HC(S)12(X) Debugger Manual

Revised: February 18, 2006

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. CodeWarrior is a trademark or registered trademark of Freescale Semiconductor, Inc. in the United States and/or other countries. All other product or service names are the property of their respective owners.

Copyright © 1989 - 2006 by Freescale Semiconductor, Inc. All rights reserved.

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	http://www.freescale.com/codewarrior
Technical Support	http://www.freescale.com/support

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 9
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

Main Window Menu Bar 13
File Menu. 14
Preferences Window 16
View Menu. 18
Customizing the Toolbar 18
Run Menu 21
Connection Menu 24
Loading a Connection 25
Connection Command File Window 27
Component Menu 29
Window Menu. 31
Help Menu. 33
About Box 34
Component Associated Menus 34
Component Main Menu 35
Component Files 35
Component Windows Object Info Bar 35
Component Popup Menu. 36
Highlights of the User Interface 37
Activating Services with Drag and Drop 37
To Drag and Drop an Object 38
Drag and Drop Combinations 39
Dragging from Assembly Component Window 39
Dragging from Data Component Window. 39
Dragging from Source Component Window 40
Dragging from the Memory Component Window 41
Dragging from Procedure Component Window 42
Dragging from Register Component Window. 42
Dragging from Module Component Window 43
Selection Dialog Box. 43

3 Debugger Components 45

Component Introduction. 45
CPU Components 45
Window Components 45

Connection Components 46
Loading Component Windows 46
General Debugger Components 48
Assembly Component 49
Assembly Menu 49
Setting Breakpoints 50
Associated Popup Menu 51
Command Line Component 54
Command Menu 56
Cache Size 57
Coverage Component 58
Coverage Operations 59
Coverage Menu 59
Output File 60
Split View Associated Popup Menu 61
DA-C Link Component 63
DA-C Link Operation 63
DA-C Link Menu 63
Drag Out 64
Drop Into 64
Demo Version Limitations 64
Data Component 65
Data Operations 65
Expression Editor 66
Expression Command file 67
Data Menu 68
Scope Submenu 68
Format Submenu 69
Format Selected & All Sub Menu 70
Mode Submenu 70
Options Submenu 72
Zoom and Sort Submenus 74
Associated Popup Menu 74
Memory Component 78
Memory Operations 79

Memory Menu 79
Display Submenu 82
Fill Memory 83
Display Address 83
CopyMem Submenu 85
Search Pattern 85
Update Rate 86
Associated Popup Menu 87
MicroC Component 90
MicroC Link Menu 90
MicroC DLLs 91
Module Component 94
Module Operations 94
Module Menu 94
Procedure Component 96
Procedure Operations 96
Procedure Menu 96
Profiler Component 98
Profiler Operations 99
Profiler Output File Functions 100
Recorder Component 102
Recorder Operations 102
Recorder Menu 103
Register Component 105
Status Register Bits 105
Editing Registers 105
Register Menu (Format Submenu) 106
Drop Into: 107
SoftTrace Component 108
SoftTrace Operations 108
SoftTrace Menu 108
Associated Popup Menu 109
Source Component 111
Folding and Unfolding 114
Source Menus 115

Open Source File 119
Go to Line 119
Find Operations 120
Find Procedure. 121
Folding Menu 121
Visualization Utilities 123
Inspector Component 124
Inspector Operations 128
Inspector Menu 129
Associated Popup Menu 129
VisualizationTool Component. 131
Edit Mode and Display Mode 132
VisualizationTool Menu. 133
Associated Popup Menu 134
VisualizationTool Properties 136
Instruments 136

4 Control Points 147

Introduction. 147
Breakpoints 148
Breakpoints Tab. 150
Multiple Selections in List Box. 151
Checking Expressions 151
Saving Breakpoints 152
Setting Breakpoints. 154
Positions Where a Breakpoint Is Definable 154
Temporary Breakpoints 155
Setting Temporary Breakpoints 155
Permanent Breakpoints 156
Setting Permanent Breakpoints 156
Counting Breakpoints 157
Setting Counting Breakpoints 157
Conditional Breakpoints 158
Setting Conditional Breakpoints 158
Deleting Breakpoints. 159

Associate a Command with a Breakpoint 160
Demo Version Limitations 160
Watchpoints 161
Watchpoints Tab 163
Multiple Selections 164
Checking Syntax 164
Setting Watchpoints 165
Setting a Read Watchpoint 165
Setting a Write Watchpoint 166
Defining a Read/Write Watchpoint 166
Defining a Counting Watchpoint 167
Defining a Conditional Watchpoint 168
Deleting a Watchpoint 169
Associate a Command with a Watchpoint 169
Demo Version Limitations 170
Markpoints 171
Markpoints Tab 173
Setting Markpoints 174
Setting a Source Markpoint 174
Setting a Data Markpoint 174
Setting a Memory Markpoint 175
Deleting a Markpoint 175
Halting on a Control Point 176

5 Real Time Kernel Awareness 177

Introduction 177
Inspecting Task State 178
RTK Interface 178
Task Description Language 178
Application Example 180
Inspecting Kernel Data Structures 181
RTK Awareness Register Assignments 182
OSEK Kernel Awareness 183
OSEK ORTI 183
ORTI File and Filename 183

ORTI File Structure 184
OSEK RTK Inspector Component 184
Inspector Task 185
Inspector Stack. 187
Inspector SystemTimer 187
Inspector Alarm 188
Inspector Message 189

6 How To ... 191

How To Configure the Debugger 192
For Use from Desktop (Win 95, Win 98,Win NT4.0 or Win2000). 192
Defining the Default Directory in the MCUTOOLS.INI 192
How To Start the Debugger 193
From WinEdit 193
Automating Debugger Startup 194
How To Load an Application 195
How To Start an Application 196
How To Stop an Application. 196
How To Step in the Application 197
On Source Level 197
On the Next Source Instruction 197
Step Over a Function Call (Flat Step) 198
Step Out from a Function Call. 198
Step on Assembly Level 199
How To Work on Variables. 199
Display Local Variable from a Function 199
Display Global Variable from a Module 199
Change Format for Variable Value Display 200
Modify a Variable Value 201
Get the Address Where a Variable is Allocated 202
Inspect Memory Starting at a Variable Location Address 202
Load an Address Register with the Address of a Variable 202
How To Work on the Register. 203
Change Format of Register Display 203
Modify a Register Content 203

Modify Index or Accumulator Register Content . . . 203
Modify Bit Register Content . . . 204
Start Memory Dump at Address Where Register Is Pointing . . . 204
Modify Content of Memory Address . . . 205
How to Consult Assembler Instructions Generated by a Source Statement . . . 205
How To View Code . . . 206
How to Communicate with the Application . . . 207
About startup.cmd, reset.cmd, preload.cmd, postload.cmd . . . 207

7 CodeWarrior Integration 209

Debugger Configuration . . . 209

8 Debugger DDE Capabilities 211

Introduction . . . 211
DDE Implementation . . . 211
Driving Debugger through DDE . . . 211

9 Synchronized Debugging Through DA-C IDE 213

Configuring DA-C IDE for Freescale Tool Kit . . . 213
Create New Project . . . 214
Configure Working Directories . . . 214
Configure File Types . . . 216
Configure Library Path . . . 216
Adding Files to Project . . . 218
Building The Database . . . 219
Configuring The Tools . . . 221
Compiler Configuration . . . 221
Linker Configuration . . . 222
Maker Configuration . . . 223
Debugger Interface . . . 224
DA-C IDE and Debugger Communication . . . 225
Communication DLL Installation . . . 225
Debugger Properties Configuration . . . 226
Debugger Project File Configuration . . . 227
Synchronized Debugging . . . 229

Troubleshooting 229

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

Book II Contents 233

10 HC12 Debugging First Steps 235

Technical Considerations 235
- Full Chip Simulation Considerations 235
- HCS12 Serial Monitor Considerations 235
- SofTec HCS12 Considerations 236
- ICD-12 Considerations 236
- BDIK Considerations 236

Debugging First Steps Using the Wizard 237
Switching Connections 247
Loading the Full Chip Simulation Connection 247
Loading the ICD-12 Connection 249
Switching to SofTec HC12 253
Switching to HCS12 Serial Monitor Connection 255

11 HC(S)12(X) Full Chip Simulation Connection 259

- Full Chip Simulation Menu 259
  - Debugger Status Bar with Full Chip Simulation 260
  - Open I/O Component Dialog Box 261
  - Command File Window 262
- Memory Configuration 263
  - Memory Configuration Dialog Box Features 263
  - Access Details Dialog Box 268
  - Output 269
- Clock Frequency Setup 269
- Bus Trace 270

Full Chip Simulation Warnings. 271
WARNING_SETUP Command 272
MESSAGE_HIDE_ID Command 274
MESSAGE_SHOW_ID Command. 274
MESSAGE_HIDE_RESET Command 274
FCS and Silicon On-chip Peripherals Simulation. 275
Supported Derivatives. 276
Communication Modules 285
Byteflight (BF) 285
J1850 Bus (BLCD) 285
Motorola Scalable CAN (MSCAN) 285
Inter-IC Bus (IIC) 286
Serial Communication Interface 286
Serial Peripheral Interface 288
Converter Modules 288
Analog to Digital Converter 288
Memory Modules 290
EEPROM (EETS) 290
Flash (FTS) 290
Miscellaneous Modules. 290
Voltage Regulator (VREG) 290
Debug Module (DBG). 291
S12X_INT 291
XGATE 291
Port I/O Modules. 292
External Bus Interface (EBI) 292
Module Mapping Control (MMC). 293
Multiplexed External Bus Interface (MEBI) 293
Port Integration Module (PIM) 294
Timer Modules 294
Clock and Reset Generator (CRG) 294
Blocks:. 294
Enhanced Capture Timer (ECT) 296
Periodic Interrupt Timer (PIT). 299
Pulse Width Modulator (PWM). 299

Timer Module (TIM) 301
Legacy HC12 (CPU12) Derivatives Simulation 301
MC68HC812A4. 301
HC912DG128x, HC912DT128x 317
FCS Visualization Utilities 323
Analog Meter Component. 323
Analog Meter Operations 324
Analog Meter Menu. 324
IO_LED Component 325
IO_LED Operations. 325
IO_LED Menu 325
Demo Version Limitations. 326
LED Component 327
LED Operations. 327
LED Menu 327
Phone Component 329
Phone Operations. 330
ADC/DAC Component 331
ADC/DAC Menu. 333
ADC/DAC - Setup Dialog Box 333
Conversion Parameters Dialog Box 334
Display Properties Dialog Box 335
IT_Keyboard Component 336
IT_Keyboard Menu 337
Interruption Keyboard Setup 337
LCD Component 339
LCD Operation 339
Instruction Listing 340
The Initialization Step 342
LCD Menus 343
LCD Display 344
Monitor Component 345
Monitor Menu 345
Add Channel 346
Monitor Settings 346

Change Colors . . . 347
Push Buttons Component . . . 349
Push Buttons Menu . . . 349
Push Buttons Setup . . . 349
Use with IO_Ports . . . 350
Use with LEDs Component . . . 350
Programmable IO_Ports Component . . . 352
Programmable IO_Ports Menu . . . 352
Port Address . . . 353
7-Segments Display Component . . . 354
7-Segments Display Menu . . . 355
7-Segments Display Setup . . . 355
Stimulation Component . . . 357
Stimulation Popup Menu . . . 357
Example of a Stimulation File . . . 358
TestTerm Component . . . 360
Output Redirection . . . 361
How to Redirect . . . 362
Using TestTerm . . . 362
TestTerm Menu . . . 363
Terminal Component . . . 365
Configure Terminal Connections . . . 366
Input and Output File . . . 367
File Control Commands . . . 367
How to Use Virtual SCI . . . 369
Wagon Component . . . 370
Wagon Menu . . . 370
Wagon Setup . . . 370
True Time I/O Stimulation . . . 372
Stimulation Program Examples . . . 372
Running an Example Program Without Stimulation . . . 372
Example Program with Periodical Stimulation of a Variable . . . 374
Example Program with Stimulated Interrupt . . . 375
Example of a Larger Stimulation File . . . 377
Stimulation Input File Syntax . . . 380

EBNF 380
Electrical Signal Generators and Signals Application to Device Pins 382
Signal IO Component 382
Signal Description File EBNF. 382
Signal File Format 382
File Example 1. 383
File Example 2. 384
File Parameters 384
Signal IO Usage. 385
Signal Commands 385
Remarks 386
Base Signal Files Provided 386
Virtual Wire Connections with the Pinconn IO Component. 387
Pinconn IO. 387
Command Set to Apply Signal on ATD Pin 388
FCS Tutorials 389
Guess the Number. 389
Step 1 - Environment Setup. 389
Step 2 - Creating the project 389
Step 3 - ‘Target CPU’ Window 390
Step 4 - ‘Bean Selector’ Window 391
Step 5 - ‘Project Panel’ Window 392
Step 6 - ‘Bean Inspector AS1:AsynchroSerial’ Window 393
Step 7 - Generation of Driver Code. 394
Step 8 - Verification of Files Created 394
Step 9 - Entering User Code 395
Step 10 - Run 397
PWM Channel 0 397
Step 1 - Environment Setup. 398
Step 2 - Creating Project 398
Step 3 - ‘Target CPU’ Window 398
Step 4 - Creating PWM Bean 399
Step 5 - ‘Project Panel’ Window 399
Step 6 - ‘Bean Inspector PWM8.PWM 399
Step 7 - Generate Driver Code. 399

Step 8 - Verification of Files Created 399
Step 9 - Entering User Code 400
Step 10 - Run 400
VisualizationTool Properties 401
'Chart' Properties 401
Period 'Bar Properties' 401
Duty Time 'Bar Properties' 402

12 P&E Multilink/Cyclone Pro Connection 405

Introduction. 405
Windows NT Installation Notice 405
Interfacing Your System and P&E Multilink/Cyclone Pro. 406
Loading the P&E Multilink/Cyclone Pro Connection 407
Default Connection Setup. 411

13 Softec HCS12 Connection 413

Technical Considerations 413
inDart-HCS12 Menu Options 413
MCU Configuration Option. 414
MCU Configuration Dialog Box. 414
Communication Settings Dialog Box 414
User's Manual Option 416
About Option. 416

14 HCS12 Serial Monitor Connection 417

Serial Monitor Technical Considerations 417
CodeWarrior and Serial Monitor Connection 417
HCS12 Serial Monitor Interface 418
MONITOR-HCS12 Menu Options 421
Monitor Communication... 422
Vector Mirroring Setup... 422
Erase Flash. 422
Trigger Module Settings... 422
Bus Trace 422
Select Derivative 422

Monitor Setup Window . . . 422
Monitor Communication Tab . . . 423
Vector Table Mirroring Tab . . . 424
Derivative Selection Dialog Box. . . . 425

15 Abatron BDI Connection 427

Abatron BDI Highlights . . . 427
Abatron BDI Requirements . . . 428
Abatron BDI Connection Introduction . . . 428
Interfacing Abatron BDI andYour System . . . 429
BDI Interface Software Setup. . . . 430
Running the ABATRON Configuration Tool . . . 430
Example with B10C12.EXE Configuration Tool . . . 430
Firmware Loading . . . 431
Initialization List (Startup Init List) Loading . . . 432
Communication with the Debugger Setup. . . . 433
BDI Working Mode and Setup/List Transmission. . . . 434
Loading the Abatron BDI Connection . . . 434
Abatron BDI Connection Menu Entries . . . 437
Load... . . . 437
Reset . . . 437
Communication... or Connect... . . . 437
Setup.... . . . 438
Configure BDI Box... . . . 438
Set Bank... . . . 438
Command Files . . . 438
Help . . . 438
Abatron BDI Connection Windows, Edit and Dialog Boxes . . . 438
Communication Device Specification Edit Box . . . 439
Setup Dialog Box . . . 440
Abatron BDI Status Bar Information . . . 441
Status Messages. . . . 441
BDI ready V x.xx. . . . 441
No Link To Target . . . 441
RUNNING. . . . 441

HALTED 442
RESET 442
Stepping and Breakpoint Messages. 442
STEPPED 442
STEPPED OVER 442
STOPPED 442
TRACED 442
BREAKPOINT 442
WATCHPOINT 443
Terminal Emulation 443
Example for CPU12 Targets:. 443
Flash Programming 445
Examples of HC12/CPU12 Direct Commands 445
Abatron BDI Connection Environment 447
Default Connection Setup 447
HC12 and HCS12 Banked Memory support 447
Banked Memory Location Window 447
PPAGE Tab 448
DPAGE Tab 449
EPAGE Tab 449
Various Tab (Not For All Connections). 450

Book III - HC(S)12(X) Debug Connections - Common Features

Book III Contents 451

15 HCS12 On-chip DBG Module 453

16 HCS12X On-chip DBG Module 455

17 Debugging Memory Map 457

Introduction. 457
Debugging Memory Map GUI 458
Edition Dialog Box 459
Types 460
Priorities. 461
Remarks. 462
CPU Core Types and Priorities 462
HC12(CPU12) Core 462
Priorities: 462
Types:. 463
HCS12 Core 464
Priorities: 464
Types:. 465
HCS12X Core 465
Priorities: 465
Types:. 465
DMM Commands 466

18 HC(S)12(X) Flash Programming 467

Non-volatile Memory Control Utility Introduction 467
Automated Application Programming 467
Setup 468
Advanced Options: Preventing Erasing 469
NVMC Graphical User Interface 470
Modules and Module States. 470
NVMC Dialog Box 471
Flash Module Handling 473
MCU Speed Information 474
Configuration: FPP File Loading 474
Loading an Application in Flash 475

Preparing, Loading an Application in Flash 477
Hardware Considerations 478
HC12 (CPU12) CPU devices 478
HC12B32. 478
HC12D60. 478
HC12DG128 479
HCS12 and HCS12X CPU devices. 480
HCS12 EEPROM’s Relocation. 481
EB386 Compliancy and RAM Moving. 482
Legacy Flash Programming Commands in Preload and Postload Command Files 482

Book IV - Commands

Book IV Contents 485

20 Debugger Engine Commands 487

Commands Overview 487
Command Syntax 488
Available Command Lists 488
Kernel Commands 488
Base Commands 489
Environment Commands 491
Component Commands 492
Component Specific Commands 493
Command Syntax Terms 496
Module Names. 497
Debugger Commands 497
A 498
ACTIVATE 498
ADDXPR 499
ATTRIBUTES. 499
In the Command Component. 499

In the Procedure Component 500
In the Assembly Component 500
In the Register Component 501
In the Source Component 503
In the Data Component 504
In the Memory Component 505
In the Inspector Component 507
AT 509
AUTOSIZE 509
BASE. 510
BC 510
BCKCOLOR 511
BD 512
BS 513
CALL 515
CD 516
CF 517
CLOCK 519
CLOSE 519
COPYMEM. 520
CMDFILE 520
CR 521
CYCLE 521
DASM 522
DB 523
DDEPROTOCOL 525
DEFINE. 526
DETAILS. 527
DL 528
DUMP 529
DW 529
E 530
ELSE 531
ELSEIF 531
ENDFOCUS 532

ENDFOR 533
ENDIF 533
ENDWHILE 534
EXECUTE. 534
EXIT 535
FILL. 535
FILTER 536
FIND 537
FINDPROC 538
FOCUS 539
FOLD. 540
FONT. 540
FOR 541
FPRINTF 542
FRAMES 542
G 543
GO 544
GOTO 545
GOTOIF. 546
GRAPHICS 547
HELP 547
IF 548
INSPECTOROUTPUT 549
INSPECTORUPDATE 549
LF. 550
LOAD 551
LOADCODE 552
LOADSYMBOLS 552
LOG 553
More About Logging of IF, FOR, WHILE and REPEAT 554
LS. 557
MEM 558
MS 559
NB 560
NOCR 562

NOLF 562
OPEN 563
OUTPUT 564
P. 565
PAUSETEST 566
PRINTF 567
PTRARRAY 567
RD 568
RECORD. 569
REPEAT 569
RESET. 570
RESTART 570
RETURN. 571
RS 572
S. 573
SAVE. 574
SAVEBP 575
SET 576
SETCOLORS 576
SLAY. 577
SLINE 577
SMEM 578
SMOD 579
SPC 580
SPROC 581
SREC. 582
STEPINTO 583
STEPOUT 584
STEPOVER. 585
STOP 586
T 587
TESTBOX. 588
TUPDATE. 588
UNDEF 589
UNFOLD. 591

UNTIL 592
UPDATERATE 592
VER 593
WAIT 594
WB 595
WHILE 596
WL 597
WW 597
ZOOM 598

21 Debugger Connection-specific Commands 599

Abatron BDI Connection Commands 599
Banked Memory Location-associated Commands 602
BANKWINDOW Examples 604
ICD-12 Commands 606
Command Files and ICD-12 Commands. 606
ICD-12 Memory Configuration Commands 606
NVMC Commands 608
FLASH. 608
[<*blockNo*>] 611
DMM Commands 614
Debugging Memory Map Manager Commands .614
"DMM" Command 614
"DMM ADD" command. 614
"DMM DEL" Command 615
"DMM SAVE" Command 615
"DMM DELETEALLMODULES" Command 615
"DMM RELEASECACHES" Command 616
"DMM CACHINGON" Command 616
"DMM CACHINGOFF" Command 616
"DMM HCS12MERHANDLINGON" Command 616
"DMM HCS12MERHANDLINGOFF" Command. 616
"DMM OPENGUI" Command 617
Full Chip Simulator Commands 617
FCS-Associated Component-specific Commands 617

ADDCHANNEL 619
ADCPORT 619
CPORT 620
DELCHANNEL 620
ITPORT 621
ITVECT 621
KPORT 622
LCDPORT 622
LINKADDR 623
PBPORT 623
PORT 624
PSMODE 624
REGBASE 625
RESETCYCLES 625
RESETMEM 626
RESETRAM 627
RESETSTAT 627
SEGPORT 628
SETCONTROL 628
SETCPU 629
SHOWCYCLES 629
WPORT 630

Book V - Environment Variables

Book V Contents 631

22 Debugger Engine Environment Variables 633

Debugger Environment 634
The Current Directory 635
Global Initialization File (MCUTOOLS.INI - PC Only) 636
Local Configuration File (usually project.ini) 637
Default Layout Configuration (PROJECT.INI) 638

Ini File Activation 640
Environment Variable Paths 641
Line Continuation 642
Environment Variables 643
ABSPATH: Absolute Path 644
DEFAULTDIR: Default Current Directory 645
ENVIRONMENT=: Environment File Specification 646
GENPATH: #include “File” Path 647
LIBRARYPATH: ‘include <File>’ Path 648
OBJPATH: Object File Path 649
TMP: Temporary directory 650
USELIBPATH: Using LIBPATH Environment Variable 651
Search Order for Source Files 652
In the Debugger for C Source Files (*.c, *.cpp) 652
In the Debugger for Assembly Source Files (*.dbg) 652
In the Debugger for Object Files (HILOADER) 652
Debugger Files 652

23 Connection-specific Environment Variables 655

Connection-specific Environment Variables 655
Abatron BDI Connection Environment Variables 655
Example 658
Banked Memory Location-associated Environment Variables 660
ICD-12 Environment 663
ICD-12 Environment Variables 663
ICDPORT Variable 663
BMDELAY Variable 664

Index 665

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 avilable in the debugger.

- Chapter 1.1 "Introduction" on page 5
- Chapter 1.2 "Debugger Interface" on page 7
- Chapter 1.3 "Debugger Components" on page 45
- Chapter 1.4 "Control Points" on page 147
- Chapter 1.5 "Real Time Kernel Awareness" on page 177
- Chapter 1.6 "How To ..." on page 191
- 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 "HC12 Debugging First Steps" on page 235
- Chapter 2.2 "HC(S)12(X) Full Chip Simulation Connection" on page 259
- Chapter 2.3 "P&E Multilink/Cyclone Pro Connection" on page 405
- Chapter 2.4 "Softec HCS12 Connection" on page 413
- Chapter 2.5 "HCS12 Serial Monitor Connection" on page 417
- Chapter 2.6 "Abatron BDI Connection" on page 427

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

- Chapter 3.1 "HCS12 On-chip DBG Module" on page 453
- Chapter 3.2 "HCS12X On-chip DBG Module" on page 455
- Chapter 3.3 "Debugging Memory Map" on page 457

- Chapter 3.4 "HC(S)12(X) Flash Programming" on page 467

Book 4: Commands

- Chapter 4.1 Debugger Engine Commands on page 487
- Chapter 4.2 Debugger Connection-specific Commands on page 599

Book 5: Environment Variables

- Chapter 5.1 Debugger Engine Environment Variables on page 633
- Chapter 5.2 Connection-specific Environment Variables on page 655

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" on page 5 Chapter introduces the Debugger concept.
- The "Debugger Interface" on page 7 Chapter provides all details about the Debugger user interface environment i.e., menus, toolbars, status bars and drag and drop facilities.
- The "Debugger Components" on page 45 Chapter contains descriptions of each basic component and visualization utility.
- The "Control Points" on page 147 Chapter is dedicated to the control points and associated windows.
- The "Real Time Kernel Awareness" on page 177 Chapter contains descriptions of the Real Time concept and related applications.
- The "How To ..." on page 191 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 on page 211 describe the debugger DDE features.
- The Synchronized Debugging Through DA-C IDE on page 213 chapter explains the use of tools with the DA-C IDE from RistanCase

1

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.

2

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 on page 7
- Application Programs on page 7
- Starting the Debugger on page 8
- Debugger Main Window on page 11
- Component Associated Menus on page 34
- Highlights of the User Interface on page 37

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 windoe 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

`hibase.dll`	Debugger main function dll
`elfload.dll`	Debugger loader dll
`*.wnd`	Debugger component
`*.tgt`	Debugger target file
`*.cpu`	Debugger CPU awareness file

Starting the Debugger

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

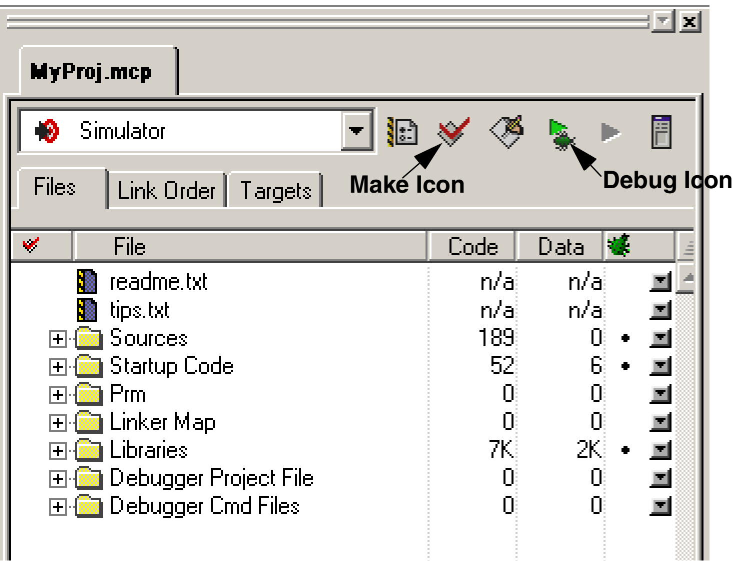
Starting from within the IDE

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 on page 8), at the top of the Project window.

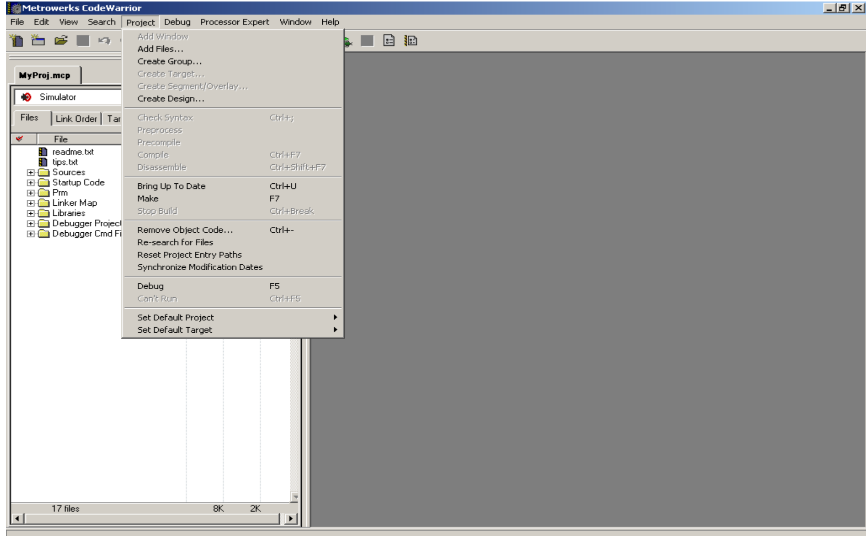
Figure 2.1 Project Window Make and Debug Icons



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**.)

Figure 2.2 Main Window Project Menu



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. Preceed 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:

```
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 l(W) was specified
3. Execute command file <cmdFile> if specified
4. Execute command if specified
5. *Start running unless option l(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.

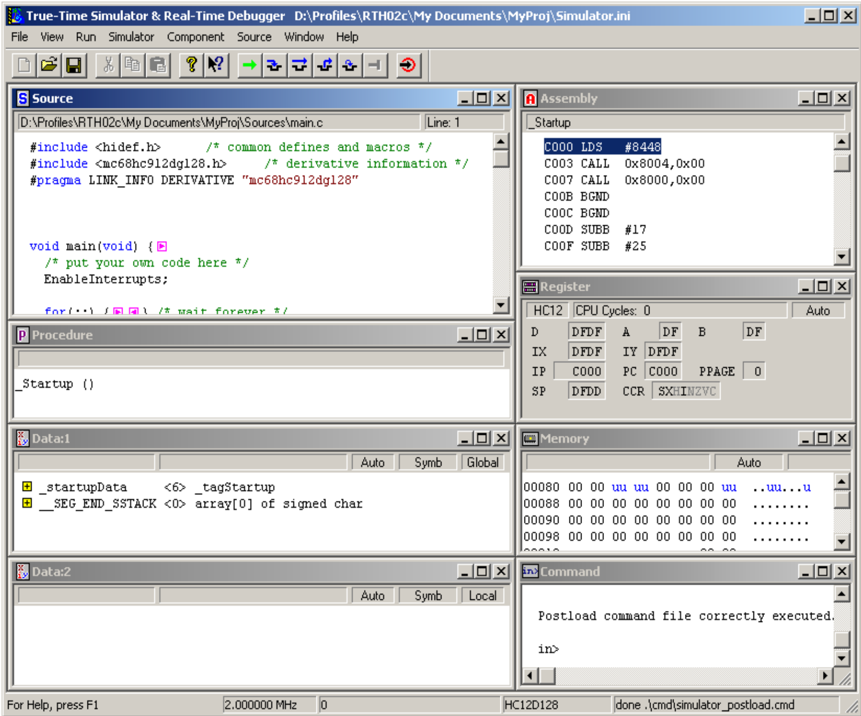
Example

```
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.

