the EEPROM module.

**EB386 Compliancy and RAM Moving**
NVMIF2 (format) new FPP drivers can be relocated in RAM. This new format for HCS12 devices is based on PIC code runtimes. Therefore, the NVM handling runtime can be moved in RAM if necessary.
First the “FLASH” command must be type in a Command window to verify if the FPP file is “(NVMIF2)”.
The “FLASH NVMIF2WORKSPACE” can be executed to relocate the driver workspace in RAM, according to an eventual relocation of the RAM by the user via INITRM setup with a debugger “WB” command. Please see the debugger FLASH commands section.
This can provide more flexibility for EB386 (EB386/D, Rev. 3, 07/2002, Engineering Bulletin) "Example 1 Layout" device ram memory relocation. However, if the relocation is performed by the application itself, the usage of the FPP relocation is useless, as the programming is performed with the default location of the RAM.

**CAUTION** The FPP files/drivers do not support HCS12 onchip Registers block moving from default/reset position.

**Legacy Flash Programming Commands in Preload and Postload Command Files**
The legacy FLASH commands created by the project wizard to program automatically an application is given here below.

**In xxxx_Preload.cmd file:**
```bash
// reset the device to get default settings
RESET

// initialize flash programming process
FLASH
```
// select the flash modules
FLASH SELECT
// erase the flash modules
FLASH ERASE
// arm the flash for programming
FLASH ARM

In xxxx_Postload.cmd file:
// The following commands must be enabled to terminate the programming process
with the ICD12

// disarm the flash modules
FLASH DISARM
// unselect the flash modules
FLASH UNSELECT
// reset the target board
RESET

TIP This Legacy implementation can be replaced by the Automated Application
Programming feature discussed at the beginning if this manual section. Simply
clean or disable both command files then engage the “Automatically erase and
program...” option in debugger Preferences.
XGATE Debugging Support
Book IV - Commands

Book IV Contents

Each section of the Debugger manual includes information to help you become more familiar with the Debugger, to use all its functions and help you understand how to use the environment. This book, the Debugger Commands, defines the HC12, HCS12 and HC(S)12(X) Commands, both those commands used by the debugger engine and those specific to individual debugger connections.

This book is divided into the following chapters:

- Chapter 4.1 Debugger Engine Commands
- Chapter 4.2 Debugger Connection-specific Commands
Debugger Engine
Commands

Commands Overview

The debugger supports scripting with the use of commands and command files. When you script the debugger, you can automate repetitive, time-consuming, or complex tasks.

You do not need to use or have knowledge of commands to run the Simulator/Debugger. However these commands are useful for editing debugger command files, for example, after a recording session, to generate your own command files, or to set up your applications and targets, etc.

This section provides a detailed list of all Simulator/Debugger commands. All command names and component names are case insensitive. The command EBNF syntax is:

```
component [:component number] < ] command
```

where **component** is the name of the component that you can read in each component window title. For example: Data, Register, Source, Assembly, etc. **Component number** is the number of the component. This number does not exist in the component window title if only one component of this type is open. For example, you will read **Register** or **Memory**. If you open a second Memory component window, the initial one will be renamed **Memory:1** and the new one will be called **Memory:2**. A number is automatically associated with a component if there are several components of the same type displayed.

**Command Example:**

```
in>Memory:2 < SMEM 0x8000,8
```

`<` redirects a command to a specific component (in this example: **Memory:2**). Some commands are valid for several or all components and if the command is not redirected to a specific component, all components will be affected. Also, a mismatch could occur due to the fact that a command’s parameters could differ for different components.
Debugger Engine Commands

Commands Overview

Command Syntax

To display the syntax of a command, type the command followed by a question mark.

Syntax Example:

```
in>printf?
PRINTF (<format>, <expression>, <expression>, ...)
```

Available Command Lists

Commands described on the following pages are sorted into 5 groups, according to their specific actions or targets. However, these groups have no relevance in the use of these commands. A list of all commands in their respective group is given below:

Kernel Commands

Kernel commands are commands that can be used to build command programs. They can only be used in a debugger command file, since the Command Line component can only accept one command at a time. It is possible to build powerful programs by combining Kernel commands with Base commands, Common commands and Component specific commands. Table 20.1 contains all available Kernel commands.

Table 20.1 List of Kernel Commands

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Affects a value</td>
</tr>
<tr>
<td>AT</td>
<td>Sets a time delay for command execution</td>
</tr>
<tr>
<td>CALL fileName[]</td>
<td>Executes a command file</td>
</tr>
<tr>
<td>DEFINE symbol [ ]</td>
<td>Defines a user symbol</td>
</tr>
<tr>
<td>ELSE</td>
<td>Other operation associated with IF command</td>
</tr>
<tr>
<td>ELSEIF condition</td>
<td>Other conditional operation associated with IF command</td>
</tr>
<tr>
<td>ENDFOCUS</td>
<td>Resets the current focus (refer to FOCUS command)</td>
</tr>
<tr>
<td>ENDFOR</td>
<td>Exits a FOR loop</td>
</tr>
</tbody>
</table>
Table 20.1 List of Kernel Commands (continued)

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENDF</td>
<td>Exits an IF condition</td>
</tr>
<tr>
<td>ENDWHILE</td>
<td>Exits a WHILE loop</td>
</tr>
<tr>
<td>FOCUS component</td>
<td>Sets the focus on a specified component</td>
</tr>
<tr>
<td>FOR [variable =]range [&quot;,&quot; step]</td>
<td>FOR loop instruction</td>
</tr>
<tr>
<td>FPRINTF (fileName,format,parameters)</td>
<td>FPRINTF instruction</td>
</tr>
<tr>
<td>GOTO label</td>
<td>Unconditional branch to a label in a command file</td>
</tr>
<tr>
<td>GOTOIF condition Label</td>
<td>Conditional branch to a label in a command file</td>
</tr>
<tr>
<td>IF condition</td>
<td>Conditional execution</td>
</tr>
<tr>
<td>PAUSETEST</td>
<td>Displays a modal message box</td>
</tr>
<tr>
<td>PRINTF (*Text;:&quot; value])</td>
<td>PRINT instruction</td>
</tr>
<tr>
<td>REPEAT</td>
<td>REPEAT loop instruction</td>
</tr>
<tr>
<td>RETURN</td>
<td>Returns from a CALL command</td>
</tr>
<tr>
<td>TESTBOX</td>
<td>Displays a message box with a string</td>
</tr>
<tr>
<td>UNDEF symbol</td>
<td>Undefines a userdefined symbol</td>
</tr>
<tr>
<td>UNTIL condition</td>
<td>Condition of a REPEAT loop</td>
</tr>
<tr>
<td>WAIT [time] [:s]</td>
<td>Command file execution pause</td>
</tr>
<tr>
<td>WHILE condition</td>
<td>WHILE loop instruction</td>
</tr>
</tbody>
</table>

**Base Commands**

Base commands are used to monitor the Simulator/Debugger target execution. Target input/output files, target execution control, direct memory editing, breakpoint management and CPU register setup are handled by these commands. Base commands can be executed independent of components that are open. Table 20.2 contains all available Base commands.
### Table 20.2  List of Base Commands

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BC</strong> address*</td>
<td>Deletes a breakpoint (breakpoint clear)</td>
</tr>
<tr>
<td><strong>BS</strong> address[function [P][T][state]]</td>
<td>Sets a breakpoint (breakpoint set)</td>
</tr>
<tr>
<td><strong>CD</strong> [path]</td>
<td>Changes the current working directory</td>
</tr>
<tr>
<td><strong>CR</strong> [fileName][:A]</td>
<td>Opens a record file (command records)</td>
</tr>
<tr>
<td><strong>DASM</strong> [address[range]][:OBJ]</td>
<td>Disassembles</td>
</tr>
<tr>
<td><strong>DB</strong> [address[range]]</td>
<td>Displays memory bytes</td>
</tr>
<tr>
<td><strong>DL</strong> [address[range]]</td>
<td>Displays memory bytes as longwords</td>
</tr>
<tr>
<td><strong>DW</strong> [address[range]]</td>
<td>Displays memory bytes as words</td>
</tr>
<tr>
<td><strong>G</strong> [address]</td>
<td>Starts execution of the application currently loaded</td>
</tr>
<tr>
<td><strong>GO</strong> [address]</td>
<td>Starts execution of the application currently loaded</td>
</tr>
<tr>
<td><strong>LF</strong> [fileName][:A]</td>
<td>Opens a log file</td>
</tr>
<tr>
<td><strong>LOG</strong> type [=] state ([,] type [=] state)</td>
<td>Enables or disables logging of a specified information type</td>
</tr>
<tr>
<td><strong>MEM</strong></td>
<td>Displays the memory map</td>
</tr>
<tr>
<td><strong>MS</strong> range list</td>
<td>Sets memory bytes</td>
</tr>
<tr>
<td><strong>NOCR</strong></td>
<td>Closes the record file</td>
</tr>
<tr>
<td><strong>NOLF</strong></td>
<td>Closes the log file</td>
</tr>
<tr>
<td><strong>P</strong> [address]</td>
<td>Single assembly steps into program</td>
</tr>
<tr>
<td><strong>RESTART</strong></td>
<td>Restarts the loaded application</td>
</tr>
<tr>
<td><strong>RD</strong> list[*]</td>
<td>Displays the content of registers</td>
</tr>
<tr>
<td><strong>RS</strong> register[=]value[,register[=]value]</td>
<td>Sets a register</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>Stops execution of the loaded application</td>
</tr>
</tbody>
</table>
Table 20.3  List of Environment Commands

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEPINTO</td>
<td>Steps to the next source instruction of the loaded application</td>
</tr>
<tr>
<td>STEPOUT</td>
<td>Executes program out of a function call</td>
</tr>
<tr>
<td>STEPOVER</td>
<td>Steps over the next source instruction of the loaded application</td>
</tr>
<tr>
<td>STOP</td>
<td>Stops execution of the loaded application</td>
</tr>
<tr>
<td>SAVEBP on</td>
<td>off</td>
</tr>
<tr>
<td>T [address],[count]</td>
<td>Traces program instructions at the specified address</td>
</tr>
<tr>
<td>WB range list</td>
<td>Writes bytes</td>
</tr>
<tr>
<td>WL range lis</td>
<td>Writes longwords</td>
</tr>
<tr>
<td>WW range list</td>
<td>Writes words</td>
</tr>
</tbody>
</table>

Environment Commands

Simulator/Debugger environment commands are used to monitor the debugger environment, specific component window layouts and framework applications and targets. Table 20.3 contains all available Environment commands.

Table 20.3  List of Environment Commands

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVATE component</td>
<td>Activates a component window</td>
</tr>
<tr>
<td>AUTOSIZE on</td>
<td>off</td>
</tr>
<tr>
<td>BCKCOLOR color</td>
<td>Sets the background color</td>
</tr>
<tr>
<td>CLOSE component</td>
<td>Closes a component</td>
</tr>
<tr>
<td>DDEPROTOCOL ON</td>
<td>OFF</td>
</tr>
<tr>
<td>FONT 'fontName' [size][color]</td>
<td>Sets text font</td>
</tr>
</tbody>
</table>
Component Commands

Component common commands are used to monitor component behaviors. They are common to more than one component and for better usage, they should be redirected (as explained in the introduction to this chapter. Table 20.4 contains all available Component commands.

Table 20.4 List of Component Commands

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD applicationName</td>
<td>Loads a framework application (code and debug information)</td>
</tr>
<tr>
<td>LOADCODE applicationName</td>
<td>Loads the code of a framework application</td>
</tr>
<tr>
<td>LOADSYMBoLS applicationName</td>
<td>Loads debugging information of a framework application</td>
</tr>
<tr>
<td>OPEN component [[x y width height][[]]]</td>
<td>Opens a Windows component</td>
</tr>
<tr>
<td>SET targetName</td>
<td>Sets a new target</td>
</tr>
<tr>
<td>SLAY fileName</td>
<td>Saves the general window layout</td>
</tr>
</tbody>
</table>
### Component Specific Commands

Component specific commands are associated with specific components. Table 20.5 contains all available Component Specific commands.

#### Table 20.5 List of Component Specific Commands

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADDXPR</strong> &quot;expression&quot;</td>
<td>Adds a new expression in the data component</td>
</tr>
<tr>
<td><strong>ATTRIBUTES</strong> list</td>
<td>Sets up the display inside a component window</td>
</tr>
<tr>
<td><strong>BASE</strong> code</td>
<td>module</td>
</tr>
<tr>
<td><strong>BD</strong></td>
<td>Displays a list of all breakpoints</td>
</tr>
<tr>
<td><strong>CF</strong> fileName</td>
<td>[;C][;NL]</td>
</tr>
<tr>
<td><strong>CLOCK</strong> frequency</td>
<td>Sets the clock speed</td>
</tr>
<tr>
<td><strong>COPYMEM</strong> &lt;Source addr range&gt; dest-addr</td>
<td>Copies memory</td>
</tr>
<tr>
<td><strong>CYCLE</strong> on</td>
<td>off</td>
</tr>
<tr>
<td><strong>DETAILS</strong> assembly</td>
<td>source</td>
</tr>
<tr>
<td><strong>DUMP</strong></td>
<td>Displays data component content</td>
</tr>
<tr>
<td><strong>E</strong> expression</td>
<td>[:O</td>
</tr>
<tr>
<td><strong>EXECUTE</strong> fileName</td>
<td>Executes a stimulation file</td>
</tr>
<tr>
<td><strong>FILL</strong> range value</td>
<td>Fills a memory range with a value</td>
</tr>
<tr>
<td><strong>FILTER</strong> Options [&lt;range&gt;]</td>
<td>Selects the output file filter options</td>
</tr>
</tbody>
</table>
### Debug Engine Commands

**Commands Overview**

Table 20.5  List of Component Specific Commands (continued)

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND &quot;string&quot; [:B] [:MC] [:WW]</td>
<td>Finds and highlights a pattern</td>
</tr>
<tr>
<td>FINDPROC ProcedureName</td>
<td>Opens a procedure file</td>
</tr>
<tr>
<td>FOLD [*]</td>
<td>Folds a source block</td>
</tr>
<tr>
<td>FRAMES number</td>
<td>Sets the maximum number of frames</td>
</tr>
<tr>
<td>GRAPHICS on/off</td>
<td>Switches graphic bars on/off</td>
</tr>
<tr>
<td>INSPECTOROUTPUT [name {subname}]</td>
<td>Prints content of Inspector to Command window</td>
</tr>
<tr>
<td>INSPECTORUPDATE</td>
<td>Updates content of Inspector</td>
</tr>
<tr>
<td>LS [symbol</td>
<td>*][;C</td>
</tr>
<tr>
<td>NB [base]</td>
<td>Sets the base of arithmetic operations</td>
</tr>
<tr>
<td>OUTPUT fileName</td>
<td>Redirects the coverage component results</td>
</tr>
<tr>
<td>PTRARRAY on/off</td>
<td>Switches on/off the pointer as array display</td>
</tr>
<tr>
<td>RECORD on/off</td>
<td>Switches on/off the frame recorder</td>
</tr>
<tr>
<td>SLINe linenumber</td>
<td>Shows the desired line number</td>
</tr>
<tr>
<td>SAVE range fileName [offset][;A]</td>
<td>Saves a memory block in S-Record format</td>
</tr>
<tr>
<td>SETCOLORS ( &quot;Name&quot; ) ( Background) ( Cursor ) ( Grid ) ( Line ) ( Text )</td>
<td>Changes the colors attributes of the &quot;Name&quot; channel from the Monitor component</td>
</tr>
<tr>
<td>SREC fileName [offset]</td>
<td>Loads a memory block in S-Record format</td>
</tr>
<tr>
<td>UPDATE on/off</td>
<td>Switches on/off time update for statistics</td>
</tr>
<tr>
<td>UNFOLD [*]</td>
<td>Unfolds a source block</td>
</tr>
<tr>
<td>UPDATERATE rate</td>
<td>Sets the data and memory update mode</td>
</tr>
<tr>
<td>ZOOM address in</td>
<td>out</td>
</tr>
</tbody>
</table>
Command Syntax Terms

address
A number matching a memory address. This number must be in the ANSI format (i.e. $ or 0x for hexadecimal value, 0 for octal, etc.).
Example: 255, 0377, 0xFF, $FF

NOTE address can also be an “expression” if “constant address” is not specially mentioned in the command description. An “expression” can be: Global variables of application, I/O registers defined in DEFAULT.REG, definitions in the command line, numerical constants.

Example: DEFINE IO_PORT = 0x210
WB IO_PORT 0xFF

range
A composition of 2 addresses to define a range of memory addresses. Syntax is shown below:
address..address
or
address, size
where size is an ANSI format numerical constant.
Example:
0x2F00..0x2FFF
Refers to the memory range starting at 0x2F00 and ending at 0x2FFF (256 bytes).
Example:
0x2F00,256
Refers to the memory range starting at 0x2F00, which is 256 bytes wide. Both previous examples are equivalent.

fileName
A DOS file name and path that identifies a file and its location. The command interpreter does not assume any file name extension. Use backslash (\) or slash (/) as a directory delimiter.
The parser is case insensitive. If no path is specified, it looks for (or edits) the file in the current project directory, i.e. when no path is specified, the default directory is the project directory.
Debugger Engine Commands

Debugger Commands

Example:
d:/demo/myfile.txt
Example:
layout.hwl
Example:
d:/work/project.hwc

**component**
The name of a debugger component. A list of all debugger components is given by choosing **Component>Open**. The parser is case insensitive.

Example:
Memory
Example:
SoUrCe

**Module Names**
Correct module names are displayed in the Module component window. Make sure that the module name of a command that you implement is correct:

If the .abs is in **HIWARE** format, some debug information is in the object file (.o), and module names have a .o extension (e.g., fibo.o).

In **ELF** format, module name extensions are .c, .cpp or .dbg (.dbg for program sources in assembler) (e.g., fibo.c), since all debugging information is contained in the .abs file and object files are not used.

**Debugger Commands**

The commands available when you use the Simulator/Debugger are defined on the following pages.
A

The A command assigns an expression to an existing variable. The quoted expression must be used for string and enum expressions.

Usage

A variable = value or A variable = "value"

Components

Debugger engine.

Example:

    in>a counter=8
    The variable counter is now equal to 8.
    in>A day1 = "monday_8U"    (Monday_8U is defined in an Enum)
    The variable day1 is now equal to monday_8U.
    in>A value = "3.3"
    The variable value is now equal to 3.3

ACTIVATE

ACTIVATE activates a component window as if you clicked on its title bar. The window is displayed in the foreground and its title bar is highlighted. If the window is iconized, its title bar is activated and displayed in the foreground.

Usage

ACTIVATE component

Components

Debugger engine.

Example:

    in>ACTIVATE Memory
    Activates the Memory Component and brings the window to the foreground.
**ADDXPR**

The **ADDXPR** command adds a new expression in the data component.

**Usage**

ADDXPR “expression”

Where the parameter expression is an expression to be added and evaluated in the data component.

**Components**

Data component.

**Example:**

\[ \text{in>ADDXPR “counter + 10”} \]

The expression “counter +10” is added in the data component.

**ATTRIBUTES**

This command is effective for various components as described in the next sections.

**In the Command Component**

The ATTRIBUTES command allows you to set the display and state options of the Command component window. The CACHESIZE command sets the cache size in lines for the Command Line window: The cache size value is between 10 and 100000.

**NOTE**

Usually this command is not specified interactively by the user. However this command can be written in a command file or a layout (".HWL") file to save and reload component window layouts. An interactive equivalent operation is typically possible, using Simulator/Debugger menus and operations, drag and drops, etc., as described in the following sections in “Equivalent Operations”.

**Usage**

ATTRIBUTES list

where list=command{,command})

command=CACHESIZE value

**Example:**

\[ \text{command < ATTRIBUTES 2000} \]
In the Procedure Component

The ATTRIBUTES command allows you to set the display and state options of the Procedure component window. The VALUES and TYPES commands display or hide the Values or Types of the parameters.

Usage

ATTRIBUTES list
where list=command{,command})

command=VALUES (ON|OFF)| TYPES (ON|OFF)

Example:
Procedure < ATTRIBUTES VALUES ON,TYPES ON

In the Assembly Component

The ATTRIBUTES command allows you to set the display and state options for the Assembly component window. The ADR command displays or hides the address of a disassembled instruction. ON | OFF is used to switch the address on or off. SMEM (show memory range) and SPC (show PC address) scroll the Assembly component to the corresponding address or range code location and select/highlight the corresponding assembler lines or range of code. The CODE command displays or hides the machine code of the disassembled instruction. ON | OFF is used to switch on or off the machine code. The ABSADR command shows or hides the absolute address of a disassembled instruction like ‘branch to’. ON | OFF is used to switch on or off the absolute address. The TOPPC command scrolls the Assembly component in order to display the code location given as an argument on the first line of Assembly component window. The SYMB command displays or hides the symbolic names of objects. ON | OFF is used to switch the symbolic display on or off.

Usage

ATTRIBUTES list
where list=command{,command})

command= ADR (ON|OFF) | SMEM range | SPC address | CODE(ON|OFF) | ABSADR (ON|OFF) | TOPPC address | SYMB (ON|OFF)

NOTE Also refer to SMEM and SPC command descriptions for more detail about these commands. The SPC command is similar to the TOPPC command but also highlights the code and does not scroll to the top of the component window.
Equivalent Operations

ATTRIBUTES ADR ~ Select menu Assembly>Display ADR

ATTRIBUTES SMEM ~ Select a range in Memory component window and drag it to the Assembly component window.

ATTRIBUTES SPC ~ Drag a register to the Assembly component window.

ATTRIBUTES CODE ~ Select menu Assembly>Display Code

ATTRIBUTES SYMB ~ Select menu Assembly>Display Symbolic

Example:

Assembly < ATTRIBUTES ADR ON, SYMB ON, CODE ON, SMEM 0x800,16

Addresses, hexadecimal codes, and symbolic names are displayed in the Assembly component window, and assembly instructions at addresses 0x800,16 are highlighted.

In the Register Component

The ATTRIBUTES command allows you to set the display and state options of the Register component window.

The FORMAT command sets the display format of register values.

The VSCROLLPOS command sets the current absolute position of the vertical scroll box (the vposition value is in lines: each register and bitfield have the same height, which is the height of a line). vposition is the absolute vertical scroll position. The value 0 represents the first position at the top.

The HSCROLLPOS command sets the position of the horizontal scroll box (the hposition value is in columns: a column is about a tenth of the greatest register or bitfield width). hposition is the absolute horizontal scroll position. The value 0 represents the first position on the left.

The parameters vposition and hposition can be constant expressions or symbols defined with the DEFINE command.

The COMPLEMENT command sets the display complement format of register values: one sets the first complement (each bit is reversed), none unselects the first complement.

An error message is displayed if:

• the parameter is a negative value

• the scroll box is not visible

If the given scroll position is bigger than the maximum scroll position, the current absolute position of the scroll box is set to the maximum scroll position.
**Equivalent Operations**

ATTRIBUTES FORMAT ~ Select menu Register>Options

ATTRIBUTES VSCROLLPOS ~ Scroll vertically in the Register component window.

ATTRIBUTES HSCROLLPOS ~ Scroll horizontally in the Register component window.

ATTRIBUTES COMPLEMENT ~ Select menu Register>Options

**Usage**

ATTRIBUTES list

where list=command(,command)

command= FORMAT (hex|bin|dec|udec|oct) | VSCROLLPOS vposition | HSCROLLPOS hposition | COMPLEMENT(none|one)

Where vposition=expression and hposition=expression

**Example:**

in>Register < ATTRIBUTES FORMAT BIN

Contents of registers are displayed in binary format in the Register component window.

in>Register < ATTRIBUTES VSCROLLPOS 3

Scrolls 3 positions down. The third line of registers is displayed on the top of the register component.

in>Register < ATTRIBUTES VSCROLLPOS 0

Returns to the default display. The first line of registers is displayed on the top of the register component.

in>DEFINE vpos = 5

in>Register < ATTRIBUTES HSCROLLPOS vpos

Scrolls 5 positions right. The second column of registers is displayed on the left of the register component.

in>Register < ATTRIBUTES HSCROLLPOS 0

Returns to the default display. The first column of registers is displayed on the left of the register component.

in>Register < ATTRIBUTES COMPLEMENT One

Sets the first complement display option. All registers are displayed in reverse bit.
In the Source Component

The **ATTRIBUTES** command allows you to set the display and state options of the Source component window. The **SMEM** (show memory range) command and **SPC** (show PC address) command loads the corresponding module’s source text, scrolls to the corresponding text range location or text address location and highlights the corresponding statements. The **SMOD** (show module) command loads the corresponding module’s source text. If the module is not found, a message is displayed in the Component Windows Object Info Bar. The **SPROC** (show procedure) command loads the corresponding module’s source text, scrolls to the corresponding procedure and highlights the statement, that is in the procedure chain of this procedure. The **numberAssociatedToProcedure** is the level of the procedure in the procedure chain. The **MARKS** command (**ON** or **OFF**) displays or hides the marks.

**NOTE**

Also refer to **SMEM** **SPC** **SPROC** and **SMOD** command descriptions for more detail about these commands.

Equivalent Operations

- **ATTRIBUTES SPC** ~ Drag and drop from Register component to Source component.
- **ATTRIBUTES SMEM** ~ Drag and drop from Memory component to Source component.
- **ATTRIBUTES SMOD** ~ Drag and drop from Module component to Source component.
- **ATTRIBUTES SPROC** ~ Drag and drop from Procedure component to Source component.
- **ATTRIBUTES MARKS** ~ Select menu **Source>Marks**.