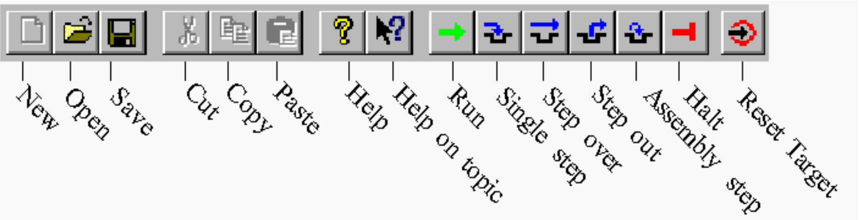
Figure 2.3 Debugger Main Window



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 on page 12 identifies each default icon.

Figure 2.4 Debugger Main Window Toolbar

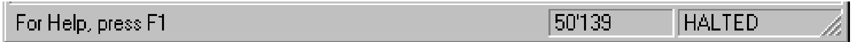


A tool tip is available when you point the mouse at an icon.

Debugger Main Window Status Bar

The status bar at the bottom of the Debugger Main Window, shown in Figure 2.5 on page 13 contains a context sensitive help line for connection specific information, e.g., number of CPU cycles for the **Simulator** connection and execution status.

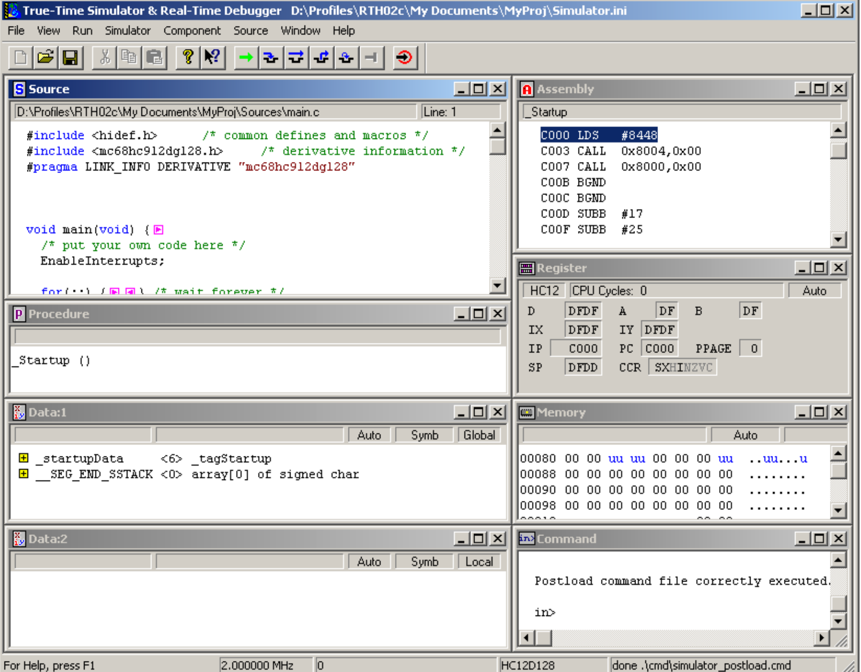
Figure 2.5 The Debugger Status Bar



Main Window Menu Bar

The Debugger Main Window Menu Bar, shown in Figure 2.6 on page 13 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 on page 14 describes menu entries available in the menu bar.

Table 2.1 Description of the Main Menu Toolbar Entries

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

File Menu

The **File** menu shown in Figure 2.7 on page 15 is dedicated to the debugger project.

Figure 2.7 File Menu

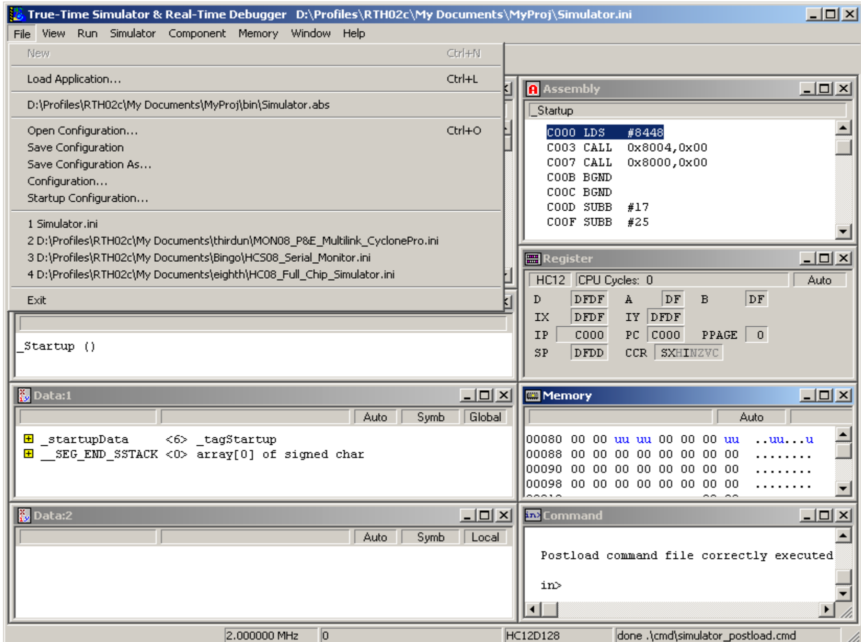


Table 2.2 on page 15 describes File Menu entries.

Table 2.2 File Menu Entry Description

Menu Entry	Description
New	Creates a new project.
Load Application	Loads an executable file (or debugger connection if nothing is selected).
...\restart.abs ...\await.abs ...	Recent applications list
Open Configuration	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.

Table 2.2 File Menu Entry Description

Menu Entry	Description
Save Configuration	Saves the project file
Save Project As	Opens the debugger project window to save the project file under a different path and name, and format (PJT; INI...).
Configuration	Opens the Preferences window to set environment variables for current project.
1.Project.ini 2.Test.ini 3...	Recent project file list
Exit	Quits the Debugger.

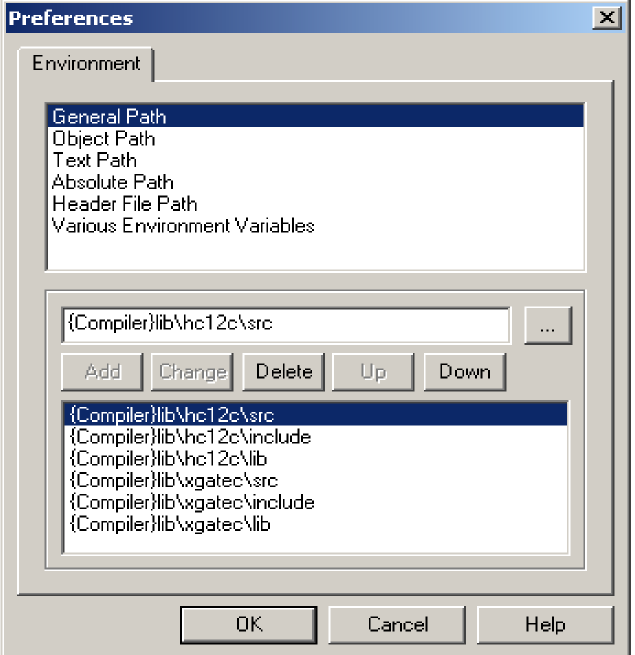
You can shortcut some of these functions by clicking toolbar icons (refer to the Debugger Main Window Toolbar on page 12 section).

Preferences Window

Open the Preferences window by selecting **Configuration** from the **Files** menu. With this window (Figure 2.8 on page 17) it is possible to set up environment variables for the current project. New variables are saved in the current project file when you clickthe **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.

View Menu

In the Main Window View menu (Figure 2.9 on page 18) you can choose to show or hide the toolbar, status bar, window component titles and headlines (see the Component Windows Object Info Bar on page 35). You can select smaller window borders and customize the toolbar. Table 2.3 on page 18 describes the View Menu entries.

Figure 2.9 View Menu

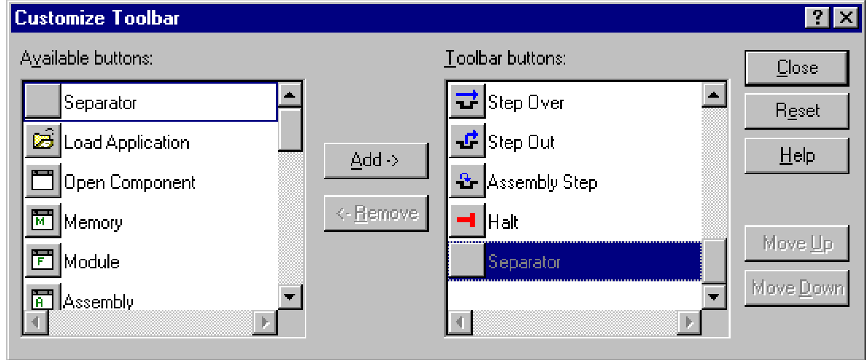
Table 2.3 View Menu Description

Menu Entry	Description
Toolbar	Check / uncheck Toolbar if you want to display or hide it.
Status Bar	Check / uncheck Status Bar if you want to display or hide it.
Hide Tile	Check / uncheck Hide Title if you want to hide or display the window title.
Hide Headline	Check / uncheck Hide Headline if you want to hide or display the headline.
Small Borders.	Check / uncheck Small Border if you want to display or hide small window borders.
Customize	Opens the debugger Customize Toolbar window.

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 on page 19.

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 on page 19 shows a typical component window display.

Figure 2.11 Typical Component Window Display

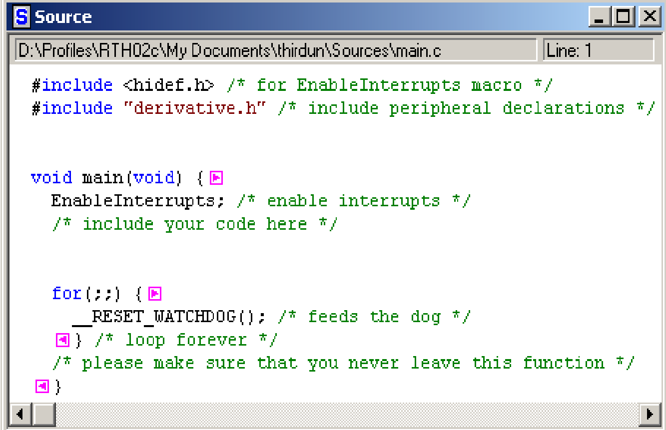


Figure 2.12 on page 20 shows a component window without a title and headline.

Figure 2.12 Component Window without Title and Headline

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

  void main(void)
```

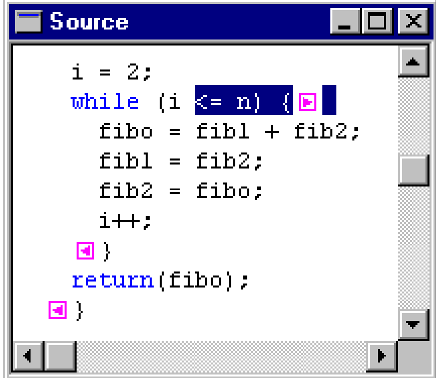
Figure 2.13 on page 20 shows a component window without a title and headline, and with a small border.

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

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

  void main(void)
```

Figure 2.14 on page 21 shows a component window without headline and small border

Figure 2.14 Component Window without Headline and Small Border



Run Menu

The Main Window Run menu, shown in Figure 2.15 on page 22 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 on page 22.

Figure 2.15 Run Menu

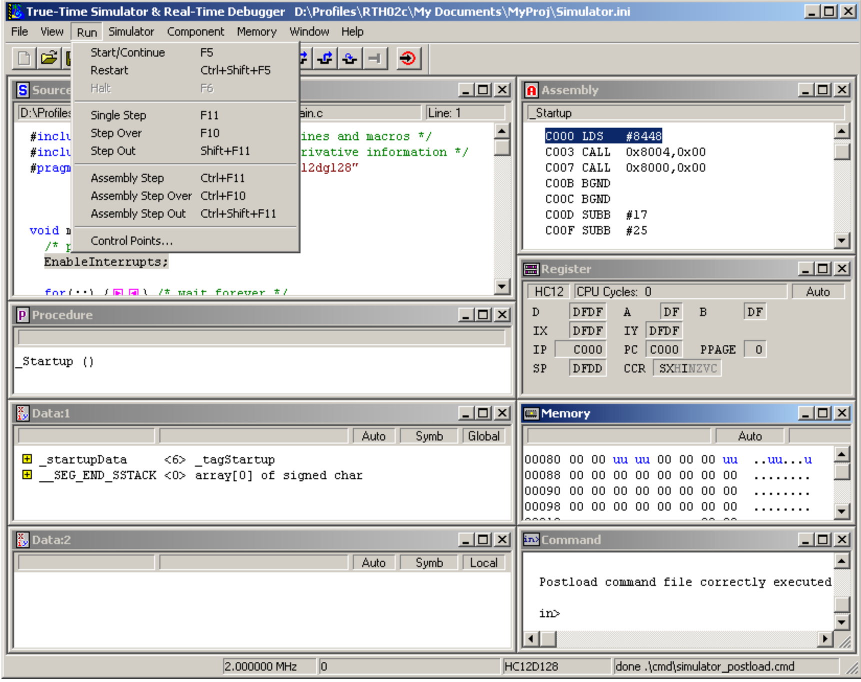


Table 2.4 Run Menu Description

Menu entry	Description
Start/Continue	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 -> Halt. Shortcut: **F5 key**
Restart	Starts execution of the loaded application from its entry point. Shortcut: **CTRL + Shift + F5 keys**
Halt	Interrupts and halts a running application. You can examine the state of each variable in the application, set breakpoints, watchpoints, and inspect source code. Shortcut: **F6 key**

Table 2.4 Run Menu Description (*continued*)

Menu entry	Description
Single Step	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. Shortcut: **F11 key**
Step Over	Similar to the **Single Step** command, but does not step into called functions. A function call is treated as one statement. Shortcut: **F10 key**
Step Out	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. Shortcut: **Shift + F11 keys**
Assembly Step	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. Shortcut: **CTRL + F11 keys**
Assembly Step Over	Similar to the **Step Over** command, but steps over subroutine call instructions. Shortcut: **CRTL + F10 keys**
Assembly Step Out	If the application is halted inside a function, this command continues execution and stops on the CPU instruction following the current function invocation. This command is similar to the **Step Out** command, but stops before the assignment of the result from the function call. Shortcut: **CTRL + Shift + F11 keys**
Control Points	Opens the Controlpoints Configuration Window which contains tabs that allow you to control Breakpoints, Watchpoints and Markpoints (refer to Control Points on page 147 chapter).

You can provide shortcuts for some of these functions using the toolbar. Refer to the Debugger Main Window Toolbar on page 12 and Customizing the Toolbar on page 18 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 on page 147 chapter.

Connection Menu

This menu entry (Figure 2.16 on page 24) 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 on page 29 section for details.

Figure 2.16 Connection Menu

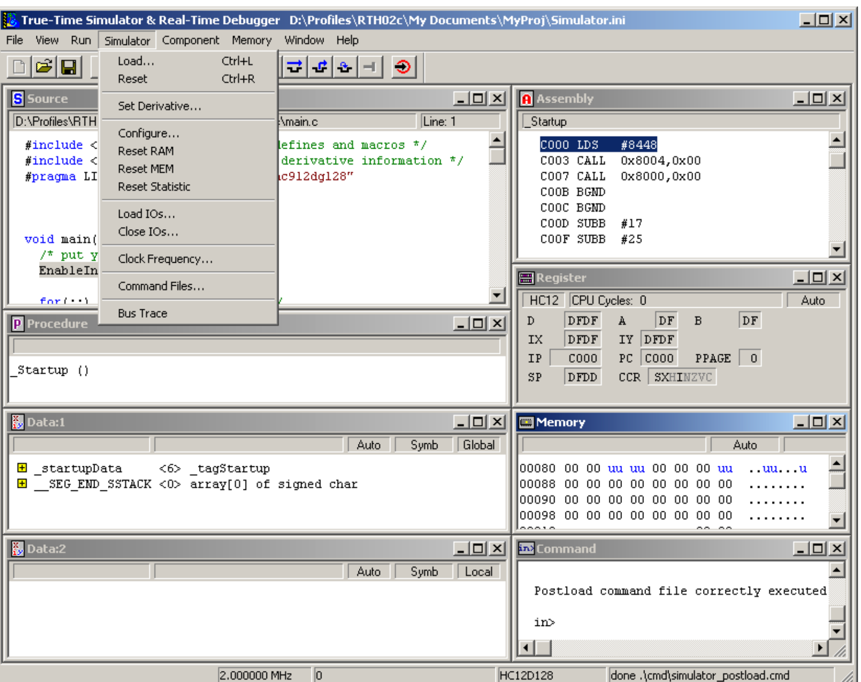


Table 2.5 on page 25 describes the Connection Menu entries.

Table 2.5 Connection Menu Common Option Description

Menu Entry	Description
Load	Loads a connection.
Reset	Resets the current connection.

Loading a Connection

Use the Connection menu to load a debugger connection.

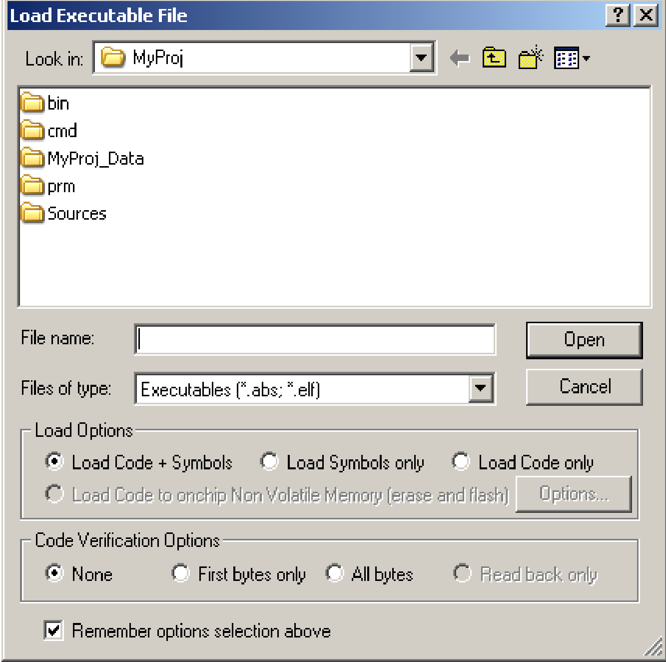
1. Choose **Connection>Load...**

   The Load Executable File window shown in Figure 2.17 on page 26 is displayed:

Load Executable File Window

From the Connection menu, choose **Load...** to open the Load Executable File window, shown in Figure 2.17 on page 26, then set the load options and choose a Simulation Execution Framework (an `.ABS` application file).

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

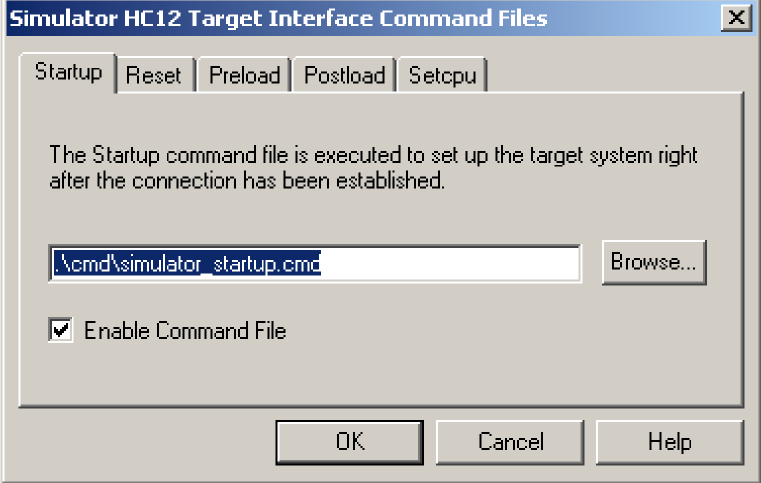
The checkboxes and buttons of this area of the Load Executable File window offer the dollowing options:

- 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 on page 27 corresponds to an event on which a command file can be automatically run from the . See the Startup Command File on page 28, Reset Command File on page 28, Preload Command File on page 28, and Postload Command File on page 29, sections that follow.

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 on page 27.

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 on page 27.

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 on page 27.

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 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 on page 27.

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 on page 29.

Figure 2.19 Component Menu

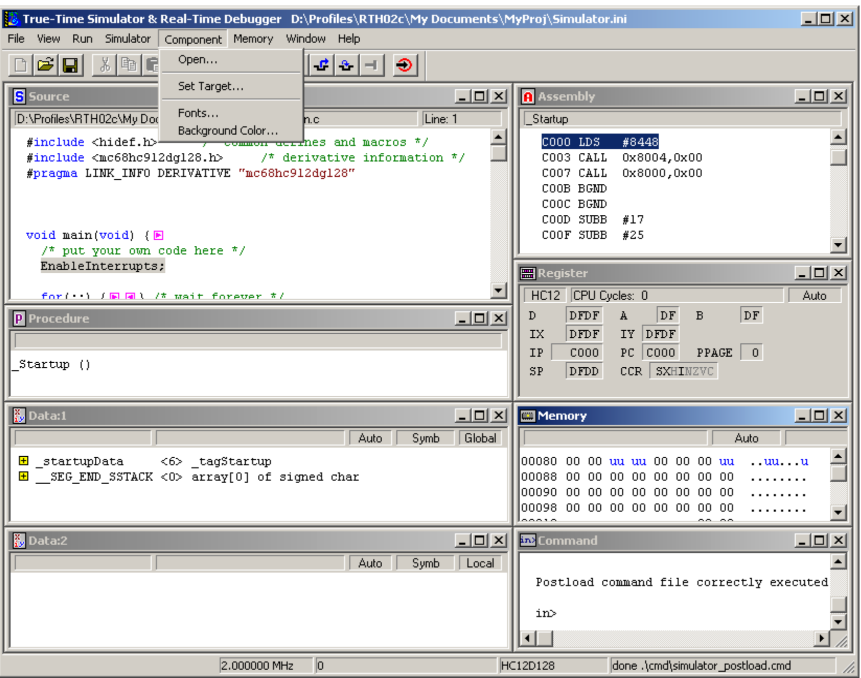


Table 2.6 on page 30 describes the Component Menu entries.

Table 2.6 Component Menu Description

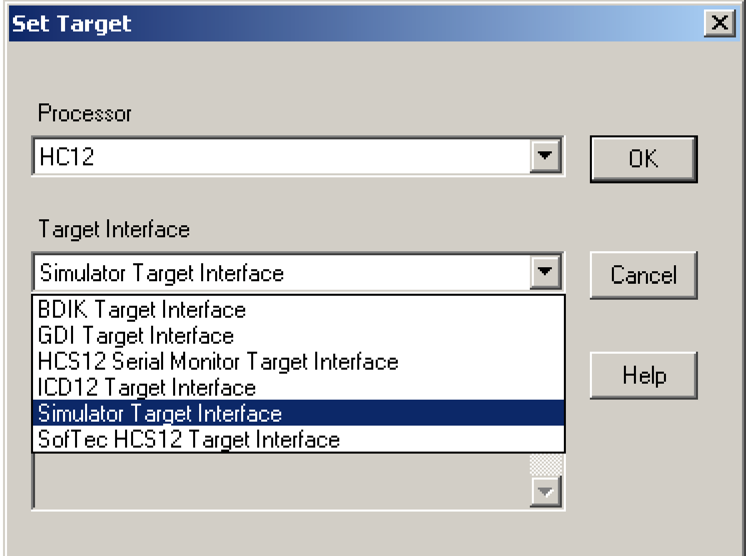
Menu entry	Description
Open	Loads an extra component window that has not been loaded by the Debugger at startup. The popup dialog presents a set of different components that are introduced in the Typical Component Window Display on page 19 section.
Set Connection	Sets the Debugger connection.
Fonts	Opens a standard Font Selection dialog, where you can set the font used by Debugger components.
Background Color	Opens a standard Color Selection dialog, where you can set the background color used by the Debugger component windows.

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 on page 45.

Select **Component>Set Connection...** and the **Set Connection** dialog box shown in Figure 2.20 on page 31 is opened.

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 on page 32, you can set the component windows general arrangement. The Submenu **Window>Options** is shown in Figure 2.22 on page 32 and the Submenu **Window>Layout** in Figure 2.23 on page 32.

Figure 2.21 Window Menu

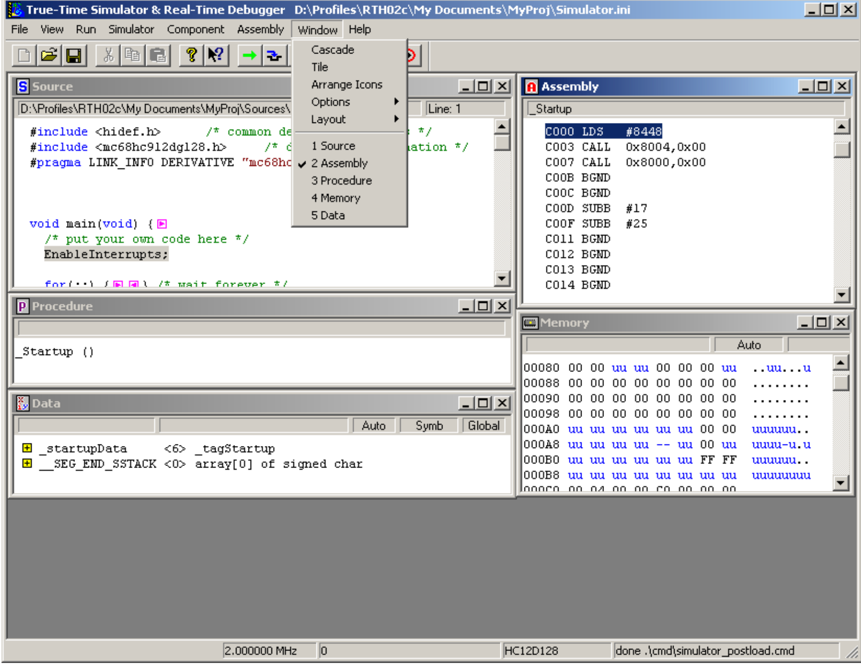


Figure 2.22 Window Menu Options SubMenu

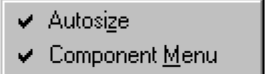


Figure 2.23 Window Menu Layout SubMenu

Table 2.7 on page 33 specifies the Window Menu entries.

Table 2.7 Window Menu Description

Menu entry	Description
Cascade	Option to arrange all open windows in cascade (so they overlap).
Tile	Option to display all open windows in tile format (non overlapping).
Arrange Icons	Arranges icons at the bottom of windows.
Options - Autosize	Component windows always fit into the debugger window whenever you modify the debugger window size.
Options - Component Menu	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.
Layout - Load/Store	Option to **Load / Store** your arrangements from a `.HWL` file.

NOTE Autosize and Component Menu are checked by default.

Help Menu

This is the Debugger Main window Help menu (Figure 2.24 on page 33). Table 2.8 on page 33 shows menu entries.

Figure 2.24 Help Menu

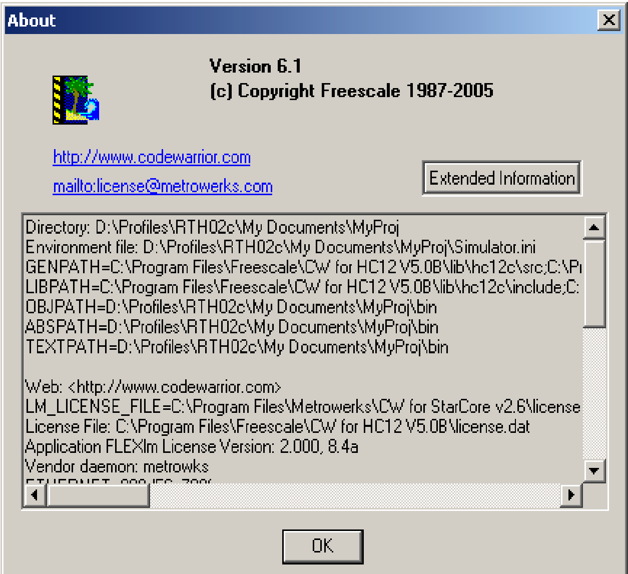
Table 2.8 Help Menu Description

Menu entry	Description
Help Topics	Choose **Help Topics** in the menu for online help or if you need specific information about a topic.
About ...	Information about the debugger version and copyright, and license information is displayed.

About Box

Select **Help>about** to display the About box, shown in Figure 2.25 on page 34. 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`.

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.

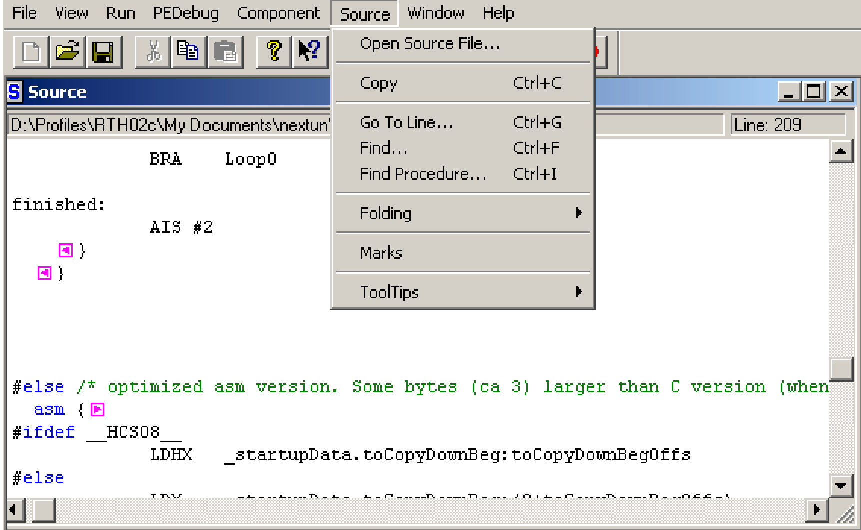
Component Associated Menus

Vaious Debugger Component windows are shown in Figure 2.3 on page 12. 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 on page 35 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.**

Figure 2.26 Example of Component Main Menu



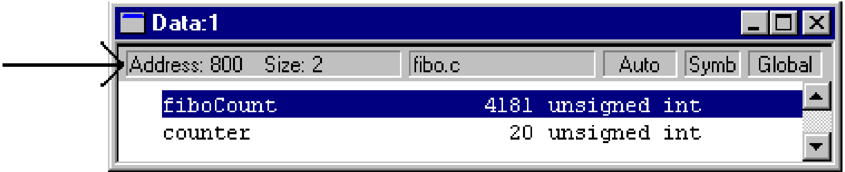
Component Files

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 on page 36, provides information about the selected object.

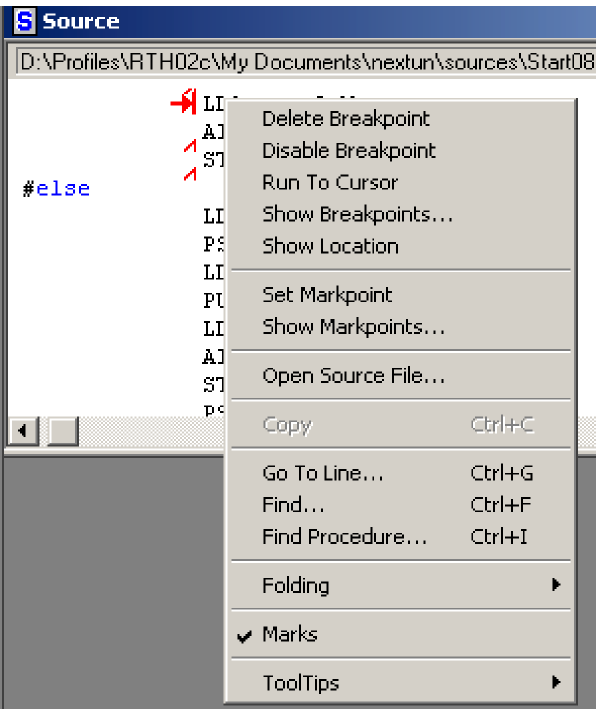
Figure 2.27 Object Info Bar of Debugger Component Windowss



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.

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 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 on page 37.

Figure 2.29 Drag and Drop Example

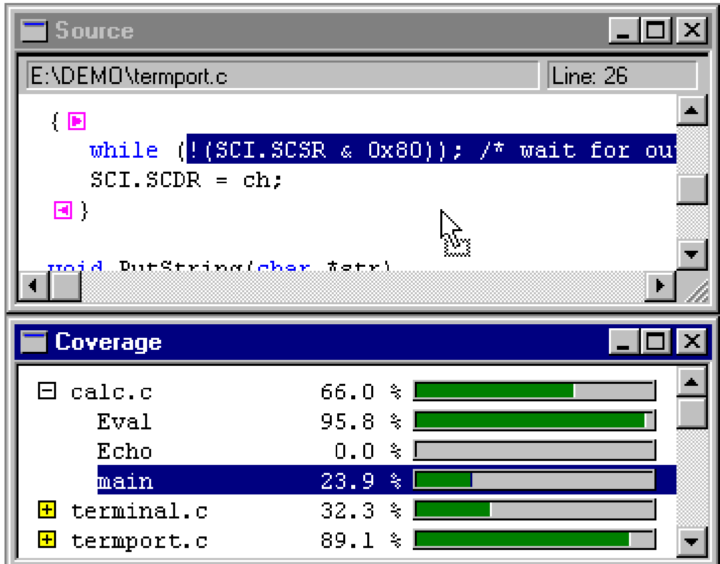
```
Source
E:\DEMO\fibo.c                          Line: 26
  i = 2;
  while (i <= n) {
    fibo = fib1 + fib2;
    fib1 = fib2;
    fib2 = fibo;
    i++;
  }
  return(fibo);
}
```

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 on page 38).

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 on page 39 summarizes dragging from the Assembly Component.

Table 2.9 Dragging from the Assembly Component Window

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

Dragging from Data Component Window

Table 2.10 on page 40 summarizes dragging from the Data Component.

Table 2.10 Dragging from the Data Component Window

Destination Component Window	Action
Command Line	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.
Memory	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.
Register	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.
Source	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.

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 on page 40 summarizes dragging from the Source Component.

Table 2.11 Dragging from the Source Component Window

Destination Component Window	Action
Assembly	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
Register	Loads the destination register with the PC of the first instruction selected.

Table 2.11 Dragging from the Source Component Window

Destination Component Window	Action
Memory	Displays the memory area corresponding with the high level language source code selected. The memory area corresponding to the selected instructions are greyed in the memory component.
Data	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. (please see Data Component on page 65 and Expression Editor on page 66)

Dragging from the Memory Component Window

Table 2.12 on page 41 summarizes dragging from the Memory Component.

Table 2.12 Dragging from the Memory Component Window

Destination Component Window	Action
Assembly	Displays disassembled instructions starting at the first address selected. Instructions corresponding to the selected memory area are highlighted in the Assembly component.
Command Line	Appends the selected memory range to the Command Line window
Register	Loads the destination register with the start address of the selected memory block.
Source	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.

Dragging from Procedure Component Window

Table 2.13 on page 42 summarizes dragging from the Procedure Component.

Table 2.13 Dragging from the Procedure Component Window

Destination Component Window	Action
Data > Local	Displays local variables from the selected procedure in the data component
Source	Displays source code of the selected procedure. Current instruction inside the procedure is highlighted in the Source component.
Assembly	The current assembly statement inside the procedure is highlighted in the Assembly component.

Dragging from Register Component Window

Table 2.14 on page 42 summarizes dragging from the Register Component Window.

Table 2.14 Dragging from the Register Component Window

Destination Component Window	Action
Assembly	Assembly component receives an address range, scrolls to the corresponding instruction and highlights it.
Memory	Dumps memory starting at the address stored in the selected register. The corresponding address is selected in the memory component.

Dragging from Module Component Window

Table 2.15 on page 43 summarizes dragging from the Register Component.

Table 2.15 Dragging from the Module Component Window

Destination Component Window	Action
Data > Global	Displays global variables from the selected module in the data component
Memory	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.
Source	Displays source code from selected module.

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

3

Debugger Components

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

- Component Introduction on page 45
- Loading Component Windows on page 46
- General Debugger Components on page 48
- Visualization Utilities on page 123

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.

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.

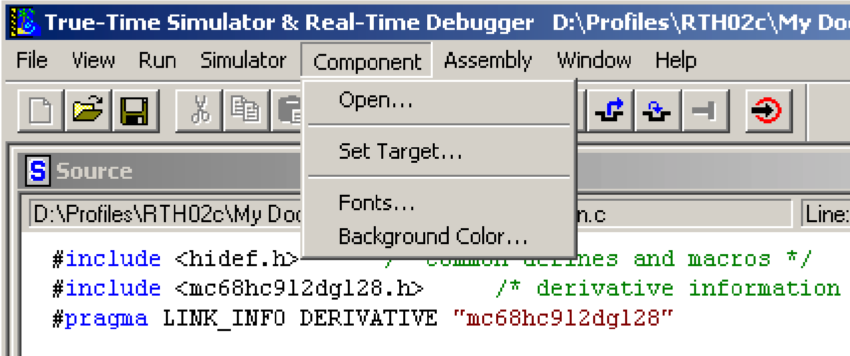
Different connections are available to connect the target system (hardware) to the debugger. For example, the connection may be connected using a Full Chip Simulator, an Emulator, a ROM monitor, a BDM pod cable, or any other supported device.

NOTE Connection components are introduced in their respective manuals.

Loading Component Windows

In the Debugger Main Window Menu Bar, shown in Figure 3.1 on page 46, 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

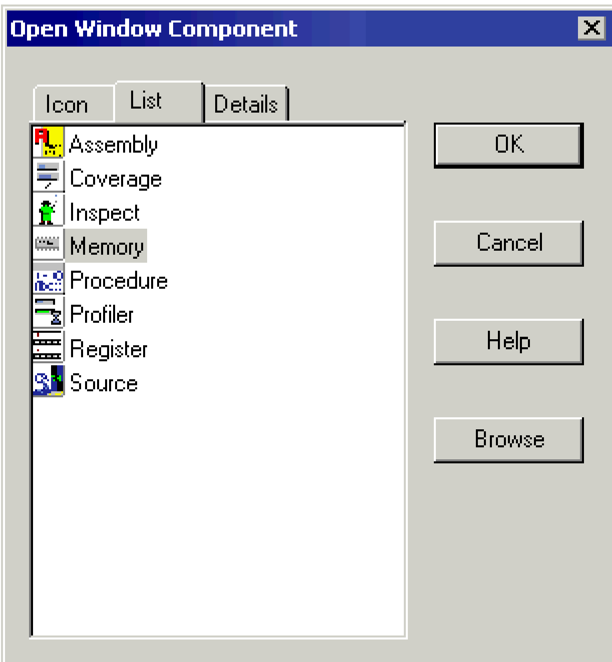


To open the window that lets you choose one or more components:

1. Choose **Component>Open...**
2. In the Open Window Component window shown in Figure 3.2 on page 47, select the desired component.

NOTE To open more than one component, select multiple components.

Figure 3.2 Open Window Component Window



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.

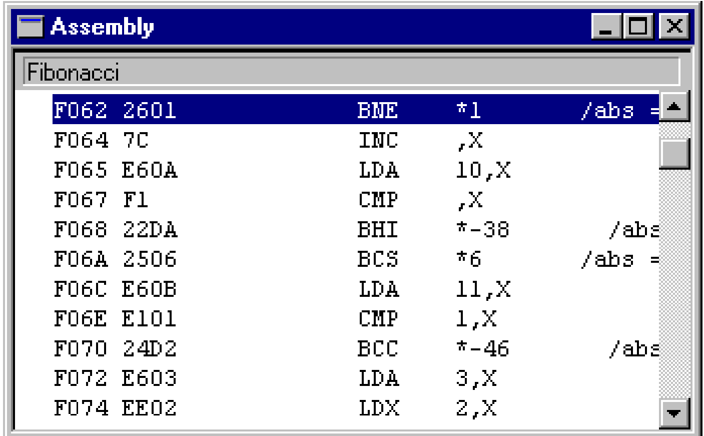
General Debugger Components

This chapter describes the various features and usage of the debugger components.

Assembly Component

The Assembly window, shown in Figure 3.3 on page 49, 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 on page 50 contains all functions associated with the assembly component. Theses entries are described in Table 3.1 on page 50.

Figure 3.4 Assembly Menu

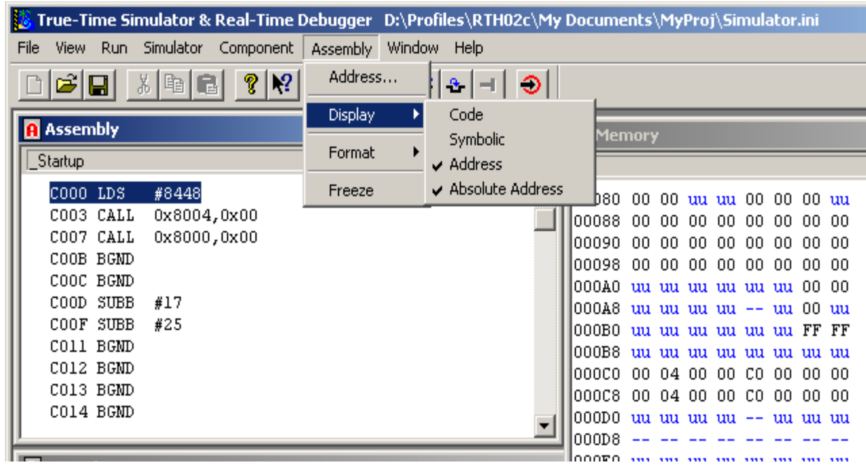


Table 3.1 Assembly Menu Description

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

Setting Breakpoints

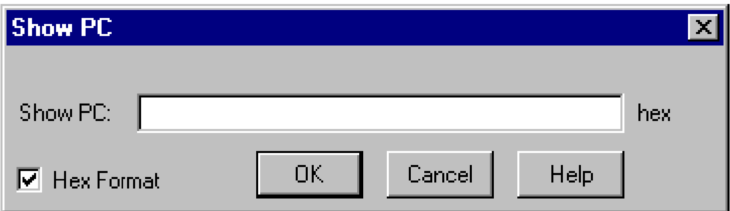
Breakpoints can be set, edited and deleted when using the popup menu. Right-click on any statement in the Source component window, then choose Set Breakpoint, Delete Breakpoint, etc.

NOTE For information on using breakpoints, see Control Points on page 147 chapter.

Show PC Dialog Box

If a hexadecimal address is entered in the Show PC dialog box shown in Figure 3.5 on page 51, 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 on page 52; depending on the current state of the debugger.

Figure 3.6 Assembly Popup Menu

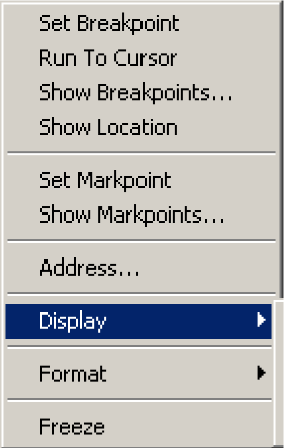


Table 3.2 Assembly Popup Menu Description

Menu Entry	Description
Set Breakpoint	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.
Delete Breakpoint	Appears in popup menu if a breakpoint is set or disabled on the specified instruction. When selected, deletes this breakpoint.
Enable Breakpoint	Appears only in popup menu if a breakpoint is disabled on an instruction. When selected, enables this breakpoint.
Disable Breakpoint	Appears in the popup menu if a breakpoint is set on an instruction. When selected, disables this breakpoint.
Run To Cursor	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.
Show Breakpoints	Opens the Controlpoints Configuration Window Brealpoints Tab and displays list of breakpoints defined in the application (refer to Control Points on page 147).
Show Location	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.
Set Markpoint	When selected, enables you to set a markpoint at this location.
Delete Markpoint	Appears in the Popup Menu only if a markpoint is set at the nearest code position (visible with marks). When selected, disables this markpoint.
Show Markpoints	Opens the Controlpoints Configuration Window Markpoints Tab and displays list of markpoints defined in the application (refer to Control Points on page 147).
Address...	For a description of the remaining popup menu entries see Table 3.1 “Assembly Menu Description” on page 50.

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 on page 53 shows the drag actions possible from the Assembly component.

Table 3.3 Assembly Component Drag Actions

Destination Component Window	Action
Command Line	The Command Line component appends the address of the pointed to instruction to the current command.
Memory	Dumps memory starting at the selected instruction PC. The PC location is selected in the memory component.
Register	Loads the destination register with the PC of the selected instruction.
Source	Source component scrolls to the source statements and highlights it.

Drop Into:

Table 3.4 on page 54 shows the drop actions possible in the Assembly component

Table 3.4 Drop Into Assembly Component

Source Component Window	Action
Source	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
Memory	Displays disassembled instructions starting at the first address selected. Instructions corresponding to the selected memory area are highlighted in the Assembly component.
Register	Displays disassembled instructions starting at the address stored in the source register. The instruction starting at the address stored in the register is highlighted.
Procedure	The current assembly statement inside the procedure is highlighted in the Assembly component.

Demo Version Limitations

No limitation

Associated Commands

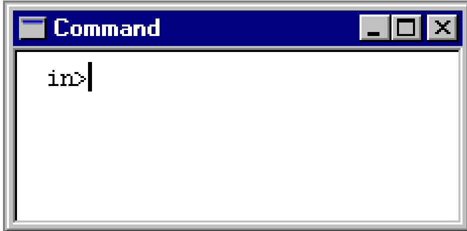
Following commands are associated with the Assembly component:

ATTRIBUTES on page 499, SMEM on page 578, SPC on page 580.

Command Line Component

The Command Line window shown in Figure 3.7 on page 55 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.

Figure 3.7 Command Line Window



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 funtion 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 on page 519 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 on page 56 shows the Command menu, which is identical to the Command Popup menu.

Figure 3.8 Command Menu

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.

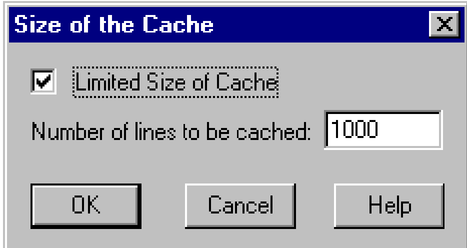
The first line of text contained in the clipboard can be pasted where the caret is blinking (end of current line) by:

- 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 on page 57.

Figure 3.9 Cache Size Dialog Box



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 on page 58. The command line component appends corresponding items of the current command.

Table 3.5 Drop Into Command Component

Source Component Window	Action
Assembly	The Command Line component appends the address of the pointed to instruction to the current command.
Data	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.
Memory	Appends the selected memory range to the Command Line window
Register	The address stored in the pointed to register is appended to the current command.

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 on page 512, CF on page 517, E on page 530, HELP on page 547, NB on page 560, LS on page 557, SREC on page 582, SAVE on page 574.

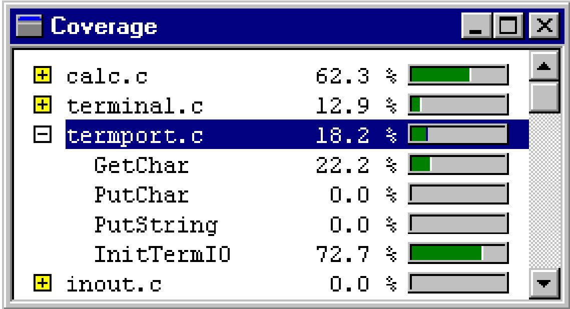
NOTE For more details about commands, refer to Debugger Engine Commands on page 487.

Coverage Component

The Coverage window, shown in Figure 3.10 on page 59 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.

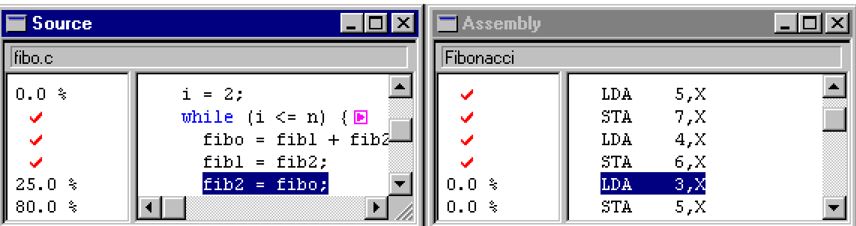
Figure 3.10 Coverage Window



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 on page 59. 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.

Figure 3.11 Split Views



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 on page 60.

Figure 3.12 Coverage Menu

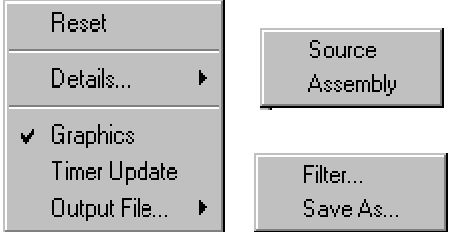


Table 3.6 Coverage Menu Description

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

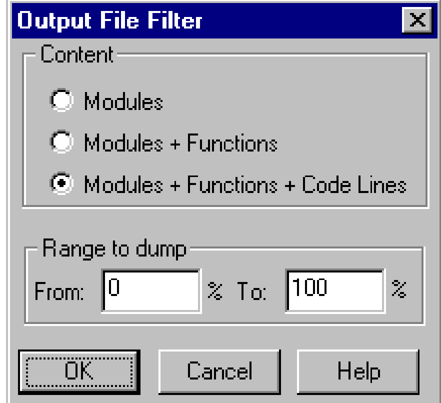
Output File

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 on page 61. 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.

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 on page 61.

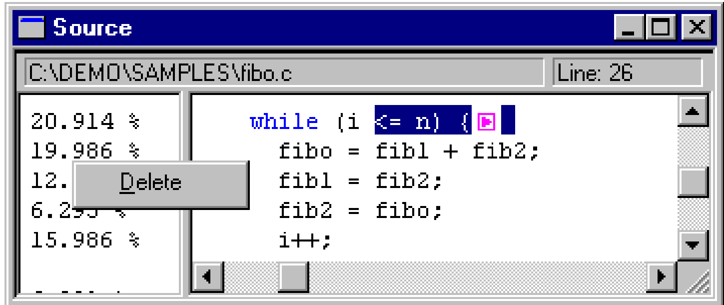
Listing 3.1 Example Output File with Modules and Functions:

```
------------------------------------------------------------
Coverage:      Item:
------------------------------------------------------------
 94.4 %        Application
 FULL          fibo.c
                  Fibonacci()
 FULL             main()
 86.0 %        startup.c
 80.5 %           Init()
 FULL             _Startup()
```

Split View Associated Popup Menu

The popup menu for the split view (Figure 3.14 on page 62) contains the **Delete** entry, which is used to remove the split view.

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 on page 527, FILTER on page 536, GRAPHICS on page 547, OUTPUT on page 564, RESET on page 570, TUPDATE on page 588

DA-C Link Component

The DA-C Link window shown in Figure 3.15 on page 63 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 on page 213.

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

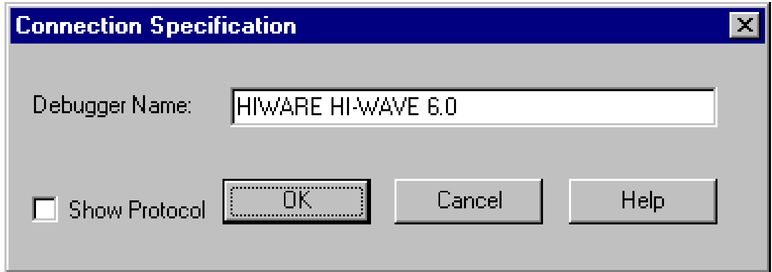
Table 3.7 DA-C Link Menu Description

Menu Entry	Description
Setup	Opens the Connection Specification dialog box.

Connection Specification Dialog Box

In the Connection Specificaton dialog box you can set the DA-C debugger name.

Figure 3.17 Connection Specification Dialog Box



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 on page 229 section in the DA-C documentation.

Drag Out

Nothing can be dragged out.

Drop Into

Nothing can be dropped into the DAC Component window.

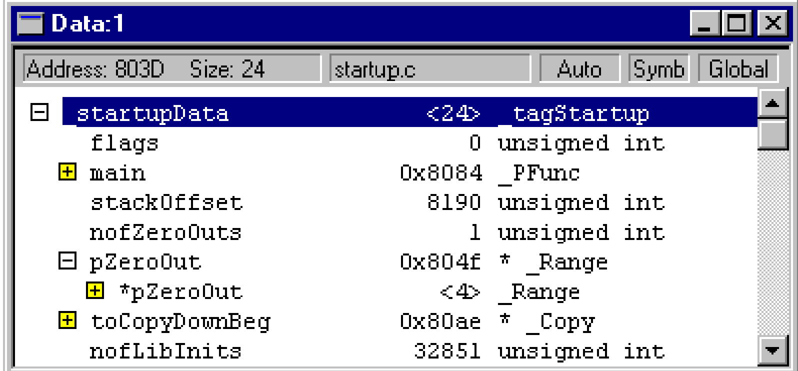
Demo Version Limitations

None.

Data Component

The Data window shown in Figure 3.18 on page 65 contains the names, values and types of global or local variables.

Figure 3.18 Data Window



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 on page 35 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 on page 147 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.

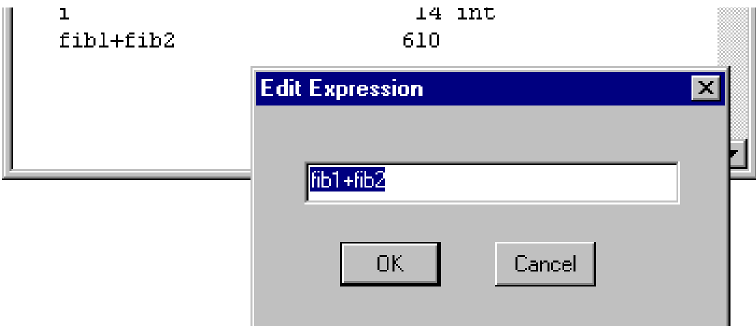
- Select a variable in the Data component, and left mouse button + W key to set a “Write” watchpoint on the selected variable. A red vertical bar is displayed on the left side of the variables on which a write watchpoint has been defined. If write access is detected on the variable during execution, 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 on page 67, 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.

Figure 3.19 Edit Expression Dialog Box



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`

Data Menu

Figure 3.20 on page 68 shows the Data component menu, the Zoom submenu is shown in Figure 3.29 on page 74, the Scope submenu is shown in Figure 3.21 on page 69, the Format submenu in Figure 3.22 on page 69, the Mode submenu in Figure 3.24 on page 71, the Options submenu in Figure 3.26 on page 72 and the Zoom and Sort submenus in Figure 3.29 on page 74. Data Menu entries are described in Table 3.8 on page 68.

Figure 3.20 Data Menu

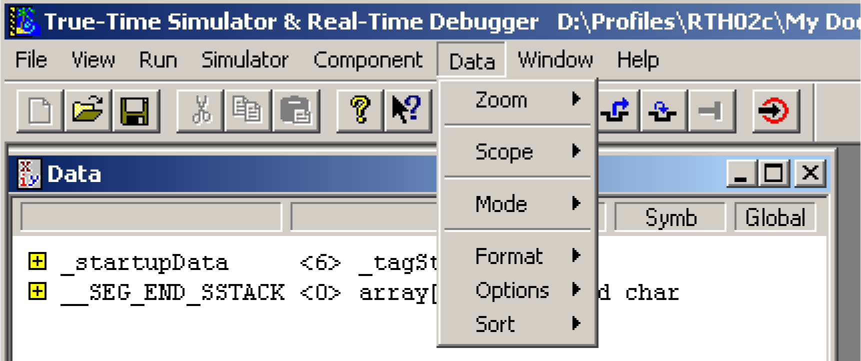


Table 3.8 Data Menu Entry Description

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

Scope Submenu

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

Figure 3.21 Scope Submenu

Table 3.9 on page 69 describes the Scope submenu entries.

Table 3.9 Scope Submenu Entries

Menu Entry	Description
Global	Switches to **Global** variable display in the Data component.
Local	Switches to **Local** variable display in the Data component.
User	Switches to **User** variable display in the Data component. Displays user defined expression (variables are erased).

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 on page 69 describes the Format submenu entries.

Table 3.10 Format Sub Menu Entries

Menu Entry	Description
Selected	The changes will be applied to the selection only
All	The changes will be applied to all items

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

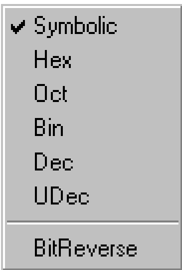


Table 3.11 on page 70 describes the Format Selected Mode & Format All Mode Sub Menu entries.

Table 3.11 Format Selected & All Sub Menu

Menu entry	Description
Symbolic	Select the **Symbolic** (display format depends on the variable type) display format. This is the default display.
Hex	Select the hexadecimal data display format
Bin	Select the binary data display format
Oct	Select the octal data display format
Dec	Select the signed decimal data display format
UDec	Select the unsigned decimal data display format
Bit Reverse	Select the bit reverse data display format (Each bit is reversed).

Mode Submenu

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

Figure 3.24 Mode Submenu

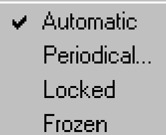


Table 3.12 on page 71 describes the Mode Submenu entries.

Table 3.12 Mode Submenu

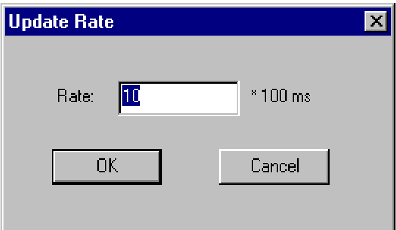
Menu Entry	Description
Automatic	Switches to **Automatic** mode (default), variables are updated when the connection is stopped. Variables from the currently executed module or procedure are displayed in the data component.
Periodical	Switches to **Periodical** mode: variables are updated at regular 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).
Locked	Switches to **Locked** mode, value from variables displayed in the data component are updated when the connection is stopped.
Frozen	Switches to **Frozen** mode: value from variables displayed in the data component are not updated when the connection is stopped.

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 on page 72 allows you to modify the default update rate using steps of 100 ms.

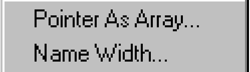
Figure 3.25 Update Rate Dialog Box



Options Submenu

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

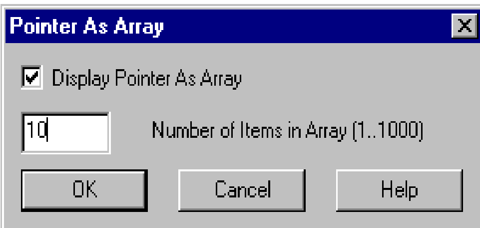
Figure 3.26 Options Submenu



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 on page 72.

Figure 3.27 Pointer as Array Dialog Box

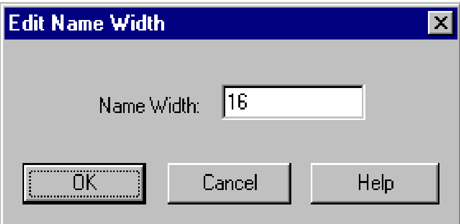


Within this dialog box, you can display pointers as arrays, assuming that the pointer points to the first item (**pointer[0]**). Note that this setup is valid for all pointers displayed in the Data window. Check the **Display Pointer as Array** checkbox and set the number of items that you want to be 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 on page 73.

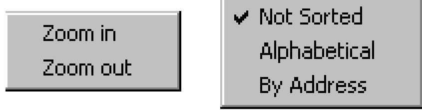
Figure 3.28 Edit Name Width Dialog Box



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.

Zoom and Sort Submenus

Figure 3.29 Zoom and Sort Submenus



Associated Popup Menu

Figure 3.30 Data Popup Menu

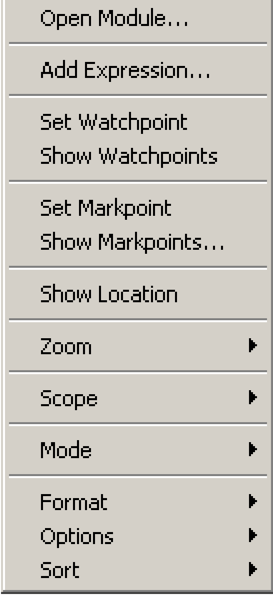


Table 3.13 on page 75 specifies the Data Popup Menu entries.