Usage

**ATTRIBUTES list**

where

- **list**=\texttt{command{,command}}
- **command**= \texttt{SPC address | SMEM range | SMOD module (without extension) | SPROC numberAssociatedToProcedure | MARKS (ON|OFF)}

**Example:**

\texttt{in>Source < ATTRIBUTES MARKS ON}

Marks are visible in the Source component window.
In the Data Component

The **ATTRIBUTES** command allows you to set the display and state options of the Data component window. The **FORMAT** command selects the format for the list of variables. The format is one of the following: binary, octal, hexadecimal, signed decimal, unsigned decimal or symbolic.

**Usage**

ATTRIBUTES list
where list=command{,command}

command=FORMAT(bin|oct|hex|signed|unsigned|symb)| SCOPE (global|local|user) | MODE (automatic|periodical|locked|frozen) | SPROC level | SMOD module | UPDATERATE rate | COMPLEMENT(none|one) | NAMEWIDTH width

The **MODE** command selects the display mode of variables.

- In **Automatic** mode (default), variables are updated when the target is stopped. Variables from the currently executed module or procedure are displayed in the data component. Variables are updated when target is stopped.
- In **Locked** and **Frozen** mode, variables from a specific module are displayed in the data component. The same variables are always displayed in the data component.
- In **Locked** mode, values from variables displayed in the data component are updated when the target is stopped.
- In **Frozen** mode, values from variables displayed in the data component are not updated when the target is stopped.
- In **Periodical** mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second, but it can be modified by steps of up to 100 ms using the associated dialog box or the **UPDATERATE** command.

The **UPDATERATE** command sets the variables update rate (see also **UPDATERATE** command).

The **SPROC** (show procedure) and **SMOD** (show module) commands display local or global variables of the corresponding procedure or module.

The **SCOPE** command selects and displays global, local or user defined variables.

The **COMPLEMENT** command sets the display complement format of Data values: one sets the first complement (each bit is reversed), none unselects the first complement.

The **NAMEWIDTH** command sets the length of the variable name displayed in the window.

**NOTE** Refer to **SPROC**, **UPDATERATE** and **SMOD** command descriptions for more detail about these commands.
**Debugger Engine Commands**

**Equivalent Operations**

ATTRIBUTES FORMAT ~ Select menu Data>Format...
ATTRIBUTES MODE ~ Select menu Data>Mode...
ATTRIBUTES SCOPE ~ Select menu Data>Scope...
ATTRIBUTES SPROC ~ Drag and drop from Procedure component to Data component.
ATTRIBUTES SMOD ~ Drag and drop from Module component to Data component.
ATTRIBUTES UPDATERATE ~ Select menu Data>Mode>Periodical.
ATTRIBUTES COMPLEMENT ~ Select menu Data>Format...
ATTRIBUTES NAMEWIDTH ~ Select menu Data>Options...>Name Width...

**Example:**

Data:1 < ATTRIBUTES MODE FROZEN

In **Data:1** (global variables), variables update is frozen mode. Variables are not refreshed when the application is running.

**In the Memory Component**

The **ATTRIBUTES** command allows you to set the display and state options of the Memory component window. The **WORD** command selects the word size of the memory dump window. The word size number can be 1 (for “byte” format), 2 (for “word” format - 2 bytes) or 4 (for “long” format - 4 bytes). The **ADR** command **ON** or **OFF** displays or hides the address in front of the memory dump lines. The **ASC** command **ON** or **OFF** displays or hides the ASCII dump at the end of the memory dump lines. The **ADDRESS** command scrolls the corresponding memory dump window and displays the corresponding memory address lines (memory **WORD** is not selected). **SPC** (show pc), **SMEM** (show memory) and **SMOD** (show module) commands scroll the Memory component accordingly, to display the code location given as argument, and select the corresponding memory area (SPC selects an address, SMEM selects a range of memory and SMOD selects the module name whom global variable would be located in the window).

The **FORMAT** command selects the format for the list of variables. The format is one of the following: binary, octal, hexadecimal, signed decimal, unsigned decimal or symbolic.

The **COMPLEMENT** command sets the display complement format of memory values: one sets the first complement (each bit is reversed), none unselects the first complement.

The **MODE** command selects the display mode of memory words.

- In **Automatic** mode (default), memory words are updated when the target is stopped. Memory words from the currently executed module or procedure are displayed in the Memory component. Memory words are updated when target is stopped.
In **Frozen** mode, value from memory words displayed in the Memory component are not updated when the target is stopped.

In **Periodical** mode, memory words are updated at regular time intervals when the target is running. The default update rate is 1 second, but it can be modified by steps of up to 100 ms using the associated dialog box or `UPDATERATE` command.

The `UPDATERATE` command sets the variables update rate (see also `UPDATERATE` command).

**NOTE** Also refer to SMEM, SPC and SMOD command descriptions for more detail about these commands.

### Equivalent Operations

- **ATTRIBUTES FORMAT** ~ Select menu `Memory>Format`
- **ATTRIBUTES WORD** ~ Select menu `Memory>Word Size`
- **ATTRIBUTES ADR** ~ Select menu `Memory>Display>Address`
- **ATTRIBUTES ASC** ~ Select menu `Memory>Display>ASCII`
- **ATTRIBUTES ADDRESS** ~ Select menu `Memory>Address...
- **ATTRIBUTES COMPLEMENT** ~ Select menu `Memory>Format`
- **ATTRIBUTES SMEM** ~ Drag and drop from Data component (variable) to Memory component.
- **ATTRIBUTES SMOD** ~ Drag and drop from Source component to Memory component.
- **ATTRIBUTES MODE** ~ Select menu `Memory>Mode...
- **ATTRIBUTES UPDATERATE** ~ Select menu `Memory>Mode>Periodical`

### Usage

**ATTRIBUTES** list

where **list=command[,command]**

- `FORMAT(bin|oct|hex|signed|unsigned)` | `WORD number` | `ADR (ON|OFF)` | `ASC (ON|OFF)` | `ADDRESS address` | `SPC address` | `SMEM range` | `SMOD module` | `MODE (automatic|periodical|frozen)` | `UPDATERATE rate` | `COMENT (NONE|ONE)`

### Example:

`Memory < ATTRIBUTES ASC OFF, ADR OFF`

ASCII dump and addresses are removed from the Memory component window.
In the Inspector Component

The **ATTRIBUTES** command allows you to set the display and state of the Inspector component window.

**Usage**

`ATTRIBUTES list`

where `list=command{,command}`

`command= COLUMNWIDTH columnname columnfield columnsize | EXPAND [name [subname]] deep | COLLAPSE name [subname] | SELECT name [subname] | SPLIT pos | MAXELEM ( ON | OFF ) [number] | FORMAT (Hex|Int)`

The **COLUMNWIDTH** command sets the width of one column entry on the right pane of the Inspector Window. The first parameter (columnname) specifies which column. The following column names currently exist:

- Names - simple name list
- Interrupts - interrupt list
- SymbolTableFunction - function in the Symbol Table
- ObjectPoolObject - Object in Object Pool without additional information
- Events - event list
- Components - component list
- SymbolTableVariable - variable or differentiation in the Symbol Table
- ObjectPoolIOBase - Object in Object Pool with additional information
- SymbolTableModules - non IOBase derived Object in the Object Pool

The column field is the name of the specific field, which is also displayed in the Inspector Window.

The following commands set the width of the function names to 100:

```
inspect < ATTRIBUTES COLUMNWIDTH SymbolTableModules Name 100
```

**NOTE** Due to the “inspect <“ redirection, only the Inspector handles this command.

The **EXPAND** command computes and displays all subitems of a specified item up to a given depth. An item is specified by specifying the complete path starting at one of the root items like “Symbol Table” or “Object Pool”. Names with spaces must be surrounded by double quotes.
To expand all subitems of TargetObject in the Object Pool up to 4 levels, the following command can be used:

```
inspect < ATTRIBUTES EXPAND "Object Pool" TargetObject 4
```

**NOTE**  
Because the name Object Pool contains a space, it must be surrounded by double quotes.

**NOTE**  
The symbol Table, Stack or other Items may have recursive information. So it may occur that the information tree grows with the depth. Therefore, specifying large expand values may use a large amount of memory.

The **COLLAPSE** command folds one item. The item name must be given. The following command folds the TargetObject:

```
inspect < ATTRIBUTES COLLAPSE "Object Pool" TargetObject
```

The **SELECT** command shows the information of the specified item on the right pane. The following command shows all Objects attached to the TargetObject:

```
inspect < ATTRIBUTES SELECT "Object Pool" TargetObject
```

The **SPLIT** command sets the position of the split line between the left and right pane. The value must be between 0 and 100. A value of 0 only shows the right pane, a value of 100 shows the left pane. Any value between 0 and 100 makes a relative split. The following command makes both panes the same size:

```
inspect < ATTRIBUTES SPLIT 50
```

The **MAXELEM** command sets the number of subitems to display. After the following command, the Inspector will prompt for 1000 subitems:

```
inspect < ATTRIBUTES MAXELEM ON 1000
```

The **FORMAT** command specifies whether integral values like addresses should be displayed as hexadecimal or decimal. The following command specifies the hexadecimal display:

```
inspect < ATTRIBUTES FORMAT Hex
```

**Equivalent Operations**

- ATTRIBUTES COLUMNWIDTH ~ Modify column width with the mouse.
- ATTRIBUTES EXPAND ~ Expand any item with the mouse.
- ATTRIBUTES COLLAPSE ~ Collapse the specified item with the mouse.
- ATTRIBUTES SELECT ~ Click on the specified item to select it.
- ATTRIBUTES SPLIT ~ Move the split line between the panes with the mouse.
- ATTRIBUTES MAXELEM ~ Select max. Elements... from the context menu.
**AT**

The AT command temporarily suspends a command file from executing until after a specified delay in milliseconds. The delay is measured from the time the command file is started. In the event that command files are chained (one calling another), the delay is measured from the time the first command file is started.

**NOTE** This command can only be executed from a command file. The time specified is relative to the start of command file execution.

**Usage**

AT time

where time=expression and expression is interpreted in milliseconds.

**Components**

Debugger engine.

**Example:**

AT 10 OPEN Command

This command (in command file) opens the Command Line component 10 ms after the command file is executed.

---

**AUTOSIZE**

AUTOSIZE enables/disables windows autosizing. When on, the size of component windows are automatically adapted to the Simulator/Debugger main window when it is resized.

**Usage**

AUTOSIZE on|off

**Components**

Debugger engine.

**Example:**

in> AUTOSIZE off

Windows autosizing is disabled.
BASE
In the Profiler component, the BASE command sets the profiler base to code (total code) or module (each module code).

Usage
BASE code|module

Components
Profiler component.

Example:
in>BASE code

BC
BC deletes a breakpoint at the specified address. When * is specified, all breakpoints are deleted.

You can point to the breakpoint in the Assembly or Source component window, right-click and choose Delete Breakpoint in the popup menu, or open the ControlPoints Window, select the breakpoint from the list and click Delete.

NOTE Correct module names are displayed in the Module component window. Make sure that the module name of your command is correct: if the .abs is in HIWARE format, some debug information is in the object file (.o), and module names have a .o extension (e.g., fibo.o). In ELF format, module name extensions are .c, .cpp or .dbg (.dbg for program sources in assembler) (e.g., fibo.c), since all debugging information is contained in the .abs file and object files are not used. Adapt the following examples with your .abs application file format.

Usage
BC address|*

address is the address of the breakpoint to be deleted. This address is specified in ANSI C or standard Assembler format. address can also be replaced by an expression as shown in the example below.

When * is specified all breakpoints are deleted.

Components
Debugger engine.
Debugger Engine Commands
Debugger Commands

Example 1:
in>BC 0x8000
This command deletes the breakpoint set at the address 0x8000. The breakpoint symbol is removed in the source and assembly window. The breakpoint is removed from the breakpoint list.

Example 2:
in>BC &FIBO.C:Fibonacci
In this example, an expression replaces the address. FIBO.C is the module name and Fibonacci is the function where the breakpoint is cleared.

BCKCOLOR

BCKCOLOR sets the background color.
The background color defined with the BCKCOLOR command is valid for all component windows. Avoid using the same color for the font and background, otherwise text in the component windows will not be visible. Also avoid using colors that have a specific meaning in the command line window. These colors are:
Red: used to display error messages.
Blue: used to echo commands.
Green: used to display asynchronous events.

NOTE When WHITE is given as a parameter, the default background color for all component windows is set, for example, the register component is lightgrey.

Usage

BCKCOLOR color
Where color can be one of the following: BLACK, GREY, LIGHTGREY, WHITE, RED, YELLOW, BLUE, CYAN, GREEN, PURPLE, LIGHTRED, LIGHTYELLOW, LIGHTBLUE, LIGHTCYAN, LIGHTGREEN, LIGHTPURPLE

Components

Debugger engine.

Example:
in>BCKCOLOR LIGHTCYAN
The background color of all currently open component windows is set to Lightcyan. To return to the original display, enter BCKCOLOR WHITE.
In the Command Line component, the **BD** command displays the list of all breakpoints currently set with addresses and types (temporary, permanent).

**Usage**

BD

**Components**

Debugger engine.

**Example:**

```plaintext
in>BD
Fibonacci 0x805c T
Fibonacci 0x8072 P
Fibonacci 0x8074 T
main 0x8099 T
```

One permanent and two temporary breakpoints are set in the function **Fibonacci**, and one temporary breakpoint is set in the **main** function.

**NOTE**  From the list, it is not possible to know if a breakpoint is disabled or not.
BS

BS sets a temporary (T) or a permanent (P) breakpoint at the specified address. If no P or T is specified, the default is a permanent (P) breakpoint.

Equivalent Operation

You can point at a statement in the Assembly or Source component window, right-click and choose Set Breakpoint in the popup menu, or open the Controlpoints Configuration Window and choose Show Breakpoint, then select the breakpoint and set its properties.

NOTE
Correct module names are displayed in the Module component window. Make sure that the module name of your command is correct:
If the .abs is in HIWARE format, some debug information is in the object file (.o), and module names have a .o extension (e.g., fibo.o). In ELF format, module name extensions are .c, .cpp or .dbg ( .dbg for program sources in assembler) (e.g., fibo.c), since all debugging information is contained in the .abs file and object files are not used. Adapt the following examples with .abs application file format.

Usage

BS address| function [[{mark}]]
[P|T [state]][;cond="condition"] [state]
[;cmd="command"] [state][;cur=current[ inter=interval]]
[;cdSz=codeSize[ srSz=sourceSize]]

address is the address where the breakpoint is to be set. This address is specified in ANSI C format. address can also be replaced by an expression as shown in the example below.

function is the name of the function in which to set the breakpoint.

mark (displayed mark in Source component window) is the mark number where the breakpoint is to be set. When mark is:
  • > 0: the position is relative to the beginning of the function.
  • = 0: the position is the entry point of the function (default value).
  • < 0: the position is relative to the end of the function.

P specifies the breakpoint as a permanent breakpoint.

T specifies the breakpoint as a temporary breakpoint. A temporary breakpoint is deleted once it is reached.
State is E or D where E is for enabled (state is set by default to E if nothing is specified), and D is for disabled.
**condition** is an expression. It matches the *Condition* field in the Controlpoints Configuration window for a conditional breakpoint.

**command** is any Debugger command (at this level, the commands `G`, `GO` and `STOP` are not allowed). It matches the *Command* field in the Controlpoints Configuration window, for associated commands. For the *Command* function, the states are `E` (enabled) or `C` (continue).

**current** is an expression. It matches the *Current* field (Counter) in the Controlpoints Configuration window, for counting breakpoints.

**interval** is an expression. It matches the *Interval* field (Counter) in the Controlpoints Configuration window, for counting breakpoints.

**codeSize** is an expression. It is usually a constant number to specify (for security) the code size of a function where a breakpoint is set. If the size specified does not match the size of the function currently loaded in the .ABS file, the breakpoint is set but disabled.

**sourceSize** is an expression. It is usually a constant number to specify (for security) the source (text) size of a function where a breakpoint is set. If the size specified does not match the size of the function in the source file, the breakpoint is set but disabled.

**Components**

Debugger engine.

**Example:**

```
in>BS 0x8000 T
```

This command sets a temporary breakpoint at the address 0x8000.

```
in>BS $8000
```

This command sets a permanent breakpoint at the address 0x8000.

```
BS &FIBO.C:Fibonacci
```

In this example, an expression replaces the address. `FIBO.C` is the module name and `Fibonacci` is the function where the breakpoint is set.

**More Examples:**

```
in>BS &main + 22 P E ; cdSz = 66 srSz = 134
```

Sets a breakpoint at the address of the main procedure + 22, where the code size of the main procedure is 66 bytes and its source size is 134 characters.

```
in>BS Fibo.c:main{3}
```

Sets a breakpoint at the 3rd mark of the procedure `main`, where `main` is a function of the `FIBO.C` module.

```
in>BS &counter + 5; cond ="fib1>fib2";cmd="bckcolor red"
```

Debugger Engine Commands

**CALL**

Executes a command in the specified command file.

**Usage**

CALL FileName [:C][;NL]

**Components**

Debugger engine.

**Example:**

```
in>cf \util\config.cmd
```

Loads the config command file.
The CD command changes the current working directory to the directory specified in path. When the command is entered with no parameter, the current directory is displayed. The directory specified in the CD command must be a valid directory. It should exist and be accessible from the PC. When specifying a relative path in the CD command, make sure the path is relative to the project directory.

**NOTE**

When no path is specified, the default directory is the project directory. When using the CD command, all commands referring to a file with no path specified could be affected.

**Usage**

CD [path]

**path:** The pathname of a directory that becomes the current working directory (case insensitive).

**Components**

Debugger engine.

**Example:**

```
in>cd..
C:\Freescale\demo
in>cd
C:\Freescale\demo
in>cd /Freescale/prog
C:\Freescale\prog
```

The new project directory is `C:\Freescale\prog`
Debugger Engine Commands

CF

The CF command reads the commands in the specified command file, which are then executed by the command interpreter. The command file contains ASCII text commands. Command files can be nested. By default, after executing the commands from a nested command file, the command interpreter resumes execution of remaining commands in the calling file. Any error halts execution of CF file commands. When the command is entered with no parameter, the Open File dialog is displayed. The CALL command is equivalent to the CF command.

NOTE If no path is specified, the destination directory is the current project directory.

Usage

CF fileName [;C][;NL]

Where fileName is a file (and path) containing Simulator/Debugger commands.

;C specifies chaining the command file. This option is meaningful in a nested command file only.

When the ;C option is given in the calling file, the command interpreter quits the calling file and executes the called file. (i.e. in the calling file, commands following the CF ... ;C command are never executed).

When the option is omitted, execution of the remaining commands in the calling file is resumed after the commands in the called file have been executed.

;NL: when set, the commands that are in the called file are not logged in the Command Line window (and not to log file, when a file has been opened with an LF command), even if the CMDFILE type is set to ON (see also the LOG command).

Components

Debugger engine.

Examples:

Example Without “;C” Option:

```
bckcolor green
cf command2.txt
```
if a command2.txt file contains:

bckcolor red

Execution:

in>cf command1.txt
executing command1.txt

!bckcolor green
!cf command2.txt
executing command2.txt

1!bckcolor red
1!
1!
done command2.txt

!bckcolor white
!
done command1.txt

---

**Example With “;C” Option:**

if a command1.txt file contains:

bckcolor green
cf command2.txt ;C
bckcolor white

if a command2.txt file contains:

bckcolor red

Execution:

in>cf command1.txt
executing command1.txt

!bckcolor green
!cf command2.txt ;C
executing command2.txt

1!bckcolor red
1!
CLOCK

In the SoftTrace component, the **CLOCK** command sets the clock speed.

**Usage**

CLOCK frequency

Where number is a decimal number, which is the CPU frequency in Hertz.

**Components**

SoftTrace component.

**Example:**

```
in>CLOCK 4000000
```

CLOSE

The **CLOSE** command is used to close a component.

Component names are: Assembly, Command, Coverage, Data, Inspect, IO_Led, Led, Memory, Module, Phone, Procedure, Profiler, Recorder, Register, SoftTrace, Source, Stimulation.

**Usage**

CLOSE component | *

where * means “all components”.

**Components**

Debugger engine.

**Example:**

```
in>CLOSE Memory
```

The Memory component window is closed (unloaded).
COPYMEM
The COPYMEM command is used to copy a memory range to a destination range defined by the beginning address. This command works on defined memory only. The source range and destination range are tested to ensure they are not overlayed.

Usage
COPYMEM <Source address range> dest-address

Components
Memory.

Example:
in>copymem 0x3FC2A0..0x3FC2B0 0x3FC300
The memory from 0x3FC2A0 to 0X3FC2B0 is copied to the memory at 0x3FC300 to 0x3FC310. This Memory range appears in red in the Memory Component.

CMDFILE
The CMDFILE command allows you to define all target specific commands in a command file. For example, startup, preload, reset, and path of this file.

Usage
CMDFILE <Command File Kind> ON|OFF ["<Command File Full Name>"]

Components
Simulator/target engine.

Example:
in>cmdfile postload on "c:\temp\mynosloadfile.cmd"
The mynosloadfile command file will be executed after loading the absolute file.
**CR**

The **CR** command initiates writing records of commands to an external file. Writing records continues until a close record file (**NOCR**) command is executed.

**NOTE** Drag & drop actions are also translated into commands in the record file.

**NOTE** If no path is specified, the destination directory is the current project directory.

**Usage**

```
CR [fileName][:A]
```

If **fileName** is not specified, a standard **Open File** dialog is opened.

;A specifies to open a file **fileName** in append mode. Records are appended at the end of an existing record file.

If the ;A option is omitted and **fileName** is an existing file, the file is cleared before records are written to it.

**Components**

Debugger engine.

**Example:**

```
in>cr /Freescale/demo/myrecord.txt ;A
```

The **myrecord.txt** file is opened in “Append” mode for a recording session.

---

**CYCLE**

In the **SoftTrace component**, the **CYCLE** command displays or hides cycles. When cycle is off, milliseconds (ms) are displayed.

**Usage**

```
CYCLE on|off
```

**Components**

Softtrace component.

**Example:**

```
in>CYCLE on
```
**DASM**

The **DASM** command displays the assembler code lines of an application, starting at the address given in the parameter. If there is no parameter, the assembler code following the last address of the previous display is displayed.

This command can be stopped by pressing the **Esc** key.

**Equivalent Operation**

Right-click in the Assembly component window, select **Address...** and enter the address to start disassembly in the **Show PC** dialog.

**Usage**

DASM [**address**|**range**][;**OBJ**]

- **address**: A constant expression representing the address where disassembly begins.
- **range**: An address range constant that specifies addresses to be disassembled. When **range** is omitted, a maximum of sixteen instructions are disassembled.

When **address** and **range** are omitted, disassembly begins at the address of the instruction that follows the last instruction that has been disassembled by the most recent **DASM** command. If this is the first **DASM** command of a session, disassembly begins at the current address in the program counter.

- **;OBJ**: Displays assembler code in hexadecimal.

**Components**

Debugger engine.

**Example:**

```
in>dasm 0xf04b
00F04B LDHX   #0x0450
00F04E TXS
00F04F CLRH
00F050 CLRX
00F051 STX   0x80
00F053 INC   0x80
00F055 LDX   0x80
00F057 JSR   0xF000
00F05A STX   0x82
00F05C STA   0x81
00F05E LDA   #0x17
00F060 CMP   0x80
00F062 BEQ   ***-20** /abs = F050
00F064 BRA   ***-19** /abs = F053
```
Debugger Engine Commands

NOTE

Depending on the target, the above code may vary.

Disassembled instructions are displayed in the Command Line component window. Therefore, it is necessary to open the Command Line component before executing this command to see the dumped code.

DB

The DB command displays the hexadecimal and ASCII values of the bytes in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first byte displayed in the line, followed by the number of specified hexadecimal byte values. The hexadecimal byte values are followed by the corresponding ASCII characters, separated by spaces. Between the eighth and ninth values, a hyphen (-) replaces the space as the separator. Each non-displayable character is represented by a period (.).

This command can be stopped by pressing the Esc key.

Usage

DB [address|range]

When address and range are omitted, the first longword displayed is taken from the address following the last longword displayed by the most recent DB, DW, or DL command, or from address 0x0000 (for the first DB, DW, DL command of a session).

Examples:

in>DB 0x8000..0x800F

8000: FE 80 45 FD 80 43 27 10-35 ED 31 EC 31 69 70 83 þ_Eý_C'.5í1ì1ipƒ

Memory bytes are displayed in the Command Line component window, with matching ASCII characters. So, it is necessary to open the Command Line component before executing this command to see the dumped code.

in>DB &TCR

0012: 5AZ

displays the byte that is at the address of the TCR I/O register. I/O registers are defined in a DEFAULT.REG file.
DDEPROTOCOL

The DDEPROTOCOL command is used to configure the Debugger/Simulator dynamic data exchange (DDE) protocol.

By default the DDE protocol is activated and not displayed in the command line component.

Usage

DDEPROTOCOL ON|OFF|SHOW|HIDE|STATUS

Where:

- ON enables the DDE communication protocol
- OFF disables the DDE communication protocol
- SHOW displays DDE protocol information in the command line component
- HIDE hides DDE protocol information in the command line component
- STATUS provides information if the DDE protocol is active (on or off) and if display is active (Show or Hide)

Components

Debugger engine.

Example:

```
$> DDEPROTOCOL ON
$> DDEPROTOCOL SHOW
$> DDEPROTOCOL STATUS
DDEPROTOCOL ON - DISPLAYING ON
```

The DDE protocol is activated and displayed, and status is given in the command line component.