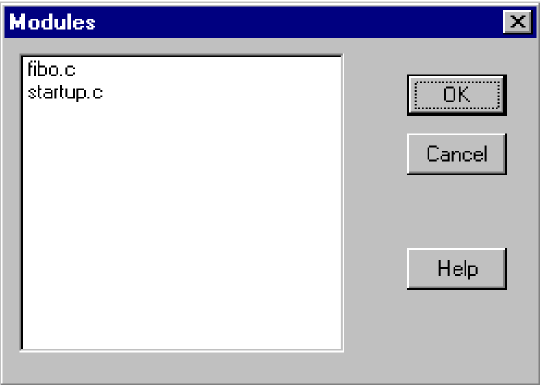
Table 3.13 Data Popup Menu

Menu Entry	Description
Open Module...	Opens the Open Module dialog box.
Set Watchpoint	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.
Delete Watchpoint	Appears only in the popup menu if a watchpoint is set or disabled on the pointed to variable. When selected, deletes this watchpoint.
Enable Watchpoint	Appears only in the popup menu if a watchpoint is disabled on the pointed to variable. When selected, enables this watchpoint.
Disable Breakpoint	Appears only in the popup menu if a breakpoint is set on the pointed to instruction. When selected, disables this watchpoint.
Show Watchpoints	Opens the Watchpoints Setting dialog box and allows you to view the list of watchpoints defined in the application. (Refer to Control Points on page 147).
Show location	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).

SUBMENU Open Module

The dialog shown in Figure 3.31 on page 76 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.

Figure 3.31 Open Modules Dialog Box



Drag Out:

Table 3.14 on page 76 describes the drag actions possible from the Data component.

Table 3.14 Dragging Data Possibilities

Destination Component Window	Action
Command Line	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.
Memory	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.
Source	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.
Register	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.

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 on page 77 describes the drop actions possible in the Data component.

Table 3.15 Data Drop Possibilities

Source Component Window	Action
Source	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 on page 65, Expression Editor on page 66.
Module	Displays the global variables from the selected module in the data component.

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.

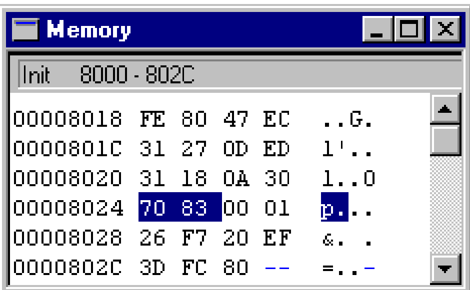
Associated Commands

ADDXPR on page 499, ATTRIBUTES on page 499, DUMP on page 529, PTRARRAY on page 567, SMOD on page 579, SPROC on page 581, UPDATERATE on page 592, ZOOM on page 598.

Memory Component

The Memory window shown in Figure 3.32 on page 78 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 on page 147 for more information about watchpoints.

Memory areas can be initialized with a fill pattern using the Fill Memory on page 83 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 on page 35 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 on page 80 provides access to memory commands. Table 3.16 on page 80 describes the menu entries.

Figure 3.33 Memory Menu

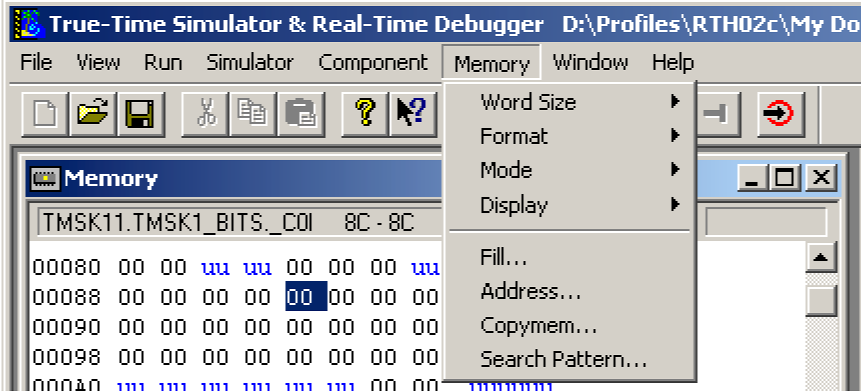


Table 3.16 Memory Menu Description

Menu Entry	Description
Word size	Opens a submenu to specify the display unit size.
Format	Opens a submenu to select the format to display items.
Mode	Opens a submenu to choose the update mode.
Display	Opens a submenu to toggle the display of addresses and ASCII dump.
Fill...	Opens the Fill Memory on page 83 to fill a memory range with a bit pattern.
Address...	Opens the memory dialog and prompts for an address.
CopyMem	Opens the CopyMem dialog box that allows you to copy memory range values to a specific location.
Search Pattern	Opens the Search Pattern dialog box.

Word Size Submenu

With the Word Size submenu shown in Figure 3.34 on page 81, you can set the memory display unit. Table 3.17 on page 81 describes the menu entries.

Figure 3.34 Word Size Submenu

Table 3.17 Word Size Submenu Description

Menu Entry	Description
Byte	Sets display unit to byte size.
Word	Sets display unit to word size (=2 bytes).
Lword	Sets display unit to Lword size (=4 bytes).

Format Submenu

With the Format Submenu shown in Figure 3.35 on page 81, you can set the memory display format. Table 3.18 on page 81 describes the menu entries.

Figure 3.35 Format Submenu

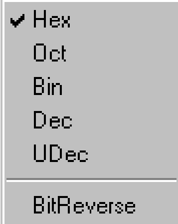


Table 3.18 Format Submenu Description

Menu Entry	Description
Hex	Selects the hexadecimal memory display format
Bin	Selects the binary memory display format
Oct	Selects the octal memory display format
Dec	Selects the signed decimal memory display format

Table 3.18 Format Submenu Description (*continued*)

Menu Entry	Description
UDec	Selects the unsigned decimal memory display format
Bit Reverse	Selects the bit reverse memory display format (each bit is reversed).

Mode Submenu

With the Mode submenu shown in Figure 3.36 on page 82, you can set the memory mode format. Table 3.19 on page 82 describes the menu entries.

Figure 3.36 Mode Submenu

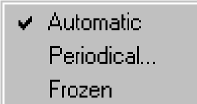


Table 3.19 Mode Submenu Description

Menu Entry	Description
Automatic	Selects **Automatic** mode (default), memory dump is updated when the connection is stopped.
Periodical	Selects the **Periodical** 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).
Frozen	Selects the **Frozen** mode, memory dump displayed in the memory component is not updated when the connection is stopped.

Display Submenu

With the Display submenu shown in Figure 3.37 on page 82, you can set the memory display (address/ascii). Table 3.20 on page 83 describes the menu entries.

Figure 3.37 Display Submenu

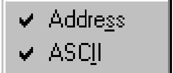


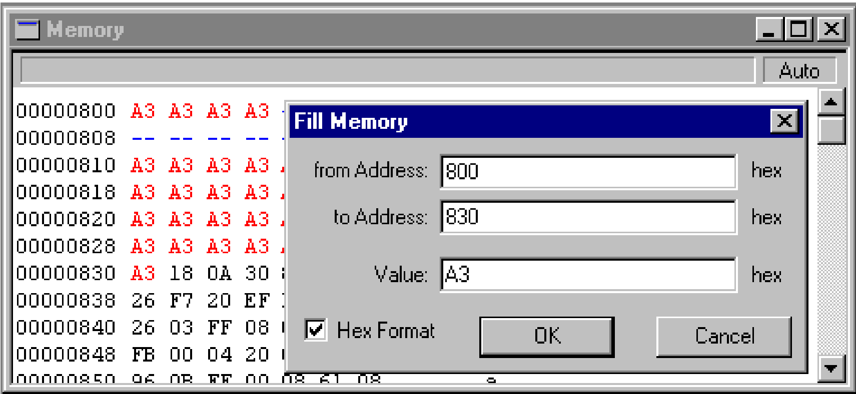
Table 3.20 Display Submenu Description

Menu Entry	Description
Address	Allows you to toggle the display of address dump.
ASCII	Allows you to toggle the display of ASCII dump.

Fill Memory

The Fill Memory dialog box shown in Figure 3.38 on page 83 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

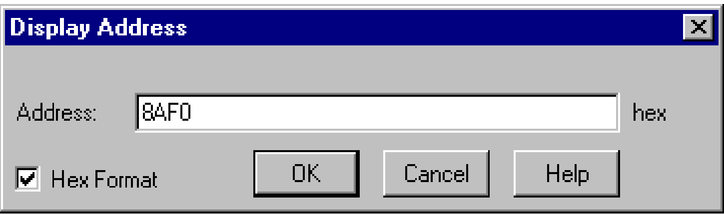


NOTE If “Hex Format” is checked, numbers and letters are interpreted as hexadecimal numbers. Otherwise, expressions can be typed and Hex numbers should be prefixed with “**Ox**” or “**$**”.

Display Address

With the Display Address dialog box, shown in Figure 3.39 on page 84,, the memory component dumps memory starting at the specified address.

Figure 3.39 Display Address Dialog Box

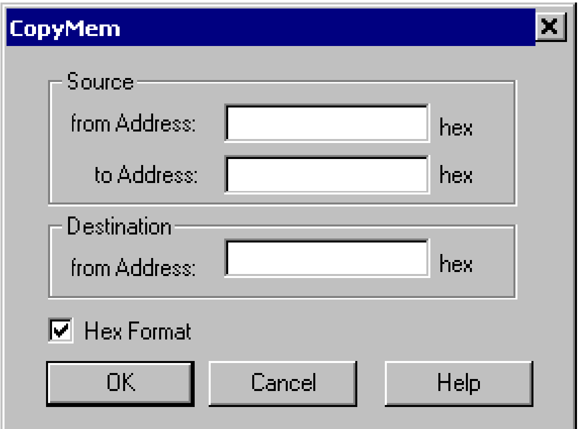


NOTE The **Show PC** dialog box is the same as the Display Address dialog box. In this dialog box, the Assembly component dumps assembly code starting at the specified address.

CopyMem Submenu

The CopyMem dialog box shown in Figure 3.40 on page 85 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.

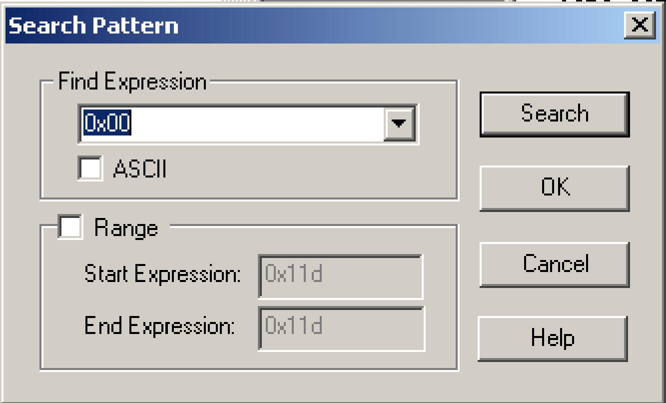
If "**Hex Format**" is checked, all given values are in Hexadecimal Format. You don't need to add "0x". For instance type 1000 instead of 0x1000.

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 on page 86 allows you to search memory or a memory range for a specific expression.

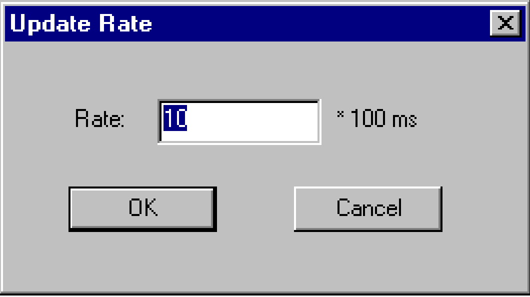
Figure 3.41 Search Pattern Dialog Box



Update Rate

This dialog box shown in Figure 3.42 on page 86 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

The Memory popup menu entries shown in Table 3.21 on page 87 allow you to execute memory associated commands.

Table 3.21 Memory Popup Menu Description

Menu Entry	Description
Set Watchpoint	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.
Delete Watchpoint	Appears in the Popup Menu only if a watchpoint is set or disabled on the selected memory range. When selected, deletes this watchpoint.
Show Watchpoints	When selected, brings up the Controlpoints Configuration Window - Watchpoints Tab. This is the interface through which watchpoints are controlled. (See “Control Points” chapter)

Table 3.21 Memory Popup Menu Description (*continued*)

Menu Entry	Description
Set Markpoint	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.
Show Markpoints	When selected, brings up the Controlpoints Configuration Window - Markpoints Tab. This is the interface through which markpoints are controlled. (See “Control Points” chapter)
Show Location	Forces all opened windows to display information about the selected memory area.
Word Size, etc.	The remaining entries in this menu are explained in table 3.17 Memory Menu Description on page 80

Drag Out:

Table 3.22 on page 88 Describes the drag actions possible from the Memory component.

Table 3.22 Memory Component Drag Possibilities

Destination Component Window	Action
Assembly	Displays disassembled instructions starting at the first address selected. The instructions corresponding to the selected memory area are highlighted in the Assembly component.
Command Line	Appends the selected memory range to the Command Line window
Register	Loads the destination register with the start address of the selected memory block.
Source	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.

Drop Into:

Table 3.23 on page 89 shows the drop actions possible in the Memory component.

Table 3.23 Memory Component Drop Possibilities

Source Component Window	Action
Assembly	Dumps memory starting at the selected PC instruction. The PC location is selected in the memory component.
Data	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.
Register	Dumps memory starting at the address stored in the selected register. The corresponding address is selected in the memory component.
Module	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.

Demo Version Limitations

No limitation

Associated Commands

ATTRIBUTES on page 499, FILL on page 535, SMEM on page 578, SMOD on page 579, SPC on page 580, UPDATERATE on page 592.

MicroC Component

The MicroC window shown in Figure 3.44 on page 90 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 on page 90 shows the MicroC menu and its entries are described in Table 3.24 on page 91.

Figure 3.45 MicroC Link Menu

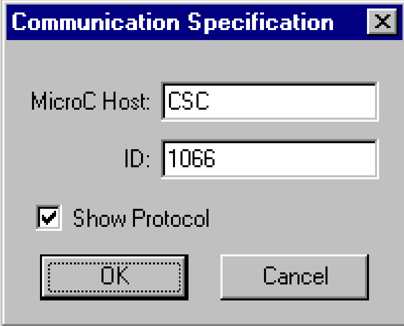
Table 3.24 MicroC Link Menu Description

Menu Entry	Description
Setup	Opens the communication setup Window.
Connect	Establishes communication with RHAPSODY in MicroC.

MicroC Communication Specification

Within the Communication Specification dialog box shown in Figure 3.46 on page 91, 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



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:

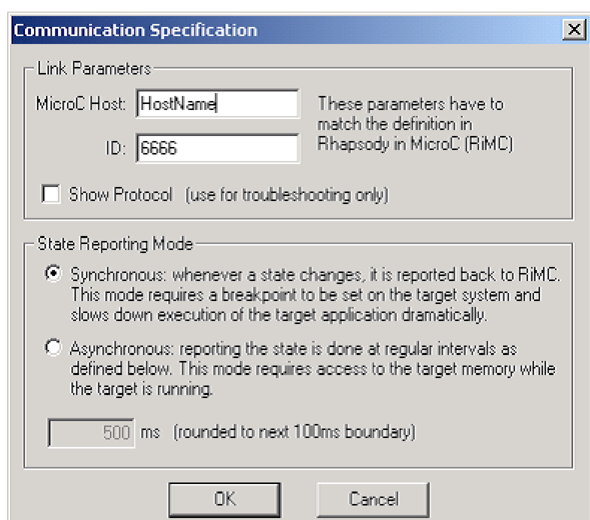
  This mode corresponds to the legacy implementation and lets RiMC update the state whenever a change of state is detected on the target system. This is implemented by setting a breakpoint on the target on a function that is called whenever that state of the application is changed. When hit, the state 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:

Figure 3.47 Communication Specification



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.

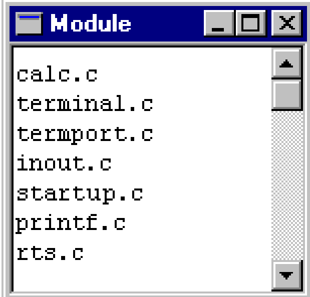
Each of these commands will close any pending communication and re-establish communication with the new parameters.

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 on page 94 gives an overview of source modules building the application.

Figure 3.48 Module Window



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 on page 95 shows the drag actions possible from the Module component.

Table 3.25 Module Component Drag Possibllities

Destination Component Window	Action
Data > Global	Displays the global variables from the selected module in the data component
Memory	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.
Source	Displays the source code from the selected module.

Drop Into:

Nothing can be dropped into the Module Component window.

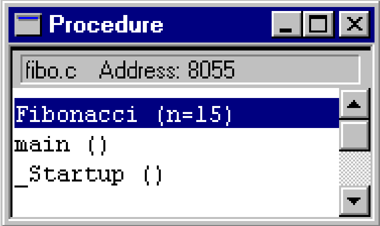
Demo Version Limitations

Only 2 modules are displayed

Procedure Component

The Procedure window shown in Figure 3.49 on page 96 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 on page 97 shows the Procedure menu and its entries are described in Table 3.26 on page 97.

Figure 3.50 Procedure Menu

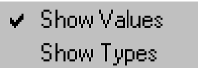


Table 3.26 Procedure Menu Description

Menu Entry	Description
Show Values	Switches to the display of function parameter values in the procedure component.
Show Types	Toggles to the display of function parameter types in the procedure component.

Drag Out:

Table 3.27 on page 97 shows the drag actions possible from the Procedure component.

Table 3.27 Procedure Component Drag Possibilities .

Destination Component Window	Action
Data > Local	Displays the local variables from the selected procedure in the data component
Source	Displays source code of the selected procedure. Current instruction inside the procedure is highlighted in the Source component.
Assembly	The current assembly statement inside the procedure is highlighted in the Assembly component.

Drop Into:

Nothing can be dropped into the Procedure component.

Demo Version Limitations

Only the last two procedures are displayed.

Associated Commands

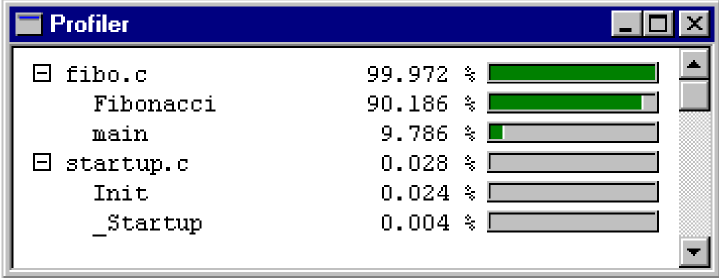
ATTRIBUTES on page 499, FINDPROC on page 538

Profiler Component

The Profiler window shown in Figure 3.51 on page 98 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.

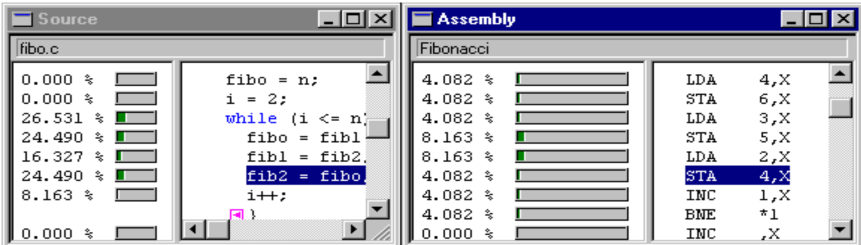
Figure 3.51 Profiler Window



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 on page 98. To obttain a split view in either the Source or Assembly windows, select: **Details>Source** or **Details>Assembly** or both from the Profilier menu and submenu. The split windows effect ends when the Profilier 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 on page 99 shows the Profiler Menu entries, with the Details submenu and the Base submenu. Figure 3.54 on page 99 shows the Profiler Output File submenu. Entries are described in Table 3.28 on page 99.

Figure 3.53 Profiler Menu and Submenus

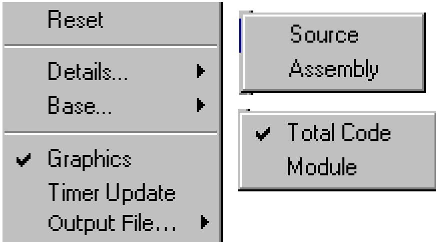


Figure 3.54 Profiler Output File Submenu

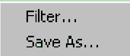


Table 3.28 Profiler Menu Entries Description

Menu Entry	Description
Reset	Resets all statistics.
Details	Sets a split view in the chosen component (Source or Assembly)
Base	Sets the base of percentage (total code or module code).
Graphics	Toggles the display from graphics bar.

Table 3.28 Profiler Menu Entries Description (*continued*)

Menu Entry	Description
Timer Update	Switches on/off the periodic update of the Coverage component. If activated, statistics are updated each second.
Output File	Setup the Profiler Output File Functions on page 100.

Split View Associated Popup Menu

Figure 3.55 on page 100 shows the Profiler popup menu, the **Delete** and **Graphics** menu entries are described in Table 3.29 on page 100.

Figure 3.55 Profiler Split View Associated Popup Menu

Table 3.29 Profiler Split View Associated Popup Menu Description

Menu Entry	Description
Delete	Removes the split view from the host component.
Graphics	Toggles the graphic bars display in the split view.

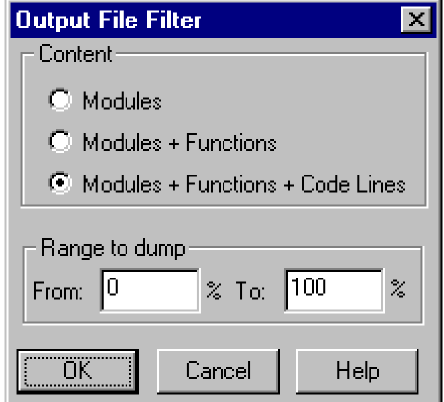
Profiler Output File Functions

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 on page 101 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.

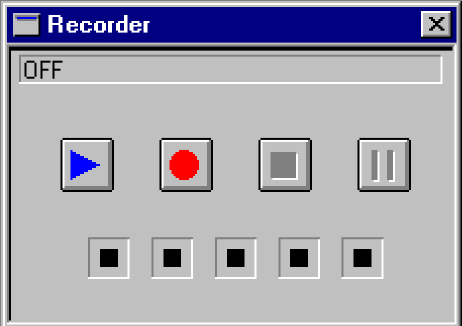
Associated Commands:

GRAPHICS on page 547, TUPDATE on page 588, DETAILS on page 527, RESET on page 570, BASE on page 510.

Recorder Component

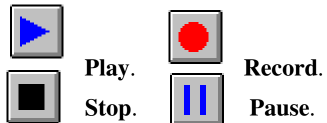
The Recorder window shown in Figure 3.57 on page 102 provides record and replay facilities for debug sessions.

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.

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 on page 103 changes according to the current session. The menu items are described in Table 3.30 on page 103.

Figure 3.58 Recorder Menu

Table 3.30 Recorder Menu Description

Menu Entry	Description
Record	Starts recording from a debug session.
Replay	Starts replaying from a debug session.
Record Time	If set, the evolution time is also recorded. Instant 0 corresponds to the beginning of the recording.

In Listing 3.2 on page 103, 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

```
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.

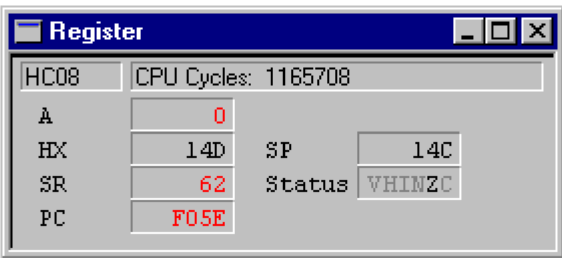
Demo Version Limitations

Only 20 commands are recorded and replayed.

Register Component

The Register window, shown in Figure 3.59 on page 105, 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 on page 106. Table 3.31 on page 106 describes the menu entries.

Figure 3.60 Register Menu

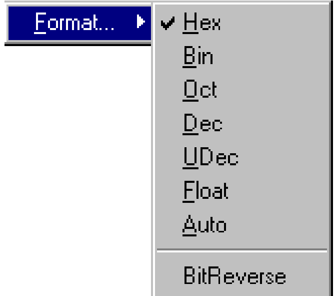


Table 3.31 Register Menu Description

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

Drag Out:

Table 3.32 on page 107 contains the drag actions possible from the Register window.

Table 3.32 Register Component Drag Possibilities

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

Drop Into:

Table 3.33 on page 107 shows the drop actions possible into the Register component.

Table 3.33 Register Component Drop Possibilities

Source Component Window	Action
Assembler	Loads the destination register with the PC of the selected instruction.
Data	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.
Source	Loads the destination register with the PC of the first instruction selected.
Memory	Loads the destination register with the start address of the selected memory block.

Demo Version Limitations

No limitation

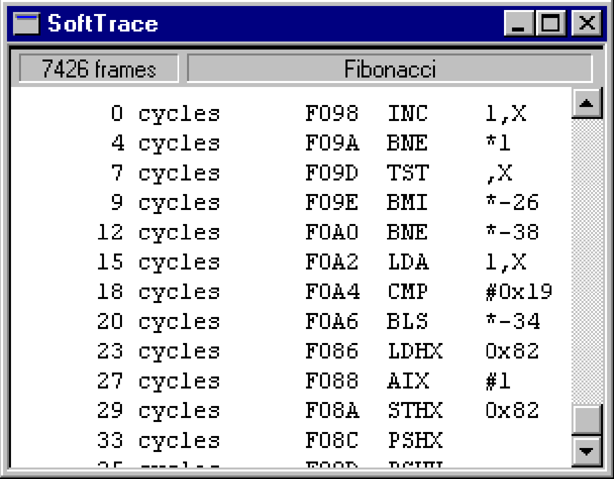
Associated Commands

ATTRIBUTES on page 499.

SoftTrace Component

The SoftTrace window shown in Figure 3.61 on page 108 records and displays instruction frames and time or cycles.

Figure 3.61 SoftTrace Window



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 on page 109 contains the functions described in Table 3.34 on page 109.

Figure 3.62 SoftTrace Menu

Table 3.34 SoftTrace Menu Description

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

Associated Popup Menu

The SoftTrace popup menu shown in Figure 3.63 on page 109 contains functions (described in Table 3.35 on page 110) associated with the pointed to frame.

Figure 3.63 SoftTrace Associated Popup Menu

Table 3.35 SoftTrace Associated Popup Menu Description

Menu Entry	Description
Set Zero Base	Sets the zero base frame to the pointed to frame.
Show Location	Forces open component windows to show the code corresponding to the pointed to frame.

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 on page 519, CYCLE on page 521, FRAMES on page 542, RECORD on page 569, RESET on page 570.

Source Component

The Source window shown in Figure 3.64 on page 111 displays the source code of your program, i.e. your application file.

Figure 3.64 Source Window