NOTE  For more information on Debugger/Simulator DDE implementation, please refer to the chapter that deals with Debugger DDE Capabilities.
**DEFINE**

The DEFINE command creates a symbol and associates the value of an expression with it. Arithmetic expressions are evaluated when the command is interpreted. The symbol can be used to represent the expression until the symbol is redefined, or undefined using the UNDEF command. A symbol is a maximum of 31 characters long. In a command line, all symbol occurrences (after the command name) are substituted by their values before processing starts. A symbol cannot represent a command name. Note that a symbol definition precedes (and hence conceals) a program variable with the same name.

Defined symbols remain valid when a new application is loaded. An application variable or I/O register can be overwritten with a DEFINE command.

**NOTE**  
This command can be used to assign meaningful names to expressions, which can be used in other commands. This increases the readability of command files and avoids re-evaluation of complex expressions.

**Usage**

DEFINE symbol [=] expression

**Components**

Debugger engine.

**Example:**

```plaintext
in>DEFINE addr $1000
in>DEFINE limit = addr + 15
```

First addr is defined as a constant equivalent to $1000. Then limit is defined and affected with the value ($1000 + 15)

A symbol defined in the loaded application can be redefined on the command line using the DEFINE command. The symbol defined in the application is not accessible until an UNDEF on that symbol name is detected in the command file.
Example:
A symbol named ‘testCase’ is defined in the test application.

```c
/* Loads application test.abs */
LOAD test.abs
/* Display value of testCase. */
DB testCase
/* Redefine symbol testCase. */
DEFINE testCase = $800
/* Display value stored at address $800. */
DB testCase
/* Redefine symbol testCase. */
UNDEF testCase
/* Display value of testCase. */
DB testCase
```

NOTE Also refer to examples given for the command UNDEF.

DETAILS
In the Profiler component, the DETAILS command opens a profiler split view in the Source or Assembly component.

Usage
DETAILS assembly|source

Components
Profiler components.

Example:
in>DETAILS source
Debugger Engine Commands

### DL

The **DL** command displays the hexadecimal values of the longwords in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first longword displayed in the line, followed by the number of specified hexadecimal longword values.

When a size is specified in the range, this size represents the number of longwords that should be displayed in the command line window.

This command can be stopped by pressing the **Esc** key.

**NOTE**  
Open the Command Line component before executing this command to see the dumped code.

#### Usage

**DL** [address|range]

When **range** is omitted, the first longword displayed is taken from the address following the last longword displayed by the most recent **DB, DW, or DL** command, or from address 0x0000 (for the first **DB, DW, DL** command of a session).

#### Components

Debugger engine.

#### Example:

```plaintext
in>DL 0x8000..0x8007
8000: FE8045FD 80432710
```

The content of the memory range starting at 0x8000 and ending at 0x8007 is displayed as longword (4-bytes) values.

```plaintext
in>DL 0x8000,2
8000: FE8045FD 80432710
```

The content of 2 longwords starting at 0x8000 is displayed as longword (4-bytes) values.

Memory longwords are displayed in the Command Line component window.
DUMP

The DUMP command writes everything visible in the Data component to the command line component.

Usage
DUMP

Components
Data component.

Example:
\texttt{in > Data:1 < DUMP}

DW

The DW command displays the hexadecimal values of the words in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first word displayed in the line, followed by the number of specified hexadecimal word values.

When a size is specified in the range, this size represents the number of words that should be displayed in the command line window.

This command can be stopped by pressing the \texttt{Esc} key.

\textbf{NOTE} Open the Command Line component before executing this command to see the dumped code.

Usage
\texttt{DW \{address \mid range\}}

When \texttt{address} is an address constant expression, the address of the first word is displayed.

When \texttt{address} and \texttt{range} are omitted, the first word displayed is taken from the address following the last word displayed by the most recent \texttt{DB}, \texttt{DW}, or \texttt{DL} command, or from address \texttt{0x0000} (for the first \texttt{DB}, \texttt{DW}, \texttt{DL} command of a session).

Components
Debugger engine.

Example:
\texttt{in > DW 0x8000,4}
The content of 4 words starting at 0x8000 is displayed as word (2-bytes) values.
Memory words are displayed in the Command Line component window.

**E**
The E command evaluates an expression and displays the result in the Command Line component window. When the expression is the only parameter entered (no option specified) the value of the expression is displayed in the default number base. The result is displayed as a signed number in decimal format and as unsigned number in all other formats.

**Usage**

E expression[;O|D|X|C|B]

where:

;O: displays the value of expression as an octal (base 8) number.
;D: displays the value of expression as a decimal (base 10) number.
;X: displays the value of expression as an hexadecimal (base 16) number.
;C: displays the value of expression as an ASCII character. The remainder resulting from dividing the number by 256 is displayed. All values are displayed in the current font. Control characters (<32) are displayed as decimal.
;B: displays the value of expression as a binary (base 2) number.

**Components**
Debugger engine.

**Example:**

```plaintext
in>define a=0x12
in>define b=0x10
in>e a+b
in>=34
```

The addition operation of the two previously defined variables a and b is evaluated and the result is displayed in the Command Line window. The output can be redirected to a file by using the LF command (refer to LF and LOG command descriptions).
**ELSE**

The `ELSE` keyword is associated with the `LF` command.

**Usage**

Else

**Components**

Debugger engine.

**Example:**

```plaintext
if CUR_TARGET == 1000         /* Condition */
    set sim
else set bdi                 /* Other Condition */
```

**ELSEIF**

The `ELSEIF` keyword is associated with the `IF` command.

**Usage**

ELSEIF condition

where `condition` is same as defined in “C” language.

**Components**

Debugger engine.

**Example:**

```plaintext
if CUR_TARGET == 1000         /* Simulator */
    set sim
elseif CUR_TARGET == 1001  /* BDI */
    set bdi
```
### ENDFOCUS

The **ENDFOCUS** command resets the current focus. It is associated with the **FOCUS** command. Following commands are broadcast to all currently open components. This command is only valid in a command file.

#### Usage

ENDFOCUS

#### Components

Debugger engine.

#### Example:

```
FOCUS Assembly
ATTRIBUTES code on
ENDFOCUS
FOCUS Source
ATTRIBUTES marks on
ENDFOCUS
```

The **ATTRIBUTES** command is first redirected to the Assembly component by the **FOCUS** Assembly command. The code is displayed next to assembly instructions. Then the Assembly component is released by the **ENDFOCUS** command and the second **ATTRIBUTES** command is redirected to the Source component by the **FOCUS** Source command. Marks are displayed in the Source window.
**ENDFOR**

**Description**
The `ENDFOR` keyword is associated with the `FOR` command.

**Usage**

```plaintext
ENDFOR
```

**Components**
Debugger engine.

**Example:**
```
for i = 1..5
    define multi5 = 5 * i
endfor
```

After the `ENDFOR` instruction, `i` is equal to 5.

**ENDIF**

**Description**
The `ENDIF` keyword is associated with the `IF` command.

**Usage**

```plaintext
ENDIF
```

**Components**
Debugger engine.

**Example:**
```
if (CUR_CPU == 12)
    DW &counter
else
    DB &counter
endif
```
**ENDWHILE**

**Description**

The ENDWHILE keyword is associated with the WHILE command.

**Usage**

ENDWHILE

**Components**

Debugger engine.

**Example:**

```plaintext
while i < 5
    define multi5 = 5 * i
    define i = i + 1
endwhile
```

After the ENDWHILE instruction, i is equal to 5

**EXECUTE**

**Description**

In the Stimulation component, the EXECUTE command executes a file containing stimulation commands. Refer to the I/O Stimulation document.

**Usage**

EXECUTE fileName

**Components**

Stimulation component.

**Example:**

```plaintext
in>EXECUTE stimu.txt
```
EXIT

Description
In the Command line component, the EXIT command closes the Debugger application.

Usage
EXIT

Components
Debugger engine.

Example:
in>EXIT
The Debugger application is closed.

FILL

Description
In the Memory component, the FILL command fills a corresponding range of Memory component with the defined value. The value must be a single byte pattern (higher bytes ignored).

Usage
FILL range value
the syntax for range is: LowAddress..HighAddress

Components
Memory component.

Equivalent Operation
The File Memory dialog is available from the Memory popup menu and by selecting Fill... or Memory>Fill... menu entry.

Example:
in>FILL 0x8000..0x8008 0xFF
The memory range 0x8000..0x8008 is filled with the value 0xFF.
FILTER

Description
In the Memory component, with the FILTER command, you select what you want to display, for example modules: modules only, functions: modules and functions, or lines: modules and functions and code lines. You can also specify a range to be logged in your file. Range must be between 0 and 100.

Usage
FILTER Options [<range>]
Options = modules|functions|lines

Components
Coverage component.

Example:

in>coverage < FILTER functions 25..75
FIND

Description
In the Source component, the FIND command is used to search a specified pattern in the source file currently loaded. If the pattern has been found, it is highlighted. The search is forward (default), backward (;B), match case sensitive (;MC) or match whole word sensitive (;WW). The operation starts form the currently highlighted statement or from the beginning of the file (if nothing is highlighted). If the item is found, the Source window is scrolled to the position of the item and the item is highlighted in grey.

Equivalent Operation
You can select Source>Find... or open the Source popup menu and select Find... to open the Find dialog.

Usage
FIND “string” [:B] [:MC] [:WW]
Where string is the “pattern” to match. It has to be enclosed in double quotes. See the example below.
;B the search is backwards, default is forwards.
;MC match case sensitive is set.
;WW match whole word is set.

Components
Source component.

Example:
in> FIND "this" ;B ;WW
The “this” string (considered as a whole word) is searched in the Source component window. The search is performed backward.
FINDPROC

Description
If a valid procedure name is given as parameter, the source file where the procedure is defined is opened in the Source Component. The procedure’s definition is displayed and the procedure’s title is highlighted.

Equivalent Operation
You can select Source>Find Procedure... or open the Source popup menu and select Find Procedure... to open the Find Procedure dialog.

Usage
FINDPROC procedureName

Components
Source component.

Example:
in>findproc Fibonacci
The “Fibonacci” procedure is displayed and the title is highlighted.
**FOCUS**

**Description**

The **FOCUS** command sets the given component (component) as the destination for all subsequent commands up to the next **ENDFOCUS** command. Hence, the focus command releases the user from repeatedly specifying the same command redirection, especially in the case where command files are edited manually. This command is only valid in a command file.

**NOTE**

It is not possible to visually notice that a component is “FOCUSed”. However, you can use the **ACTIVATE** command to activate a component window.

**Usage**

FOCUS component

**Components**

Debugger engine.

**Example:**

FOCUS Assembly  
ATTRIBUTES code on  
ENDFOCUS  
FOCUS Source  
ATTRIBUTES marks on  
ENDFOCUS

The ATTRIBUTES command is first redirected to the Assembly component by the **FOCUS** Assembly command. The code is displayed next to assembly instructions. Then the Assembly component is released by the ENDFOCUS command and the second ATTRIBUTES command is redirected to the Source component by the **FOCUS** Source command. Marks are displayed in the Source window.
FOLD

Description
In the Source component, the FOLD command hides the source text at the program block level. Folded program text is displayed as if the program block was empty. When the folded block is unfolded, the hidden program text reappears. All text is folded once or (*) completely, until there are no more folded parts.

Usage
FOLD [*]
Where * means fold completely, otherwise fold only one level.

Components
Source component.

Example:
in>FOLD *

FONT

Description
FONT sets the font type, size and color.

Equivalent Operation
The Font dialog is available by selecting the Component>Fonts... menu entry.

Usage
FONT ‘FontName’ [size][color]

Components
Debugger engine.

Example:
FONT ‘Arial’ 8 BLUE
The font type is “Arial” 8 points and blue.
FOR

Description
The FOR loop allows you to execute all commands up to the trailing ENDFOR a predefined number of times. The bounds of the range and the optional steps are evaluated at the beginning. A variable (either a symbol or a program variable) may be optionally specified, which is assigned to all values of the range that are met during execution of the for loop. If a variable is used, it must be defined before executing the FOR command, with a DEFINE command.

Assignment happens immediately before comparing the iteration value with the upper bound. The variable is only a copy of the internal iteration value, therefore modifications on the variable don't have an impact on the number of iterations.

This command can be stopped by pressing the Esc key.

Usage
FOR[variable =]range [";" step]
Where variable is the name of a defined variable.
range: This is an address range constant that specifies addresses to be disassembled.
step: constant number matching the step increment of the loop.

Components
Debugger engine.

Example:

```
DEFINE loop = 0
FOR loop = 1..6,1
T
ENDFOR
```

The T Trace command is performed 6 times.
Debugger Engine Commands

Debugger Commands

FPRINTF

Description
FPRINTF is the standard ANSI C command: Writes formatted output string to a file.

Usage
FPRINTF (<filename>, <&format>, <expression>, <expression>, ...)

Components
Debugger engine.

Example:
`fprintf (test.txt,"%s %2d","The value of the counter is:",counter)`
The content of the file `test.txt` is: The value of the counter is: 25

FRAMES

Description
In the SoftTrace component, the FRAMES command sets the maximum number of frame records.

Usage
FRAMES number
Where number is a decimal number, which is the maximum number of recorded frames.
This number must not exceed 100000.

Components
SoftTrace component.

Example:
FRAMES 10000
**G**

**Description**

The **G** command starts code execution in the emulated system at the current address in the program counter or at the specified address. You can therefore specify the entry point of your program, skipping execution of the previous code.

**Usage**

**G** [address]

When no **address** is entered, the address in the program counter is not altered and execution begins at the address in the program counter.

**Alias**

**GO**

**Components**

Debugger engine.

**Example:**

```
G 0x8000
```

Program execution is started at 0x8000. **RUNNING** is displayed in the status bar. The application runs until a breakpoint is reached or you stop the execution.
### GO

**Description**

The GO command starts code execution in the emulated system at the current address in the program counter or at the specified address. You can therefore specify the entry point of your program, skipping execution of previous code.

**Usage**

GO [address]

When no address is entered, the address in the program counter is not altered and execution begins at the address in the program counter.

**Alias**

G

**Components**

Debugger engine.

**Example:**

in>GO 0x8000

Program execution is started at address 0x8000. RUNNING is displayed in the status bar. The application runs until a breakpoint is reached or you stop execution.
GOTO

Description
The GOTO command diverts execution of the command file to the command line that follows the Label. The Label must be defined in the current command file. The GOTO command fails, if the Label is not found. A label can only be followed on the same line by a comment.

Usage
GOTO Label

Components
Debugger engine.

Example:

GOTO MyLabel

...  
...

MyLabel:  // comments

When the instruction GOTO MyLabel is reached, the program pointer jumps to MyLabel and follows program execution from this position.
Debugger Engine Commands
Debugger Commands

**GOTOIF**

**Description**
The **GOTOIF** command diverts execution of the command file to the command line that follows the label if the condition is true. Otherwise, the command is ignored. The **GOTOIF** command fails, if the condition is true and the label is not found.

**Usage**
GOTOIF condition Label
where condition is same as defined in “C” language.

**Components**
Debugger engine.

**Example:**
```
DEFINE jump = 0
...
DEFINE jump = jump + 1
...
GOTOIF jump == 10 MyLabel
T
...
MyLabel: // comments
```

The program pointer jumps to MyLabel only if jump equals 10. Otherwise, the next instruction (T Trace command) is executed.
GRAPHICS

Description
In the Profiler component, GRAPHICS switches the percentages display in the graph bar on/off.

Usage
GRAPHICS on|off

Components
Profiler component.

Example:
in>GRAPHICS off

HELP
In the Command line component, the HELP command displays all available commands. Subcommands from the ATTRIBUTES command are not listed. Component specific commands, which are not open, will not be listed either.

Usage
HELP

Components
Debugger engine.

Example:
in>HELP

HI-WAVE Engine:
VER
LF
NOLF
CR
NOCR
....
**Debugger Engine Commands**

**Debugger Commands**

**IF**

The conditional commands (IF, ELSEIF, ELSE and ENDIF) allow you to execute different sections depending on the result of the corresponding condition. The conditional command may be nested. Conditions of the IF and ELSEIF commands, respectively, guard all commands up to the next ELSEIF, ELSE or ENDIF command on the same nesting level. The ELSE command guards all commands up to the next ENDIF command on the same nesting level. Any occurrence of a subcommand not in sequence of “IF, zero or more ELSEIF, zero or one ELSE, ENDIF” is an error.

**Usage**

IF condition

Where condition is same as defined in “C” language.

**Components**

Debugger engine.

**Example:**

```plaintext
DEFINE jump = 0
...
DEFINE jump = jump + 1
...
IF jump == 10
    T
    DEFINE jump = 0
ELSEIF jump == 100
    DEFINE jump = 1
ELSE
    DEFINE jump = 2
ENDIF
```

The jump == 10 condition is evaluated and depending on the test result, the T Trace instruction is executed, or the ELSEIF jump == 100 test is evaluated.
**INSPECTOROUTPUT**

The Inspector dumps the content of the specified item and all computed subitems to the command window. Uncomputed subitems are not printed. To compute all information, the `ATTRIBUTES EXPAND` command is used.

**Usage**

`INSPECTOROUTPUT [name {subname}]`

The *name* specifies any of the root items. The *subname* specifies a recursive path to subitems.

If a name contains a space, it must be surrounded by double quotes ("").

**Components**

Inspector component.

**Example:**

```
in>loadio swap
in>Inspect<ATTRIBUTES EXPAND 3
in>INSPECTOROUTPUT “Object Pool” Swap
```

<table>
<thead>
<tr>
<th>Swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Name Value Address Init...</td>
</tr>
</tbody>
</table>
| - IO_Reg_1 0x0 0x1000 0x0 ...
| - IO_Reg_2 0x0 0x1001 0x0 ...

**INSPECTORUPDATE**

The Inspector displays various information. Some types of information are automatically updated. To make sure that displayed values correspond to the current situation, the `INSPECTORUPDATE` command updates all information.

**Usage**

`INSPECTORUPDATE`

**Components**

Inspector component.

**Example:**

```
in>INSPECTORUPDATE
```
**LF**

The LF command initiates logging of commands and responses to an external file or device. While logging remains in effect, any line that is appended to the command window is also written to the log file.

Logging continues until a close log file (NOLF) command is executed. When the LF command is entered with no filename, the Open File Dialog is displayed to specify a filename.

Use the logging option (LOG) command to specify information to be logged.

If a path is specified in the file name, this path must be a valid path. When a relative path is specified, ensure that the path is relative to the project directory.

**Usage**

LF [fileName][;A]

fileName is a DOS filename that identifies the file or device where the log is written. The command interpreter does not assume a filename extension.

;A opens the file in append mode. Logged lines are appended at the end of an existing log file.

If the ;A option is omitted and fileName is an existing file, the file is cleared before logging begins.

**Components**

Debugger engine.

**Example**

in>lf /mcuez/demo/logfile.txt ;A

The logfile.txt file is opened as a Log File in “append” mode.

**NOTE** If no path is specified, the destination directory is the current project directory.
LOAD

The **LOAD** command loads a framework application (.abs file) for a debugging session. When no application name is specified, the **LoadObjectFile** dialog is opened.

If no target is installed, the following error message is displayed:

"Error: no target is installed"

If no target is connected, the following error message is displayed:

"Error: no target is connected"

**Usage**

LOAD[applicationName] [CODEONLY]$SYMBOLSONLY$ [NOPROGRESSBAR] [NOBPT] [NOXPR] [NOPRELOADCMD] [NOPOSTLOADCMD] [DELAY] [VERIFYFIRST]$VERIFYALL$[VERIFYONLY] [VERIFYOPTIONS][SYMBOLSOPTIONS]

Where:

- **applicationName** is the name of the application to load
- **CODEONLY** and **SYMBOLSONLY** loads only the code or symbols
- **NOPROGRESSBAR** loads the application without progress bar
- **NOBPT** loads the application without loading breakpoints file (with BPT extension)
- **NOXPR** loads the application without playing Expression file (with XPR extension)
- **NOPRELOADCMD** loads the application without playing PRELOAD file
- **NOPOSTLOADCMD** loads the application without playing POSTLOAD file
- **DELAY** loads the application and waits one second
- **VERIFYFIRST** matches the "First bytes only" code verification option.
- **VERIFYALL** matches the "All bytes" code verification option.
- **VERIFYONLY** matches the "Read back only" code verification option.
- **VERIFYOPTIONS** displays the "Code Verification Options" group in the "Load Executable File" dialog. If this option is missing, the group is not displayed. However, the verification mode can still be specified with options above.
- **SYMBOLSOPTIONS** displays the "Load Options" group in the "Load Executable File" dialog. If this option is missing, the group is not displayed. However, the code+symbols mode can still be specified with options CODEONLY and SYMBOLSONLY.

**NOTE** By default, the LOAD command is "code+symbols" with no verification.
Debugger Engine Commands

NOTE If the "SYMBOLSONLY" parameter is passed, verification parameters are ignored and NO verification is performed.

Components
Debugger engine.

Example:
LOAD FIBO.ABS
The FIBO.ABS application is loaded.

NOTE If no path is specified, the destination directory is the current project directory.

LOADCODE
This command loads code into the target system. This command can be used if no debugging is needed. If no target is installed, the following error message is displayed:
“Error: no target is installed”
If no target is connected, the following error message is displayed:
“Error: no target is connected”

Usage
LOADCODE [applicationName]

Components
Debugger engine.

Example:
LOADCODE FIBO.ABS
Code of the FIBO.ABS application is loaded.

NOTE If no path is specified, the destination directory is the current project directory.

LOADSYMBOLS
This command is similar to the LOAD command but only loads debugging information into the debugger. This can be used if the code is already loaded into the target system or programmed into a non-volatile memory device.
If no target is installed, the following error message is displayed:

“Error: no target is installed”

If no target is connected, the following error message is displayed:

“Error: no target is connected”

Usage

LOADSYMBOLES [applicationName]

Components

Debugger engine.

Example:

LOADSYMBOLES FIBO.ABS

Debugging information of the FIBO.ABS application is loaded. If no path is specified, the destination directory is the current project directory.

LOG

The LOG command enables or disables logging of information in the Command Line component window (and to logfile, when it as been opened with an LF command). If LOG is not used, all types are ON by default i.e. all information is logged in the Command Line component and log file.

NOTE  - about RESPONSES: Responses are results of commands. For example, for the DB command, the displayed memory dump is the response of the command. Protocol messages are not responses. - about ERRORS: Errors are displayed in red in Command Line component. Protocol messages are not errors. - about NOTICES: Notices are displayed in green in the Command Line.

Usage

LOG type [=] state [,] type [=] state

Where type is one of the following types:

CMDLINE: Commands entered on the command line.
CMDFILE: Commands read from a file.
RESPONSES: Command output response.
ERRORS: Error messages.
NOTICES: Asynchronous event notices, such as breakpoints.
Where \texttt{state} is \texttt{on} or \texttt{off}.

\texttt{state} is the new state of \texttt{type}:
- When \texttt{ON}, enables logging of the \texttt{type}.
- When \texttt{OFF}, disables logging of the \texttt{type}.

**Components**

Debugger engine.

**Example:**

\texttt{LOG ERRORS = OFF, CMDLINE = on}

Error messages are not recorded in the Log File. Commands entered in the Command Line component window are recorded.

**More About Logging of IF, FOR, WHILE and REPEAT**

When commands executed from a command file are logged, all executed commands that are in a \texttt{IF} block are logged. That is, a command file executed with the \texttt{CF} or \texttt{CALL} command without the \texttt{NL} option and with \texttt{CMDFILE} flag of the \texttt{LOG} command set to \texttt{TRUE}. All commands in a block that are not executed because the corresponding condition is false are also logged but preceded with a “-”.

**Example:**

When executing the following command file:

```plaintext
define truth = 1
IF truth
  bckcolor blue
  at 2000 bckcolor white
else
  bckcolor yellow
  at 1000 bckcolor white
ENDIF
```

The following log file is generated:

```plaintext
!define truth = 1
!IF truth
  !  bckcolor blue
  !  at 2000 bckcolor white
else
  !  bckcolor yellow
  !  at 1000 bckcolor white
!ENDIF
```
When commands executed from a command file are logged, all executed commands that are in the **FOR** loop are logged the number of times they have been executed. That is, a command file executed with the **CF** or **CALL** command without the **NL** option and with the **CMDFILE** flag of the **LOG** command set to **TRUE**.

**Example 2:**

When executing the following file:

```plaintext
define i = 1
FOR i = 1..3
   ls
ENDFOR
```

The following log file is generated:

```
!define i = 1
!FOR i = 1..3
!   ls
!       i 0x1 (1)
!ENDFOR
!   ls
!       i 0x2 (2)
!ENDFOR
!   ls
!       i 0x3 (3)
!ENDFOR
```

When commands executed from a command file are logged, all executed commands that are in the **WHILE** loop are logged the number of times they have been executed. That is, a command file executed with the **CF** or **CALL** command without the **NL** option and with the **CMDFILE** flag of the **LOG** command set to **TRUE**.

**Example 3:**

When executing the following file:

```plaintext
define i = 1
WHILE i < 3
   define i = i + 1
   ls
ENDWHILE
```

```
```
The following log file is generated:

```plaintext
!define i = 1
!WHILE i < 3
  ! define i = i + 1
  ! ls
  i    0x2 (2)
!ENDWHILE
  ! define i = i + 1
  ! ls
  i    0x3 (3)
!ENDWHILE
```

When commands executed from a command file are logged, all executed commands that are in the `REPEAT` loop are logged the number of times they have been executed. That is, a command file executed with the `CF` or `CALL` command without the `NL` option and with the `CMDFILE` flag of the `LOG` command set to `TRUE`.

**Example 4:**

When executing the following file:

```plaintext
define i = 1
REPEAT
  define i = i + 1
ls
UNTIL i == 4
```

The following log file is generated:

```plaintext
repeat
until condition
!define i = 1
!REPEAT
  ! define i = i + 1
  ! ls
  i    0x2 (2)
!UNTIL i == 4
  ! define i = i + 1
  ! ls
  i    0x3 (3)
!UNTIL i == 4
  ! define i = i + 1
  ! ls
  i    0x4 (4)
!UNTIL i == 4
```
**LS**

In the Command Line window, the LS command lists the values of symbols defined in the symbol table and by the user. There is no limit to the number of symbols that can be listed. The size of memory determines the symbol table size. Use the DEFINE command to define symbols, and the UNDEF command to delete symbols.

The symbols that are listed with the LS command are split in two parts: Applications Symbols and User Symbols.

**Usage**

LS [symbol | *][;C|S]

Where **symbol** is a restricted regular expression that specifies the symbol whose values are to be listed.

* specifies to list all symbols.

;C specifies to list symbols in canonical format, which consists of a DEFINE command for each symbol.

;S specifies to list symbol table statistics following the list of symbols.

**Components**

Debugger engine.

**Example:**

in>ls

<table>
<thead>
<tr>
<th>User Symbols:</th>
<th>Application Symbols:</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>0x2 (2)</td>
</tr>
<tr>
<td>counter</td>
<td>0x80 (128)</td>
</tr>
<tr>
<td>fiboCount</td>
<td>0x81 (129)</td>
</tr>
<tr>
<td>j</td>
<td>0x83 (131)</td>
</tr>
<tr>
<td>n</td>
<td>0x84 (132)</td>
</tr>
<tr>
<td>fib1</td>
<td>0x85 (133)</td>
</tr>
<tr>
<td>fib2</td>
<td>0x87 (135)</td>
</tr>
<tr>
<td>fibo</td>
<td>0x89 (137)</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>0xF000 (61440)</td>
</tr>
<tr>
<td>Entry</td>
<td>0xF041 (61505)</td>
</tr>
</tbody>
</table>

When LS is performed on a single symbol (e.g., **in>ls counter**) that is an application variable as well as a user symbol, the application variable is displayed.
Debugger Engine Commands

Example with j being an application symbol as well as a user symbol:
in>ls j

Application Symbol:
j  0x83 (131)

MEM

The MEM command displays a representation of the current system memory map and lower and upper boundaries of the internal module that contains the MCU registers.

Usage

MEM

Components

Debugger engine.

Example:
in>mem

<table>
<thead>
<tr>
<th>Type</th>
<th>Addresses</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>0.. 3F</td>
<td>PRU or TOP TOP board resource or the PRU</td>
</tr>
<tr>
<td>NONE</td>
<td>40.. 4F</td>
<td>NONE</td>
</tr>
<tr>
<td>RAM</td>
<td>50.. 64F</td>
<td>RAM</td>
</tr>
<tr>
<td>NONE</td>
<td>650.. 7FF</td>
<td>NONE</td>
</tr>
<tr>
<td>EEPROM</td>
<td>800.. A7F</td>
<td>EEPROM</td>
</tr>
<tr>
<td>NONE</td>
<td>A80..3DF</td>
<td>NONE</td>
</tr>
<tr>
<td>ROM</td>
<td>3E00..FDFF</td>
<td>ROM</td>
</tr>
<tr>
<td>IO</td>
<td>FE00..FE1F</td>
<td>PRU or TOP TOP board resource or the PRU</td>
</tr>
<tr>
<td>NONE</td>
<td>FE20..FFD</td>
<td>NONE</td>
</tr>
<tr>
<td>ROM</td>
<td>FFDC..FFFE</td>
<td>ROM</td>
</tr>
<tr>
<td>COP</td>
<td>FFFF..FFFF</td>
<td>special ram for cop</td>
</tr>
<tr>
<td>RT MEM</td>
<td>0.. 3FF</td>
<td>(enabled)</td>
</tr>
</tbody>
</table>
The `MS` command sets a specified block of memory to a specified list of byte values. When the `range` is wider than the `list` of byte values, the `list` of byte values is repeated as many times as necessary to fill the memory block.

When the `range` is not an integer multiple of the length of the `list`, the last copy of the `list` is truncated appropriately. This command is identical to the write bytes (WB) command.

**Usage**

MS range list

*range*: is an address range constant that defines the block of memory to be set to the values of the bytes in the list.

*list*: is a list of byte values to be stored in the block of memory.

**Components**

Debugger engine.

**Example:**

```plaintext
in>MS 0x1000..0x100F 0xFF
```

The memory range between addresses 0x1000 and 0x100F is filled with the 0xFF value.
NB

Description
The NB command changes or displays the default number base for the constant values in expressions. The initial default number base is 10 (decimal) and can be changed to 16 (hexadecimal), 8 (octal), 2 (binary) or reset to 10 with this command. The base is always specified as a decimal constant.

Independent of the default base number, the ANSI C standard notation for constant is supported inside an expression. That means that independent of the current number base you can specify hexadecimal or octal constants using the standard ANSI C notation shown in Table 20.6.

Usage
NB [base]
Where base is the new number base (2, 8, 10 or 16).

Components
Debugger engine.

Table 20.6  ANSI C Constant Notation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x----</td>
<td>Hexadecimal constant</td>
</tr>
<tr>
<td>0----</td>
<td>Octal constant</td>
</tr>
</tbody>
</table>

Table Example:

0x2F00, /* Hexadecimal Constant */

043,   /* Octal Constant */

255    /* Decimal Constant */

In the same way, the Assembler notation for constant is also supported. That means that independent of the current number base you can specify hexadecimal, octal or binary constants using the Assembler prefixes shown in Table 20.7.
Table Example:

$2F00, /* Hexadecimal Constant */
@43,  /* Octal Constant */
%10011 /* Binary Constant */

When the default number base is 16, constants starting with a letter A, B, C, D, E or F must be prefixed either by 0x or by $, as shown in Table 20.8. Otherwise, the command line interpreter cannot detect if you are specifying a number or a symbol.

Table 20.7 Assembler Notation for Constant

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-----</td>
<td>Hexadecimal constant</td>
</tr>
<tr>
<td>@-----</td>
<td>Octal constant</td>
</tr>
<tr>
<td>%-----</td>
<td>Binary constant</td>
</tr>
</tbody>
</table>

Table Example:

in>NB 16
The number base is hexadecimal.
**NOCR**
The **NOCR** command closes the current record file. The record file is opened with the **CR** command.

**Usage**
NOCR

**Components**
Debugger engine.

**Example:**
in>NOCR
The current record file is closed.

**NOLF**
The **NOLF** command closes the current Log File. The log file is opened with the **LF** command.

**Usage**
NOLF

**Components**
Debugger engine.

**Example:**
in>NOLF
The current Log File is closed.
OPEN

The OPEN command is used to open a window component.

Usage

OPEN "component" [x y width height][;I | ;MAX]

where:

- component is the component name with an optional path
- x is the X-axis of the upper left corner of the window component
- y is the Y-axis of the upper left corner of the window component
- width is the width of the window component
- height the height of the window component

When I is specified, the component window will be iconized; when MAX is specified, the component window will be maximized.

Component names are: Assembly, Command, Coverage, Data, Inspect, IO_Led, Led, Memory, Module, Phone, Procedure, Profiler, Recorder, Register, SoftTrace, Source, Stimulation.

Components

Debugger engine.

Example:

in>OPEN Terminal 0 78 60 22

The Terminal component and window is opened at specified positions and with specified width and height.
OUTPUT

With OUTPUT, you can redirect the Coverage component results to an output file indicated by the file name and his path.

Usage
OUTPUT FileName
Where FileName is file name (path + name).

Components
Coverage component.

Example:
in>coverage < OUTPUT c:\Freescale\myfile.txt
The Coverage output results are redirected to the file myfile.txt from the directory C:\Freescale.
P

The P command executes a CPU instruction, either at a specified address or at the current instruction, (pointed to by the program counter). This command traces through subroutine calls, software interrupts, and operations involving the following instructions (two are target specific):

- Branch to SubRoutine (BSR)
- Long Branch to Subroutine (LBSR)
- Jump to Subroutine (JSR)
- Software Interrupt (SWI)
- Repeat Multiply and Accumulate (RMAC)

For example: if the current instruction is a BSR instruction, the subroutine is executed, and execution stops at the first instruction after the BSR instruction. For instructions that are not in the above list, the P and T commands are equivalent.

When the instruction specified in the P command has been executed, the software displays the content of the CPU registers, the instruction bytes at the new value of the program counter and a mnemonic disassembly of that instruction.

Usage

P [address]

address: is an address constant expression, the address at which execution begins.

If address is omitted, execution begins with the instruction pointed to by the current value of the program counter.

Components

Debugger engine.

Example:

in>p

A=0x0  HX=0x450  SR=0x70  PC=0xF04E  SP=0xFF

00F04E 94 TXS
STEPPED

Contents of registers are displayed and the current instruction is disassembled.
PAUSETEST

Displays a modal message box shown in Figure 20.1 for testing purpose.

Figure 20.1  Test Pause Message Box

Usage

PAUSETEST

Components

Debugger engine.

Example:

in> pausetest
PRINTF

The **PRINTF** is the standard ANSI C command: Prints formatted output to the standard output stream.

**Usage**

PRINTF ("[Text ]%format specification", value)

**Components**

Debugger engine.

**Example**

```
in>PRINTF("The value of the counter is: %d", counter)
The output is: The value of the counter is: 2
```

PTRARRAY

The **PTRARRAY** command allows to specify if a pointer should be displayed as an array.

**Usage**

PTRARRAY on|off [nb]

Where:

- **on** displays pointers as arrays.
- **off** displays pointers as usual (*pointer).
- **nb** is the number of elements to display in the array when unfolding a pointer displayed as array.

**Components**

Data component.

**Example:**

```
in>Ptrarray on 5
Display content of pointers as array of 5 items.
```
**Debugger Engine Commands**

**Debugger Commands**

**RD**

The **RD** command displays the content of specified registers. The display of a register includes both the name and hexadecimal representation. If the specified register is not a CPU register, then it looks for this register in a register file as an I/O register. This file is called: **MCUIxxxx.REG** (where **xxxx** is a number related to the MCU).

**NOTE** This command is processor/derivative specific and will not display banked registers if the processor does not support banking.

**Usage**

RD { <list> | CPU | * }

where **list** is a list of registers to be displayed. Registers to be displayed are separated by a space. When "**RD CPU**" is specified, all CPU registers are displayed. If no CPU is loaded, "**No CPU loaded**" is displayed as an error message.

When * is specified, the **RD** command lists the content of the register file that is currently loaded. If no register file is loaded, following error message is displayed: "**No register file loaded**".

When there is no parameter, the previous **RD** command is processed again. If there is no previous **RD** command, all CPU registers are displayed.

If **list** is omitted, the list and any other parameters of the previous **RD** command are used.

For the first **RD** command of a session, all CPU registers are displayed.

**Components**

Debugger engine.

**Example1:**

```bash
in>rd a hx
A=0x14
HX=0x2
```

**Example2:**

```bash
in>rd cpu
A=0x0 HX=0x450 SR=0x70 PC=0xF04E SP=0xFF
```
RECORD
In the SoftTrace component, the RECORD command switches frame recording on/off while the target is running.

Usage
RECORD on|off

Components
SoftTrace component.

Example:
in>RECORD on

REPEAT
The REPEAT command allows you to execute a sequence of commands until a specified condition is true. The REPEAT command may be nested.
Press the Esc key to stop this command.

Usage
REPEAT

Components
Debugger engine.

Example:

```
DEFINE var = 0
...
REPEAT
  DEFINE var = var + 1
...
UNTIL var == 2
```

The REPEAT-UNTIL loop is identical to the ANSI C loop. The operation DEFINE var = var + 1 is done twice, then var == 2 and the loop ends.
RESET
In the Profiler and Coverage component, the RESET command resets all recorded frames (statistics).
In the SoftTrace component, the RESET command resets statistics and recorded frames.

NOTE Make sure that the RESET command is redirected to the correct component. Targets also have their own RESET command and if RESET is not redirected, the target is reset.

Usage
RESET

Components
Profiler and Coverage.

Example:  
in>Profiler < RESET

RESTART
Resets execution to the first line of the current application and executes the application from this point.

Usage
RESTART

Components
Engine component.

Example  
in>RESTART
After the RESTART command, the cycle counter is initialized to zero.
RETURN

The **RETURN** command terminates the current command processing level (returns from a **CALL** command). If executed within a command file, control is returned to the caller of the command file (i.e. the first instance that did not chain execution).

**Usage**

**RETURN**

**Components**

Debugger engine.

**Example:**

In file `d:\demo\cmd1.txt`:

```plaintext
... CALL d:\demo\cmd2.txt T ...
```

In file `d:\demo\cmd2.txt`

```plaintext
... ...
RETURN // returns to the caller
```

The command file `cmd1.txt` calls a second command file `cmd2.txt`. It is so necessary to insert the **RETURN** instruction to return to the caller file. Then the **T** Trace instruction is executed.
RS

The RS command assigns new values to specified registers. The RS mnemonic is followed by register name and new value(s).

An equal sign (=) may be used to separate the register name from the value to be assigned to the register; otherwise they must be separated by a space. The contents of any number of registers may be set using a single RS command. If the specified register is not a CPU register, then the register is searched in a register file as an I/O register. This file is called: MCUIxxxx.REG (where xxxx is a number related to the MCU).

Usage

RS register[=]value{,register[=]value}

register: Specifies the name of a register to be changed. String register is any of the CPU register names, or name of a register in the register file.

value: is an integer constant expression (in ANSI format representation).

Components

Debugger engine.

Example:

in>rs a=0xff hx=0x7fff
S

The S command stops execution of the emulation processor. Use the Go G command to start the emulator.

NOTE The S command ends as soon as the PC is changed.

Usage
S

Alias
STOP

Components
Debugger engine.

Example:
  in>s

  STOPPING
  HALTED

  Current application debugging is stopped/halted.
SAVE
The SAVE command saves a specified block of memory to a specified file in Freescale S-record format. The memory block can be reloaded later using the load S-record (SREC) command.

NOTE If no path is specified, the destination directory is the current project directory.

Usage
SAVE range fileName [offset][;A]
offset: an optional offset to add or subtract from addresses when writing S-records. The default offset is 0.
;A: appends the saved S-records to the end of an existing file. If this option is omitted, and the file specified by fileName exists, the file is cleared before saving the S-records.

Components
Debugger engine.

Example:
in>SAVE 0x1000..0x2000 DUMP.SX ;A
The memory range 0x1000..0x2000 is appended to the DUMP.SX file.
SAVEBP

The SAVEBP command saves all breakpoints of the currently loaded .ABS file into the matching breakpoints file. Also, the matching file has the name of the loaded .ABS file but its extension is .BPT (for example, the Fibo.ABS file has a breakpoint file called FIBO.BPT). This file is generated in the same directory as the .ABS file, when the user quits the Simulator/Debugger or loads another .ABS file.

If on is set, all breakpoints defined in the current application will be stored in the matching .BPT file.

If off is set, all breakpoints defined in the current application will not be stored in the matching .BPT file.

This command is only used in .BPT files and is related to the checkbox Save & Restore on load in the Controlpoints Configuration Window. It is used to store currently defined breakpoints (SAVEBP on) when the user quits the Simulator/Debugger or loads another .ABS file.

NOTE For more information about this syntax, refer to BS command and to the Control Points chapter.

Usage
SAVEBP on|off

Components
Debugger engine.

Example:
content of the FIBO.BPT file

```
savebp on
BS &fibo.c:Fibonacci+19 P E; cond = "fibo > 10" E; cdSz = 47 srSz = 0
BS &fibo.c:Fibonacci+31 P E; cdSz = 47 srSz = 0
BS &fibo.c:main+12 P E; cdSz = 42 srSz = 0
BS &fibo.c:main+21 P E; cond = "fiboCount==5" E; cmd = "Assembly < spc 0x800" E; cdSz = 42 srSz = 0
```
Debugger Engine Commands

SET

Sets a new current target for the debugger by loading the targetName component.

Usage

SET targetName

where targetName is name without extension of the target to set.

Components

Debugger engine.

Example:

in>SET Sim

The debugger’s current target is Simulator.

SETCOLORS

The SETCOLORS command is used to change the colors for a specific channel from the Monitor component.

Usage

SETCOLORS ( "Name" ) ( Background ) ( Cursor ) ( Grid ) ( Line ) ( Text

Name is the name of the channel to modify.

Background is the new color for the channel background (the format is: 0x00bbggrr).

Cursor is the new color for the channel cursor (the format is: 0x00bbggrr).

Grid is the new color for the channel grid (the format is: 0x00bbggrr).

Line is the new color for the channel lines (the format is: 0x00bbggrr).

Text is the new color for the channel text (the format is: 0x00bbggrr).

Components

Monitor component.

Example:

in>SETCOLORS "Leds.Port_Register bit 0" 0x00123456
0x00234567 0x00345678 0x00456789 0x00567891

The color attributes from the channel Leds.Port_Register bit 0 will be changed with these new values.
SLAY

The SLAY command is used to save the layout of all window components in the main application window to a specified file.

NOTE Layout files usually have a .HWL extension. However, you can specify any file extension.

NOTE If no path is specified, the destination directory is the current project directory.

Usage
SLAY fileName

Components
Debugger engine.

Example:
in>slay /hiwave/demo/mylayout.hwl

The current debugger layout is saved to the mylayout.hwl file in the /hiwave/demo directory.

SLINE

With the SLINE command, a line of the source file is made visible. If the line is not currently visible, the source will scroll so that it appears on the first line. If the line is currently in a folded part, it is unfolded so that it becomes visible.

NOTE The given line number should be between 1 and number of lines in source file, or else an error message is displayed.

Usage
SLINE line number

Components
Source component

Example:
in>sline 15
SMEM

In the **Source component**, the SMEM command loads the corresponding module’s source text, scrolls to the corresponding text location (the code address) and highlights the statements that correspond to this code address range.

In the **Assembly component**, the SMEM command scrolls the Assembly component, shows the location (the assembler address) and select/highlights the memory lines of the address range given as the parameter.

In the **Memory component**, the SMEM command scrolls the memory dump component, shows the locations (the memory address) of the address range given as the parameter.

**Usage**

SMEM range

**Components**

Source, Assembly and Memory components.

**Example:**

```
in>Memory < SMEM 0x8000,8
```

The Memory component window is scrolled and specified memory addresses are highlighted.
SMOD

In the **Source component**, the SMOD command loads/displays the corresponding module’s source text. If the module is not found, a message is displayed in Command Line window.

In the **Data component**, the SMOD command loads the corresponding module’s global variables.

In the **Memory component**, the SMOD command scrolls the memory dump component and highlights the first global variable of the module.

**NOTE**  
Correct module names are displayed in the Module component window. Make sure that the module name of your command is correct. If the .abs is in HIWARE format, some debug information is in the object file (.o), and module names have a .o extension (e.g., fibo.o). In ELF format, module name extensions are .c .cpp or .dbg (.dbg or program sources in assembler) (e.g., fibo.c), since all debugging information is contained in the .abs file and object files are not used. Please adapt the following examples with your .abs application file format.

**Usage**

SMOD module

Where **module** is the name of a module taking part of the application. The module name should contain no path. The module extension (i.e. .DBG for assembly sources or .C for C sources, etc.) must be specified.

The module name is searched in the directories associated with the **GENPATH** environment variable. An error message is displayed:

- If the module specified does not take part of the current application loaded.
- If no application is loaded.

**Components**

Data, Memory and source components.

**Example:**

```
in> Data:1 < SMOD fibo.c
```

Global variables found in the fibo.c module are displayed in the Data:1 component window.
SPC

In the **Source component**, the **SPC** command loads the corresponding module’s source text, scrolls to the corresponding text location (the code address) and highlights the statement that corresponds to this code address.

In the **Assembler component**, the **SPC** command scrolls the Assembly component, shows the location (the assembler address) and select/highlights the assembler instruction of the address given as parameter.

In the **Memory component**, the **SPC** command scrolls the memory dump component, shows the location (the memory address) of the address given as parameter.

**Usage**

```
SPC address
```

**Components**

Assembler, Memory and Source component.

**Example:**

```
in>Assembly < SPC 0x8000
```

The Assembly component window is scrolled to the address **0x8000** and the associated instruction is highlighted.
SPROC

In the **Data component**, the SPROC command shows local variables of the corresponding procedure stack level.

In the **Source component**, the SPROC command loads the corresponding module’s source text, scrolls to the corresponding procedure and highlights the statement of this procedure that is in the procedure chain.

`level = 0` is the current procedure level, `level = 1` is the caller stack level and so on.

**NOTE** This command is relevant when “C-source” debugging.

**NOTE** When a procedure of a level greater than 0 is given as parameter to the SPROC command, the statement corresponding to the call of the lower procedure is selected.

**Usage**

SPROC level

**Components**

Data and Source components.

**Example:**

```plaintext
in>Source < SPROC 1
```

This command displays the source code associated with the caller function in the Source component window.
SREC

The **SREC** command initiates the loading of Freescale S-Records from a specified file.

**NOTE**  If no path is specified, the destination directory is the current project directory.

**Usage**

SREC fileName [offset]

*offset*: is a signed value added to the load addresses in the file when loading the file contents.

**Components**

Debugger engine.

**Example:**

```
in>SREC DUMP.SX
```

The **DUMP.SX** file is loaded into memory.
STEPINTO

The **STEPINTO** command single-steps through instructions in the program, and enters each function call that is encountered.

**NOTE**  
This command works while the application is paused in break mode (program is waiting for user input after completing a debugging command).

**Usage**

**STEPINTO**

**Components**

Debugger engine.

**Example:**

```plaintext
in>STEPINTO
```

<table>
<thead>
<tr>
<th>STEP INTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACED</td>
</tr>
</tbody>
</table>

**TRACED** in the status line indicates that the application is stopped by an assembly step function.
STEPOUT

The STEPOUT command executes the remaining lines of a function in which the current execution point lies. The next statement displayed is the statement following the procedure call. All of the code is executed between the current and final execution points. Using this command, you can quickly finish executing the current function after determining that a bug is not present in the function.

NOTE  This command works while the application is paused in break mode (program is waiting for user input after completing a debugging command).

Usage

STEPOUT

Components

Debugger engine.

Example:

in>STEPOUT

STEP OUT
STARTED
RUNNING
STOPPED

STOPPED in the status line indicates that the application is stopped by a step out function.
**STEPOVER**

The STEPOVER command executes the procedure as a unit, and then steps to the next statement in the current procedure. Therefore, the next statement displayed is the next statement in the current procedure regardless of whether the current statement is a call to another procedure.

**NOTE** This command works while the application is paused in break mode (program is waiting for user input after completing a debugging command).

**Usage**

STEPOVER

**Components**

Debugger engine.

**Example:**

```
in>STEPOVER
```

<table>
<thead>
<tr>
<th>STEP OVER</th>
<th>STARTED</th>
<th>RUNNING</th>
<th>STOPPED</th>
</tr>
</thead>
</table>

STEPPED OVER (or STOPPED) in the status line indicates that the application is stopped by a step over function.
STOP
The STOP command stops execution of the emulation processor. Use the Go G command to start the emulator.

NOTE  The STOP command ends as soon as the PC is changed.

Usage
STOP

Alias
S

Components
Debugger engine.

Example:
in>STOP

STOPPING
HALTED

Current application debugging is stopped.
The **T** command executes one or more instructions at a specified address, or at the current address (the address in the program counter). The **T** command traces into subroutine calls and software interrupts. For example, if the current instruction is a Branch to Subroutine instruction (**BSR**), the **BSR** is traced, and execution stops at the first instruction of the subroutine. After executing the last (or only) instruction, the **T** command displays the contents of the CPU registers, the instruction bytes at the new address in the program counter and a mnemonic disassembly of the current instruction.

This command can be stopped by typing the **Esc** key.

**Usage**

*T [address][,count]*

*address*: is an address constant expression, the address where execution begins. If *address* is omitted, the instruction pointed to by the current value of the program counter is the first instruction traced.

*count*: is an integer constant expression, in the decimal integral interval [1, 65535], that specifies the number of instructions to be traced. If *count* is omitted, one instruction is traced.

**Components**

Debugger engine.

**Example:**

```
in>T 0xF030
```

<table>
<thead>
<tr>
<th>TRACED</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=0x0</td>
</tr>
<tr>
<td>00F032 B787</td>
</tr>
</tbody>
</table>

Contents of registers are displayed and current instruction is disassembled.
**TESTBOX**

Displays a modal message box shown in Figure 20.2 with a given string.

**Figure 20.2  Test Box Message Box**

Usage

TESTBOX "<String>"

Components

Debugger engine.

Example:

in>TESTBOX "Step 1: init all vars"

**TUPDATE**

In Profiler and Coverage components, the TUPDATE command switches the time update feature on/off.

Usage

TUPDATE on|off

Components

Profiler and Coverage components.

Example:

in>TUPDATE on
UNDEF

The UNDEF command removes a symbol definition from the symbol table. This command does not undefine the symbols defined in the loaded application. Program variables whose names were redefined using the UNDEF command are visible again. Undefining an undefined symbol is not considered an error.

Usage
UNDEF symbol | *
If * is specified, all symbols defined previously using the command DEFINE are undefined.

Components
Debugger engine.

Example:

```
DEFINE test = 1
...
UNDEF test
```

When the test variable is no longer needed in a command program, it can be undefined and removed from the list of symbols. After UNDEF test, the test variable can no longer be used without (re)defining it.

NOTE See also examples of the DEFINE command.