```
Source
E:\DEMO\fibo.c                    Line: 18
unsigned int Fibonacci(unsigned int n)
{
  unsigned fib1, fib2, fibo;
  int i;

  fib1 = 0;
  fib2 = 1;
  fibo = n;
  i = 2;
  while (i <= n) {
    fibo = fib1 + fib2;
    fib1 = fib2;
    fib2 = fibo;
    i++;
  }
  return(fibo);
}

void main(void)
{ }
```

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: `fib1 = 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 on page 112) 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.

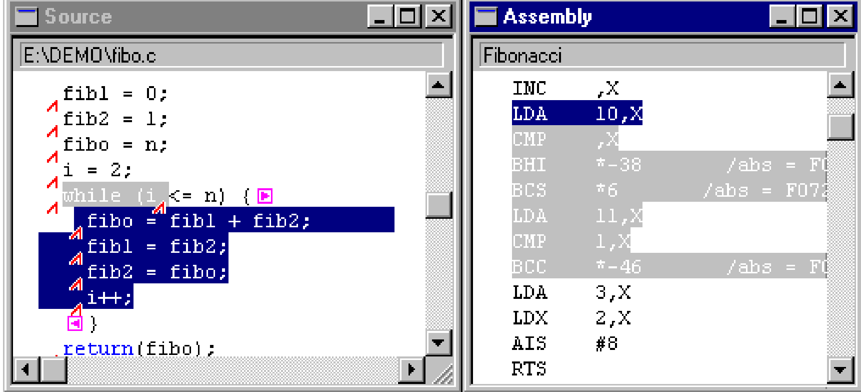
Select **ToolTips>Format** from the source menu entry to select the tool tip display format (Decimal, Hexadecimal, Octal, Binary or ASCII).

On Line Disassembling

For information about performing on line disassembly, refer to section How to Consult Assembler Instructions Generated by a Source Statement on page 205.

- 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 on page 113.
- 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

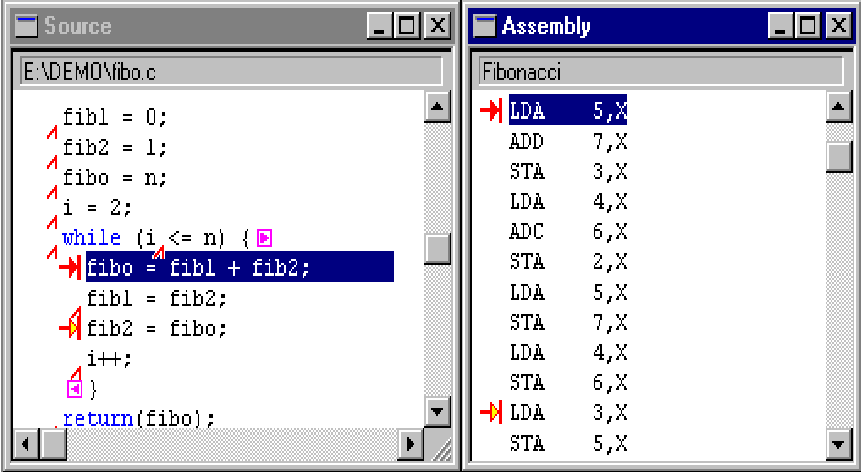


Setting Temporary Breakpoints

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 butoon 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 on page 114.

Figure 3.67 Setting Breakpoints



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 ia shown in Figure 3.68 on page 115 and Figure 3.69 on page 116 shows the functions associated with the Source Popup Menu, while Table 3.36 on page 116 describes these functions.

Figure 3.68 Source Menu

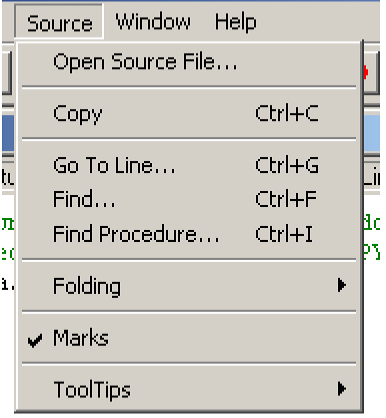


Figure 3.69 Source Associated Popup Menu

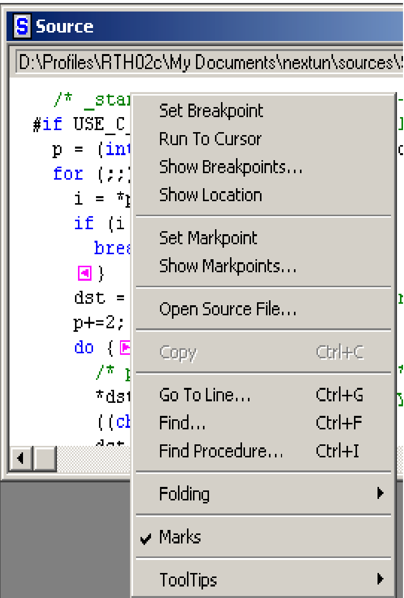


Table 3.36 Associated Pop - Up Menu Description

Menu Entry	Description
Set Breakpoint	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.
Delete Breakpoint	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.
Enable Breakpoint	Appears only in the Popup Menu if a breakpoint is disabled at the nearest code position (visible with marks). When selected, enables this breakpoint.
Disable Breakpoint	Appears only in the Popup Menu if a breakpoint is set at the nearest code position (visible with marks). When selected, disables this breakpoint.

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

Menu Entry	Description
Run To Cursor	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.
Show Breakpoints	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).
Show Location	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.
Set Markpoint	Appears only in the Popup Menu if a markpoint is disabled at the nearest code position (visible with marks). When selected, enables this markpoint.
Delete Markpoint	Appears only in the Popup Menu if a markpoint is set at the nearest code position (visible with marks). When selected, disables this markpoint.
Show Markpoints	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 "Control Points" chapter).
Set Program Counter	The Program Counter is set to the address of the selected source code.
Open Source File	Opens the Source File Dialog if a CPU is loaded (see chapter below).
Copy (CTRL+C)	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.
Go to Line (CTRL+G)	Opens a dialog box to scroll the window to a number line (see chapter below).

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

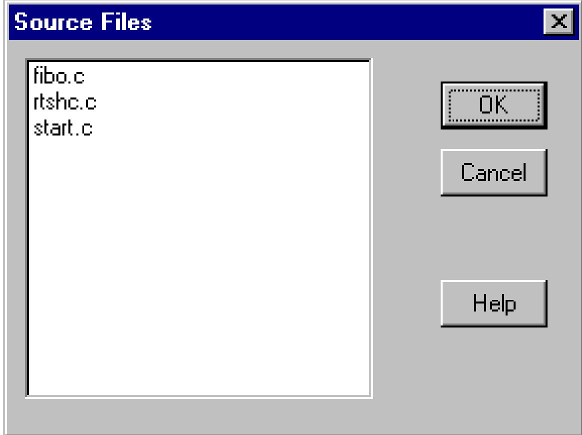
Menu Entry	Description
Find... (CTRL+F)	Opens a dialog box prompting for a string and then searches the file displayed in the source component. To start searching, click **Find Next**, the search is started at the current selection or at the first line visible in the source component (see chapter below).
Find Procedure (CTRL+I)	Opens a dialog box for searching a procedure (see chapter below).
Foldings	Opens the folding window (see chapter below)
Marks	Toggles the display of source positions where breakpoints may be set. If this switch is on, these positions are marked by small triangles.
ToolTips	Allows you to enable or disable the source tool tips feature, to set up the tool tip mode, and tool tip format.

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 on page 119 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.

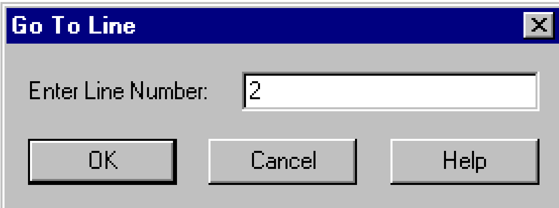
Figure 3.70 Open Source File Dialog Box



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 on page 120. 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.

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 on page 120 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:

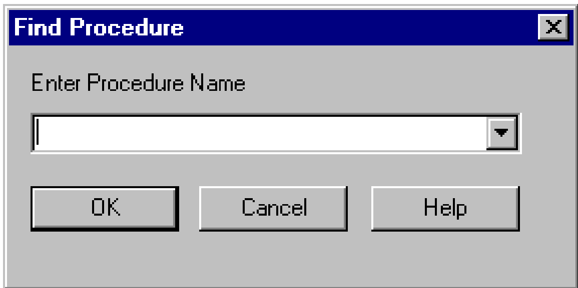
- **Match whole word only:** If this box is checked, only strings separated by special characters will be recognized.
- **Match case:** If this box is checked, the search is case sensitive.

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 on page 121 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



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 on page 121 allows you to select the Fold functions described in Table 3.37 on page 122.

Figure 3.74 Folding Menu

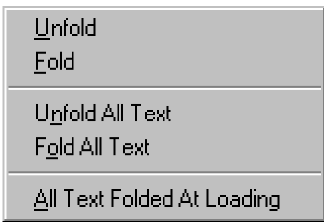


Table 3.37 Folding Menu Description

Menu Entry	Description
Unfold	Unfolds the displayed source code
Fold	Folds the displayed source code
Unfold All Text	Unfolds all displayed source code
Fold All Text	Folds all displayed source code
All Text Folded At Loading	Folds all source code at load time

Drag Out:

Table 3.38 on page 122 shows the drag actions possible from the Source component.

Table 3.38 Source Drag Possibilities

Destination Component Window	Action
Assembly	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
Register	Loads the destination register with the PC of the first instruction selected.
Data	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 on page 65 or Expression Editor on page 66)

Drop Into:

Table 3.39 on page 123 shows the drop actions possible into the Source component.

Table 3.39 Source Drop Possibilities

Source Component Window	Action
Assembly	Source component scrolls to the source statements corresponding with the pointed to assembly instruction and highlights it.
Memory	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.
Module	Displays source code from the selected module.

Demo Version Limitations

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

Associated Commands

ATTRIBUTES on page 499, FIND on page 537, FOLD on page 540, FINDPROC on page 538, SPROC on page 581, SMOD on page 579, SPC on page 580, SMEM on page 578, UNFOLD on page 591.

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.

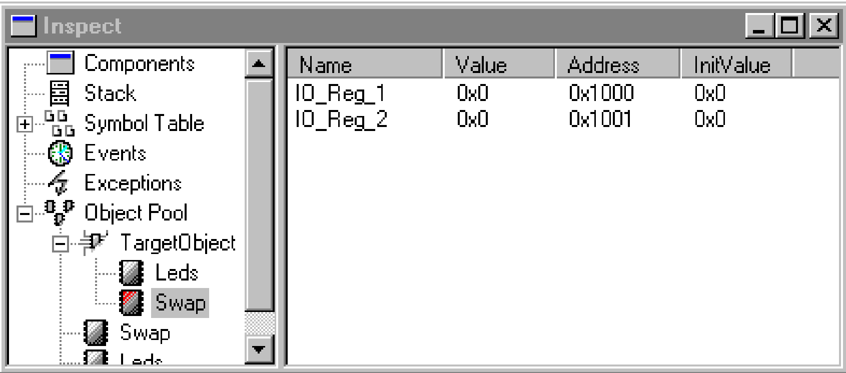
Among these components, there are visualization utilities that graphically display values, registers, memory cells, etc., or provide an advanced graphical user interface to simulated I/O devices, program variables, and so forth.

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 on page 124 displays information about several topics. It displays loaded components, the visible stack, pending events, pending exceptions and loaded I/O devices.

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 on page 125, 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.

Figure 3.76 Inspect Window Components Icon

Inspect

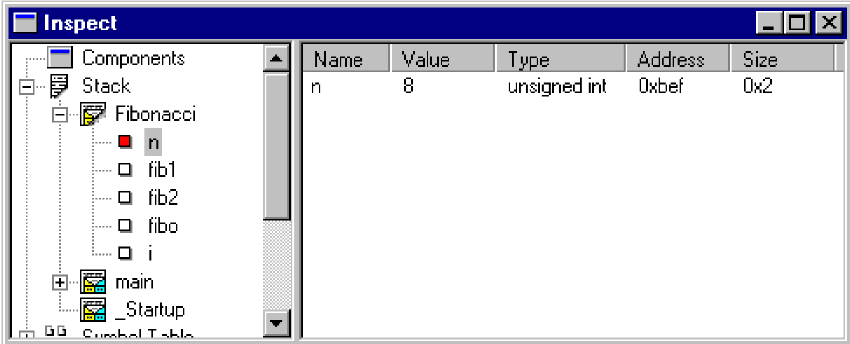
Components
Stack
Symbol Table
Events
Exceptions
Object Pool
TargetObject
Swap
Leds

Name	Type
HiwaveBase	Undefined
Source	Window
Assembly	Window
Procedure	Window
Register	Window
Memory	Window
Data:1	Window
Data:2	Window
Inspect	Window
Target	Target
ElfLoader	Object File Loader
Swap	IODevice
IO_Led	Window
Command	Window

Stack Icon

The Stack icon shown in Figure 3.77 on page 125 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.

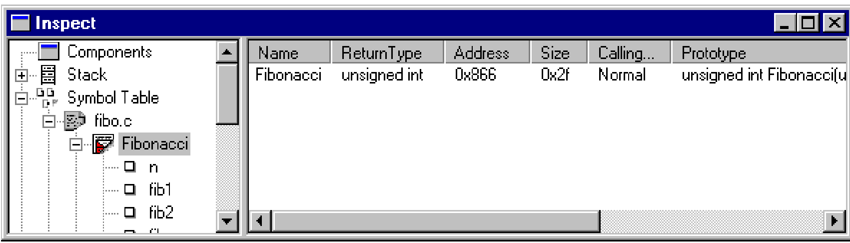
Figure 3.77 Inspector Window Stack Icon



Symbol Table

The symbol table shown in Figure 3.78 on page 126 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.

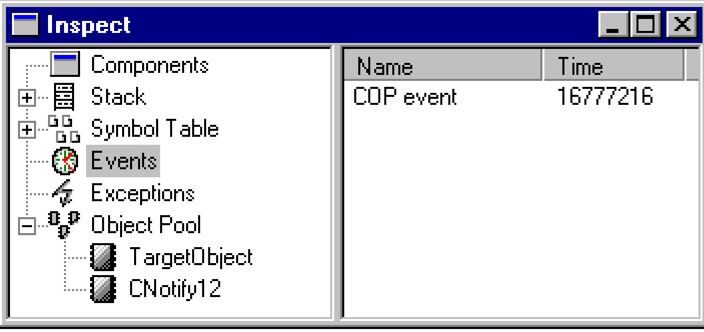
Figure 3.78 Inspector Window Symbol Table



Events Icon

The Inspect window Events icon shown in Figure 3.79 on page 126 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 on page 127 shows all currently raised exceptions. Exceptions are pending interrupts.

Figure 3.80 Inspector Window Exceptions 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.

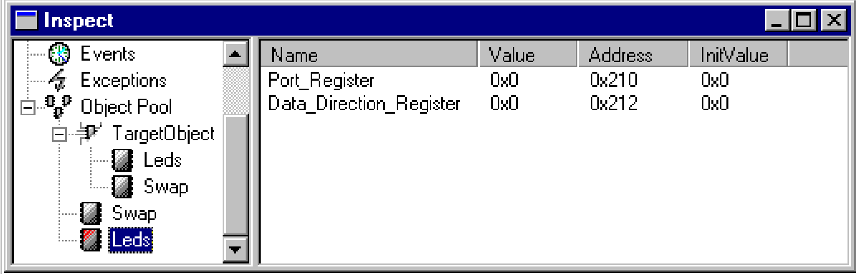
Since interrupts are usually simulated immediately when they are raised, the Exceptions are usually empty. Only when interrupts are disabled or an interrupt is handled, something is visible in this item.

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 on page 127 is a pool of objects. It can contain any number of Objects, which can communicate together and also with other parts of the Debugger.

Figure 3.81 Inspector Window Object Pool



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.

Inspector Operations

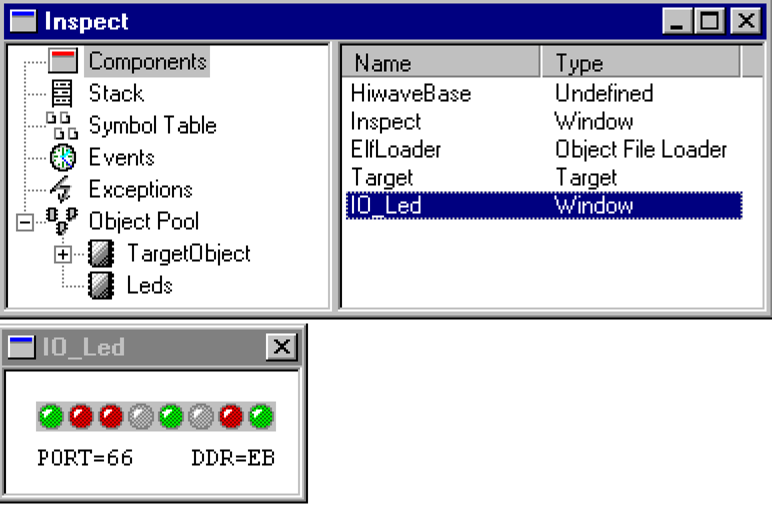
Click the folded/unfolded icons ⊞ ⊟ to unfold/fold the tree and display/hide additional information.

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 **&**).

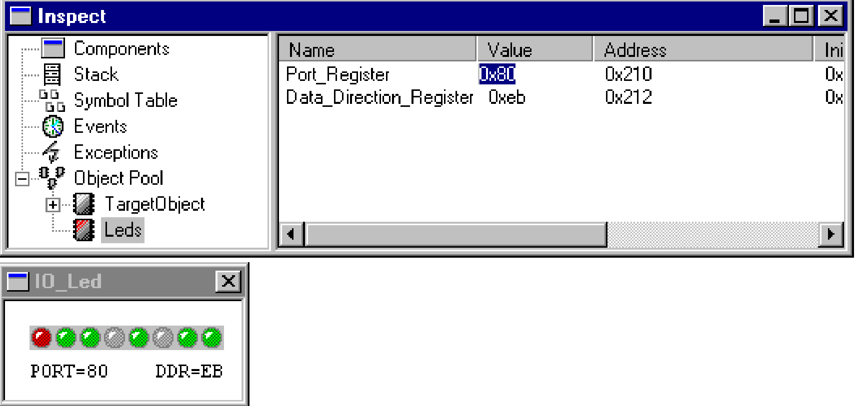
To see the IO_Led in the Inspector, as shown in Figure 3.82 on page 128, open the IO_Led with the context menu **Component**-Open and then open the Inspector. If the Inspector is already loaded, select **Update** from the context menu in the Inspector. Then click on the Components icon to see the Component 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 on page 129).

Figure 3.83 Changing “Data_Direction_Register” Value



Inspector Menu

The Inspector menu contains entries described in Table 3.40 on page 129.

Table 3.40 Inspector Menu Entries

Menu Entry	Description
Update	All displayed information is updated Items that no longer exist are removed and new items are added.

Associated Popup Menu

Commands in the Inspector context menu depend on the selected item. It can contain entries described in Table 3.41 on page 130.

Table 3.41 Inspector Popup Menu Entries Description

Menu Entry	Context	Description
Update	All items	All displayed information is updated Items that no longer exist are removed and new items are added.
Max. Elements...	All items	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.
Format	All items	Numerical values can be displayed in different formats.
Close	single selected Component only	Closes the corresponding component

Drag Out:

Items that can be dragged, depends on which icon is selected. Table 3.42 on page 130 gives a brief description.

Table 3.42 Inspector Component Drag Possibilities

Dragging Item	Description
Components	The components cannot be dragged
Stack	The Stack Icon itself cannot be dragged. All subitems can be dragged the same way as the Symbol Table subitems, described below.
Symbol Table	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.

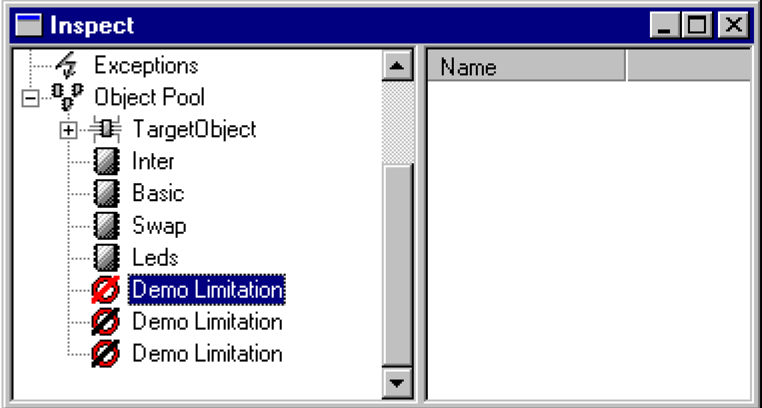
Drop Into:

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 on page 131.

Figure 3.84 Inspector Component Demo Version Limitations

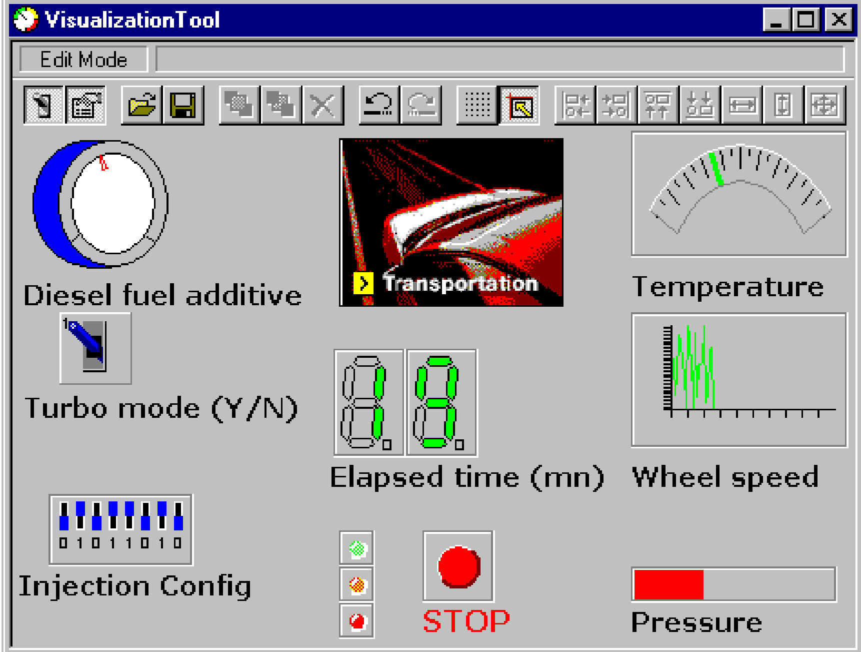


VisualizationTool Component

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 on page 132, consists of a plain workspace that can be equipped with many different instruments.

Figure 3.85 VisualizationTool Window



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 on page 133 on page 133) 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 instruments has its own size minimum.

VisualizationTool 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 on page 133.

Table 3.43 Visualization Tool Menu Description

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

Table 3.43 Visualization Tool Menu Description (*continued*)

Menu Entry	Description
Load Layout	Loads a VisualizationTool-Layout (`*.vtl`). The actual instruments will not be removed. Shortcut: <Ctrl+L>
Save Layout	Saves the current layout to a file (`*.vtl`). Shortcut: <Ctrl+S>

Associated Popup Menu

The context menu of the VisualizationTool depends on the current selection. It can contains the entries described in Table 3.44 on page 134.

Table 3.44 VisualizationTool Popup Menu

Menu entry	Context	Description
Edit mode	Always	Switches between Display mode and Edit mode. In Edit mode, this entry is checked.
Setup	Always	Shows the Setup dialog of the VisualizationTool.
Load Layout	Edit mode	Loads a VisualizationTool-Layout (`*.vtl`).
Save Layout	Always	Saves the current layout to a file (`*.vtl`).
Add New Instrument	Edit mode	Shows a new popup menu with all available instruments.
Properties	Only one instrument selected	Shows up the property dialog box for the currently selected instrument. Shortcut: Ctrl + P
Remove	At least one selection	Removes all currently selected instruments. Shortcut: Delete
Copy	At least one selection	Copies the data of the currently selected instruments into the clipboard. Shortcut: Ctrl + C

Table 3.44 VisualizationTool Popup Menu (*continued*)

Menu entry	Context	Description
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 on page 136. 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

Menu Entry	Description
Edit mode	Switches from Edit mode to Display mode.
Display Scrollbars	Switches the scrollbars on, off, or sets it to automatic mode.
Display Headline	Switches the headline on or off.
Backgroundcolor	Specifies the background color of the VisualizationTool.
Grid Mode	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.’
Grid Size	Specifies the distance between two grid points (vertical, horizontal).
Grid Color	Specifies the color of the grid points.
Refresh Mode	Specifies the way the window will be refreshed. You may choose between: “Automatic, Periodical, Each access, Cpu Cycles”.

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 on page 136.

Table 3.46 Instruments Properties Attributes

Attribute	Description
X-Position	Specifies the X-coordinate of the upper left corner.
Y-Position	Specifies the Y-coordinate of the upper left corner.
Height	Specifies the instruments height.

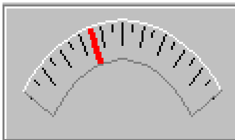
Table 3.46 Instruments Properties Attributes (*continued*)

Attribute	Description
Width	Specifies the instruments width.
Bounding Box	Specifies the look of the bounding box. Available displays are: No Box, Flat (outline only), Raised, Sunken, Etched, and Shadowed.
Backgroundcolor	Defines the color of the instrument's background. The checkbox enables to set a color or let the instrument be transparent.
Kind of Port	Specifies the kind of port to be used to get the value to display. The location must be specified in the 'Port to Display' field.
Port to Display	Defines the location of the value be used for the instrument's visualization. Here are some Examples: Substitute: *TargetObject.#210* Subscribe: *TargetObject.#210* Subscribe: *PORTB.PORTB (check exact spelling using Inspector)* Variable: *counter* Register: *SP* Memory: *0x210*
Size of Port	If you use the Memory Port, you can also specify the width of memory to display (up to 4 Bytes).

Analog Instrument

The Analog instrument (Figure 3.86 on page 137) 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 on page 138.

Table 3.47 Analog Instrument Attributes

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

Bar Instrument

Using the Bar instrument, values are displayed by a bar strip. This instrument (See Figure 3.87 on page 138) 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 on page 138

Table 3.48 Bar Instrument Attributes

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

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 on page 139

Table 3.49 Bitmap Instrument Attributes

Attribute	Description
Filename	Specifies the location of the bitmap. With the button behind, you can browse for files.
AND Mask	Performs a bitwise-AND operation with this value. AND the value of the selected port. Default value is 0.
EQUAL Mask	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.

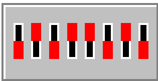
In general, for showing the bitmap, following condition has to be true:
(port_memory & ANDmask) == EQUALmask

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

DILSwitch Instrument

The DILSwitch instrument is also known as Dual-in-Line Switch (Figure 3.89 on page 140). It is mainly used for configuration purpose. You can use it for viewing or setting bits of one to four bytes.

Figure 3.89 DILSwitch Instrument



OILSwitch instrument attributes are listed in the Table 3.50 on page 140.

DIL Switch instruments attributes

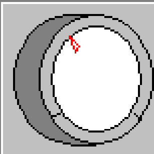
Table 3.50 DILSwitch Instrument Attributes

Attribute	Description
Display 0/1	When enabled, displays the value of the bit under each plot of the DILSwitch instrument.
Switch Color	Specifies the color of the switch.

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 on page 140).

Figure 3.90 Knob Instrument



Knob instrument attributes are shown in the Table 3.51 on page 140

Table 3.51 Knob Instrument Attributes

Attribute	Description
Low Display Value	Defines the zero point of the indicator. The values below this definition will not be displayed.
High Display Value	Defines the highest position of the indicator. It defines the value on which the indicator reads 100%.

Table 3.51 Knob Instrument Attributes (*continued*)

Attribute	Description
Indicator Color	Defines the color and the width of the pen used to draw the indicator.
Knob Color	Defines the color of the knob side.

LED Instrument

The LED instrument is used for observing one definite bit of one byte (Figure 3.91 on page 141). There are only two states: On and Off.

Figure 3.91 Led Instrument

LED instrument attributes are shown in Table 3.52 on page 141.

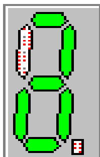
Table 3.52 LED Instrument Attributes

Attribute	Description
Bitnumber to Display	Defines the bit of the given byte to be displayed.
Color if Bit = = 1	Defines the color if the given bit is set.
Color if Bit = = 0	Defines the color if the given bit is not set.

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 on page 141).

Figure 3.92 7-Segment Display Instrument



7 Segment Display instrument attributes are shown in Table 3.53 on page 142

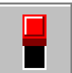
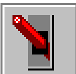
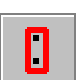
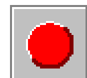
Table 3.53 7 Segment Display Instrument Attributes

Attribute	Description
Decimalmode	Displays the first four or the second four bits of one byte in hexadecimal mode. When it is switched off, each segment will represent one bit of one byte.
Sloping	Switches the sloping on or off.
Display Version	Selects the appearance of the instrument. There are two versions available.
Color if Bit = = 1	Defines the color of an activated segment. You may also set the color to transparent.
Color if Bit = = 0	Defines the color of a deactivated segment. You may also set the color to transparent.
Outlinecolor	Defines the color of the segment outlines. You may also set the color to transparent.

Switch Instrument

Use the Switch instrument to set or view a definite bit (Figure 3.93 on page 142). 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 on page 142.

Table 3.54 Switch Instrument Attributes

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

Table 3.54 Switch Instrument Attributes (*continued*)

Attribute	Description
Kind of Switch	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.
Switch Color	Specifies the color of the switch.
Bounces	If enabled, gives access to the following other attributes to configure the way the switch will bounce.
Nb Bounces	Specifies the number of bounces before stabilization.
Bounces on Edge	Specifies wether the switch will bounce on falling, rising or both edges.
Type of Unit	Synchronizes the frequency of the bouncing either on the timer of your host machine, or on CPU cycles.
Pulse Width (100ms)	Defines the duration of one bounce. This attribute should be filled in if you chose “Host Periodical” in the “Type of Unit” attribute.
CPU Count	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.

Text Instrument

The Text instrument has several functions: Static Text, Value, Relative Value, and Command (Figure 3.94 on page 143).

Figure 3.94 Text Instrument

Please use ’Text Mode’ to switch between the five available modes. Text instrument common attributes are shown in the Table 3.55 on page 144

Table 3.55 Text Instrument Attributes

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

'Static Text' is used for adding descriptions on the workspace. Its attributes are shown in Table 3.56 on page 144.

Table 3.56 Static Text Attributes

Attribute	Description
Field Description	Contains the text to be displayed.

'Value' is used for displaying a value in different ways (decimal, hexadecimal, octal, or binary). Its attributes are shown in Table 3.57 on page 144.

Table 3.57 Value Attributes

Attribute	Description
Field Description	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 "Value: "
Format mode	Defines the format. Choose among this list: Decimal, Hexadecimal, Octal, and Binary formats.

'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 on page 145

Table 3.58 Relative Value Attributes

Attribute	Description
Field Description	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 "Value: "
Low Display Value	Fixes the minimal value that will represent 0%. Values below this definition will appear as an error: #ERROR.
High Display Value	Fixes the maximal value that will represent 100%. Values above this definition will appear as an error: #ERROR..
Relative Mode	Switches between percent and permill.

'Command'. 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 the Table 3.59 on page 145

Table 3.59 Command Attributes

Attribute	Description
Field Description	Contains the text that will be displayed on the button.
Command	Contains the command-line command to be executed after pressing the button.

'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 on page 145

Table 3.60 CMD Callback Attributes

Attribute	Description
Field Description	Warning: there is no use to fill out his field as the text will be overwritten the first time you execute the specified command.
Command	Contains the command line command to be executed after pressing the button.

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.

4

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 on page 147
- Breakpoints on page 148
- Setting Breakpoints on page 154
- Watchpoints on page 161
- Setting Watchpoints on page 165
- Markpoints on page 171
- Setting Markpoints on page 174
- Halting on a Control Point on page 176

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**.

- 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.

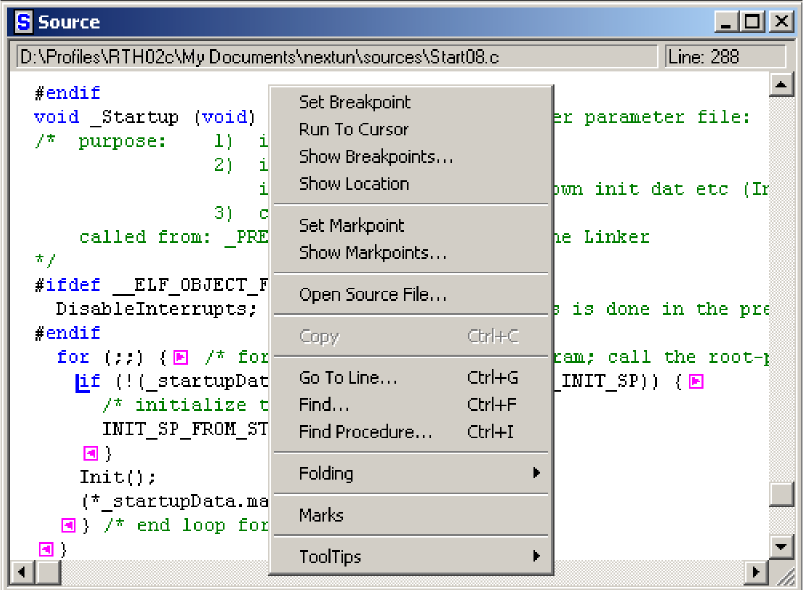
Breakpoints

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:

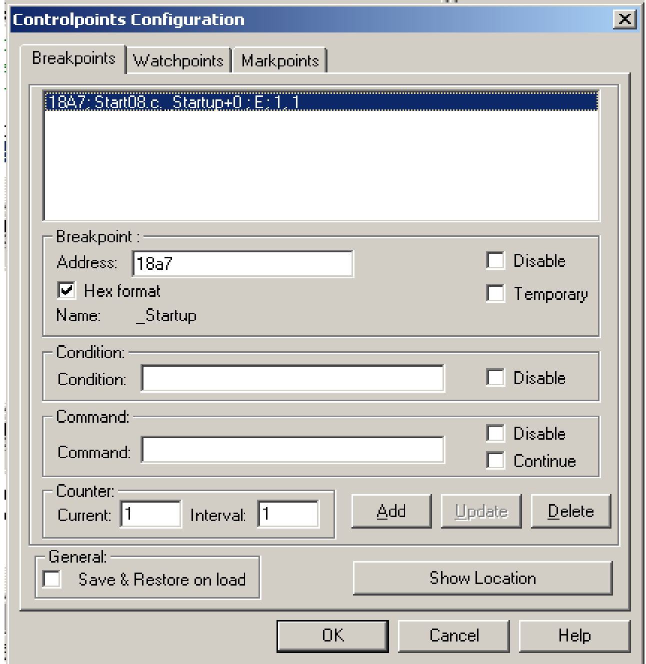
Figure 4.1 Source Window Popup Menu



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) on page 150 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) on page 150 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.

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) on page 150): the **SAVEBP** command is used: **SAVEBP on** when checked, **SAVEBP off** when unchecked.

NOTE For more information about this, see the SAVEBP on page 575 command.

- List of defined breakpoints: the **BS** command is used, as shown in Listing 4.1 on page 152.

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) on page 150 for conditional breakpoint.

command is any debugger command. It matches the **Command** field in the Controlpoints Configuration Window (Breakpoints Tab) on page 150, for associated commands.

current is an **expression.** It matches the **Current** field (**Counter**) in the Controlpoints Configuration Window (Breakpoints Tab) on page 150, for counting breakpoints.

interval is an **expression.** It matches the **Interval** field (**Counter**) in the Controlpoints Configuration Window (Breakpoints Tab) on page 150, 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 on page 153.

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**s 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.

NOTE For more information about this syntax, see BS on page 513 and SAVEBP on page 575 commands.

Setting Breakpoints

The Debugger supports different types of breakpoints:

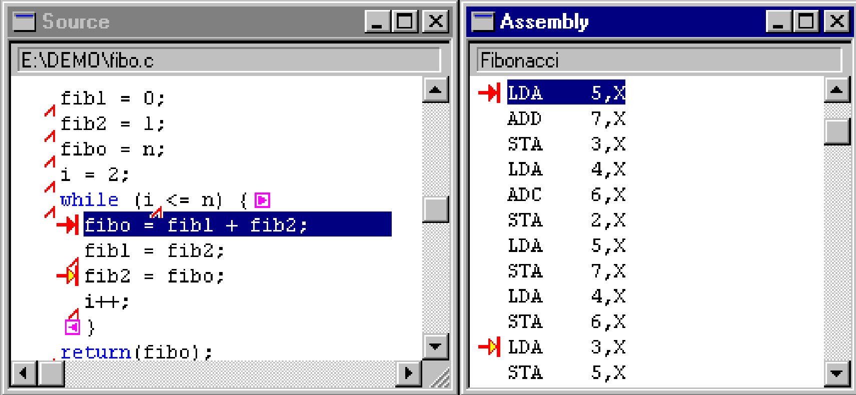
- Temporary breakpoints, which are activated next time the instruction is executed.
- Permanent breakpoints, which are activated each time the instruction is executed.
- Counting breakpoints, which are activated after the instruction has been executed a certain number of times.
- Conditional breakpoints,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

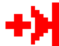
B. Holding down the left mouse button, pressing the P key:

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.

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) on page 150. 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) on page 150 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) on page 150 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) on page 150 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.

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) on page 163. 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.

Figure 4.4 Memory Popup Menu

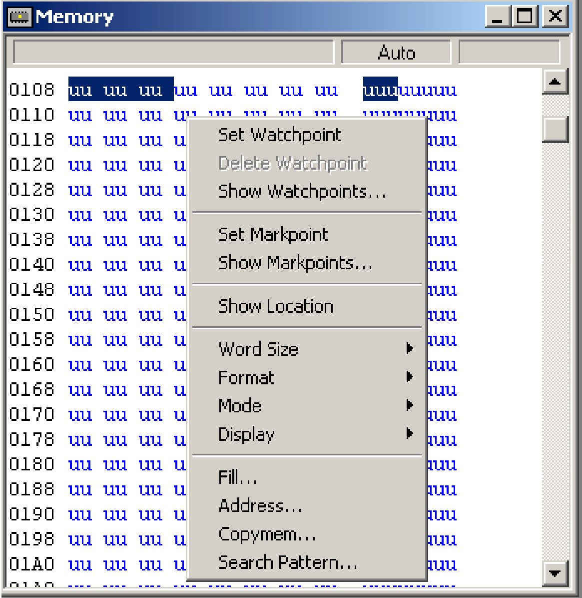
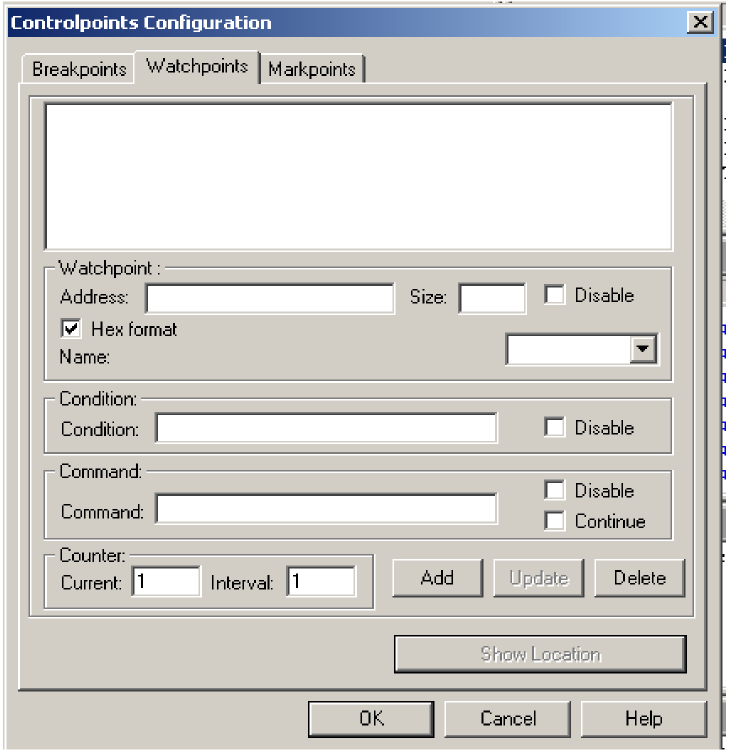


Figure 4.5 Data Popup Menu

Figure 4.6 Controlpoints Configuration Window (Watchpoints tab)



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).
- **Update** button to Update all modifications in the dialog.
- "**Command**:" group box that displays the command string to execute and state of the command (disabled or continue after command execution).

- **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 on page 162 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

stops. **Current** is reloaded with the value stored in the interval field to enable the counting watchpoint again.

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 Watchpoint*s*** 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 on page 172 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.

4. You can enter the command in the **Command** field. The command is a single debugger command. At this level, the commands G on page 543, GO on page 544 and STOP on page 586 are not allowed.
   A command file can be associated with a watchpoint using the commands CALL on page 515 or CF on page 517 (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 **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) on page 173. 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

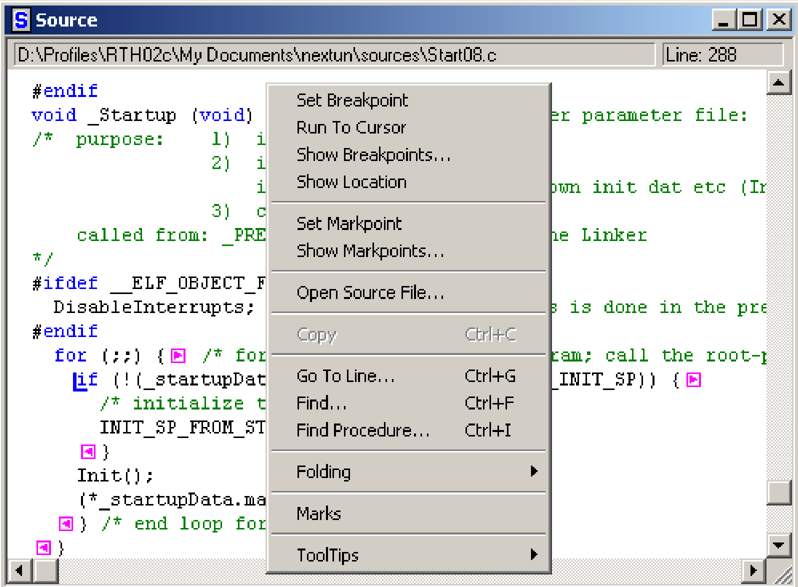


Figure 4.8 Memory Popup Menu

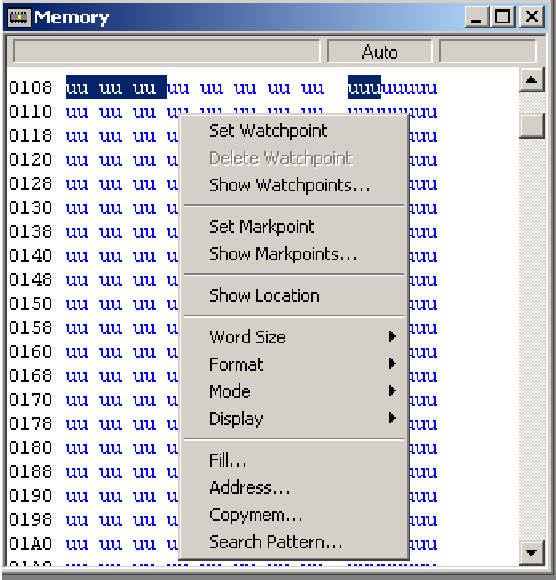
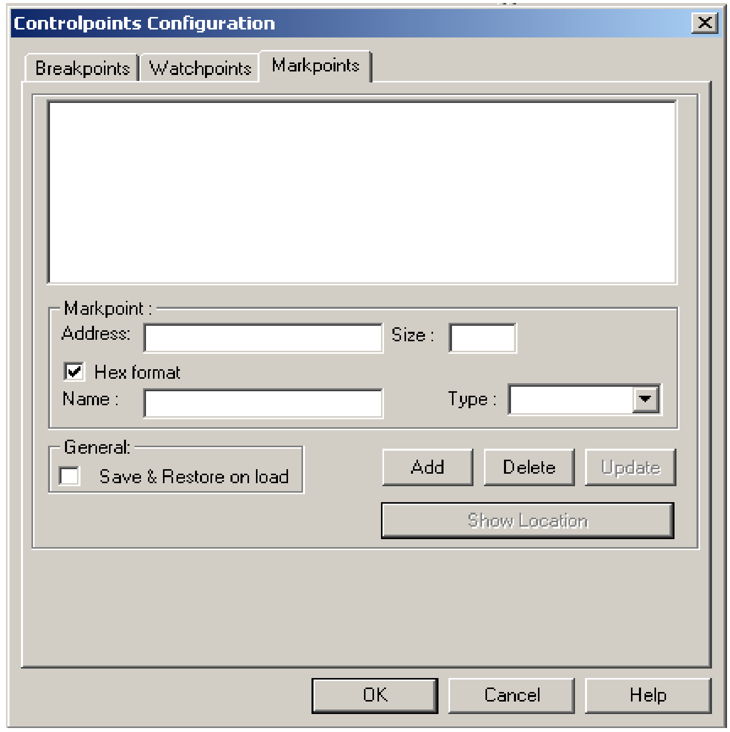


Figure 4.9 Data Popup Menu

Figure 4.10 Controlpoints Configuration Window (Markpoints Tab)



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.

- **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 on page 171 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 on page 172 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 on page 172 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.

5. Click **OK** to close the window. The markpoint is deleted and the blue letter L in front of the variable, memory range, or code line is removed.

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.

5

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 on page 177
- Task Description Language on page 178
- Application Example on page 180
- Inspecting Kernel Data Structures on page 181
- OSEK Kernel Awareness on page 183

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.PRM`', 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.PRM`" has to be created and must be stored in one of the directories specified in GENPATH: #include “File” Path on page 647

The file "`OSPARAM.PRM`" 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):