Examples:
The value of an existing symbol can be changed by applying the DEFINE command again. In this case, the previous value is replaced and lost. It is not put on a stack. Then when UNDEF is applied to the symbol, it no longer exists, even if the value of the symbol has been replaced several times:

```
in>DEFINE apple 0
in>LS
   apple          0x0 (0)    // apple is equal to 0
in>DEFINE apple = apple + 1
in>LS
   apple          0x00000001 (1)    // apple is equal to 1
```

apple          0x1 (1)    // apple is equal to 1
in>DEFINE apple = apple + 1
in>LS

apple          0x2 (2)    // apple is equal to 2
in>UNDEF apple
in>LS

// apple no longer exists

In the next example, we assume that the FIBO.ABS sample is loaded. At the beginning, no user symbol is defined:

in>UNDEF *
in>LS

User Symbols:    // there is no user symbol
Application Symbols:    // symbols of the loaded application
fiboCount 0x800 (2048)
counter    0x802 (2050)
_startupData 0x84D (2125)
Fibonacci 0x867 (2151)
main 0x896 (2198)
_Init     0x810 (2064)
_Startup 0x83D (2109)

in>DEFINE counter = 1
in>LS

User Symbols: // there is one user symbol: counter
counter 0x1 (1)
Application Symbols: // symbols of the loaded application
fiboCount 0x800 (2048)
counter 0x802 (2050)
_startupData 0x84D (2125)
Fibonacci 0x867 (2151)
main 0x896 (2198)
_Init     0x810 (2064)
_Startup 0x83D (2109)

in>undef counter
**UNFOLD**

In the Source component, the **UNFOLD** command is used to display the contents of folded source text blocks, for example, source text that has been collapsed at program block level. All text is unfolded once or (*) completely, until no more folded parts are found.

**Usage**

UNFOLD [*]

Where * means unfolding completely, otherwise unfolding only one level.

**Components**

Source component.

**Example:**

```
in>UNFOLD  *
```
Debugger Engine Commands

UNTIL
The UNTIL keyword is associated with the REPEAT command.

Usage
UNTIL condition
Where condition is defined as in “C” language definition.

Components
Debugger engine.

Example:

```plaintext
repeat
    open assembly
    wait 20
    define i = i + 1
until i==3
```

At the end of the loop, i is equal to 3.

UPDATERATE

Description
In the Data component and Memory component, the UPDATERATE command is used to set the data refresh update rate. This command only has an effect if the Data or Memory component to which it applies is set in Periodical Mode.

Usage
UPDATERATE rate
where rate is a constant number matching a quantity of time in tenths of a second, between 1 and 600 tenth of second (0.1 to 60 seconds).

Components
Data and Memory component.

Example:
in>Memory < updaterate 30
This commands sets the Memory component updaterate to 3 seconds.
**VER**

The **VER** command displays the version number of the Debugger engine and components currently loaded in the Command line window.

**Usage**

VER

**Components**

Debugger engine.

**Example:**

```
in>ver
```

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>HI-WAVE</td>
<td>6.0.27</td>
</tr>
<tr>
<td>HI-WAVE Engine</td>
<td>6.0.49</td>
</tr>
<tr>
<td>Source</td>
<td>6.0.20</td>
</tr>
<tr>
<td>Assembly</td>
<td>6.0.14</td>
</tr>
<tr>
<td>Procedure</td>
<td>6.0.10</td>
</tr>
<tr>
<td>Register</td>
<td>6.0.14</td>
</tr>
<tr>
<td>Memory</td>
<td>6.0.19</td>
</tr>
<tr>
<td>Data</td>
<td>6.0.27</td>
</tr>
<tr>
<td>Data</td>
<td>6.0.27</td>
</tr>
<tr>
<td>Simulator Target</td>
<td>6.0.17</td>
</tr>
<tr>
<td>Command Line</td>
<td>6.0.16</td>
</tr>
</tbody>
</table>

In the Command Line component window, Debugger engine and components versions are displayed.
**WAIT**

The **WAIT** command pauses command file execution for a time in tenths of second or pauses until the target is halted when the option ";s" is set.

- When no parameter is specified, it pauses for 50 tenths of a second (5 seconds).
- When only time is specified, execution of the command file is halted for the specified time.
- When only ;s is specified, execution of the command file is halted until the target is halted.
- If the target is already halted, command file execution is not halted.
- When time and ;s are specified:
  - If the target is running, command file execution is halted for the specified time only if the target is not halted. If the target is halted during the specified period of time (while command file execution is pending), the time delay is ignored and the command file is run.
  - If the target is already halted, command file execution is not halted (time delay is ignored).

**NOTE**  The Wait instruction ends as soon as the PC is changed.

**Usage**

**WAIT** [time] [:s]

**Components**

Debugger engine.

**Example:**

```
WAIT 100
T...
```

Pauses for 10 seconds before executing the T Trace instruction.
The `WB` command sets a specified block of memory to a specified list of byte values. When the range is wider than the list of byte values, the list of byte values is repeated as many times as necessary to fill the memory block. When the range is not an integer, a multiple of the length of the list and the last copy of the list is truncated accordingly. This command is identical to the memory set (`MS`) command.

**Usage**

`WB range list`

*range*: is an address range constant that defines the block of memory to be set to the values of the bytes in the list.

*list*: is a list of byte values to be stored in the block of memory.

**Alias**

`MS`

**Components**

Debugger engine.

**Example**

`in>WB 0x0205..0x0220 0xFF`

This command fills up the memory range 0x0205..0x0220 with the 0xFF byte value.
WHILE

The **WHILE** command allows you to execute a sequence of commands as long as a certain condition is true. The **WHILE** command may be nested. This command can be stopped by pressing the **Esc** key.

**Usage**

WHILE condition

Where **condition** is defined as in “C” language definition.

**Components**

Debugger engine.

**Example:**

DEFINE jump = 0
...
WHILE jump < 20
   DEFINE jump = jump + 1
ENDWHILE
T
...

While jump < 100, the jump variable is incremented by the instruction DEFINE jump = jump + 1. Then the loop ends and the T Trace instruction is executed.
WL

The WL command sets a specified block of memory to a specified list of longword values. When the range is wider than the list of longword values, the list of longword values is repeated as many times as necessary to fill the memory block. When the range is not an integer or a multiple of the length of the list, the last copy of the list is truncated accordingly.

When a size is specified in the range, this size represents the number of longwords that should be modified.

Usage

WL range list

range: is an address range constant that defines the block of memory to be set to the longword values in the list.

list: is a list of longword values to be stored in the block of memory.

Components

Debugger engine.

Example:

in>WL 0x2000 0xFFFFF0F
This command fills up memory starting at address 0x2000 with the 0xFFFFF0F longword value. The addresses 0x2000 to 0x2003 will be modified.

in>WL 0x2000, 2 0xFFFFF0F
This command fills up the memory area 0x2000 to 0x2007 with the longword value 0xFFFFF0F.

WW

The WW command sets a specified block of memory to a specified list of word values. When the range is wider than the list of word values, the list of word values is repeated as many time as necessary to fill the memory block. When the range is not an integer or a multiple of length of the list, the last copy of the list is truncated accordingly.

Usage

WW range list

range: is an address range constant that defines the block of memory to be set to the word values in the list.

list: is a list of word values to be stored in the block of memory.
Debugger Engine Commands

Debugger Commands

Components
Debugger engine.

Example:
in>WW 0x2000..0x200F 0xAF00
This command fills up the memory range 0x2000..0x200F with the 0xAF00 word value.

ZOOM
In the Data component, the ZOOM command is used to display the member fields of structures by ‘diving’ into the structure. In contrast to the UNFOLD command, where member fields are not expanded in place. The display of the member fields replaces the previous view. The ZOOM out command is used to return to the nesting level indicated by the given identifier.

NOTE Addresses are not needed to zoom out. Simply type “ZOOM out”.

NOTE This command is relevant when “C-source” debugging.

Usage
ZOOM address in|out
Where address is the address of the structure or pointer variable that should be zoomed-in or zoomed-out, respectively.

Components
Data component.

Example:
in>ZOOM 0x1FE0 in
The variable structure located at address 0x1FE0 is zoomed in.
in>zoom &_startupData
zooms in the _startupData structure (&_startupData is the address of the _startupData structure).
21

Debugger Connection-specific Commands

BDIK Connection Commands

This section describes the BDIK connection-specific commands that are used when the BDIK Target Interface is set.

The BDIK Target Interface specific commands are:

- BANKREG
- BDI
- PROTOCOL
- RESET

Those commands are entered in the Target Interface Command Files or in the Command Line component of the debugger.

This section describes each of the commands available for the BDIK Target Interface. The commands are listed in alphabetical order. Each is divided into several topics.

Table 21.1  Command Description  Parameters

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Provides a short description of the command.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the command in a EBNF format.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the command and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Small example of how to use the command.</td>
</tr>
</tbody>
</table>
Debugger Connection-specific Commands

BDIK Connection Commands

BDI

Short Description
Executes any direct BDI command

Syntax
BDI <ABATRON_direct_command>
where ABATRON_direct_command has the following syntax:
<Object>.<Action> [parName=parameterValue]...

Description
The BDI command executes any ABATRON direct command. ABATRON direct commands are described in the User Manual for your CPU from ABATRON. They are commonly used to download to non-volatile memory areas (please see also the Flash Programming section).

Example
BDI FLASH.ERASE addr=8000 size=8000 sram=0800
BANKREG

Short Description
Sets banked memory handling for HC12/CPU12 derivatives

Syntax
BANKREG [PPAGE=<PPAGE_register_adrs>] [DPAGE=<DPAGE_register_adrs>] [EPAGE=<EPAGE_register_adrs>]

Description
CAUTION This command is still available, but for compatibility only. However, it should not be used. The Banked Memory Location Window handles the banked memory handling when debugging on HC12/CPU12 derivatives.

The BANKREG command lets you define if paging is used, like PPAGE (HC912DG128, HC812A4), DPAGE (HC812A4) or EPAGE (HC812A4). This command must be inserted in the Startup Command File of your project directory. As soon as the command is executed, the specified registers are displayed in the Register component window.

Example
for HC812A4:
BANKREG PPAGE=0x35 DPAGE=0x34 EPAGE=0x36
for HC912DG128:
BANKREG PPAGE=0xFF
Debugger Connection-specific Commands
BDIK Connection Commands

PROTOCOL

Short Description
Switch on/off the Show Protocol functionality

Syntax
PROTOCOL ON|OFF

Description
If this command is used, all the messages sent to and received from the debugger are reported in the Command Line window of the debugger.

The Show Protocol facility can also be switched on/off using the corresponding check box in the Communication Device Specification Edit Box.

The state of the Show Protocol is stored in the [BDIK] section of the project file using variable SHOWPROT.

Example
PROTOCOL ON

NOTE The Show Protocol is a useful debugging feature if there is a communication problem.
## RESET

**Short Description**
Reset of the target board

**Syntax**

```
RESET
```

**Description**
Use this command to reset the target from the Command Line component of the debugger. The Reset Command File is also executed and the BDI interface automatically processes the initialization list (startup init list) stored in the interface.

**Example**

```
RESET
```

---

### Banked Memory Location-associated Commands

The following sections describe the Banked Memory Location Command Line commands which are used by the BDIK Connection. These variables are:

- `BANKWINDOW`

Those commands can be entered in the Target Interface Command Files Window or in the Command Line component of HI-WAVE.

The Banked Memory Location commands which are used by the Target Interface are described as shown in the following table.
The following sections describe each command related to the Banked Memory Location available for the Target Interface. The variables are listed in alphabetical order.

## BANKWINDOW

### Short Description
Specify a banked memory area and its status (enable/disable).

### Syntax
```bash
BANKWINDOW <bank> [OFF|ON] [range <reg> numofpages]
```

with

```
bank = (PPAGE | DPAGE | EPAGE)
```

or

```
BANKWINDOW VARIOUS [DLGATCONNECT|NODLGATCONNECT]
```

### Description
The command BANKWINDOW allows to set up the debugger to work in banked memory model.

Three different Banked Memory Area can be defined: DPAGE, EPAGE and PPAGE. Each banked memory area has an associated bank register, which is displayed in the Register component.

Using BANKWINDOW PPAGE ... command will have the same effect than using the PPAGE index tab in the Banked Memory Location Window.

Using BANKWINDOW DPAGE ... command will have the same effect than using the DPAGE index tab in the Banked Memory Location Window.
Debugger Connection-specific Commands

Using BANWINDOW EPAGE ... command will have the same effect than using the EPAGE index tab in the Banked Memory Location Window.

Using BANWINDOW VARIOUS ... command will have the same effect than using the Various index tab in the Banked Memory Location Window.

A banked memory area is defined by its start address, end address and the address of the Bank register.

The maximum number of pages parameter allows to see in the memory component only the available pages.

The status of the banking mechanism in the debugger is also monitored through this command: a command can be defined, but the debugger banking mechanism can be disabled.

Consider the command:

BANKWINDOW PPAGE ON 0x8000..0xBFFF 0x30 64

This command allows to use the banked memory model in the debugger using the MC9S12DP256B.

This commands means the PPAGE register located at address 0x30 must be used to build the PC address when the code is located in banked memory area, from 0x8000 to 0xBFFF. The 64 first page in the memory map are visible (page 0x3F is the last one).

The PPAGE register (located at address 0x30) will be displayed in the register component.

The bank settings are stored in the ["targetName"] section of the PROJECT file using variable BANKWINDOWn.
BANKWINDOW Examples

The Banking status can be gotten by typing `BANKWINDOW` without any parameters in the Command Line component.

Listing 21.1 BANKWINDOW > Banking Status

```
in>in>bankwindow
PPAGE Settings:
  Status: enabled
  Reg. Adr: 0x30
  Range: 0x8000 to 0xbfff
  Number of Pages: 64

DPAGE Settings:
  Status: disabled
  Reg. Adr: 0x34
  Range: 0x7000 to 0x7fff
  Number of Pages: 0

EPAGE Settings:
  Status: disabled
  Reg. Adr: 0x36
  Range: 0x400 to 0x7ff
  Number of Pages: 0
```

```
in>
```

The status of the PPAGE banks can be changed:

Listing 21.2 BANKWINDOW > PPPage Status Change

```
in>BANKWINDOW PPAGE OFF
in>BANKWINDOW
PPAGE Settings:
  Status: disabled
  Reg. Adr: 0x30
  Range: 0x8000 to 0xbfff
  Number of Pages: 64

DPAGE Settings:
  Status: disabled
  Reg. Adr: 0x34
  Range: 0x7000 to 0x7fff
  Number of Pages: 0
```
EPAGE Settings:
Status: disabled
Reg. Adr: 0x36
Range: 0x400 to 0x7ff
Number of Pages: 0

in>
ICD-12 Commands

Consider these hints for configuring the ICD-12 connection for a specific MCU and memory configuration. Connection initialization includes execution of startup file "startup.cmd," during ICD-12 driver loading. Triggering a reset through the Icd | Reset menu selection executes another command file, "reset.cmd." Choosing the Icd | Force BDM menu selection executes command files "forcebdm.cmd" and "reset.cmd."

[CMDL] describes the general syntax of command files.
Debugger Connection-specific Commands

NVMC Commands

The following Flash Commands can be issued through the debugger Command component window, as shown in the figure below.

Figure 21.1  NVMC Commands In Command Window

```
FLASH
FLASH parameters loaded for MC68HC12B2138 from C:\\HC12D\EMF\FLASH\cu3003.fp

NEW clock speed: 8019200

Module Name  Address Range   Status
FLASH_0000 4000 - BFFF   Enabled/Programmed - Unselected
FLASH_PAGE0 8000 - BFFF   Enabled/Blank - Unselected
FLASH_PAGE1 18000 - 1BFFF  Enabled/Programmed/Protected - Unselected
FLASH_PAGE2 28000 - 2BFFF  Enabled/Blank - Unselected
FLASH_PAGE3 38000 - 3BFFF  Enabled/Blank/Protected - Unselected
FLASH_PAGE4 48000 - 4BFFF  Enabled/Blank - Unselected
FLASH_PAGE5 58000 - 5BFFF  Enabled/Blank/Protected - Unselected
FLASH_PAGE6 68000 - 6BFFF  Enabled/Blank - Unselected
FLASH_PAGE7 78000 - 7BFFF  Enabled/Programmed/Protected - Unselected
HALTED

```

FLASH

Short Description
displays flash modules, loads .fpp file, or performs flash operations.

Syntax

FLASH [(SELECT|UNSELECT|ERASE|ENABLE|DISABLE|PROTECT|UNPROTECT|AEFSKIPERASING) [<blockNo>]]

| ARM| DISARM| SAVECONTEXT| LOADCONTEXT| MEMMAP| MEMUNMAP| RELEASE |
| OVLBACKUP| OVLRESTORE| PROTOCOL| PROTOCOLOFF| PROTOCOLON| SKIPSTATUS |

| [NVMFREQUENCY <frequency in Hz>] |
| [NVMIF2RELOCATE <address>] |
| [NVMIF2WORKSPACE <address> <address>] |
| [INIT <fileName> | AUTOID] |
Debugger Connection-specific Commands
NVMC Commands

Description
The FLASH command displays names, locations, and states of all available modules, provided that a parameter (.fpp) file is already loaded. If no parameter file is loaded, this command loads either the .fpp file for the current MCUID or the last-used .fpp file.

FLASH INIT <fileName> | AUTOID loads the parameter file according to fileName (you can specify the path). If this command includes AUTOID, the MCUID determines the parameter file (autocheck is checked in the NVMC dialog box).

FLASH RELEASE releases the current FPP file loaded by the Flash Programmer, therefore the Flash Programmer address mapping is disabled and no non volatile memory is handled.

FLASH MEMMAP maps the Flash Programmer address filtering to route the code for block programming.

FLASH MEMUNMAP unmaps the Flash Programmer address filtering. Programming is therefore disabled as long as FLASH MEMMAP is not executed.

FLASH ENABLE enables the specified modules. If no modules are specified, enables all available blocks. This command ignores modules that cannot be enabled.

FLASH DISABLE disables the specified modules. If no modules are specified, all disables all available blocks. This command ignores modules that cannot be disabled.

FLASH ERASE erases the specified modules. If no modules are specified, erases all available blocks.

FLASH AEFSKIPERASING specifies non volatile memory blocks to protect from mass erasing at application automated programming. The command should be placed in a "Startup" command file. If no modules are specified, no blocks are erased.

NOTE This command is compatible and replicated in the "NVM Programming Selection" dialog.

FLASH UNPROTECT unprotects the specified modules. If no modules are specified, unprotects all available blocks. This command ignores modules that cannot be unprotected.

FLASH PROTECT protects the specified modules. If no modules are specified, protects all available blocks. This command ignores modules that cannot be protected.

FLASH SELECT selects the specified modules for flash programming. If no modules are specified, selects all available blocks for flash programming.

FLASH UNSELECT unselects the specified modules. If no modules are specified, unselects all available blocks (The unselected state protects against accidental flash programming).
Debugger Connection-specific Commands

NVMC Commands

FLASH ARM prepares the NMVC utility for loading; as does a normal LOAD command. The system executes the VPPON.CMD file specified in the Command Files user interface. This command is required before loading flash.

FLASH DISARM ends a load process. The system executes the VPPOFF.CMD file specified in the Command Files user interface.

FLASH SAVECONTEXT backs up current SRAM content into a buffer.

FLASH LOADCONTEXT restores current buffer content into the MCU SRAM.

FLASH OVLBACKUP backups application code overlap with programming runtime/algorithm (RAM preset for debugging). The command should be executed before the application/file loading.

FLASH OVLRESTORE restores/installs (writes in RAM) the application code overlap with programming runtime/algorithm. The command should be executed after the last “FLASH” command.

FLASH PROTOCOLON displays the Flash Programmer debug protocol.

FLASH PROTOCOLOFF stops displaying the Flash Programmer debug protocol.

FLASH SKIPSTATUSON skips the Flash Programmer Device Non Volatile Memory blocks diagnostic. This can be used to speed up project application loading and programming from the IDE “debug” run: The Flash Programmer will NOT check if blocks are programmed or erased.

FLASH SKIPSTATUSOFF removes the SKIPSTATUSON mode and therefore diagnostics are performed again.

FLASH NVMFREQUENCY <frequency in Hz> specify the Non Volatile Memory programming frequency in Herz, typically the device bus speed after reset. When used, the Flash Programmer does not try to evaluate this speed and the debugger will gain 2-3 seconds at application loading time. A value of “0” engages back the speed detection.

FLASH NVMIF2RELOCATE <address> informs the flash programmer that flash driver must be loaded in ram to a different place than default (default is start of onchip ram). This will provide more flexibility for EB386 "Example 1 Layout" device ram memory relocation. The “data to program” buffer follows the same address translation. This command is Legacy and the “FLASH NVMIF2WORKSPACE” is more user friendly and performs a secured relocation. A value of “0” resets the relocation.

FLASH NVMIF2WORKSPACE <address> <address> informs the flash programmer that flash driver must be loaded in ram to a different place than default (default is start of onchip ram). The command will also resize the workspace, as a range must be passed as parameter. The command is therefore more powerful than ”FLASH NVMIF2RELOCATE”, however, the range must be correctly setup to match the targeted part. ”FLASH NVMIF2RELOCATE 0” resets any setup made with ”FLASH NVMIF2WORKSPACE” or ”FLASH NVMIF2RELOCATE” command. The command should be executed by preference from a "Startup" cmd file.
Debugger Connection-specific Commands

NVMC Commands

e.g.: FLASH NVMIF2WORKSPACE 0x3800 0x3FFF

The command implies that onchip ram is available at relocation position and range prior any flash driver is loaded. This command can provide more flexibility for EB386 "Example 1 Layout" device ram memory relocation.

[<blockNo>]

blockNo is a list of flash block/module numbers, according to this syntax:

\[ blockNo = \{ \text{number["-"number][","]} \}\]

Examples:

FLASH ERASE 2,7
This erases memory blocks 2 and 7.
FLASH ERASE 2,4-6 8
This erases memory blocks 2, 4, 5, 6, and 8.
FLASH ERASE
This erases all available memory blocks.

While flash modules are armed, execution of user code is not possible. If you enter a command such as run, step, or so forth, a message box prompts you to disarm the modules or cancel the command. If you click the OK button, the system disarms all flash modules, then executes your command. If you click the CANCEL button, the system cancels the command and leaves the flash modules armed.

Listing 21.3 Flash Programming Example from Command Line in Component Window

in>flash

FLASH parameters loaded for M68HC912DG128 from J:\HC12_EA\PROG\FPP\mcu03C4.fpp

MCU clock speed: 8025000

Module Name | Address Range | Status
---|---|---
FLASH_4000 | 4000 - 7FFF | Enabled/Blank - Unselected
FLASH_PAGE0 | 8000 - BFFF | Enabled/Blank - Unselected
FLASH_C000 | C000 - FFFF | Enabled/Blank/Protected - Unselected
FLASH_PAGE1 | 18000 - 1BFFF | Enabled/Blank/Protected - Unselected
FLASH_PAGE2 | 28000 - 2BFFF | Enabled/Blank - Unselected
FLASH_PAGE3 | 38000 - 3BFFF | Enabled/Blank/Protected - Unselected
FLASH_PAGE4 | 48000 - 4BFFF | Enabled/Blank - Unselected
FLASH_PAGE5 | 58000 - 5BFFF | Enabled/Blank/Protected - Unselected
FLASH_PAGE6 | 68000 - 6BFFF | Enabled/Blank - Unselected
FLASH_PAGE7 | 78000 - 7BFFF | Enabled/Blank/Protected - Unselected
HALTED

The FLASH command loads the applet that corresponds to the CPU derivative (MCUID) and displays the state of all modules.

If you want to program an application into module number 7 (FLASH_PAGE5), you must unprotect the module.

**Listing 21.4 Unprotect Module**

```
in>flash unprotect 7

MCU clock speed: 802500
Module Name        Address Range   Status
FLASH_4000         4000 -  7FFF   Enabled/Blank - Unselected
FLASH_PAGE0         8000 -  BFFF   Enabled/Blank - Unselected
FLASH_C000         C000 -  FFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE1        18000 - 1BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE2        28000 - 2BFFF   Enabled/Blank - Unselected
FLASH_PAGE3        38000 - 3BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE4        48000 - 4BFFF   Enabled/Blank - Unselected
FLASH_PAGE5        58000 - 5BFFF   Enabled/Blank/Unprotected - Unselected
FLASH_PAGE6        68000 - 6BFFF   Enabled/Blank - Unselected
FLASH_PAGE7        78000 - 7BFFF   Enabled/Blank/Protected - Unselected
```

The updated display shows that FLASH_PAGE5 is unprotected. Select FLASH_PAGE5 for programming.

```
in>flash select 7

Arm for programming.
```

```
in>flash arm
Arm FLASH for loading.
```

Now you can load your application.

```
in>load a:\my_page5.sx
RUNNING
```

Stop loading and disarm.

```
in>flash disarm
FLASH disarmed.
Halted
```
Debugger Connection-specific Commands
NVMC Commands

Use the FLASH command to display the final state of the modules.

**Listing 21.5 Display Module Final State**

```
in>flash
MCU clock speed: 8025000
Module Name     Address Range   Status
FLASH_4000       4000 -  7FFF   Enabled/Blank - Unselected
FLASH_PAGE0      8000 -  BFFF   Enabled/Blank - Unselected
FLASH_C000       C000 -  FFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE1     18000 - 1BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE2     28000 - 2BFFF   Enabled/Blank - Unselected
FLASH_PAGE3     38000 - 3BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE4     48000 - 4BFFF   Enabled/Blank - Unselected
FLASH_PAGE5     58000 - 5BFFF   Enabled/Programmed/Unprotected - Selected
FLASH_PAGE6     68000 - 6BFFF   Enabled/Blank - Unselected
FLASH_PAGE7     78000 - 7BFFF   Enabled/Blank/Protected - Unselected
HALTED
```

The FLASH_PAGE5 module is programmed. Now, you must protect and unselect the module.

**Listing 21.6 Protect and Unselect Module**

```
in>flash protect 7
MCU clock speed: 8025000
Module Name     Address Range   Status
FLASH_4000       4000 -  7FFF   Enabled/Blank - Unselected
FLASH_PAGE0      8000 -  BFFF   Enabled/Blank - Unselected
FLASH_C000       C000 -  FFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE1     18000 - 1BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE2     28000 - 2BFFF   Enabled/Blank - Unselected
FLASH_PAGE3     38000 - 3BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE4     48000 - 4BFFF   Enabled/Blank - Unselected
FLASH_PAGE5     58000 - 5BFFF   Enabled/Programmed/Protected - Selected
FLASH_PAGE6     68000 - 6BFFF   Enabled/Blank - Unselected
FLASH_PAGE7     78000 - 7BFFF   Enabled/Blank/Protected - Unselected

in>flash unselect 7
```

This completes the example.
DMM Commands

All DMM GUI settings can be done by debugger command line commands.

Debugging Memory Map Manager Commands

The following list of commands provides the possibility to fully script the debugging device memory mapping. However, the usage of these commands should be limited to special debugging purposes, as the default mapping is typically sufficient, and a script setup being complex and possibly leading to debugger disfunctions.

DMM
DMM ADD <parameters>
DMM DEL <module handle>
DMM SAVE <mcuid>
DMM DELETEALLMODULES
DMM RELEASECACHES
DMM CACHINGON|CACHINGOFF
DMM HCS12MERHANDLINGON|HCS12MERHANDLINGOFF
DMM OPENGUI [mcuid]

"DMM" Command

Syntax: DMM
Purpose: Displays in the Command window the current DMM "Memory Types", "Overlap Priorities" and memory memory ranges.

"DMM ADD" command

Syntax: DMM ADD <comment> <address> <size> <handle> <type> <cache locking> <priority> <mapping> <access while running>
with:
-<comment> a string for Comment field; "£" must be used for " " (space).
-<address> the start address of the memory range
-<size> the size of the memory range
-<handle> a long value for the DMM to handle the memory range (duplicated handled is not allowed).
Debugger Connection-specific Commands

DMM Commands

**WARNING!** User defined handles must be a value superior or equal to 100.

- `<type>` a value corresponding to a memory type handle, as given/listed with the DMM command.
- `<cache locking>` a "0" or "1" value, "0" forcing the memory range to be refreshed after each debugger halting.
- `<priority>` a value corresponding to an overlap priority handle/value, as given/listed with the DMM command.
- `<mapping>` a "0" or "1" value, "1" enabling the memory range mapping.
- `<access while running>` a "0" or "1" value, "1" enabling the memory range access while running.

This last parameter can be internally disabled (of no use), depending to the memory type capability.

Purpose: insert a new memory range in the DMM, as if added via the DMM dialog/user interface.

"**DMM DEL**" Command

Syntax: DEL <module handle>

with <module handle>, a memory range module handle as given/listed with the DMM command.

Purpose: Delete one specific DMM memory range module by handle reference.

"**DMM SAVE**" Command

Syntax: DMM SAVE <mcuid>

with <mcuid>, a part/device mcuid value in range $0-$FFFF.

Purpose: saves the DMM current setup in current project ini file, under "DMM_MCUIDxxxx_MODULEn=..." keys.

"**DMM DELETEALLMODULES**" Command

Syntax: DMM DELETEALLMODULES

Purpose: removes all current DMM memory range modules. Useful to start a scripted DMM setup.
"DMM RELEASECACHES" Command
Syntax: DMM RELEASECACHES
Purpose: flushes once all currently cached data for each memory range module, even if the cache locking is active, i.e. no refresh on halting is active.

"DMM CACHINGON" Command
Syntax: DMM CACHINGON
Purpose: data caching is engaged (default DMM setup). No refresh on halting is active for memory range modules defined with this option.

"DMM CACHINGOFF" Command
Syntax: DMM CACHINGOFF
Purpose: data caching is disabled. The debugger flushes all caches even for memory range modules defined without this option. Each time the debugger halts, the memory data are retrieve from the target hardware for all memory range modules.

"DMM HCS12MERHANDLINGON" Command
Syntax: DMM HCS12MERHANDLINGON
Purpose: enables the handling of Memory Expansion Registers for HCS12 devices, i.e. INITRM, INITRG and INITEE. The DMM remaps automatically memory range module addresses according to the real value of these registers when halting.

NOTE: The debugger does not poll the MER registers while running. Also the remapping is performed only on factory defined memory range modules, not user defined memory range modules.

"DMM HCS12MERHANDLINGOFF" Command.
Syntax: DMM HCS12MERHANDLINGOFF
Purpose: disables completely the feature here above.
"DMM OPENGUI" Command

Syntax: DMM OPENGUI [mcuid]
with <mcuid>, a part/device mcuid value in range $0-$FFFF.
Purpose: opens the DMM Graphical User Interface. Note that if the Mcuid is not specified, changes will not be saved for the next connection session (not saved in project), except if the connection does not care about Mcuid’s.

Full Chip Simulator Commands

Simulator environment commands are used to monitor the debugger environment, specific component window layouts and framework applications and targets. Table 21.3 contains all available Environment commands.

Table 21.3 Full Chip Simulator Commands

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETCPU ProcessorName</td>
<td>Sets a new cpu simulator</td>
</tr>
<tr>
<td>RESETCYCLES</td>
<td>Resets Simulator CPU cycles counter</td>
</tr>
<tr>
<td>RESETMEM</td>
<td>Resets all configured memory to ‘undefined’</td>
</tr>
<tr>
<td>RESETRAM</td>
<td>Resets RAM to ‘undefined’</td>
</tr>
<tr>
<td>RESETSTAT</td>
<td>Resets the statistical data</td>
</tr>
<tr>
<td>SHOWCYCLES</td>
<td>Returns executed Simulator CPU cycles</td>
</tr>
</tbody>
</table>

FCS-Associated Component-specific Commands

Component specific commands are associated with specific components supported by the Full Chip Simulator. Table 21.4 contains all available Component Specific commands.
## Debugger Connection-specific Commands

### Full Chip Simulator Commands

Table 21.4 List of Component-specific Commands

<table>
<thead>
<tr>
<th>Command, Syntax</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCPORT ( address</td>
<td>ident ) ( address</td>
</tr>
<tr>
<td>ADDCHANNEL ( &quot;Name&quot; )</td>
<td>Creates a new channel &quot;Name&quot; for the Monitor component.</td>
</tr>
<tr>
<td>CPORT ( address</td>
<td>ident ) ( address</td>
</tr>
<tr>
<td>ITPORT ( address</td>
<td>ident ) ( address</td>
</tr>
<tr>
<td>ITVECT ( address</td>
<td>ident )</td>
</tr>
<tr>
<td>KPORT ( address</td>
<td>ident ) ( address</td>
</tr>
<tr>
<td>LCDPORT ( address</td>
<td>ident ) ( address</td>
</tr>
<tr>
<td>LINKADDR ( address</td>
<td>ident ) ( address</td>
</tr>
<tr>
<td>PBPORT ( address</td>
<td>ident )</td>
</tr>
<tr>
<td>PORT address</td>
<td>Sets the LED components port address.</td>
</tr>
<tr>
<td>SEGPORT ( address</td>
<td>ident ) ( address</td>
</tr>
<tr>
<td>SETCONTROL ( &quot;Name&quot; ) ( Ticks ) ( Pixels)</td>
<td>Changes the number of ticks and pixels for the &quot;Name&quot; channel from the Monitor component.</td>
</tr>
<tr>
<td>WPORT ( address</td>
<td>ident ) ( address</td>
</tr>
</tbody>
</table>
ADDCHANNEL
The ADDCHANNEL command is used to create a new channel for the Monitor component.

Usage
ADDCHANNEL ( "Name" )
Name is the name for the new channel.

Components
Monitor component.

Example:
in>ADDCHANNEL "Leds.Port_Register bit 0"
A new channel Leds.Port_Register bit 0 is created in the Monitor component.

ADCPORT
The ADCPORT command is used to set the ports addresses used by the Adc_Dac component.

Usage
ADCPORT ( address | ident ) ( address | ident ) ( address | ident )
Address locates the port address value of the component (many formats are allowed), the default format is hexadecimal.
Ident is a known identifier, its content will define the port address.

Components
ADC_DAC component.

Example:
in>ADCPORT 0x100 0x200 0x300
The ports of the ADC_DAC component are now defined at the addresses 0x100, 0x200 and 0x300.
**CPORT**

The **CPORT** command is used to set the 5 coupler port addresses and the control port address of the coupler component.

**Usage**

CPORT ( address | ident ) ( address | ident ) ( address | ident )...

*Address* locates the port address value of the component (many formats are allowed), the default format is hexadecimal.

*Ident* is a known identifier, its content will define the port address.

**Components**

Programmable parallel Couplers component.

**Example:**

```
in>CPORT 0x100 0x200 0x300
```

The ports of the Programmable parallel Couplers will be defined at addresses 0x100, 0x200 and 0x300.

---

**DELCHANNEL**

The **DELCHANNEL** command is used to delete a specific channel for the Monitor component.

**Usage**

DELCHANNEL ( "Name" )

*Name* is the name of the channel to delete.

**Components**

Monitor component.

**Example:**

```
in>DELCHANNEL "Leds.Port_Register bit 0"
```

The channel Leds.Port_Register bit 0 will be deleted in the Monitor component.
**Debugger Connection-specific Commands**

**Full Chip Simulator Commands**

---

**ITPORT**

The ITPORT command is used to set the line and column port addresses of the IT_Keyboard component.

**Usage**

ITPORT ( address | ident ) ( address | ident ) ( address | ident )...

Address locates the port address value of the component (various formats are allowed), the default format is hexadecimal.

Ident is a known identifier, its content will define the port address.

**Components**

IT_Keyboard component.

**Example:**

in>ITPORT 0x100 0x200 0x300

Ports of the IT_Keyboard are now defined at addresses 0x100, 0x200 and 0x300.

---

**ITVECT**

The ITVECT command is used to set the interrupt vector port address of the IT_Keyboard component.

**Usage**

ITVECT ( address | ident ).

Address locates the port address value of the component (various formats are allowed), the default format is hexadecimal.

Ident is a known identifier, its content will define the port address.

**Components**

IT_Keyboard component.

**Example:**

in>ITVECT 0x400

The interrupt vector port address of the IT_Keyboard is now defined at address 0x400.
**KPORT**

The **KPORT** command is used to set the line and column ports addresses of the Keyboard component.

**Usage**

KPORT ( address | ident ) ( address | ident ) ( address | ident )...

*Address* locates the port address value of the component (many formats are allowed). The default format is hexadecimal.

*Ident* is a known identifier, its content will define the port address.

**Components**

Keyboard component.

**Example:**

```
in>KPORT 0x100 0x200 0x300
```

The ports of the Keyboard are now defined at addresses 0x100, 0x200 and 0x300.

**LCDPORT**

**Description**

The **LCDPORT** command is used to set the data port and the control port address of the Lcd component.

**Usage**

LCDPORT ( address | ident ) ( address | ident ) ( address | ident )...

*Address* locates the port address value of the component (many formats are allowed), the default format is hexadecimal.

*Ident* is a known identifier, its content will define the port address.

**Components**

Lcd component.

**Example:**

```
in>LCDPORT 0x100 0x200
```

The ports of the Lcd are now defined at addresses 0x100, 0x200 and 0x300.
LinkADDR

The LinkADDR command is used to set the components internal ports addresses used with the Programmable Couplers as memory buffers.

Usage

LinkADDR ( address | ident ) ( address | ident ) ( address | ident )...

Address locates the port address value of the component (many formats are allowed), the default format is hexadecimal.

Ident is a known identifier, its content will define the port address.

Components

Couplers, Adc_Dac, Keyboard, IT_Keyboard, IO_Led, Lcd, Push_Buttons, 7-segments display, Wagon

Example:

In>LinkADDR 0x100 0x200 0x300 0x400 0x500

Now all components working with the Programmable Couplers have PortA set to 0x100, PortB set to 0x200, PortC set to 0x300, PortD set to 0x400 and PortE set to 0x500.

PbPort

Description

The PbPort command is used to set the port address of the Push_Butons component.

Usage

PbPort ( address | ident )

Address locates the port address value of the component ( various formats are allowed), the default format is hexadecimal.

Ident is a known identifier, its content will define the port address.

Components

Push_Butons component.

Example:

In>PbPort 0x100 0x200

The ports of the Push_Butons are now defined at addresses 0x100 and 0x200.
PORT

Description
In the Led components, the PORT command sets the port Led location.

Usage
PORT address

Components
Led component.

Example:
in> PORT 0x210

PSMODE
This command changes the power save mode.
The STOP option places the CPU in is lowest power consumption mode; all internal CPU processing is halted.
The WAIT option places the CPU in low power consumption; all internal CPU processing is halted, however the internal clock, the programmable timer, SPI and SCI remain active (for more detail see the HC05 manual). This option consumes more power than the Stop option.
The WAKEUP option turns off the low power consumption mode; the processor resumes normal processing.

Usage
PSMODE (STOP | WAIT | WAKEUP)

Components
HI-WAVE engine.

Example:
in>PSMODE STOP /* The processor is completely stopped */
in>PSMODE WAKEUP /* The processor is out of power save mode */
Debugger Connection-specific Commands

Full Chip Simulator Commands

REGBASE
This command allows you to change the base address of the I/O registers or to set (Reset) this address to 0.

Usage
Regbase <Address><;R>
Where Address is an address to define the base address of the I/O registers, the 'R' option sets this address to 0 (Reset).

Components
Debugger engine.

Example:
in>regbase 0x500
0x 500 is now the base address of the I/O registers.

RESETCYCLES
This command sets the Simulator CPU cycles counter to the user defined value. If not specified, the value will be 0. The cycles counter is displayed in the Debugger status and Register Component. This command does not affect the context.

Usage
RESETCYCLES <Value>
where Value is the desired cycles. This command affects only the internal cycle counter from the Simulator/Debugger.

Components
Debugger engine.

Example:
in>SHOW CYCLES
133801
in>RESETCYCLES
in>SHOW CYCLES
0
in>RESETCYCLES 5500
in>SHOW CYCLES

5500

The _Showcycles_ command in the Command Line component displays the number of CPU cycles executed since the start of the simulation.

**RESETMEM**

This command marks the given range of memory (RAM + ROM) as uninitialized ('undefined').

**Usage**

`RESETMEM` range

**Components**

Simulator component.

**Example:**

in>RESETMEM

After the _RESETMEM_ command, all configured memory is initialized to 'undefined'.

in>RESETMEM 0x100..0x110

This command resets the memory between 0x100 and 0x110 (if configured) to 'undefined'.

in>RESETMEM 0x003F

This command resets the memory location 0x003F (if configured) to 'undefined'.

**NOTE**

In the memory configuration “Auto on Access” the full memory is defined as RAM, so in this case the command _RESETMEM_ has the same effect as _RESETRAM_.


**RESETRAM**
This command marks all RAM as uninitialized (‘undefined’).

**NOTE** In the memory configuration “Auto on Access” the full memory is defined as RAM, so in this case the command RESETMEM has the same effect as RESETRAM.

**Usage**
RESETRAM

**Components**
Simulator component.

**Example:**
in>RESETRAM
After the RESETRAM command, the content of RAM is initialized as undefined.

**RESETSTAT**
This command resets the statistics (read and write counters to zero)

**Usage**
RESETSTAT

**Components**
Simulator component.

**Example:**
in>RESETSTAT
After the RESETSTAT command, all counters are initialized to zero.
Debugger Connection-specific Commands
Full Chip Simulator Commands

SEGPORT
The SEGPORT command is used to set the display selection port and segment selection port addresses of the 7-Segments display component.

Usage
SEGPORT display selection port ( address | ident ) segment selection ( address | ident )
Address locates the port address value of the component (many formats are allowed), the default format is hexadecimal.
Ident is a known identifier, its content will define the port address.

Components
7-Segments display.

Example:
in>SEGPORT 0x100 0x200
The ports of the 7-Segments display are now defined at addresses 0x100 and 0x200.

SETCONTROL
The SETCONTROL command is used to modify the number of ticks and pixels for a Monitor component specific channel. This will change the horizontal scale of this channel.

Usage
SETCONTROL ( "Name" ) ( Ticks ) ( Pixels )
Name is the name of the channel to modify.
Ticks is the new number of ticks for this channel.
Pixels is the new number of pixels for this channel.

Components
Monitor component.

Example:
in>SETCONTROL "Leds.Port_Register bit 0" 100 1
The horizontal scale from the channel Leds.Port_Register bit 0 will be defined with the value 100 for the Ticks value and 1 for pixels value.
Debugger Connection-specific Commands
Full Chip Simulator Commands

SETCPU
Load CPU awareness for the debugger.

Usage
SETCPU ProcessorName
where ProcessorName is a supported processor (HC05, HC08, HC11, HC12, HC16, M68K, M.CORE, XA, ST7 and PPC).

Components
Simulator component.

Example:
in>SETCPU HC08
The simulator HC08.sim is loaded.

SHOWCYCLES
The SHOWCYCLES command returns the number of CPU cycles already done since the beginning of the simulation in the Command Line component (RESETCYCLES is performed internally), or since the last RESETCYCLES command. The number of cycles executed is also the number displayed in the status bar (CPU cycles counter).

Usage
SHOWCYCLES

Components
Debugger engine.

Example:
in>SHOW CYCLES
133801

in>RESETCYCLES
in>SHOW CYCLES
0

This command displays the number of CPU cycles executed since the last RESETCYCLES command in the Command Line component.
**WPORT**

The WPORT command is used to set the port addresses of the Wagon component.

**Usage**

WPORT (address | ident) (address | ident)

*Address* locates the port address value of the component (various formats are allowed), the default format is hexadecimal.

*Ident* is a known identifier, its content will define the port address.

**Components**

Wagon

**Example:**

```
in>WPORT 0x100 0x200
```

Ports of the Wagon are now defined at addresses 0x100 and 0x200.
Debugger Connection-specific Commands

Full Chip Simulator Commands
Book V - Environment Variables

Book V Contents

Each section of the Debugger manual includes information to help you become more familiar with the Debugger, to use all its functions and help you understand how to use the environment. This book, the Debugger Environment Variables, defines the HC12, HCS12, and HC(S)12(X) environment variables, both those environment variables used by the debugger engine and those specific to individual debugger connections.

Book 5: Environment Variables

- Chapter 5.1 Debugger Engine Environment Variables
- Chapter 5.2 Connection-specific Environment Variables
Debugger Engine
Environment Variables

This chapter describes the environment variables that the Debugger uses. Other tools, such as the Linker, also use some of these environment variables. For more information about other tools, see their respective manuals.

Click any of the following links to jump to the corresponding section of this chapter:

- Debugger Environment
- Local Configuration File (usually project.ini)
- ABSPATH: Absolute Path
- DEFAULTDIR: Default Current Directory
- ENVIRONMENT=: Environment File Specification
- GENPATH: #include “File” Path
- LIBRARYPATH: 'include <File>’ Path
- OBJPATH: Object File Path
- TMP: Temporary directory
- USELIBPATH: Using LIBPATH Environment Variable
- Search Order for Source Files
- Debugger Files
Debugger Environment

Various parameters of the Debugger may be set using environment variables. The syntax is always the same:

Parameter = KeyName "=" ParamDef.

**NOTE** Do not use blanks in the definition of an environment variable.

For example:

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/lib;/home/me/my_project
```

The Debugger parameters may be defined in several ways:

- Using system environment variables supported by your operating system.
- Putting the definitions in a file called `DEFAULT.ENV` in the default directory.

**NOTE** The maximum length of environment variable entries in the `DEFAULT.ENV` is 4096 characters.

- Putting definitions in a file given by the value of the system environment variable `ENVIRONMENT`.

**NOTE** The default directory mentioned above can be set by using the system environment variable `DEFAULTDIR`; Default Current Directory.

When looking for an environment variable, all programs first search the system environment, then the `DEFAULT.ENV` file and finally the global environment file given by `ENVIRONMENT`. If no definition can be found, a default value is assumed.

**NOTE** Ensure that no spaces exist at the end of environment variables.
The Current Directory

The most important environment for all tools is the current directory. The current directory is the base search directory where the tool begins to search for files (for example, the DEFAULT.ENV /.hidefaults file).

Normally, the current directory of a tool is determined by the operating system or program that launches another one (for example, WinEdit).

For MS Windows-based operating systems, the current directory definition is more complex.

- If the tool is launched using a File Manager/Explorer, the current directory is the location of the executable launched.
- If the tool is launched using an Icon on the Desktop, the current directory is the one specified and associated with the Icon.
- If the tool is launched by dragging a file on the icon of the executable under Windows 95, 98, Windows NT 4.0 or Windows 2000, the desktop is the current directory.
- If the tool is launched by another tool with its own current directory specified (for example, WinEdit), the current directory is the one specified by the launching tool (for example, current directory definition in WinEdit).
- For the Debugger tools, the current directory is the directory containing the local project file. Changing the current project file also changes the current directory, if the other project file is in a different directory. Note that browsing for a C file does not change the current directory.

To overwrite this behavior, the environment variable DEFAULTDIR: Default Current Directory may be used.
Global Initialization File (MCUTOOLS.INI - PC Only)

All tools may store global data in MCUTOOLS.INI. The tool first searches for this file in the directory of the tool itself (path of executable). If there is no MCUTOOLS.INI file in this directory, the tool looks for the file in the MS Windows installation directory (for example, C:\WINDOWS).

Example:

<table>
<thead>
<tr>
<th>C:\WINDOWS\MCUTOOLS.INI</th>
</tr>
</thead>
<tbody>
<tr>
<td>D:\INSTALL\PROG\MCUTOOLS.INI</td>
</tr>
</tbody>
</table>

If a tool is started in the D:\INSTALL\PROG\DIRECTORY, the project file in the same directory as the tool is used (D:\INSTALL\PROG\MCUTOOLS.INI).

If the tool is started outside the D:\INSTALL\PROG directory, the project file in the Windows directory is used (C:\WINDOWS\MCUTOOLS.INI).

**NOTE** For more information about MCUTOOLS.INI entries, see the compiler manual.
Local Configuration File (usually project.ini)

The Debugger does not change the default.env file. Its content is read only. All configuration properties are stored in the configuration file. The same configuration file can be used by different applications.

The shell uses the configuration file with the name “project.ini” in the current directory only. That is why this name is also suggested to be used with the Debugger. Only when the shell uses the same file as the compiler, the editor configuration written and maintained by the shell can be used by the Debugger. Apart from this, the Debugger can use any file name for the project file. The configuration file has the same format as windows.ini files. The Debugger stores its own entries with the same section name as in the global mcutools.ini file.

The current directory is always the directory containing the configuration file. If a configuration file in a different directory is loaded, then the current directory also changes. When the current directory changes, the default.env file is reloaded. Always when a configuration file is loaded or stored, options in the environment variable COMPOPTIONS are reloaded and added to the project options. Beware of this behavior when a different default.env file exists in different directories, which contain incompatible options in COMPOPTIONS.

When a project is loaded using the first default.env, its COMPOPTIONS are added to the configuration file. If this configuration is stored in a different directory, where a default.env file exists with incompatible options, the Debugger adds options and marks the inconsistency. Then a message box appears to inform the user that the default.env options were not added. In such a situation the user can either remove the option from the configuration file with the option settings dialog or remove the option from default.env with the shell or a text editor, depending on which options should be used in the future.

At startup there are three ways to load a configuration:

- use the command line option prod
- the project.ini file in the current directory
- or Open Project entry from the file menu.

If the option prod is used, then the current directory is the directory the project file is in. If prod is used with a directory, the project.ini file in this directory is loaded.
Default Layout Configuration (PROJECT.INI)

The default layout activated when starting the Debugger is defined in the PROJECT.INI file located in the project directory, as shown in Listing 22.1. All default layout related parameters are stored in section [DEFAULTS].

Listing 22.1 Example Content of PROJECT.INI:

```ini
[HI-WAVE]
Window0=Source     0   0  60  30
Window1=Assembly   60  0  40  30
Window2=Procedure  0  30  50  15
Window3=Terminal   0  45  50  15
Window4=Register   50  30  50  30
Window5=Memory     50  60  50  30
Window6=Data       0  60  50  15
Window7=Data       0  75  50  15
Target=Sim
```

**Target**: Specifies the target used when starting the Debugger (loads the file `<target>` with a .tgt extension), for example, Target=Sim for HC(S)12(X) Freescale Full Chip Simulator, or Target=Motosil, Target=Bdi.

**Window<n>**: Specifies coordinates of the windows that must be open when the Debugger is started. The syntax for a window is:

```
Window<n>=<component> <XPos> <YPos> <width> <height>
```

where `n` is the index of the window. This index is incremented for each window and determines the sequence windows are opened. This index is relevant in case of overlapping windows, because it determines which window will be on top of the other. Values for the index have to be in the range 0..99.

**component** specifies the type of component that should be opened, for example, Source, Assembly, etc.

**XPos** specifies the X coordinate of the top left corner of the component (in percentage relative to the width of the main application client window).

**YPos** specifies the Y coordinate of the top left corner of the component (in percentage relative to the height of the main application client window).

**width** specifies the width of the component (in percentage relative to the width of the main application client window).

**height** specifies the height of the component (in percentage relative to the height of the main application client window).

**Example:**

```
Window5=Memory  50 60 50 30
```
Window number 5 is a Memory component, its starting position is at: 50% from main window width, 60% from main window height. Its width is 50% from main window width and its height 30% from main window height.

**Other Parameters**

- It is possible to load a previously saved layout from a file by inserting the following line in your `PROJECT.INI` file:

```
Layout=<LayoutName>
```

Where `LayoutName` is the name of the file describing the layout to be loaded, for example, `Layout=lay1.hwl`

**NOTE** The layout path can be specified if the layout is not in the project directory.

**NOTE** If `Layout` is defined in `PROJECT.INI`, the `Layout` parameter overwrites any `Window<n>` definition, describing the default windows layout.

- It is possible to load a previously saved project from a file by inserting the following line in your `PROJECT.INI` file:

```
Project=<ProjectName>
```

where `ProjectName` is the name of the file describing the project to be loaded, for example, `Project=Proj1.hwc`

**NOTE** The project path can be specified if the project is not in the project directory. This option can be used for compatibility with the old .hwp format (Project=oldProject.hwp) and will be opened as a new project file.

See **File Menu** section for more details about Projects.

**NOTE** If `Layout` and `Project` are defined in `PROJECT.INI`, the `Project` parameter overwrites the `Layout` parameter, also containing layout information.