```
StatSequence         =     [Statement] {';' Statement;}.
Statement            =     Assignment | ErrorMsg | If.
Assignment           =     Ident ':=' Expression.
ErrorMsg             =     'MSG' ':=' String.
IfStatemen           =     'IF' BoolExpr 'THEN' StatSequence
                     {ELSIFPart} [ELSEPart] 'END'.
ELSIFPart            =     'ELSIF' BoolExpr 'THEN' StatSequence.
ELSEPart             =     'ELSE' StatSequence.
String               =     '"' {char} '"'.
BoolExpr             =     Expression RelOp Expression.
Expression           =     Term {Op Term}.
Term                 =     Ident | Function | Number.
Ident                =     'a'..'z' | 'R00'..'R31' | 'DL' | 'SP' |
                     'SR' | 'PC' | 'STATUS' | 'B'.
Function             =     ('MB' | 'MW' | 'MD' | 'MA') '['
                     Expression ']'.
RelOp                =     '#' | '<' | '<=' | '=' | '>=' | '>'.
Op                   =     '+' | '-'.
```

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,

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 on page 180 shows an example of *"OSPARAM.PRM"* file for SOOM System/REM.

Listing 5.1 OSPARAM.PRM File

```
{ File OSParam.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;
```

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 on page 181.

Listing 5.2 Kernel Data Structure Description

```
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 on page 181).

Listing 5.3 OS_DATA Structure

```
#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 on page 181:

Listing 5.4 Linker PRM File

```
NAMES  ... rtk.o+ ... END
SECTIONS
   ...
   RTK_SEC = NO_INIT 0x1040 TO 0x1F80;
   ...
```

```
END

PLACEMENT
    ...
    OS_DATA INTO RTK_SEC;
    ...
END
```

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 on page 182 show the register assignments for the RTK awareness for the HC12 processor.

Table 5.1 HC12 RTK Awareness Register Assignments

Register	Register Name	Size (bit)
R0	A	8 (high byte of D)
R1	B	8 (low byte of D)
R2	CCR	8
R6	D	16 (concatenation of A:B)
R7	X	16
R8	Y	16
R9	SP	24 (concatenation of xPAGE:SP if in banked area)
R10	PC	16
R11	PPAGE	8

Table 5.1 HC12 RTK Awareness Register Assignments

Register	Register Name	Size (bit)
R12	EPAGE	8
R13	DPAGE	8
R14	IP	24 (concatenation of PPAGE:PC if in banked area)

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.

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:

- 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 on page 185.

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.

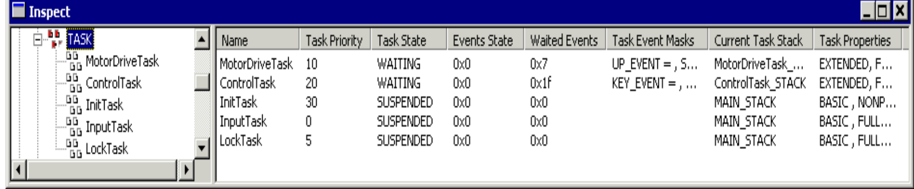
The following sections offer a description of typical objects along with their attributes and how they are presented.

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 on page 185 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, SUPENDED, 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 to1, 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 on page 187 displays the current state of OSEK stack trace.

Figure 5.4 Inspector Stack

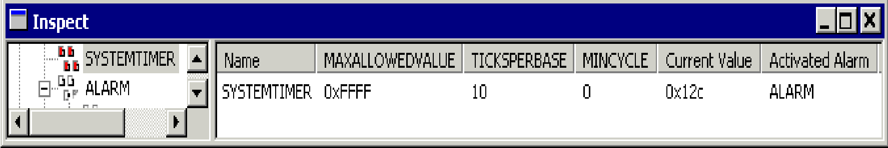
When selecting Stack in the hierarchical tree on the left side, additional information concerning the stack are displayed on the right side of the window under the following headings:

- **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 on page 187 displays the current state of OSEK SystemTimer trace.

Figure 5.5 Inspector SystemTimer



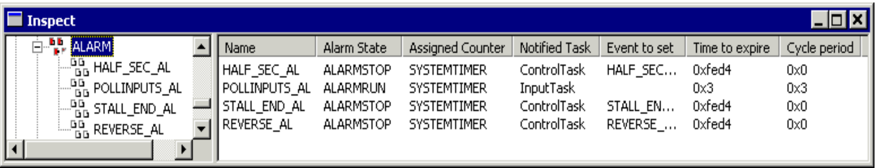
When selecting SystemTimer in the hierarchical tree on the left side, additional information concerning the timer are displayed on the right side of the window under the following headings:

- **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 on page 188 displays the current state of OSEK alarm trace.

Figure 5.6 Inspector Alarm



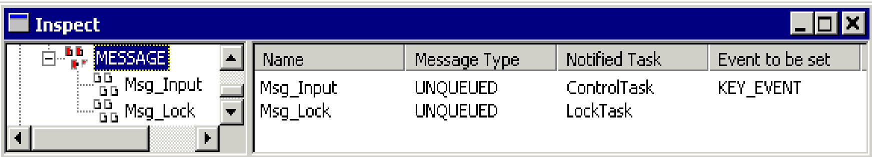
When selecting Alarm in the hierarchical tree on the left side, additional information concerning the alarm are displayed on the right side of the window under the following headings:

- **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.

Inspector Message

The Message shown in Figure 5.7 on page 189 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.

6

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 on page 192
- How To Start the Debugger on page 193
- Automating Debugger Startup on page 194
- How To Load an Application on page 195
- How To Start an Application on page 196
- How To Stop an Application on page 196
- How To Step in the Application on page 197
- How To Work on Variables on page 199
- How To Work on the Register on page 203
- Modify Content of Memory Address on page 205
- How to Consult Assembler Instructions Generated by a Source Statement on page 205
- How To View Code on page 206
- How to Communicate with the Application on page 207
- About startup.cmd, reset.cmd, preload.cmd, postload.cmd on page 207

How To Configure the Debugger

If you have installed the Debugger under Windows 95, 98, NT 4.0 and Windows2000 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**].

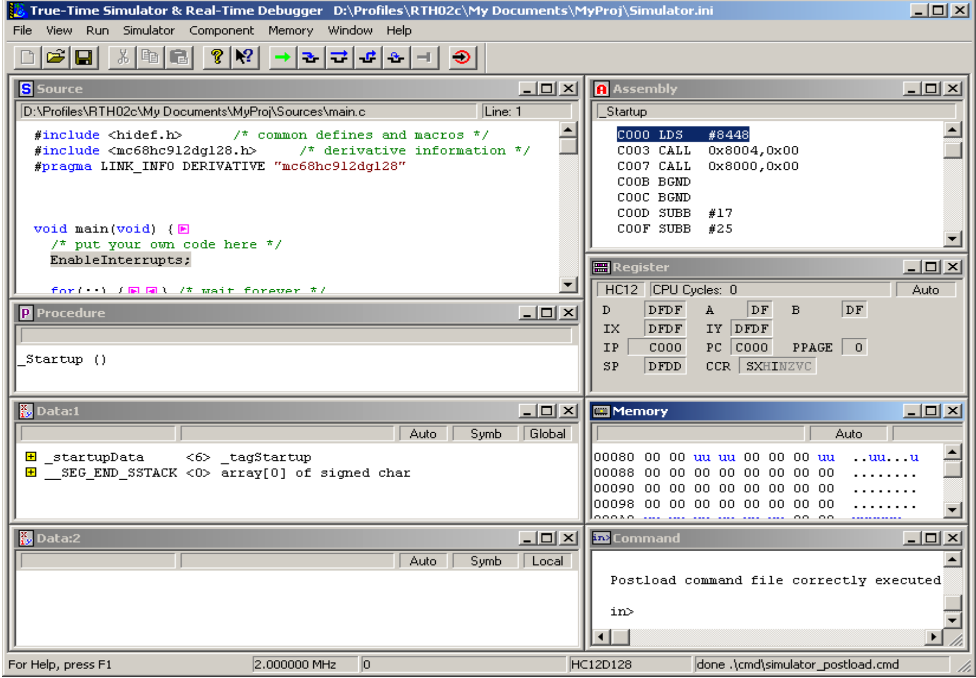
How To Start the Debugger

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 on page 193.

Figure 6.1 Debugger After Startup



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 on page 194.

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 on page 194). 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

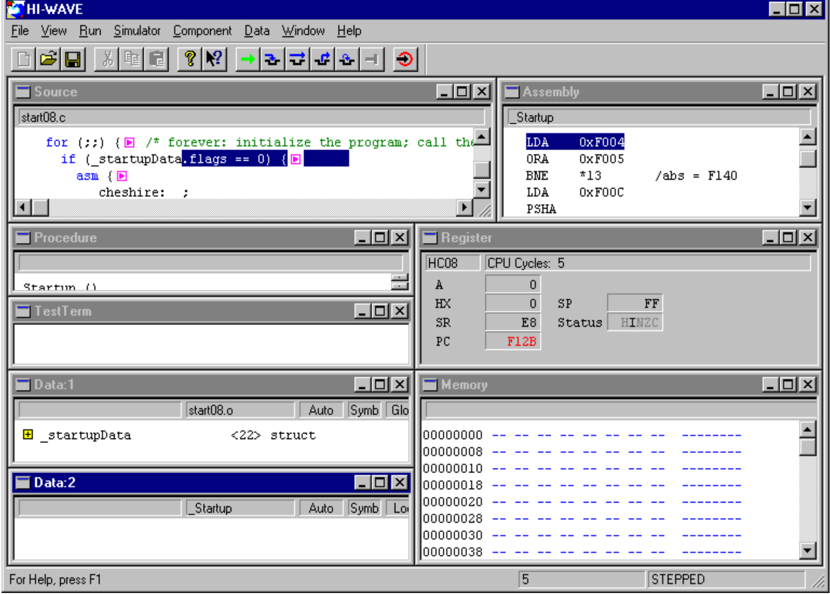
this file (for example, **CALL INIT.CMD**), it will automatically execute when the connection component is loaded.

NOTE Refer to section How To Start the Debugger on page 193.

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 on page 194).

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.

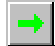
The Local Data component is empty.

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 tool bar 

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 on page 196).
- 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 tool bar

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.

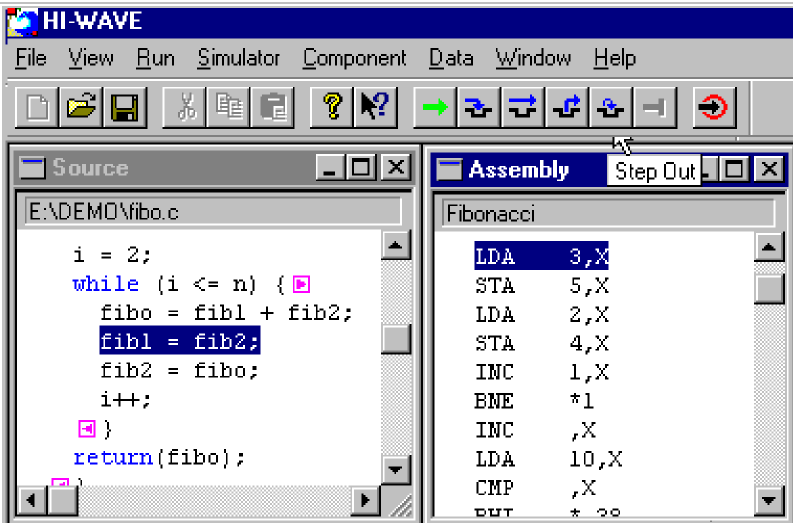
Data window with attribute **Local** displays the name and values of the local variables defined in the current procedure. The name of the procedure is specified in the Data info bar.

How To Step in the Application

The Debugger provides stepping functions at the application source level and assembler level (Figure 6.3 on page 197).

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.

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 on page 585) or STOPPED (STOP on page 586) 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 on page 586) in the status line indicates that the application is stopped by a step out function.

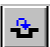
If the application was previously stopped in a function, a **Step Out** stops the application on the source instruction following the function invocation.

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

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 on page 200).

Table 6.1 Debugger Display Format

Menu entry	Description
Hex	Variable values are displayed in hexadecimal format.
Oct	Variable values are displayed in octal format.
Dec	Variable values are displayed in signed decimal format.
UDec	Variable values are displayed in unsigned decimal format.
Bin	Variable values are displayed in binary format.
Symbolic	Displayed format depends on variable type.

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.

Format menu is activated as follows:

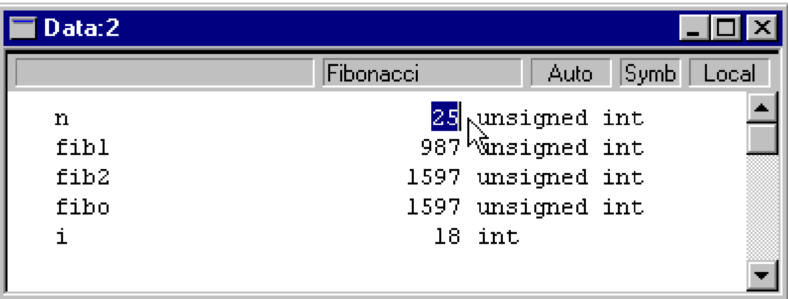
1. Right-click in the Data component. The Data Popup Menu is displayed on the screen.
2. Choose **Format** from Popup Menu. The list of all formats is displayed on the screen.

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 on page 201.

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.

- Holding down the left mouse button and pressing the **A** key

1. Point to a variable name in a Data Component.
2. Hold the left mouse button down and press 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.

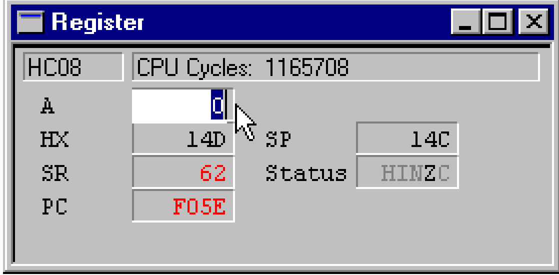
Modify a Register Content

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.

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

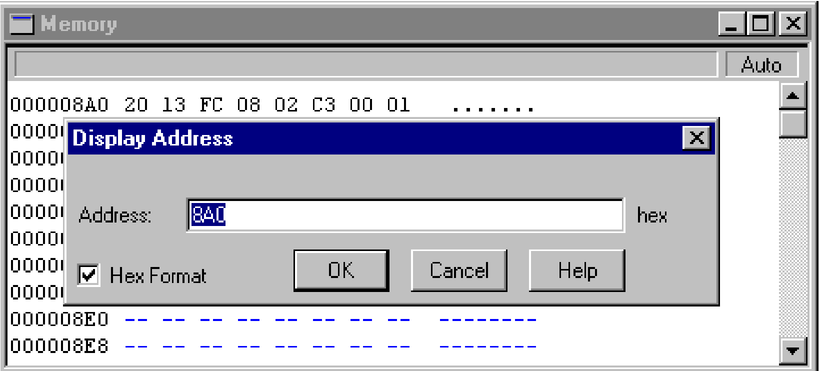
The Debugger provides two ways to dump memory starting at the address a register is pointing to.

- 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 on page 204 is opened.
3. Enter the register content in the Edit Box and choose **OK** to close the dialog box.

The memory component scrolls until it reaches the address stored in the register.

This feature allows you to display a memory dump from the application stack.

NOTE If "Hex Format" is checked, numbers and letters are considered to be hexadecimal numbers. Otherwise, expressions can be typed and Hex numbers should be prefixed with "Ox" or "$".

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

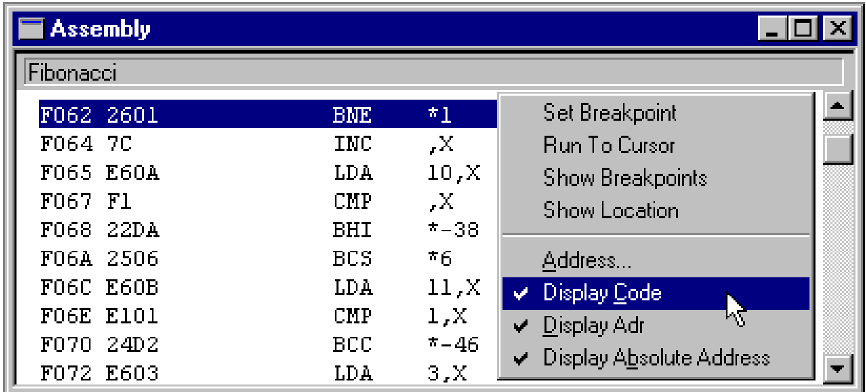
1. In the Source component window, point to the instruction you want to disassemble.
2. Hold down the left mouse button and press the R key

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 on page 206).

- 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**, 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.

7

CodeWarrior Integration

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 on page 209

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 on page 210).

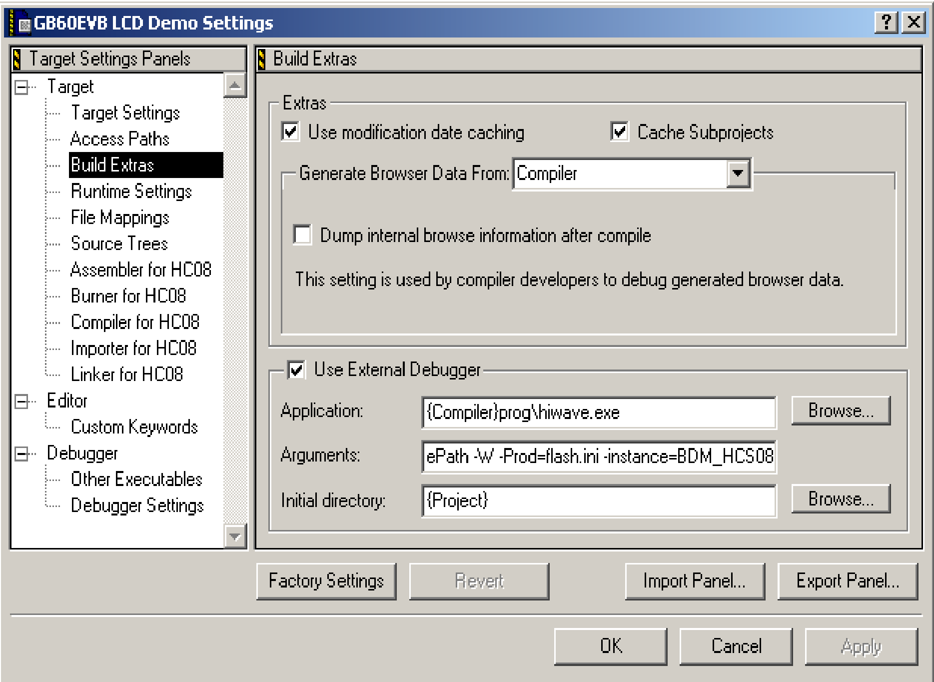
In the **Build Extras** pane check the **Use External Debugger** checkbox.

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



8

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"

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.

9

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 on page 213
- Debugger Interface on page 224
- Synchronized Debugging on page 229
- Troubleshooting on page 229

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:

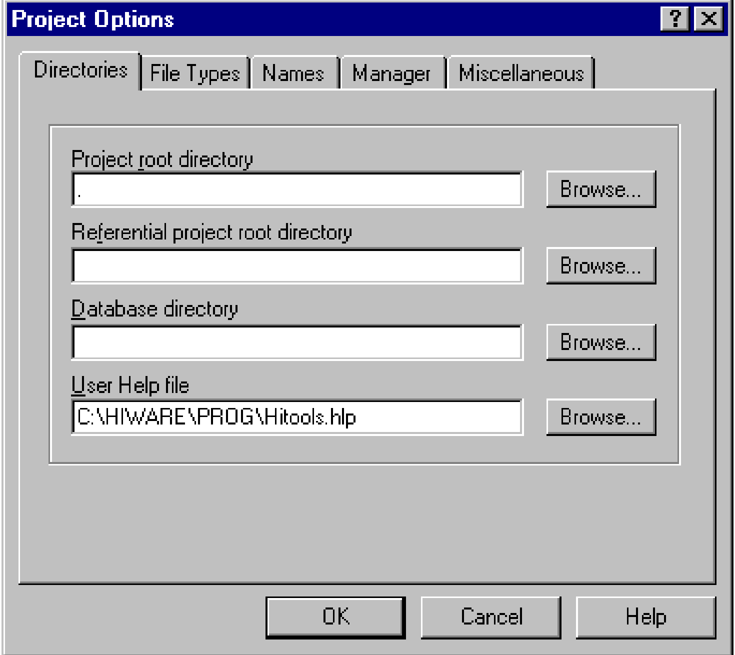
```
"C:\Freescale\work\<processor>c\myproject"
```

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 on page 214 contains options, which establish directories for the project.

Figure 9.1 DA-C Project Options Window - Directories Tab



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.

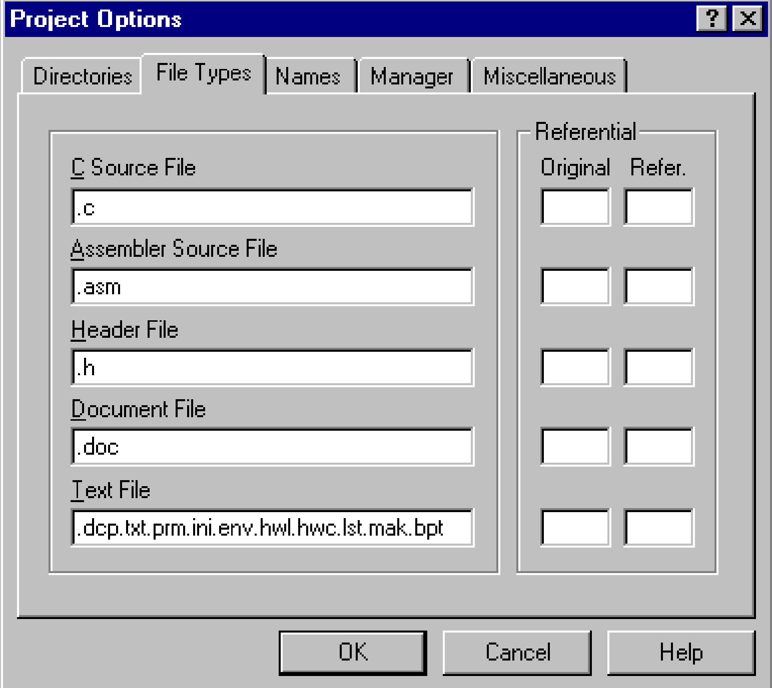
User Help File

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 on page 216 shows file extension types that can be defined.

Figure 9.2 DA-C Project Options Window - File Types Tab

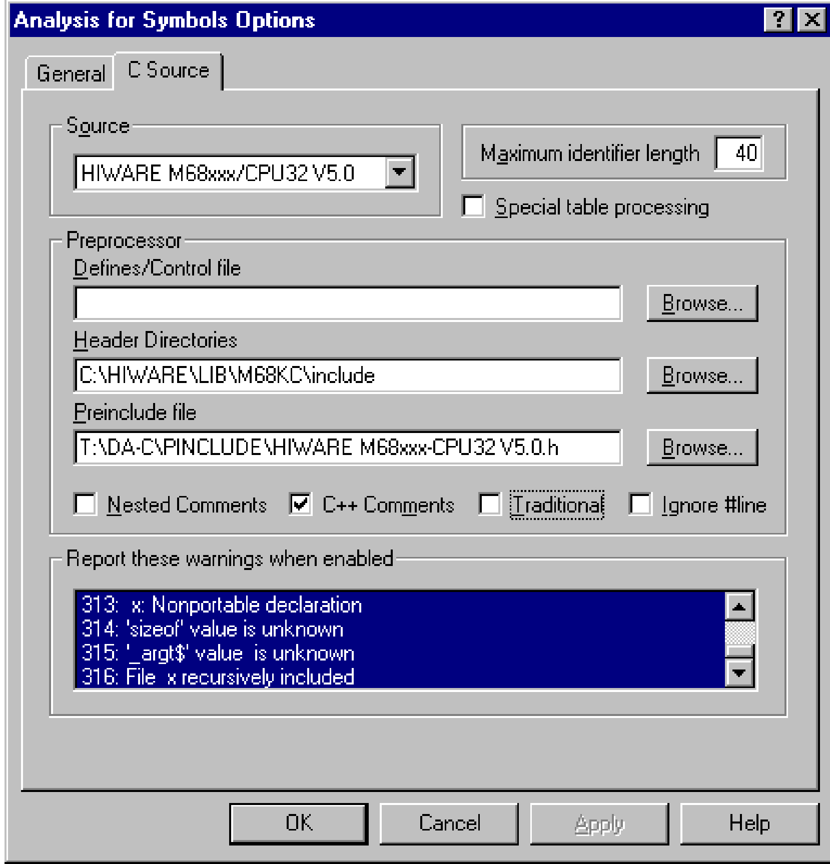


Configure Library Path

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 on page 217 contains options that determine parameters of the C source code analysis.

Figure 9.3 Analysis for Symbols Options Window - C Source Tab



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

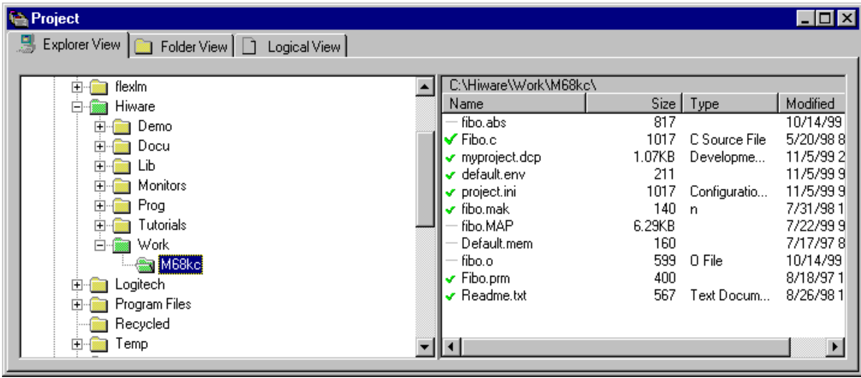
This text box determines the name of the file that will be included automatically at the beginning of every source module during analysis, in the same way as if #include "string" were present in the first line. The preinclude file can be used to specify predefined macros and variable and function declarations 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 on page 218).

Figure 9.4 Adding Files to Project Using Explorer Tab



In the same way, select "`fibo.prm`" file and add it to this project.

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.

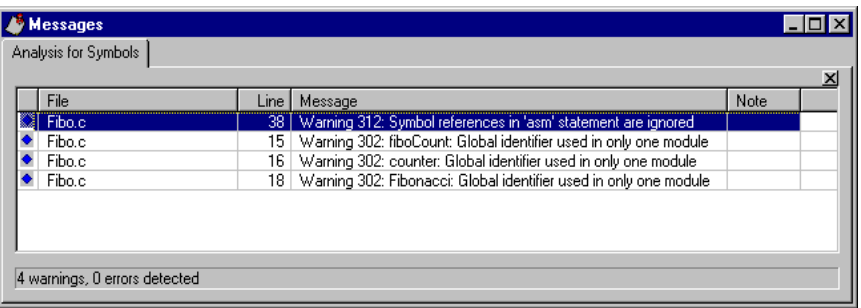
Building The Database

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 on page 219 (for `Fibo.c` sample file):

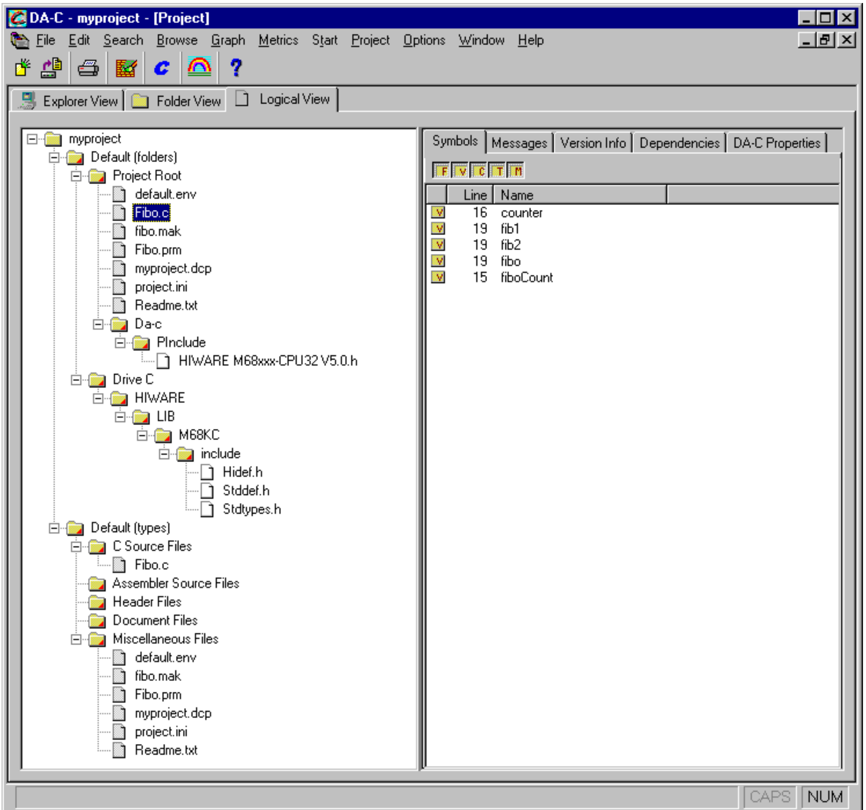
Figure 9.5 DA-C Message Window



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 on page 220 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.

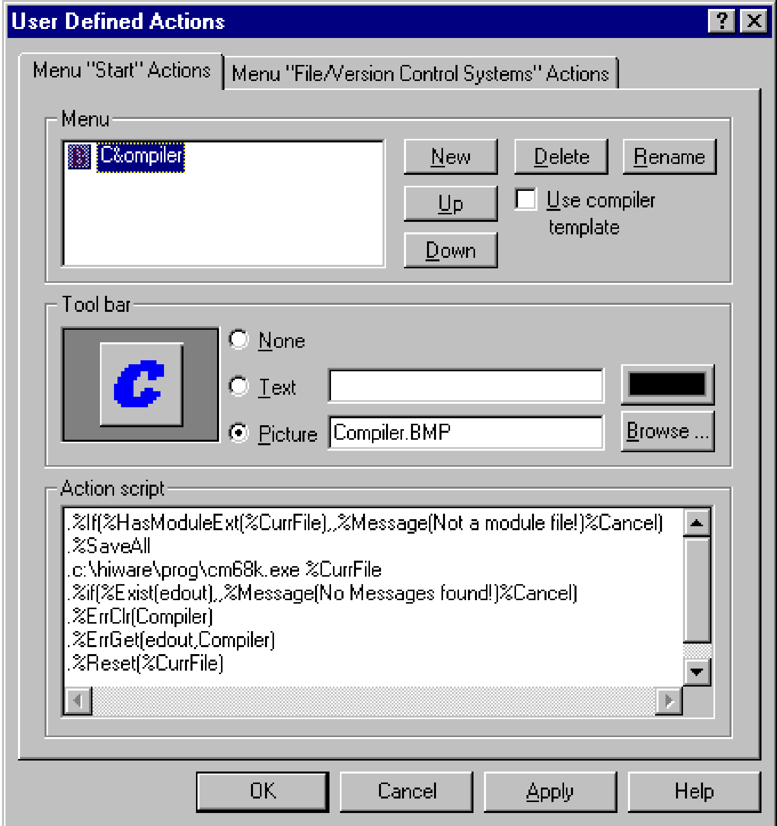
Configuring The Tools

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 on page 221). 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



Now fill in the **Action Script** field in order to associate related compiler actions. Copy the following lines shown in Listing 9.1 on page 222 in the Action Script field and change the directory to where the compiler is located.

Listing 9.1 Script for Compiler Action Association