```
MainFrame=<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>,
<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>
```

This variable is used to save and load the Debugger main window states: positions, size, maximized, minimized, iconized when opened, etc. This entry is used for internal purposes only.

- The toolbar, status bar, heading line, title bar and small border can be specified in the default section:

The toolbar can be shown or hidden with the following syntax:
Debugger Engine Environment Variables

Local Configuration File (usually project.ini)

```
Toolbar = (0 | 1)
If 1 is specified, the toolbar is shown, otherwise the toolbar is hidden.
The status bar can be shown or hidden with the following syntax:
Statusbar = (0 | 1)
If 1 is specified, the status bar is shown, otherwise the toolbar is hidden.
Title bars can be shown or hidden with the following syntax:
Hidetitle = (0 | 1)
If 1 is specified, the title bars are hidden, otherwise they are shown.
The heading lines can be shown or hidden with the following syntax:
Hideheadlines = (0 | 1)
If 1 is specified, the heading lines are hidden otherwise they are shown.
The border can be reduced with the following syntax:
Smallborder = (0 | 1)
If 1 is specified, borders are thin otherwise they are normal.

* The environment variable BPTFILE authorizes the creation of breakpoint files; they
  may be enabled or disabled. All breakpoints of the currently loaded 'abs' file are
  saved in a breakpoints file. BPTFILE may be ON (default) or OFF. When ON,
  breakpoint files are created. When OFF, breakpoint files are not created.

BPTFILE = (On | Off)
```

**NOTE**
Target specific environment variables can also be defined in the
`PROJECT.INI` file. Refer to the specific target manual for details.

---

Ini File Activation

When a project file (`PROJECT.INI`) is activated, the following occurs (from first action
to last):

- The old Project file is closed.
- Target Component is unloaded
- The environment variable (Path) is added from the Project file.

Select HI-WAVE section to retrieve value from:

- if an entry 'Windows0' or 'Target' can be retrieved from section [HI-WAVE] then:
  use [HI-WAVE]
- else if an entry 'Windows0' or 'Target' can be retrieved from section [DEFAULTS] then:
use [DEFAULTS]

- else:
  use [HI-WAVE]

The environment variables are loaded from the default.env file.
If an entry 'Layout=lll' exists, the layout file lll.hwl is loaded and executed.
The target is set (if entry 'Target=ttt' exists load target 'ttt').
If an entry 'Project=ppp' exists, the command file 'ppp' is executed.
The configuration file (*.hwc) is loaded (entry configuration=*hwc).

## Environment Variable Paths

Most environment variables contain path lists indicating where to search for files. A path list is a list of directory names separated by semicolons following the syntax below:

PathList = DirSpec {";" DirSpec}.
DirSpec = ["*"] DirectoryName.

**Example:**

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/hiwave/lib;/home/me/my_project
```

If a directory name is preceded by an asterisk ("*"), the programs recursively search the directory tree for a file, not just the given directory. Directories are searched in the order they appear in the path list.

**Example:**

```
GENPATH=.;\*S;O
```

**NOTE** Some DOS environment variables (like GENPATH, LIBPATH, etc.) are used.

We strongly recommend working with WinEdit and setting the environment by means of a DEFAULT.ENV file in your project directory. This 'project directory' can be set in WinEdit's 'Project Configure...' menu command. This way, you can have different projects in different directories, each with its own environment.

**NOTE** When using WinEdit, do **not** set the system environment variable Defaultdir. If you do and this variable does not contain the project directory given in WinEdit’s project configuration, files might not be put where you expect them.
Debugger Engine Environment Variables
Local Configuration File (usually project.ini)

Line Continuation

It is possible to specify an environment variable in an environment file (default.env/.hidefaults) over multiple lines by using the line continuation character `\`:

Example:

```plaintext
OPTIONS=\ 
   -W2 \ 
   -Wpd
```

This is the same as:

```plaintext
OPTIONS=-W2 -Wpd
```

Be careful when using the line continuation character with paths. For example:

```plaintext
GENPATH=.\ 
TEXTFILE=.	xt
```

Will result in:

```plaintext
GENPATH=.TEXTFILE=.\txt
```

To avoid such problems, use a semicolon `;` at the end of a path, if there is a `\` at the end:

```plaintext
GENPATH=.; 
TEXTFILE=.	xt
```
Environment Variables

The remainder of this section is devoted to describing each of the environment variables available for the Debugger. The options are listed in alphabetical order and each is divided into several sections described in the following table, Environment Variable Details.

Table 22.1 Environment Variable Details

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools</td>
<td>Lists of other tools that are using this variable</td>
</tr>
<tr>
<td>Synonym</td>
<td>For some environment variables a synonym also exists. The synonyms may be used for older releases of the Debugger and will be removed in the future. A synonym has lower precedence than the environment variable.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the option in EBNF format.</td>
</tr>
<tr>
<td>Arguments</td>
<td>Describes and lists optional and required arguments for the variable.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the default setting for the variable or none.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the option and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Gives an example of usage and effects of the variable where possible. The examples show an entry in the default.env file for PC.</td>
</tr>
<tr>
<td>See also</td>
<td>Names related sections.</td>
</tr>
</tbody>
</table>
**ABSPATH: Absolute Path**

**Tools**
SmartLinker, Debugger

**Synonym**
None

**Syntax**
ABSPATH=" {<path>}".

**Arguments**
<path>: Paths separated by semicolons, without spaces.

**Description**
When this environment variable is defined, the SmartLinker will store the absolute files it produces in the first directory specified. If ABSPATH is not set, the generated absolute files will be stored in the directory the parameter file was found.

**Example:**
ABSPATH=\sources\bin;\..\..\headers;\usr\local\bin
DEFAULTDIR: Default Current Directory

Tools
Compiler, Assembler, Linker, Decoder, Librarian, Maker, Burner, Debugger.

Synonym
None.

Syntax
"DEFAULTDIR="<directory>.

Arguments
<directory>: Directory specified as default current directory.

Default
None.

Description
With this environment variable the default directory for all tools may be specified. All tools indicated above will take the directory specified as their current directory instead of the one defined by the operating system or launching tool (for example, editor).

NOTE
This is an environment variable at the system level (global environment variable). It CANNOT be specified in a default environment file (DEFAULT.ENV/.hidefaults).

Example:
DEFAULTDIR=C:\INSTALL\PROJECT

See also:
The Current Directory
Global Initialization File (MCUTOOLS.INI - PC Only)
Environment File Specification

Tools
Compiler, Linker, Decoder, Librarian, Maker, Burner, Debugger.

Synonym
HIENVIRONMENT

Syntax
"ENVIRONMENT=" <file>.

Arguments
<file>: file name with path specification, without spaces

Default
None.

Description
This variable has to be specified at the system level. Normally the application looks in the
The Current Directory for an environment file named default.env. Using
ENVIRONMENT (for example, set in the autoexec.bat for DOS ), a different file name
may be specified.

NOTE This is an environment variable at the system level (global environment
variable). It CANNOT be specified in a default environment file
(DEFAULT.ENV/.hidefaults).

Example:
ENVIRONMENT=\Freescale\prog\global.env
GENPATH: #include “File” Path

Tools
Compiler, Linker, Decoder, Burner, Debugger.

Synonym
HIPATH

Syntax
"GENPATH=" {<path>}.

Arguments
<path>: Paths separated by semicolons, without spaces.

Default
Current directory

Description
If a header file is included with double quotes, the Debugger searches in the current
directory, then in the directories given by GENPATH and finally in the directories given
by LIBRARYPATH: ‘include <File>’ Path.

NOTE
If a directory specification in this environment variable starts with an asterisk
("*"), the whole directory tree is searched recursively. All subdirectories and
their subdirectories are searched. Within one level in the tree, search order is
random.

Example:
GENPATH=\sources\include;..\..\headers;
\usr\local\lib

See also:
Environment variable LIBPATH
LIBRARYPATH: ‘include <File>’ Path

Tools
Compiler, ELF tools (Burner, Linker, Decoder)

Synonym
LIBPATH

Syntax
"LIBRARYPATH=" {<path>}

Arguments
<path>: Paths separated by semicolons, without spaces.

Default
Current directory

Description
If a header file is included with double quotes, the Compiler searches in the current directory, then in the directories given by GENPATH: #include “File” Path and finally in directories given by LIBRARYPATH: ‘include <File>’ Path.

NOTE
If a directory specification in the environment variables starts with an asterisk (“*”), the whole directory tree is searched recursively. All subdirectories and their subdirectories are searched. Within one level in the tree, search order is random.

Example:
LIBRARYPATH=\sources\include;..\..\headers;\usr\local\lib

See also:
Environment variable GENPATH: #include “File” Path
Environment variable USELIBPATH: Using LIBPATH Environment Variable
OBJPATH: Object File Path

Tools
Compiler, Linker, Decoder, Burner, Debugger.

Synonym
None.

Syntax
"OBJPATH=" <path>.

Default
Current directory

Arguments
<path>: Path without spaces.

Description
If a tool looks for an object file (for example, the Linker), then it first checks for an object file specified by this environment variable, then in GENPATH: #include “File” Path and finally in HIPATH.

Example:
OBJPATH=\sources\obj
TMP: Temporary directory

Tools
Compiler, Assembler, Linker, Librarian, Debugger.

Synonym
None.

Syntax
"TMP=" <directory>.

Arguments
<directory>: Directory to be used for temporary files.

Default
None.

Description
If a temporary file has to be created, normally the ANSI function tmpnam() is used. This
library function stores the temporary files created in the directory specified by this
environment variable. If the variable is empty or does not exist, the current directory is
used. Check this variable if you get an error message “Cannot create temporary file”.

NOTE
This is an environment variable at the system level (global environment
variable). It CANNOT be specified in a default environment file (DEFAULT.ENV/.hidefaults).

Example:
TMP=C:\TEMP

See also:
The Current Directory
USELIBPATH: Using LIBPATH Environment Variable

Tools
Compiler, Linker, Debugger.

Synonym
None.

Syntax
"USELIBPATH= " ("OFF" | "ON" | "NO" | "YES")

Arguments
"ON", "YES": The environment variable LIBRARYPATH: ‘include <File>’ Path is used to look for system header files <*.h>.
"NO", "OFF": The environment variable LIBRARYPATH: ‘include <File>’ Path is not used.

Default
ON

Description
This environment variable allows a flexible usage of the LIBRARYPATH: ‘include <File>’ Path environment variable, because LIBRARYPATH: ‘include <File>’ Path may be used by other software (for example, version management PVCS).

Example:
USELIBPATH=ON

See also:
Environment variable LIBRARYPATH: ‘include <File>’ Path
Search Order for Source Files

This section describes the search order (from first to last) used by the debugger.

**In the Debugger for C Source Files (*.c, *.cpp)**
1. Path coded in the absolute file (.abs)
2. Project file directory (where the .pj or .ini file is located)
3. Paths defined in the GENPATH environment variable (from left to right)
4. Abs File directory

**In the Debugger for Assembly Source Files (*.dbg)**
1. Path coded in the absolute file (.abs)
2. Project file directory (where .pj or .ini file is located)
3. Paths defined in the GENPATH environment variable (from left to right)
4. Abs File directory

**In the Debugger for Object Files (HILOADER)**
1. Path coded in the absolute file (.abs)
2. Abs File directory
3. Project file directory (where .pj or .ini file is located)
4. Path defined in the OBJPATH environment variable
5. Paths defined in the GENPATH environment variable (from left to right)

**Debugger Files**

The Debugger comes with several program, application, configuration files and examples. These files and file extensions are listed in the following table.
### Table 22.2 Debugger File Extensions

<table>
<thead>
<tr>
<th>File Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*.ABS</td>
<td>Absolute framework application file e.g., fibo.abs</td>
</tr>
<tr>
<td>*.ASM</td>
<td>Assembler specific file e.g., macrodem.asm</td>
</tr>
<tr>
<td>*.BBL</td>
<td>Burner Batch Language file e.g., fibo.bbl</td>
</tr>
<tr>
<td>*.BPT</td>
<td>Debugger Breakpoint file e.g., fibo.bpt</td>
</tr>
<tr>
<td>*.C *.CPP</td>
<td>C and C++ source files</td>
</tr>
<tr>
<td>*.CHM</td>
<td>Compiled HTML help file</td>
</tr>
<tr>
<td>*.CMD</td>
<td>Command File Script, for example, Reset.cmd</td>
</tr>
<tr>
<td>*.CNF</td>
<td>Specific cpu configuration file</td>
</tr>
<tr>
<td>*.CNT</td>
<td>Help Contents File, for example, cxa.cnt</td>
</tr>
<tr>
<td>*.CPU</td>
<td>Central Processor Unit Awareness file</td>
</tr>
<tr>
<td>*.DBG</td>
<td>Debug listing files, for example, Fibo.dbg</td>
</tr>
<tr>
<td>DEFAULT.ENV</td>
<td>Debugger Default Environment file.</td>
</tr>
<tr>
<td>*.DLL</td>
<td>A .DLL file that contains one or more functions compiled, linked, and stored separately from the processes that use them. The operating system maps the DLLs into the process’s address space when the process is starting up or while it is running. The process then executes functions in the DLL. The DLL of the Debugger is provided for supported library and extended functions.</td>
</tr>
<tr>
<td>*.H</td>
<td>Header file</td>
</tr>
<tr>
<td>HIWAVE.EXE</td>
<td>The Debugger for Windows executable program.</td>
</tr>
<tr>
<td>*.HWL</td>
<td>Debugger Layout file, for example, default.hwl</td>
</tr>
<tr>
<td>*.HWC</td>
<td>Debugger Configuration file (project.hwc)</td>
</tr>
<tr>
<td>*.EXE</td>
<td>Other Windows executable program, for example, LINKER.EXE</td>
</tr>
<tr>
<td>*.FPP</td>
<td>Flash Programming Parameters files (CPU specific) for example, mcu0e36.fpp</td>
</tr>
<tr>
<td>*.HLP</td>
<td>Application Help file, for example, Hiwave.hlp</td>
</tr>
</tbody>
</table>
### Debugger File Extensions

#### Table 22.2 Debugger File Extensions

<table>
<thead>
<tr>
<th>File Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*.IO</td>
<td>I/O's simulation file, for example, sample11.io</td>
</tr>
<tr>
<td>*.ISU</td>
<td>Uninstall Application File</td>
</tr>
<tr>
<td>*.PJT</td>
<td>Debugger configuration Settings File, for example, Project.pjt</td>
</tr>
<tr>
<td>*.INI</td>
<td>Debugger configuration Settings File, for example, Project.ini</td>
</tr>
<tr>
<td>*.LST</td>
<td>Assembler Listing File, for example, fibo.lst</td>
</tr>
<tr>
<td>*.MCP</td>
<td>Freescale CodeWarrior IDE project file</td>
</tr>
<tr>
<td>*.MAK</td>
<td>Make file, for example, demo.mak</td>
</tr>
<tr>
<td>*.MAP</td>
<td>Mapping file, for example, macrodem.map</td>
</tr>
<tr>
<td>*.MEM</td>
<td>Memory Configuration file, for example, 000p4v01.mem</td>
</tr>
<tr>
<td>*.MON</td>
<td>Firmware loading, file for allowing to load a specified target, for example, Firm0508.mon</td>
</tr>
<tr>
<td>*.O</td>
<td>Object file code, for example, Fib0.o</td>
</tr>
<tr>
<td>*.PRM</td>
<td>Linker parameter file, for example, fibo.prm</td>
</tr>
<tr>
<td>Project.Ini</td>
<td>Debugger Project Initialization File</td>
</tr>
<tr>
<td>*.REC</td>
<td>Recorder File</td>
</tr>
<tr>
<td>*.REG</td>
<td>Register Entries files, for example, mcu081e.reg</td>
</tr>
<tr>
<td>*.SIM</td>
<td>CPU Awareness file, for example, st7.sim</td>
</tr>
<tr>
<td>*.SX</td>
<td>S-Record file, for example, fibo.sx</td>
</tr>
<tr>
<td>*.TGT</td>
<td>Target File for the Debugger, for example, xtend-g3.tgt</td>
</tr>
<tr>
<td>*.WND</td>
<td>Debugger Window Component File, for example, recorder.wnd</td>
</tr>
<tr>
<td>*.XPR</td>
<td>Debugger User Expression file, for example, Fib0.xpr</td>
</tr>
</tbody>
</table>
Connection-specific Environment Variables

Some of the environment variables that can be used in the debugging process are imported with the connection software and are specific to that connection. This chapter is intended to list and describe those variables.

Connection-specific Environment Variables

The following sections address connection environment variables that can be manually edited.

BDIK Connection Environment Variables

This section describes the environment variables which are used by the BDIK Target Interface. The BDIK Target Interface specific environment variables are:

- BDICONF
- COMDEV
- COMPRESS
- SHOWPROT
- SKIPILLEGALBREAK
- VERIFY

These variables are stored in the [BDIK] section from the project file.
Connection-specific Environment Variables

Listing 23.1 Example of the [BDIK] section from the project file:

```plaintext
[BDIK]
CMDFILE0=CMDFILE STARTUP ON "startup.cmd"
CMDFILE1=CMDFILE RESET ON "reset.cmd"
CMDFILE2=CMDFILE PRELOAD ON "preload.cmd"
CMDFILE3=CMDFILE POSTLOAD ON "postload.cmd"
COMDEV=COM1 57600
SHOWPROT=0
BDICONF=C:\tmp\B10c12.exe
SKIPILLEGALBREAK=0
VERIFY=1
COMPRESS=1
```

The remainder of this section describes each of the variables available for the BDIK Target Interface. The variables are listed in alphabetical order and are divided into several topics.

Table 23.1 Variable Description Parameters

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Provides a short description of the variable.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the variable in a EBNF format.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the default setting for the variable.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the variable and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Small example of how to use the variable.</td>
</tr>
</tbody>
</table>
**BDICONF**

**Short Description**
Defines the ABATRON configuration tool file and path

**Syntax**

BDICONF=ConfigurationToolFileNameandPath

where *ConfigurationToolFileNameandPath* is the ABATRON configuration tool file name and path.

**Default**

The default value does not exist. The string "Enter here the path to the ABATRON configuration tool." is displayed in the edit box.

**Description**

This variable defines the communication device between the computer and the BDI. It is set according to the *BDI Configuration Tool Path* edit box of the Setup dialog box. The *BDI Configuration Tool Path* edit box can be set up with the path and application name of the configuration tool from ABATRON. The application tool is automatically browsed when selecting the *Configure BDI Box...* menu entry and browsing for the application. Otherwise, press the *Browse...* button to look for the tool.

**Example**

BDICONF=C:\tmp\B10c12.exe
Connection-specific Environment Variables
BDIK Connection Environment Variables

COMDEV

**Short Description**
Defines the communication device between the computer and the BDI.

**Syntax**
COMDEV=COMn baudrate
where \( n \) is the COM port number like 1, 2, 3, etc. and where baudrate is 9600, 19200, 38400, 57600, 115200, according to the setup done in the ABATRON configuration application.

For the communication via an Ethernet:
COMDEV=NETWORK ip_address port
where ip_address is the IP address of the BDI box or bdiNet in the form xxx.xxx.xxx.xxx and port is the bdiNet port, usually "1" for BDI1000 and BDI2000.

**Default**
The default value is COM1 57600.

**Description**
This variable defines the communication device between the computer and the BDI. It is set according to the Communication Device Specification Edit Box.

**Example**
COMDEV=COM1 57600
COMPRESS

Short Description
Sets data transfer compression

Syntax
COMPRESS=1 | 0

Default
The default value is 1.

Description
This variable sets the BDI download mode with data compression. By default, data compression is enabled for asynchronous communication channels. With older computers, it is possible that download speed is faster without data compression. It is set according to the Use Data Compression check box of the Setup Dialog Box.

Example
COMPRESS=1
SHOWPROT

**Short Description**
Set Show Protocol On/Off

**Syntax**
SHOWPROT=1 | 0

**Default**
The default value is 0.

**Description**
If the Show Protocol is used, all the commands and responses sent and received are reported in the Command Line component of the debugger.

If the variable is set to 1, Show Protocol is activated.

This variable is set according to the Show Protocol check box of the Communication Device Specification Edit Box.

**Example**
SHOWPROT=1

**NOTE**
The Show Protocol is a useful debugging feature if there is a communication problem.
**SKIPILLEGALBREAK**

**Short Description**

Enables skipping illegal breakpoints

**Syntax**

```
SKIPILLEGALBREAK=1 | 0
```

**Default**

The default value is 0.

**Description**

This variable is set according to the Continue on illegal break (banked hardware breakpoint) option check box of the Setup Dialog Box.

The Continue on illegal break (banked hardware breakpoint) option check box is only available for the HC12/CPU12 derivative. You can check this check box to overcome the 2-byte address size on-chip break module which does not handle the PPAGE (e.g. HC912DG128). Note that internally, the target will be halted by the hardware breakpoint (in Flash memory), compared with the breakpoint that you set, then relaunched if not (bank) matching.

**Example**

```
SKIPILLEGALBREAK=1
```
Connection-specific Environment Variables

BDIK Connection Environment Variables

VERIFY

Short Description
Sets data transfer verification

Syntax
VERIFY=0|1|2|3
with 0 for no verification at all (fastest mode), 1 for first byte verification only, 2 for all data read back verification, and 3 for only verification (no write).

Default
The default value is 1.

Description
This variable sets the BDI download mode with data verification. By default, use Verify only first...option. If necessary, you can set a different option to improve transfer speed or security. It is set according to the Data Transfer Verification radio buttons of the Setup Dialog Box.

Example
VERIFY=1

Banked Memory Location-associated Environment Variables

The following sections describe the Banked Memory Location environment variables which are used by the BDIK connection. These variables are:

Bankwindow
These variables are stored in the ["targetName"] section from the project file.

Listing 23.2 Example of [BDIK] Target Section from Project File:

```
[BDIK]
BANKWINDOW0=BANKWINDOW PPAGE ON 0x8000..0xBFFF 0x30 64
BANKWINDOW1=BANKWINDOW DPAGE OFF 0x7000..0x7FFF 0x34 256
BANKWINDOW2=BANKWINDOW EPAGE OFF 0x400..0x7FF 0x36 256
```
The Banked Memory Location environment variables which are used by the BDIK Connection are described as shown in the following table.

Table 23.2 Variable Description Parameters

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Provides a short description of the variable.</td>
</tr>
<tr>
<td>Syntax</td>
<td>Specifies the syntax of the variable in a EBNF format.</td>
</tr>
<tr>
<td>Default</td>
<td>Shows the default setting for the variable.</td>
</tr>
<tr>
<td>Description</td>
<td>Provides a detailed description of the variable and how to use it.</td>
</tr>
<tr>
<td>Example</td>
<td>Small example of how to use the variable.</td>
</tr>
</tbody>
</table>

The following sections describe each variable available for the Target Interface. The variables are listed in alphabetical order.

BANKWINDOWn

**Short Description**
Contains a BANKWINDOW Command Line command to be used to set up the Banked Memory support.

**Syntax**
BANKWINDOW=<one BANKWINDOW Command Line command>

**Default**
All available banked memory area are disabled by default.

The default PPAGE memory banked area is 0x8000 to 0xBFFF, 8 pages allowed, with PPAGE register at address 0x35.

The default DPAGE memory banked area is 0x7000 to 0x7FFF, 256 pages allowed, with PPAGE register at address 0x34.

The default EPAGE memory banked area is 0x400 to 0x7FF, 256 pages allowed, with PPAGE register at address 0x36.
Connection-specific Environment Variables

BDIK Connection Environment Variables

The default settings for the VARIOUS page is that the bank window dialog is displayed automatically when connecting when settings are not done (do only apply to the Hitex Target Interface).

Description

The BANKWINDOWn variable specifies a command file definition using BANKWINDOW Command Line command. Three or four of those entries should be present in the project file, depending on the Target Interface.

Those variables are used to store the Banked Memory Location definition (range, address, number of pages) and status (enable/disable) specified either with the BANKWINDOW Command Line command the Banked Memory Location Window.

Example

BANKWINDOW0=BANKWINDOW PPAGE OFF 0x8000..0xBFFF 0x30 64
BANKWINDOW1=BANKWINDOW DPAGE OFF 0x7000..0x7FFF 0x34 256
BANKWINDOW2=BANKWINDOW EPAGE OFF 0x400..0x7FF 0x36 256
BANKWINDOW3=BANKWINDOW VARIOUS DLGATCONNECT
ICD-12 Default Environment

As with any HI-WAVE program, you can set ICD-12 connection component parameters in the DEFAULT.ENV file. This file should be in the working directory.

In normal use, you set these parameters in the DEFAULT.ENV file once, interactively, during installation. You use these parameter values in subsequent debugging sessions.

ICD-12 Environment Variables

The following sections introduce the environment variables associated with the ICD-12 connection.

ICDPORT Variable

This variable specifies (to the host computer) the parallel communication port to which the ICD-12 connects.

Syntax

ICDPORT=LPTn
ICDPORT=LPTn:
ICDPORT=portAddr

where

n: number of the printer port (1, 2)
portAddr: address of the printer port (1, 2). Specifying the printer-port address is only possible with Windows 95, or with Windows 3.1x with Win32s. Under Windows NT, a driver that evaluates the port address must handle access to a port, so you cannot specify a port address. First try to define the Icd-Port by name (LPT1 or LPT2). If that does not work, define the communication port by address.

Examples

ICDPORT=LPT2
// Name of the port.
ICDPORT=0x378
// Address of the port.

Default

ICDPORT=LPT1
NOTE  ICDPORT=0x378 is the MS-DOS first parallel printer port address;  
ICDPORT=0x278 is the MS-DOS second parallel printer port address. Under  
some Win 3.x installations, it could be necessary to specify the ICD-12 port by  
address.

BMDELAY Variable

This variable slows down the communication speed of the serial link (the ICD-12 cable).  
The MCU clock speed is the maximum speed available, but the PC also affects the  
communication speed. So if your target MCU clock speed is slower than 1 MHz, you may  
need a delay that is greater than 0.

Syntax

BMDELAY=x

where

x: communication delay. The x value 0 yields the fastest communication speed.

Example

BMDELAY=9

You may have to work down from a high x value, such as 150 or 100, until you find the  
optimal value for your system.

Default

The default x value is 0.
Index

Symbols
.ABS 419
.abs file 43
.cmd 54
.FPP file loading 552
.hidedefaults 715, 725, 726, 730
.hl 719
.HWP 15
.hwp 719
.INI 15
.PJT 15
.rec 100
.WND 43
.wnd 33
.xpr file 65

Numerics
16-Bit Modulus Down-Counter 290, 291

A
A 462, 500, 577
ABATRON Configuration tool 412
About Box 31
About True Time Simulator and Real Time Debugger 31
ABSPATH 724
ACTIVATE 577
ADCPORT 698
Add New Instrument 129, 130
ADDCHANNEL 698
Address 78, 81, 486, 524
Address B 478, 479, 516, 517
Address… 48
ADDXPR 578
Align 131
All Text Folded At Loading 118
Analog 132
Analog to Digital Converter 260
AND Mask 135, 136, 138
Application
  Assembly Step 195
  Embedded 5
  Loading 191
  Starting 192
  Step In 193
  Step Out 194
  Step Over 194
  Stopping 192
  Target 5
Application loading 553
Arrange Icons 30
ASCII 81
Assembly Step 21
Assembly Step Out 22
Assembly Step Over 22
Associated Commands 310
AT 588
ATTRIBUTES 578
Auto 103
auto configure 235
Auto on Access 237
Auto on Load 237
Auto select according to MCU Id 553
Automatic 66, 80
Automatically analyze the FIFO content 488, 526
AUTOSIZE 588

B
B 462, 500
Background Color 28
Background Debug Mode 410
Backgroundcolor 132, 133
Banked hardware breakpoint 423
Banked Memory Location dialog 438
Banked Memory Location Target commands 441, 681
banked memory model 290
Banking on CPU12 420
BANKREG 436, 679
Banks 290, 420
BANKWINDOW 442, 682
BANKWINDOW 445, 743
Bar 132
Barcolor 134
Bardirection 134
BASE 589
BC 589
BCKCOLOR 590
BD 591
BDI 435, 678
Abatron setup 412
Configuration 412
Interface 411
Interface setup 412
BDI Firmware 413
BDI Initialization List 414
BDI Startup Init List 414
BDI Working Mode 416
BDI1000 410
BDICONF 430, 737
BDI-HS 410
BDIK
BDIK Flash Programming with BDI 427
BDIK Target Interface Dialogs 420, 421, 422
BDIK Terminal Emulation with BDI 425
Menu Entries 419
BDIK | Command Files 420
BDIK | Communication... 419, 421
BDIK | Connect 419, 421
BDIK | Help 420
BDIK | Load... 419, 420
BDIK | Reset 419
BDIK | Set Bank... 420
BDIK | Setup... 422
BDIK Flash Programming with BDI 427
BDIK Target Interface Dialogs 420, 421, 422
BDIK Terminal Emulation with BDI 425
BDM connector/port 411
BDM port 410
Bin 68, 79, 103, 196
Binary 196, 199
Bit Reverse 80, 103
Bitnumber to Display 137
blank 550, 551
BLCD 257
Blocks 548
Bottom 131
Bounding Box 133
Box configuration 412
BREAKPOINT 424
Breakpoint 47, 109
BREAKPOINT 424
Checking condition 147
Command 156
Conditional 154, 172
Counting 153, 172
Definition 143
Deleting 155
Multiple selection 147
Permanent 143
Position 150
Temporary 143, 151
breakpoint 423, 433, 460, 498, 741
Breakpoint banked hardware 423
Breakpoint with Register Condition 154, 155
Breakpoints... 22
BRLD 292
BS 592
BSPL 292
BTST 292
Bus Trace 406, 460, 498
Byte 79
Byteflight 257
C
-C 10
CALL 290, 594
Call Chain 94
Capture 290, 482, 520
Capture / Compare Timer 291
Capture r/write values at Address B after access at Address A 482, 520
Capture Stimulation 291
Capture the read/write values at Address B 482, 520
Cascade 30
CD 595
CF 596
CFORC 291
Clear 494, 532
CLOCK 598
Clock and Reset Generator 266
Clone Attributes 131
CLOSE 598
-Cmd 10
CMDFILE 449, 599
CMDFILEn 453
Code coverage 475, 513
Code profiling 475, 513
CodeWarrior Integration 205
Color if 138

HC(S)12(X) Debugger Manual 748
Dump 493, 531
DW 607
dynamic 470, 508

E
E 608
e 468, 506
Editing
  Memory 201
  Register 199
  Variable 197
Editmode 129, 130, 132
Editor 64
EEPPROM 262
ELSE 609
ELSEIF 609
Enable Breakpoint 50, 113
enabled 550
Enabling 551
End 550
ENDFOCUS 610
ENDFOR 611
ENDIF 611
ENDWHILE 612
Enhanced Capture Timer 268, 290
Environment
  ABSPATH 724
  DEFAULTDIR 725
  ENVIRONMENT 714
  File 714
  GENPATH 727, 729
  HIENVIRONMENT 726
  HPATH 727, 729
  LIBPATH 728, 731
  LIBRARYPATH 728
  OBJPATH 729
  TMP 730
  USELIBPATH 731
  Variable 723
Environment variables 429
-ENVpath 10
EPAGE 440
EPAGE Banked Memory Area 440
EQUAL Mask 135, 138
Erase 427
Erase Flash 391, 405
Erasing 551
Ethernet 410, 411
EXECUTE 612
EXIT 613
Exit 15
Expert 474, 512
Expert mode 467, 505
Expert triggers 474, 512
Explorer 715
Expression Command File 65
Expression Editor 64
External Bus Interface 264

F
FE 293
FEE28 556
FEE32 557
Field Description 140, 141
FIFO 488, 490, 494, 526, 528, 532
FIFO Analyse remark 490, 528
FIFO Depth 494, 495, 532, 533
File
  Environment 714
  File Manager 715
  Filename 135
  FILL 613
  Fill Memory Dialog 81
  FILTER 614
  FIND 615
  Find 114, 116
  Find Procedure 114, 117
  FINDPROC 616
  Firmware 413
FLASH 687
  Commands 687
  Disabling 551
  Enabling 551
  Loading 553
  Module 549
  Module selecting 551
  Operations 551
  Protecting 551
  Select 549
  Unprotecting 551
  Unselect 549
Flash 262
Flash programming 427, 448
FLASH SELECT 549, 552
Flash.Erase 427
Flash.Idle 427
Flash.Load 427
FLASH_4000 557
FLASH_B32 556
FLASH_C000 557
FLASH_PAGE0 557
FLASH_PAGE1 557
FLASH_PAGE2 557
FLASH_PAGE3 558
FLASH_PAGE4 558
FLASH_PAGE5 558
FLASH_PAGE6 558
FLASH_PAGE7 558
FLEXlm 6
Float 103
FOCUS 617
FOLD 618
Fold 118
Fold All Text 118
Folding 111
Mark 111
Folding Menu 117
Foldings 114
FONT 618
Fonts 28
FOR 619, 632
Format 78, 196, 199
Format mode 140
Format... 66
FPP Browse 553
FPP directory 552
PRINTF 620
Frame 490, 528
FRAMES 620
Frames 105
Frozen 66, 69, 80

G
G 621
GDI 480, 489, 518, 527
GENPATH 727, 729
Global 67
Global Variable
    Displaying 195
GO 622

Go to Frame 493, 531
Go To Line 117
Go to Line 114, 115, 116
Graphical bar 57, 96
Graphical 491, 529
Graphical display 491, 529
GRAPHICS 625
Grid Color 132
Grid Mode 132
Grid Size 132
GUI Graphical User Interface 548

H
Halt 21
Halt when the fifo is full 484, 522
HALTED 424
Hardware 5, 556
hardware breakpoint 423, 433, 741
hardware breakpoints 473, 511
HC08 Full Chip Simulation 231
HC12B32 556
HC12D60 556
HC12DG128 557
HCS08 459, 497
HCS08 Serial Monitor 459, 480, 497, 518
HCS08 Serial Monitor Connection 545
Height 132
HELP 625
Help 420
Help Topics 31
Hex 67, 68, 79, 103, 196, 199
Hexadecimal 196, 199, 202
Hide Headline 17
Hide Tile 17
HIENVIRONMENT 726
High Display Value 134, 136, 141
Highlights 410
HIPATH 727
HiTOP Ready 423
Horiz. Text Alignment 140
Horizontal Size 131
How To
    Debugging an Application in RAM 455
    Starting with the BDIK Target Interface 454
How To ... 187
How to Load 555
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MainFrame</td>
<td>719</td>
</tr>
<tr>
<td>Manual Configuration</td>
<td>237</td>
</tr>
<tr>
<td>Markpoints</td>
<td>465, 470, 503, 508</td>
</tr>
<tr>
<td>markpoints</td>
<td>463, 501</td>
</tr>
<tr>
<td>Marks</td>
<td>114</td>
</tr>
<tr>
<td>match value</td>
<td>478, 480, 516, 518</td>
</tr>
<tr>
<td>MC9S12A32</td>
<td>248</td>
</tr>
<tr>
<td>MC9S12A64</td>
<td>248</td>
</tr>
<tr>
<td>MC9S12C32</td>
<td>249</td>
</tr>
<tr>
<td>MC9S12D32</td>
<td>249</td>
</tr>
<tr>
<td>MC9S12D64</td>
<td>250</td>
</tr>
<tr>
<td>MC9S12DB128A</td>
<td>250</td>
</tr>
<tr>
<td>MC9S12DB128B</td>
<td>251</td>
</tr>
<tr>
<td>MC9S12DG128B</td>
<td>251</td>
</tr>
<tr>
<td>MC9S12DG256B</td>
<td>252</td>
</tr>
<tr>
<td>MC9S12DJ128B</td>
<td>252</td>
</tr>
<tr>
<td>MC9S12DJ256B</td>
<td>253</td>
</tr>
<tr>
<td>MC9S12DJ64</td>
<td>253</td>
</tr>
<tr>
<td>MC9S12DP256B</td>
<td>254</td>
</tr>
<tr>
<td>MC9S12DP512</td>
<td>254</td>
</tr>
<tr>
<td>MC9S12DT128B</td>
<td>255</td>
</tr>
<tr>
<td>MC9S12DT256B</td>
<td>255</td>
</tr>
<tr>
<td>MC9S12XDPS12</td>
<td>256</td>
</tr>
<tr>
<td>MC9S12XDT512</td>
<td>257</td>
</tr>
<tr>
<td>MCCNT</td>
<td>291</td>
</tr>
<tr>
<td>MCCTL</td>
<td>291</td>
</tr>
<tr>
<td>MCU Communication</td>
<td>391</td>
</tr>
<tr>
<td>mcu03c1.fpp</td>
<td>556</td>
</tr>
<tr>
<td>mcu03c3.fpp</td>
<td>556</td>
</tr>
<tr>
<td>mcu03c4.fpp</td>
<td>557</td>
</tr>
<tr>
<td>mcsld</td>
<td>552</td>
</tr>
<tr>
<td>MCUTOOLS.INI</td>
<td>188, 716</td>
</tr>
<tr>
<td>MEM</td>
<td>636</td>
</tr>
<tr>
<td>Memory</td>
<td>466, 504</td>
</tr>
<tr>
<td>Dump</td>
<td>76</td>
</tr>
<tr>
<td>Word</td>
<td>76</td>
</tr>
<tr>
<td>Memory access at Address A</td>
<td>477, 515</td>
</tr>
<tr>
<td>Memory access at Address A and value on data bus match</td>
<td>477, 515</td>
</tr>
<tr>
<td>Memory access at Address A and value on data bus mismatch</td>
<td>478, 516</td>
</tr>
<tr>
<td>Memory access at Address A or at Address B</td>
<td>477, 515</td>
</tr>
<tr>
<td>Memory access at Address A then memory access at Address B</td>
<td>477, 515</td>
</tr>
<tr>
<td>Memory access inside Address A - Address B range</td>
<td>477, 515</td>
</tr>
<tr>
<td>Memory banking</td>
<td>410</td>
</tr>
<tr>
<td>Memory Configuration Modes</td>
<td>236</td>
</tr>
<tr>
<td>Memory Expansion Register</td>
<td>290</td>
</tr>
<tr>
<td>memory model</td>
<td>290</td>
</tr>
<tr>
<td>Memory Write Access</td>
<td>487, 525</td>
</tr>
<tr>
<td>Menu</td>
<td></td>
</tr>
<tr>
<td>Help</td>
<td>31</td>
</tr>
<tr>
<td>Run</td>
<td>20</td>
</tr>
<tr>
<td>Target</td>
<td>22, 27</td>
</tr>
<tr>
<td>View</td>
<td>17</td>
</tr>
<tr>
<td>Window</td>
<td>29</td>
</tr>
<tr>
<td>MicroC</td>
<td>88</td>
</tr>
<tr>
<td>misaligned access</td>
<td>235</td>
</tr>
<tr>
<td>mismatch value</td>
<td>478, 481, 516, 519</td>
</tr>
<tr>
<td>Mode</td>
<td>78</td>
</tr>
<tr>
<td>Modify Trigger</td>
<td>485, 523</td>
</tr>
<tr>
<td>Module</td>
<td>92</td>
</tr>
<tr>
<td>Modules</td>
<td>549</td>
</tr>
<tr>
<td>Modulus Down-Counter</td>
<td>291</td>
</tr>
<tr>
<td>Monitor Communication</td>
<td>405</td>
</tr>
<tr>
<td>MONITOR-HCS08 &gt; Bus Trace</td>
<td>406</td>
</tr>
<tr>
<td>MONITOR-HCS08 &gt; Erase Flash</td>
<td>405</td>
</tr>
<tr>
<td>MONITOR-HCS08 &gt; Monitor Communications...</td>
<td>405</td>
</tr>
<tr>
<td>MONITOR-HCS08 &gt; Select Derivative</td>
<td>406</td>
</tr>
<tr>
<td>MONITOR-HCS08 &gt; Trigger Module Settings...</td>
<td>405</td>
</tr>
<tr>
<td>MOTOR-HCS08 &gt; Vector Mirroring Setup...</td>
<td>405</td>
</tr>
<tr>
<td>Motorola Scalable CAN</td>
<td>257</td>
</tr>
<tr>
<td>MS</td>
<td>637</td>
</tr>
<tr>
<td>ms</td>
<td>106</td>
</tr>
<tr>
<td>Multiplexed External Bus Interface</td>
<td>265</td>
</tr>
</tbody>
</table>

**N**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>550</td>
</tr>
<tr>
<td>NB</td>
<td>638</td>
</tr>
<tr>
<td>NETWORK</td>
<td>421</td>
</tr>
<tr>
<td>New</td>
<td>15</td>
</tr>
<tr>
<td>NF</td>
<td>293</td>
</tr>
<tr>
<td>No Link To Target</td>
<td>423</td>
</tr>
<tr>
<td>NOCR</td>
<td>640</td>
</tr>
<tr>
<td>-Nodefaults</td>
<td>10</td>
</tr>
<tr>
<td>NOLF</td>
<td>640</td>
</tr>
<tr>
<td>Non Volatile Memory</td>
<td>447, 448</td>
</tr>
<tr>
<td>NV_PARAMETER_FILE</td>
<td>552</td>
</tr>
</tbody>
</table>
Object Info Bar 33
OBJPATH 729
Oct 68, 79, 103, 196
Octal 196
OP_SetValue 291
OPEN 641
Open Component 28
Open Configuration 15
Open Memory Block 238
Open Source File 114
Options 188
  Pointer As Array 66
Options - Autosize 30
Options - Component Menu 30
OR 293
OSEK Kernel Awareness 179
OSEK ORTI 179
OSEK RTK Inspector 181
OSPARAM.PRM 174
Outlinecolor 138
OUTPUT 642
Output 290

P
P 643
P&E 459, 497
Parallel port 375
Paste 131
PATH 721
Pause 100
PAUSETTEST 644
PBPORT 702
PE 292
PEDebug 459, 497
Percentage 56, 96
Periodic Interrupt Timer 271
PERIODICAL 329
Periodical 66, 80
PF 293
pins 290
PIX0 290
PIX1 290
PIX2 290
Play 100
Pointer as Array 66, 70
PORT 703

Port 375
Port Integration Module 266
Port to Display 133
PORTT 290
PORTTBitx 291
Postload command file 27, 448
postload.cmd 203
PPAGE 290, 439
PPAGE Banked Memory Area 439
PR0 291
PR1 291
PR2 291
Preference panel 16
Preferences dialog 15
Preload command file 26, 448
preload.cmd 203
PRINTF 645
Procedure Chain 94
-Prod 10
Profiling 410, 454, 475, 496, 513, 534
Program flow rebuild gap 490, 528
Program loading 553
programmed 550
Project 719
PROJECT.INI 22, 718
project.ini 552, 553, 717
Properties 130
Protect DBG FIFO content from unexpected reads 488, 526
protected 550
Protecting 551
PROTOCOL 437, 680
PSMODE 703
PT 292
PTRARRAY 645
Pulse Width Modulator 271
PVCS 731

R
R/W Access 485, 523
R8 293
RAF 293
RAMs 412
RD 646
RDRF 293
RE 292
Read 466, 485, 504, 523
Read Access 470, 508
Read/Write 466, 504
Read/Write Access 470, 508
READACCESS 469, 507
READWRITEACCESS 469, 507
real time 5
Real Time Kernel Awareness 173
Real Time Kernels 173
RECORD 647
Record 100
Record continuously and DO NOT halt on trigger hit 483, 521
Record continuously and halt on trigger hit 483, 521
REGBASE 704
Register 102
Register Block 290
Register values 154, 155, 164
Registration 31
Relative Mode 141
Remove 130
REPEAT 632, 647
Replay 101
Requirements 410
RESET 424, 437, 648, 681
Reset 419
Reset command file 26, 447
Reset Target 23, 232
reset.cmd 203
RESETCYCLES 704
RESETMEM 705
RESETRAM 706
RESETSTAT 706
RESTART 648
Restart 21
RETURN 649
RHAPSODY 88
RIE 292
Right 131
ROMs 412
RS 650
RS-232 serial communication 410
RSRC 292
RTC 290
Run To Cursor 50, 113
RUNNING 423
RWU 292
S
S 651
SAVE 652
Save and Restore on load 465, 470, 503, 508
Save Configuration 15
Save Configuration As 15
Save Layout 130
Save Memory Block 238
SAVEBP 653
SBK 292
SBR 292
SC0BDH 292
SC0BDL 292
SC0CR1 292
SC0CR2 292
SC0DRH 293
SC0DRL 293
SC0SR1 292
SC0SR2 293
SC1BDH 292
SC1BDL 292
SC1CR1 292
SC1CR2 292
SC1DRH 293
SC1DRL 293
SC1SR1 292
SC1SR2 293
SCIInput 292, 294
SCIInputH 292, 293
SCIOutput 293
SCIOutputH 293
Scope... 66
SDI 44
search order 732
Searching Order
Assembly source files 732
C source files 732
Object files source files 732
SEGPORT 707
Select Derivative 406
Selecting 549, 551, 552
Send to Back 131
Send to Front 131
Serial Communication Interface 258, 292
Serial Monitor 480, 518
Serial Peripheral Interface 260
SerialInput 292, 294
SerialOutput 294
SET 654
Set Bank... 420
Set Breakpoint 50, 113
Set DBGCA 467, 505
Set DBGCB 467, 505
Set Target 28
Set Trigger A 473, 511
Set Trigger Address 466, 504
Set Trigger Address B 467, 505
Set Trigger B 473, 511
Set TriggerAddress 461, 499
Set Zero Base 107, 328
SETCOLORS 654
SETCONTROL 707
SETCPU 708
Setcpu command file 234
Setup 130
Setup dialog 422
Setup... 420
Show Breakpoints 50, 113
Show Location 50, 113, 114, 486, 490, 524, 528
Show Markpoints... 463, 501
Show Protocol 422
SHOWCYCLES 708
SHOWPROT 432, 740
Simulation 5, 410, 454
Single Step 21
Size 131
Size of Port 133
SKIPILLEGALBREAK 433, 741
SLAY 655
SLINE 655
Sloping 138
Small Borders. 17
SMEM 656
SMOD 657
SofTec HC08 - CodeWarrior and SofTec HC08 Connection 383
SofTec HC08 - Communication Settings Dialog Box 392
SofTec HC08 - First Steps From Within an Existing Project 388
SofTec HC08 - First Steps Using the Stationery Wizard 384
SofTec HC08 - inDart-HC08 Connection Menu Options 390
SofTec HC08 - MCU Configuration Dialog Box 390
SofTec HC08 - Set Connection Dialog Box 389
SofTec HC08 - Technical Considerations 383
SofTec HC08 - Wizard Connection Selection 387
Softec HC08 Connection 383
Softec HCS08 Connection 535
SPC 658
Splitting View 57
SPROC 659
SREC 660
ST.SectionTitle 1, 3, 229, 231, 383, 395, 457, 535, 545, 563, 565, 567, 677, 711, 713, 735
ST1619-HDS
  Postload command file 27, 234
  Preload command file 26
  Reset command file 26
  Startup command file 26
Start 101, 550
Start recording after trigger hit and halt when the fifo is full 483, 484, 521, 522
Start/Continue 21
Starting an Application 192
startup 717
Startup command file 26, 447
Startup Init List 414
startup.cmd 203
State 550
States 550
Statistics 97
Status Bar 13, 17
Status bar 471, 509
Status Bar Information for the BDIK Target Interface 423
Status Message 423
  BDI Ready 423
  HALTED 424
  No Link To Target 423
  Reset 424
  RUNNING 423
Status register bits 102
Step In 193
  Assembly Instruction 195
  Source Instruction 193
Step Out 21, 193
  Function Call 194
Step Over 21, 193, 194
Using on Windows 95 or Windows NT 4.0/
WIN2000 188
TUPDATE 666
Type 485, 523

U
UDec 68, 80, 103, 196
UNDEF 667
undefined 485, 523
UNFOLD 669
Unfold 118
Unfold All Text 118
Unfolding 111
Mark 111
unprotected 550
Unprotecting 551
Unselecting 549, 552
Unsigned Decimal 196
UNTIL 670
UPDATERATE 670
USELIBPATH 731
User 67
User’s Manual 391

V
VA 676
Variable 429
Address 198
Displaying Global Variables 195
Displaying Local Variables 195
Editing Value 197
Format 63
Local and Global 63
Mode 66
Scope 63
Showing Location 198
Type 63
Value 196
Vector Mirroring Setup... 405
VER 671
VERIFY 434, 742
Version number 31
Vert. Text Alignment 140
Vertical Size 131
VisualizationTool
  7 Segment Display 137
  Analog 133

Bar 134
Bitmap 135, 136
Demo 142
Demo limitation 142
Demo Version Limitations 142
DILSwitch 136
Instrument 132
Knob 136
LED 137
Setup 132
Switch 138
Text 139
Voltage Regulator 262
Vppoff command file 448
Vppon command file 448

W
-W 10
WAIT 672
WAKE 292
WATCHPOINT 425
Watchpoint
  Checking condition 160
  Command 165
  Conditional 164, 172
  Counting 163, 172
  Definition 143
  Deleting 165, 171
  Read 161, 170
  Read, Write 144
  Read/Write 162, 171
  WATCHPOINT 425
  Write 162
  watchpoint 460, 466, 498, 504
WB 673
WHILE 632, 674
Width 133
Windows 715
WinEdit 715
WL 675
WOMS 292
Word 79
Word size 78
WorkDir 188
Working Mode 416
WorkingDirectory 188
WPORT 709
Write 466, 485, 504, 523
Write Access 470, 508
WRITEACCESS 469, 507
WW 675

X
X-Position 132

Y
Y-Position 132

Z
ZOOM 676
Zoom in 66