```
.%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.

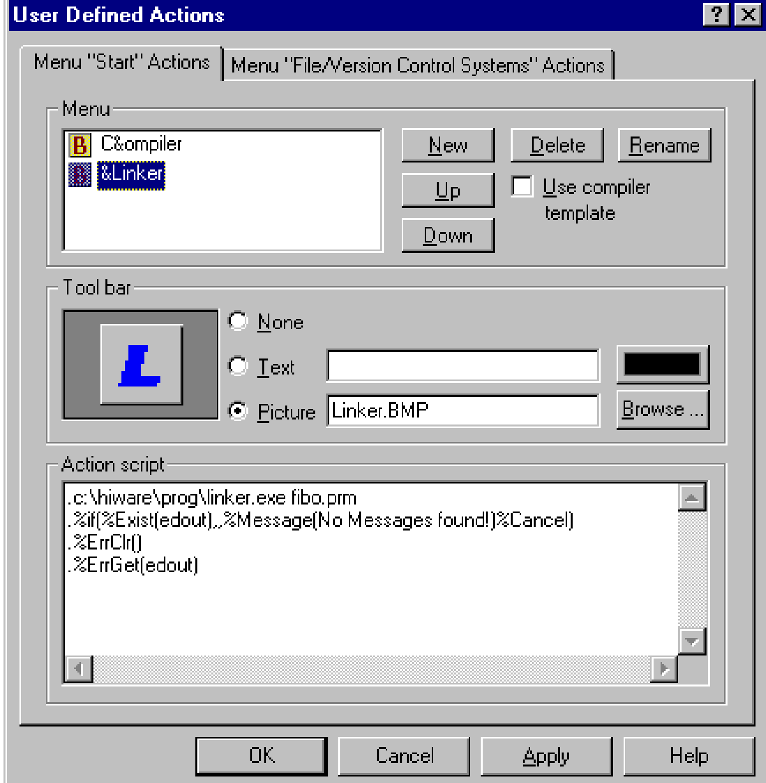
Linker Configuration

In the same way, you can now configure the linker as illustrated in Figure 9.8 on page 223. 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 on page 222 in the **Action Script** field and change the directory to where the linker is located.

Listing 9.2 Script for Linker Action Association

```
+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



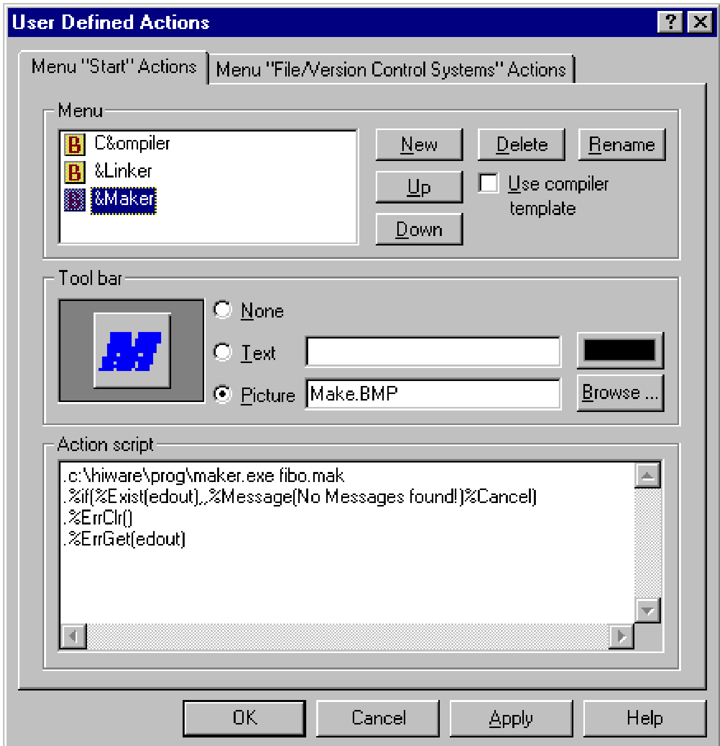
Maker Configuration

In the same way, you can now configure the maker as illustrated in Figure 9.9 on page 224. 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 on page 223 in the **Action Script** field and change the directory to where the maker is located.

Listing 9.3 Script for Maker Action Association

```
+c:\Freescale\prog\maker.exe fibo.mak
.%if(%Exist(edout),,%Message(No Messages found!)%Cancel)
.%ErrClr()
.%ErrGet(edout)
```

Figure 9.9 DA-C Maker Settings



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:

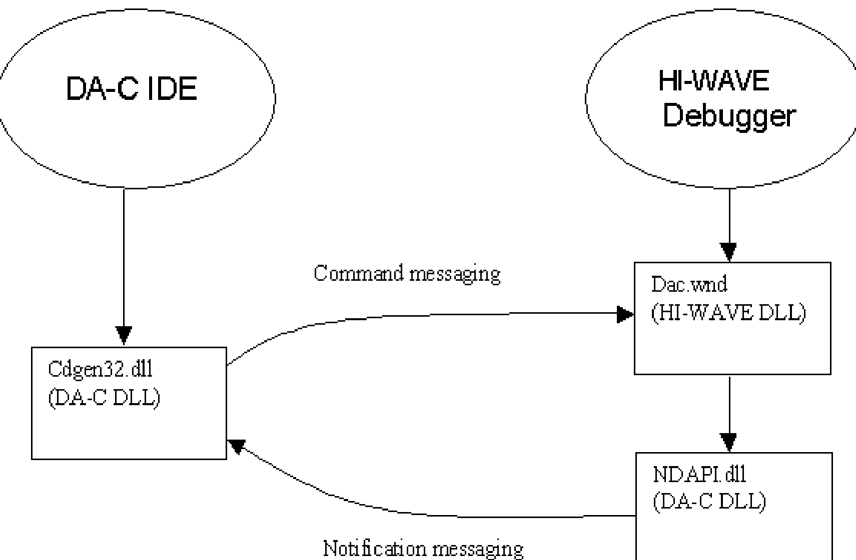
- Installation of communication DLL
- Configuration of Debugger properties
- Configuration of the Debugger project file

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 on page 225. 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

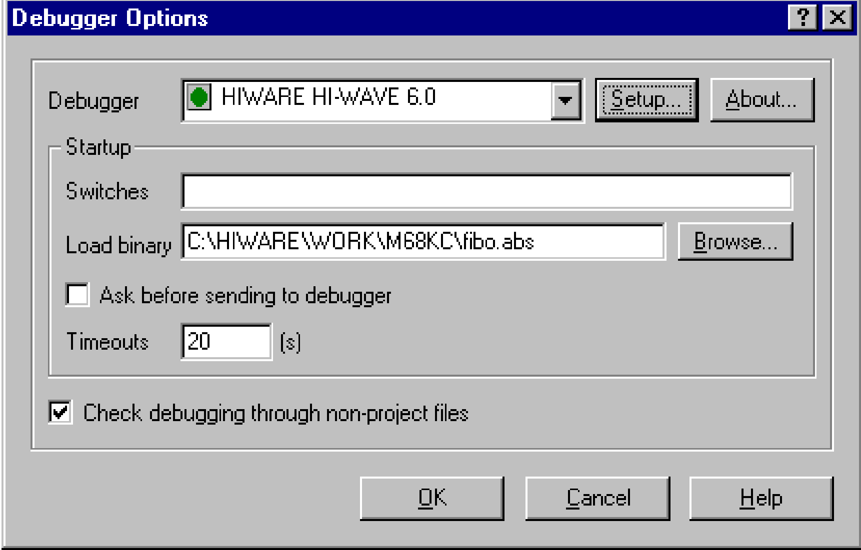
Tool Kit installation. However, if you install a new release of DA-C you have to follow this procedure:

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 on page 226 is opened.

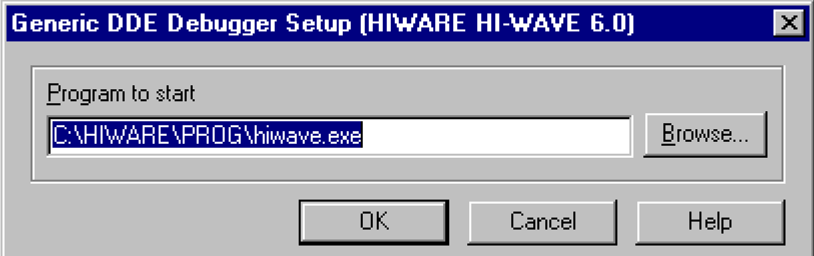
Figure 9.11 DA-C Debugger Options Dialog Box



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 on page 227 is opened.

Figure 9.12 DDE Debugger Setup Dialog Box

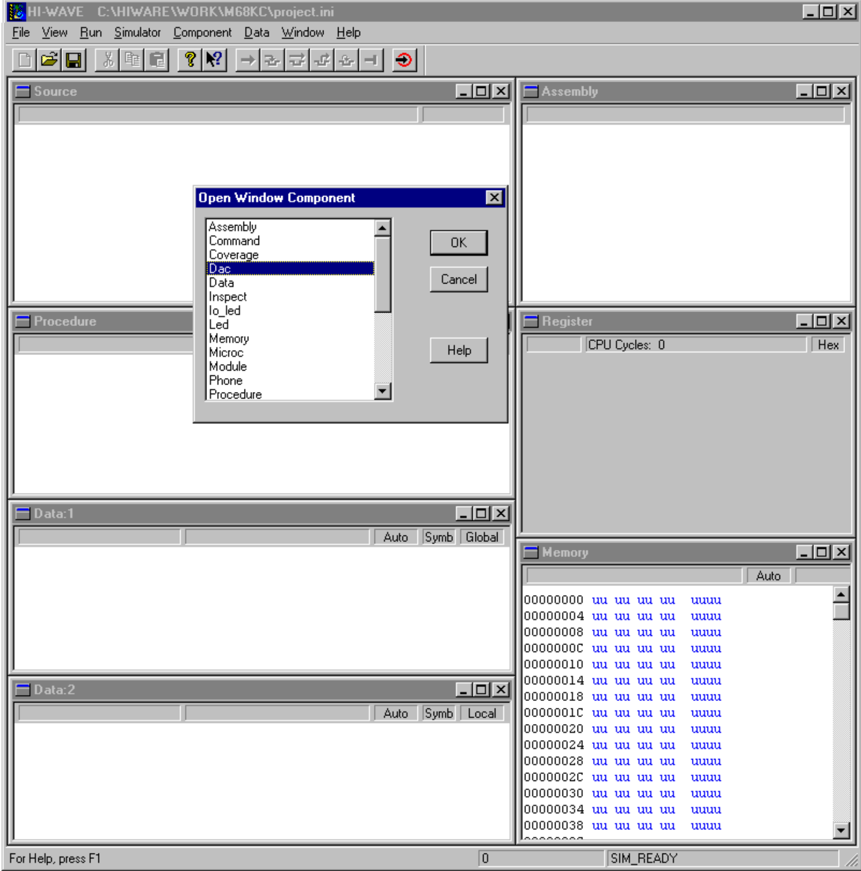


Specify the path to the "**hiwave.exe**" file or use the **Browse**… button then click on **OK**.

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 on page 228.

Figure 9.13 DA-C Component Opening



The Debugger DAC window, which is needed for communication with DA-C IDE is now opened (Figure 9.14 on page 228).

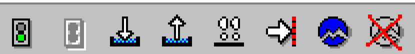
Figure 9.14 DA-C Window

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 on page 229 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.

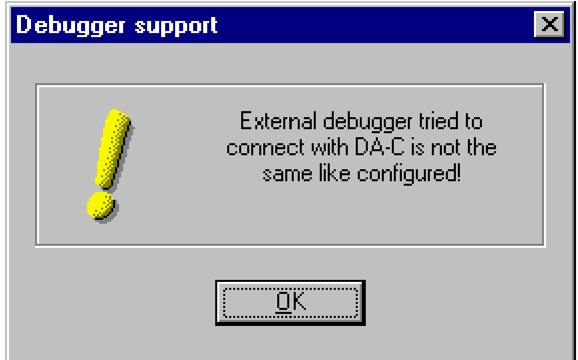
1. When loading DAC component into the Debugger, if the message box shown in Figure 9.16 on page 229 is displayed:

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.

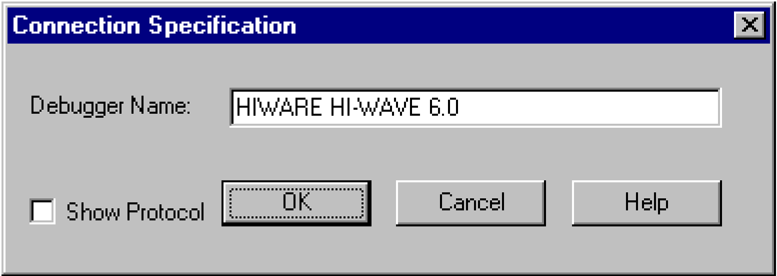
2. If the message box shown in Figure 9.17 on page 230 is displayed in DA-C IDE:

Figure 9.17 DA-C Debugger Support Message



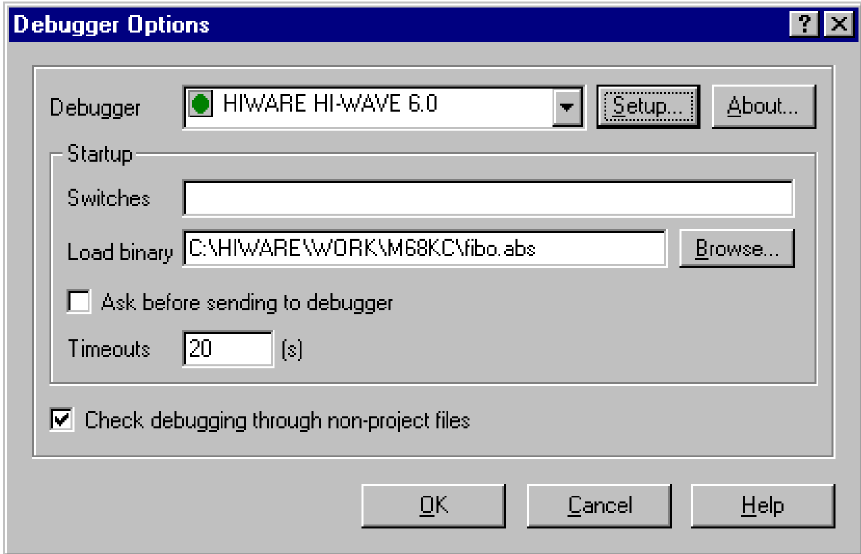
This means that the current name specified in the **Options>Debugger** main menu of DA-C doesn't match the debugger name specified in the Debugger. Open the setup dialog in the Debugger by clicking on the DA-C Link component and choose **DA-C Link>Setup…** from the main menu. The "Connection Specification" dialog box is opened (Figure 9.18 on page 230).

Figure 9.18 DA-C Connection Specification Dialog Box



Compare the "**Debugger Name**" from this dialog box with the selected Debugger in DA-C IDE (**Options>Debugger**), as shown in Figure 9.19 on page 231.

Figure 9.19 DA-C Debugger Options Dialog Box



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 on page 231.

Listing 9.4 DA-C Section in Project File.

```
[DA-C]
DEBUGGER_NAME=HI-WAVE 6.0
SHOWPROT=1
```

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 "HC12 Debugging First Steps" on page 235
- Chapter 2.2 "HC(S)12(X) Full Chip Simulation Connection" on page 259
- Chapter 2.3 "P&E Multilink/Cyclone Pro Connection" on page 405
- Chapter 2.4 "Softec HCS12 Connection" on page 413
- Chapter 2.5 "HCS12 Serial Monitor Connection" on page 417
- Chapter 2.6 "Abatron BDI Connection" on page 427

10

HC12 Debugging First Steps

Debugging HC12 code using the CodeWarrior IDE requires that a project be created or exist which specifies a connection that can be used to debug the code. This section guides you through the first steps toward HC12 code debugging with CodeWarrior and the following connections:

- *Full Chip Simulation* connection
- *HCS12 Serial Monitor* connection
- *SofTec inDart HCS12* connection
- *ICD-12* connection
- *BDIK* connection

It does not replace all the additional documentation provided in this manual, but gives you a good start.

Technical Considerations

While they can be used to debug HCS12 and HCS12X code (selectable in the Wizard), any of the connections listed in the previous paragraph may be used to debug HC12 code. Some of these connections have special technical considerations, as discussed in the following paragraphs.

Full Chip Simulation Considerations

The Full Chip Simulation (FCS) connection runs a complete simulation of all processor peripherals and I/O on the user's PC. No development board is required. Each derivative has a totally different simulation engine to accurately simulate the memory ranges, I/O, and peripherals for any given derivative.

HCS12 Serial Monitor 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.

SofTec HCS12 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 throught 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.

ICD-12 Considerations

In order to use the **P&E Cable 12** or **P&E BDN-Multilink**, the drivers from P&E must be installed on the host computer.

A parallel cable should be used for communication between the **P&E Cable 12** or **BDM-Multilink** and the host computer.

The communication protocol between the **P&E cable 12** or **BDM-Multilink** and the host is fully handled by the *unit_12z.dll* target driver which is automatically loaded with the ICD-12 connection.

BDIK Considerations

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.

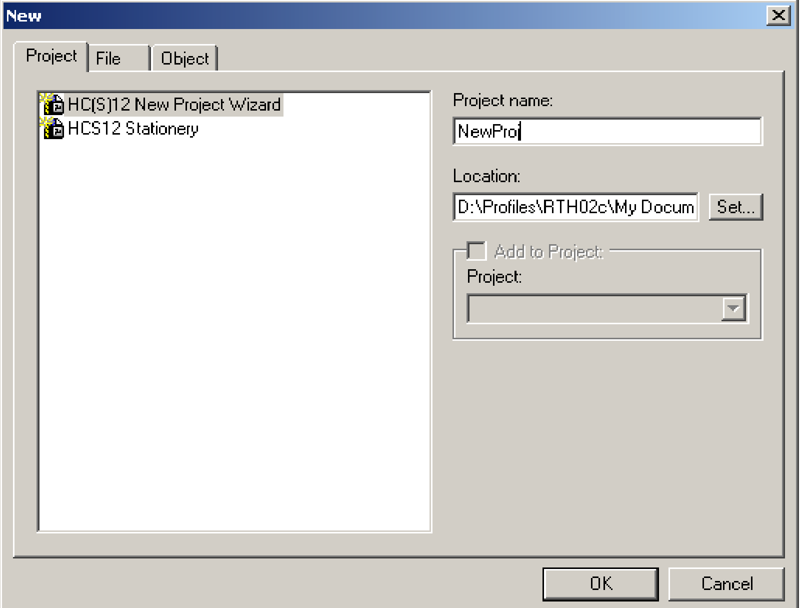
One free serial communication port of your computer is required to communicate with the BDI interface. You may need to set it up even if you will be using an Ethernet communication instead of an RS-232 serial communication.

Debugging First Steps Using the Wizard

To take the first steps toward debugging with CodeWarrior IDE using the stationery Wizard:

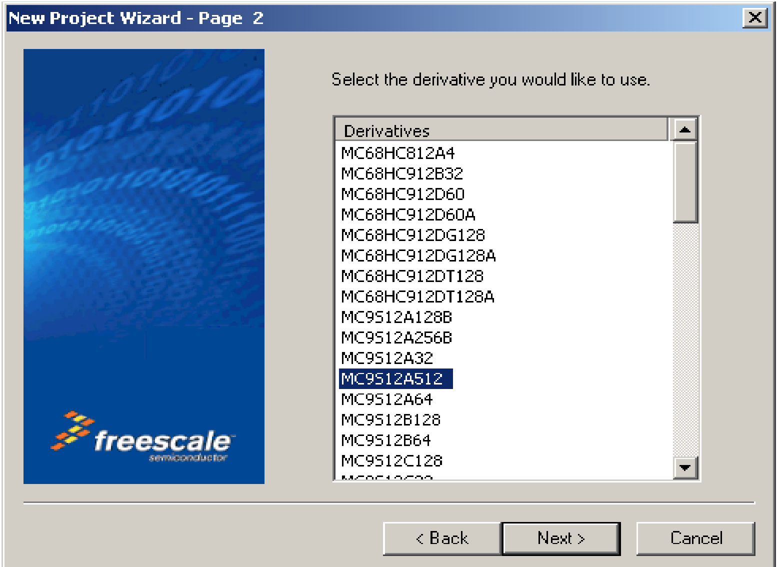
1. Run the *CodeWarrior IDE* with the shortcut created in the program group.
2. Choose the menu **File > New** to create a new project from a stationery - the ***HCS12 New Project Wizard first*** screen appears.

Figure 10.1 New Window - Project Tab



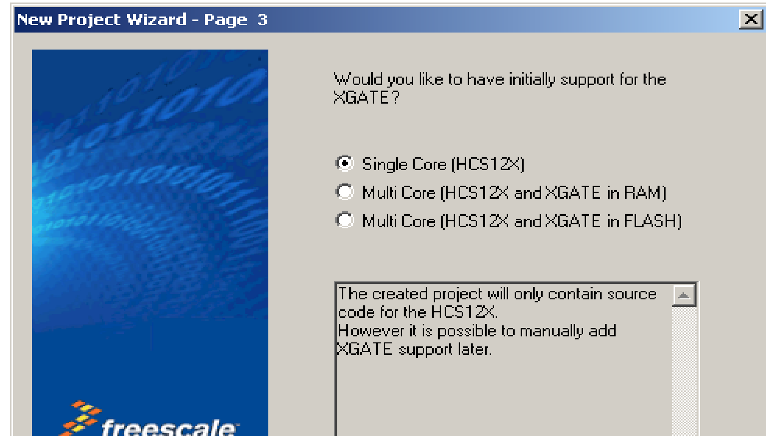
3. In the list box on theleft of the screen, select ***HC(S)12 New Project Wizard***.
4. In the Project Name textbox, type the name of your new project.
5. Click the **OK** button to proceed.

Figure 10.2 Select Derivative Screen



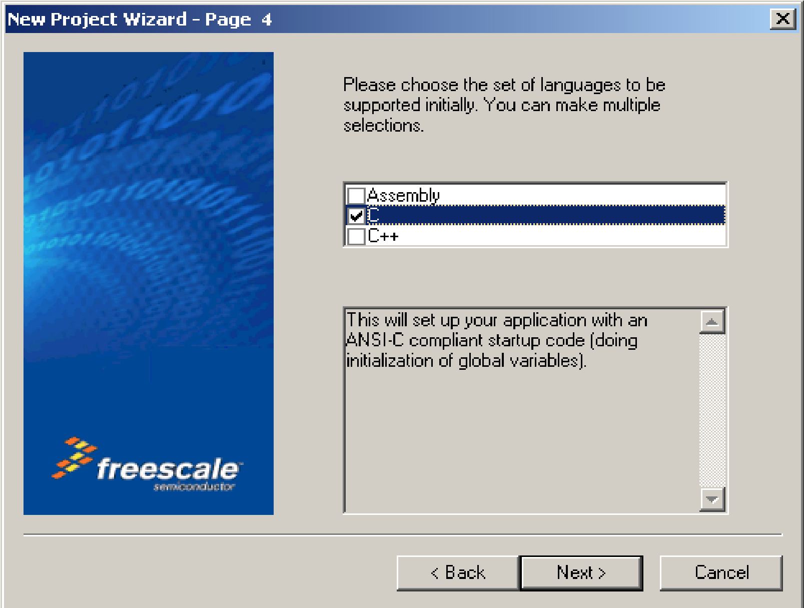
6. In the list box on the screen, select the HC12 MCU you are targeting.
7. Click the **Next** button to proceed.

Figure 10.3 XGATE Support Screen



8. Unless you need XGATE support, select the Single Core format by checking its checkbox and clicking the **Next** button to proceed.
   Selecting any of the options results in the following conditions:
   - Single Core (HCS12X) - The created project will only contain source code for the HCS12X. However, it is possible to add XGATE support at a later date manually.
   - Multi Core (HCS12X and XGATE in RAM) - The created project will contain source code for the HCS12X and the XGATE. The HCS12X code copies the XGATE code from FLASH into RAM and then configures the XGATE.
   - Multi Core (HCS12X and XGATE in FLASH) - The created project will contain source code for the HCS12X and the XGATE. The XGATE will execute out of the FLASH. Note that this setup does not work with the LC40 mask set of the MC9S12XPD512.

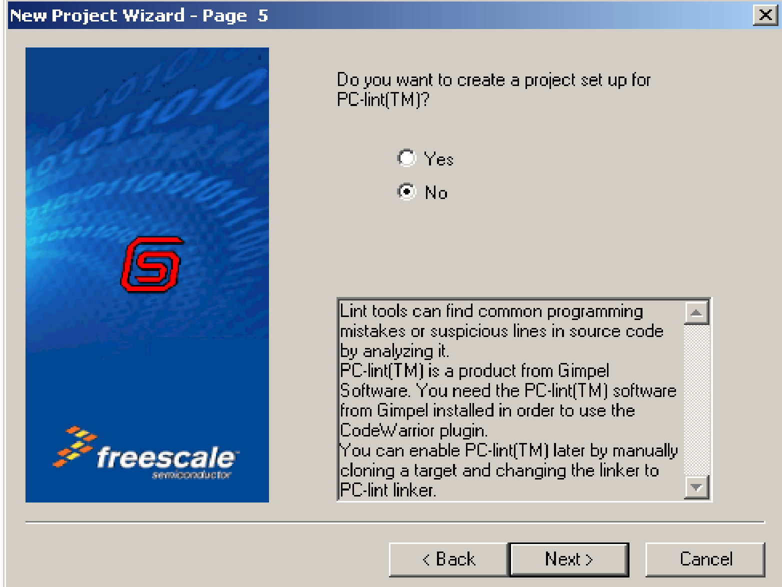
Figure 10.4 Language Support Screen



9. Select the Language format by checking its checkbox, then clicking the **Next** button to proceed. You can make multiple selections, creating the code in multiple formats. Selecting any of the options results in the following conditions:

- Assembly - If only Assembly is selected, you can later choose to use absolute/single file assembly application or relocatable assembly.
- C - This will set up your application with an ANSI C-compliant startup code, doing initialization of global variables.
- C++ - This will set up your application with an ANSI C++ startup code, doing global class object initialization.

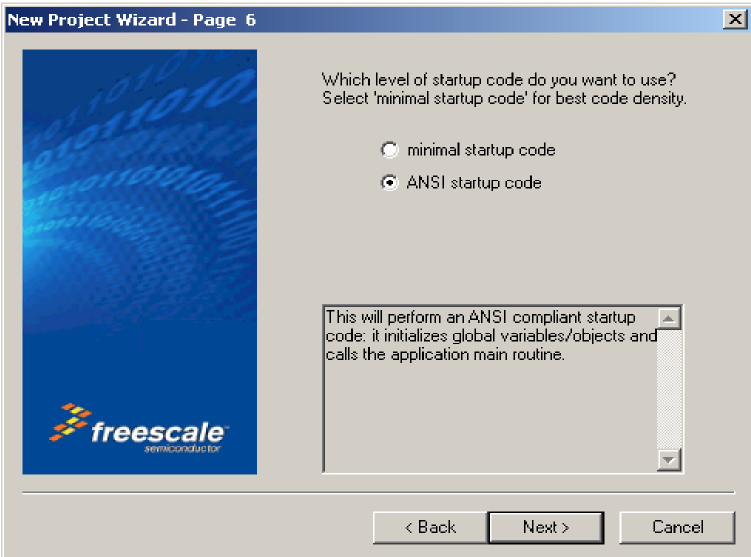
Figure 10.5 PC Lint Support Screen



10. Unless you wish to create a project set up for PC-lint, select **No** by checking its checkbox and clicking the **Next** button to proceed.
    While Lint tools can find common programming mistakes or suspicious lines in surce code by anayzing it, you need the PC-lint software from Gimpel installed in order to use the CodeWarrior plugin. You can enable PC-lint later by manually cloning a target and changing the linker to PC-lint linker.
    Selecting the Yes option results in the following conditions:

    - This will add an additional target to the project with the name PC-Lint. A professional license is required to use the PC-lint plugin.

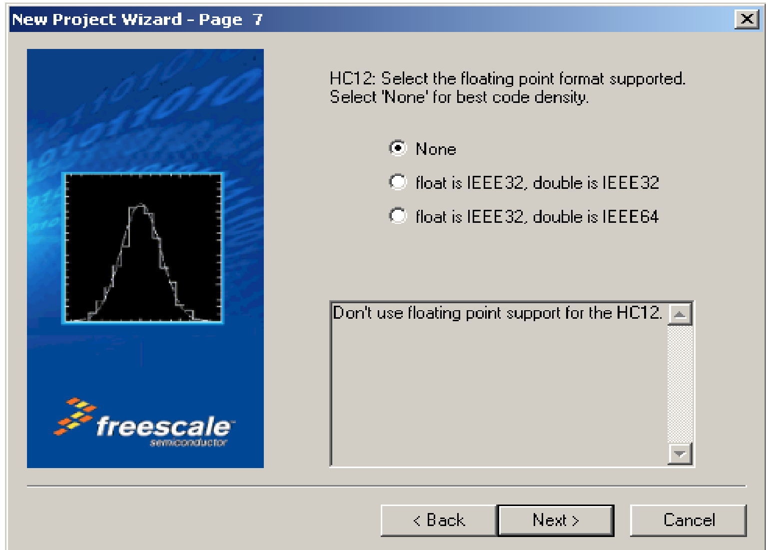
Figure 10.6 Startup Code Level Screen



11. This screen lets you select the level of startup code you wish to produce. Selecting the minimal startup code option produces the best code density. However, selecting either of the options, then clicking on the **Next** button results in the following conditions:

- Minimal startup code - This startup code initializes the stack pointer and calls the main function. No initialization of global variables is done, giving the user the best speed/code density and a fast startup time. But, the application code has to care about variable initialization. This makes this option not ANSI compliant, since ANSI requires variable initialization.
- ANSI startup code - This performs an ANSI-compliant startup code that initializes global variables/objects and calls the applicaiton main routine.

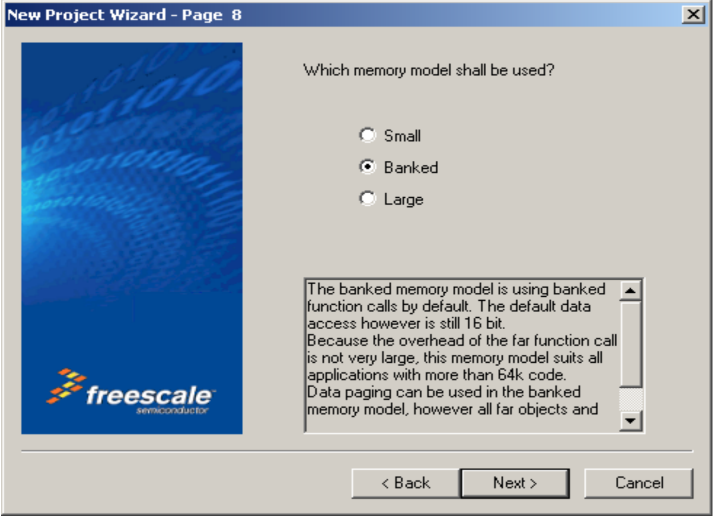
Figure 10.7 Floating Point Selection Screen



12. Select the floating point format by checking its checkbox and clicking the **Next** button to proceed. Selecting any of the options results in the following conditions:

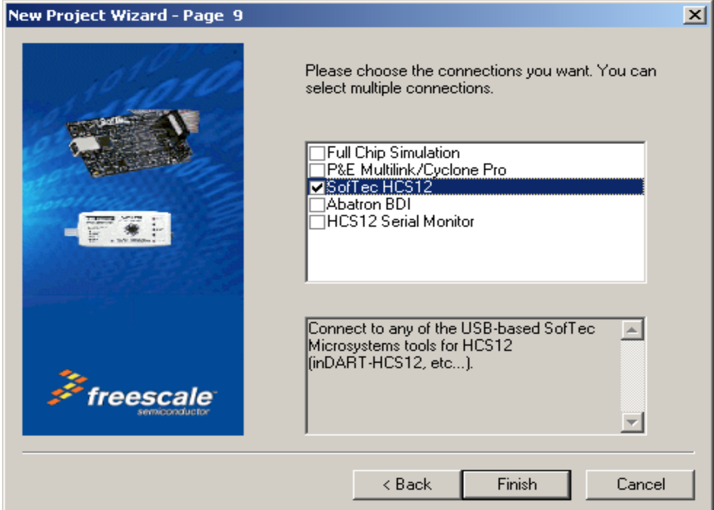
- None - Don't use floating point for the HC12.
- Float is IEEE32, double is IEEE32 - All float and double variables are 32-bit IEEE32 for the HC12.
- Float is IEEE32, double is IEEE64 - Float variables are 32-bit IEEE32. Double variables are 64-bit IEEE64 for the HC12.

Figure 10.8 Memory Model Selection Screen



13. Select the Memory Model by checking its checkbox and clicking the **Next** button to proceed. Selecting any of the options results in the following conditions:
    - Small - The Small memory model is best used if both the code and the data fit into the 64kB address space. By default all variables and functions are accessed with 16-bit addresses. the compiler does support banked functions or paged variables in this memory model, but all accesses have to be explicitly handled.
    - Banked - Banked memory model uses banked function calls by default, but the default data access is still 16-bit.Because the overhead of the far funtion call is not very large, this memory model suits all applications with more than 64k code. Data paging can be used, however all far objects and pointers to them have to be specially declared.
    - Large - The Large memory model supports both code banking and data paging by default. But, data paging causes an especially large volume of overhead and should therefore be used with care. Overhead is significant with respect to both code size and speed. If it is possible to manually use far accesses to any data which does not fit into the 64-bit address space, then the banked memory model should be used instead.

Figure 10.9 Connection Selection Screen

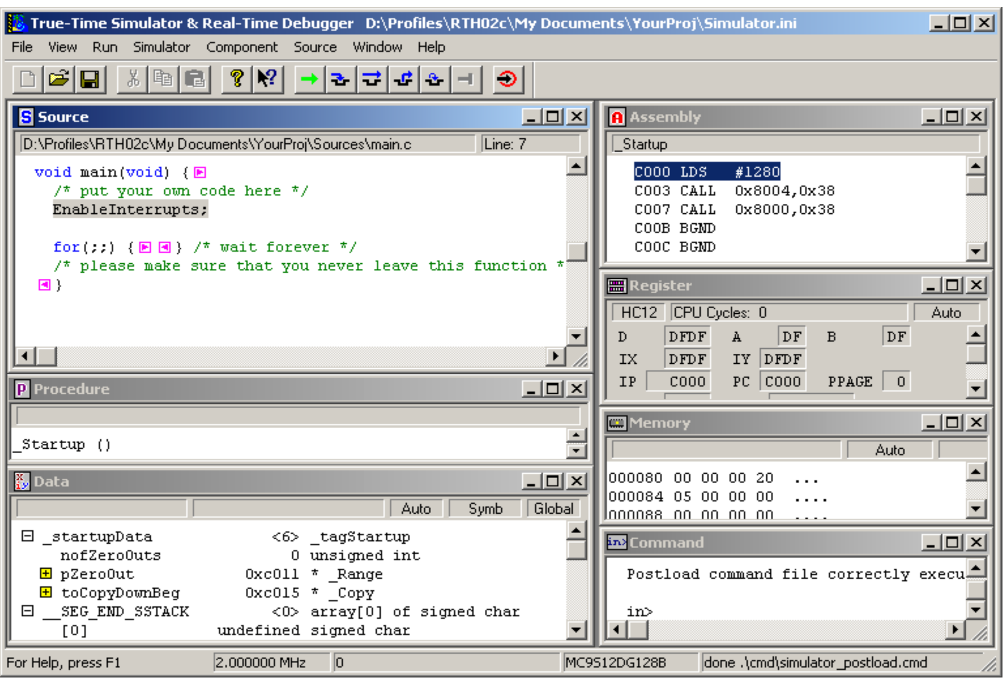


14. Select the connection by checking its checkbox and clicking the **Next** button to proceed. You can make multiple selections, which makes multiple connections available in the Debugger Main Screen later. Selecting any of the options results in the following conditions:

    - Full Chip Simulation - Connects to Freescale Full Chip Simulation with simulation of on-chip peripherals. With this selection, you can switch to hardware debugging later in the debugging session.
    - P&E Multilink/Cyclone Pro - Connects to the P&E BDM Multilink (USB and Parallel), or to P&E Cyclone Pro (USB, Serial, and TCP/IP. Choose this connection for ICD-12 debugging.
    - Softec HCS12 - Connects to any of the USB-based SofTec Microsystems tools for the HC12, (in-Dart-HCS12, etc.).
    - Abatron BDI - Connect to the hardware board using Abatron hardware (BDI-HS or BDI 1000) through the BDM connection.
    - HCS12 Serial Monitor - Connect to a board through the Freescale HCS12 Serial Monitor. The Freescale HCS12 Serial Monitor must be on the board before you connect.

15. Click on the **Finish** button - the IDE opens.
16. In the IDE main window toolbar Project menu, choose **Project > Make.**
17. Now choose **Project > Debug** to start the debugger.

Figure 10.10 Your Project in Debugger Main Window



Switching Connections

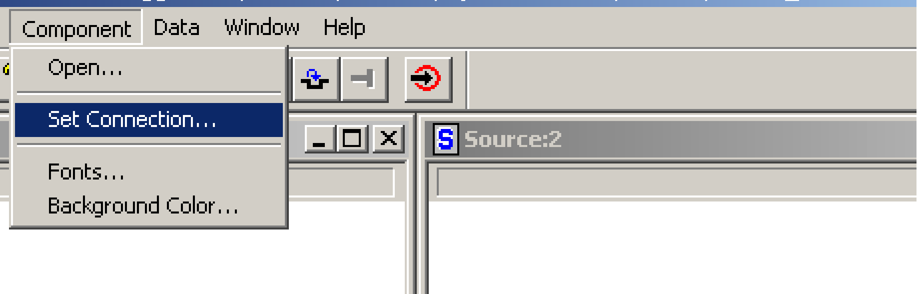
It is possible to switch connections from within an existing HC12 debugging project. the following paragraphs give directions for accompliching this.

Loading the Full Chip Simulation Connection

Because there is no actual hardware involved in switching from another project, such as the SofTec in-Dart HCS12 connection, to the FCS connection, the process is simple. To load the FCS connection from within an existing project, take the following steps:

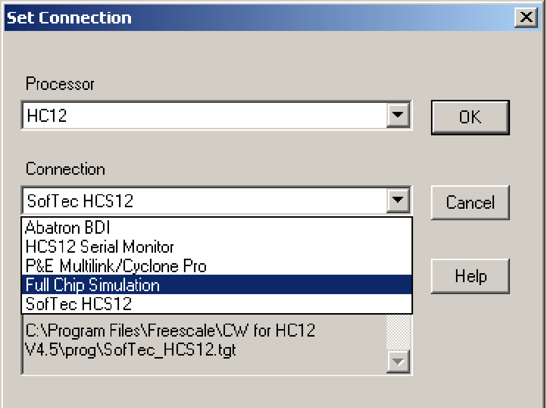
1. From the Debugger main menu, select *Component | Set Connection...*, as shown below.

Figure 10.11 Component Menu



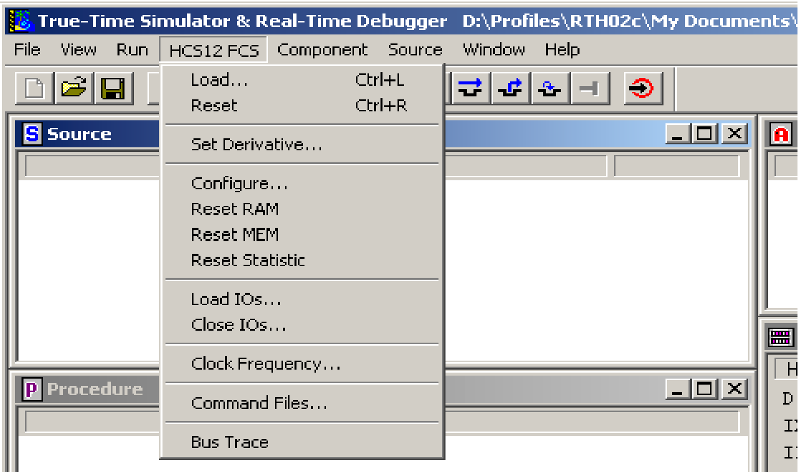
The Set Connection dialog box now appears.

Figure 10.12 Set Connection Dialog Box



2. Set the Processor as HC12 and the Connection as Full Chip Simulation.
3. Press the OK button. The Debugger main menu entry bar for the connection now changes to HCS12 FCS.

Figure 10.13 HCS12 FCS Menu



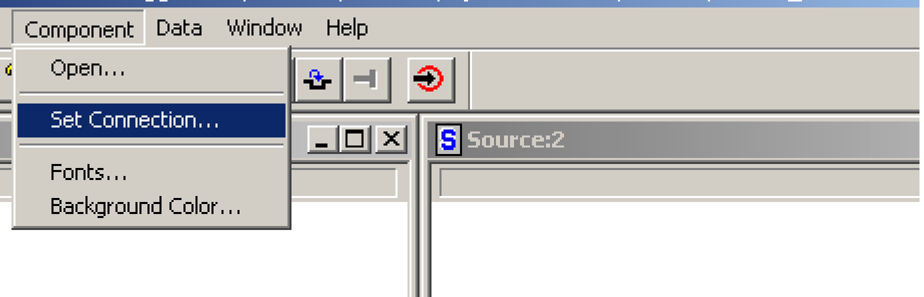
You have successfully switched connections to the FCS connection. The values and use of each HCS12 FCS menu entry is explained in the Full Chip Simulation chapter of this manual.

Loading the ICD-12 Connection

To load the ICD-12 connection from within an existing project, take the following steps:

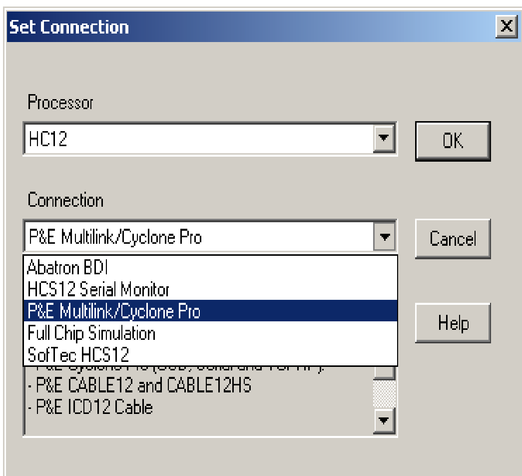
1. From the Debugger main menu, select *Component* | *Set Connection...*, as shown below.

Figure 10.14 Component Menu



The Set Connection dialog box now appears.

Figure 10.15 Set Connection Dialog Box - Connection Menu

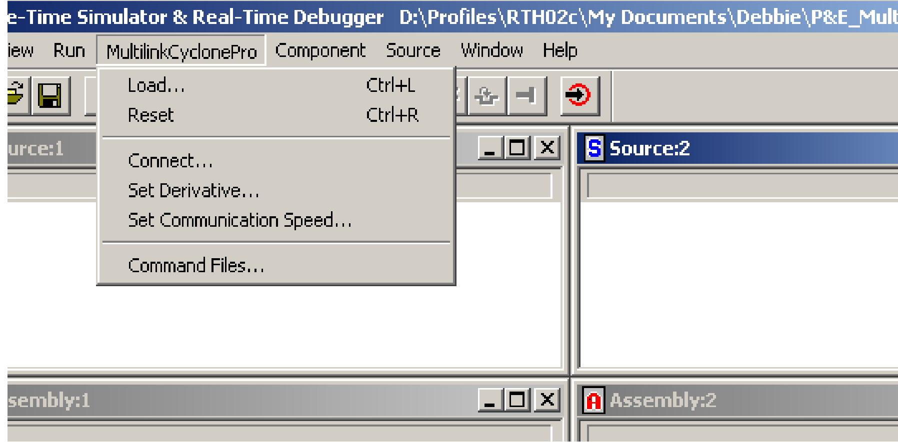


2. Within the Set Connection dialog box, press the Down Arrow button next to the Connection list box to display the list of available connections.
3. Select P&E Multilink/Cyclone Pro.

The Connection menu selection *P&E Multilink/Cyclone Pro* loads the proper drivers, etc. for the ICD-12 connection.

4. In the Debugger Main window, the Connection heading has been renamed **MultilinkCyclone Pro**. Click on this heading to display its menu with the list of possible ICD-12 selections.

Figure 10.16 MultilinkCyclone Pro (ICD-12) Menu

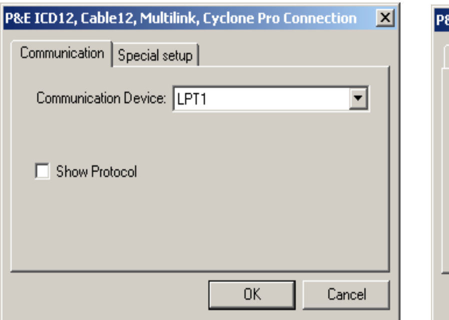


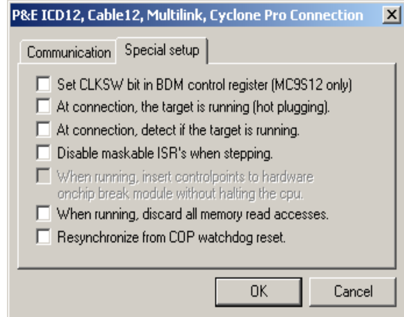
The menu selection *MultilinkCyclonePro* | *Load...* loads an executable ("`.abs`") file into connection memory. The file's program counter points to the first instruction of the startup section.

The menu selection *MultilinkCyclonePro*| *Reset* triggers a reset of the connection and executes the command file "`reset.cmd`".

The menu selection *MultilinkCyclonePro | Connect...* takes you to the P&E ICD-12, Multilink, Cyclone Pro dialog box. The two tabs of this dialog box allow you to set the Communications and Special Setup parameters for the ICD-12 connection.

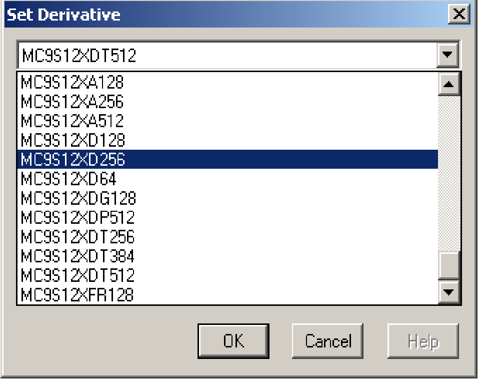
Figure 10.17 P&E Multilink, Cyclone Pro Connection Dialog Box





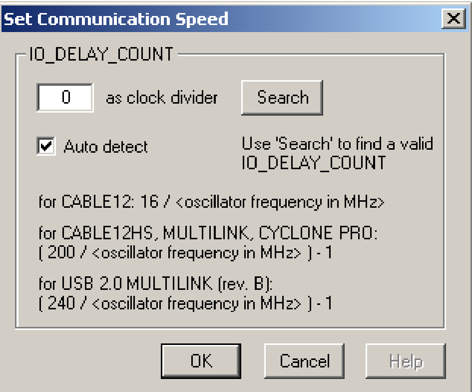
5. The menu selection *MultilinkCyclonePro | Set Derivative...*takes you to the Set Derivative dialog box. This dialog box allow you to choose the target MCU for the ICD-12 connection.

Figure 10.18 Set Derivative Dialog Box



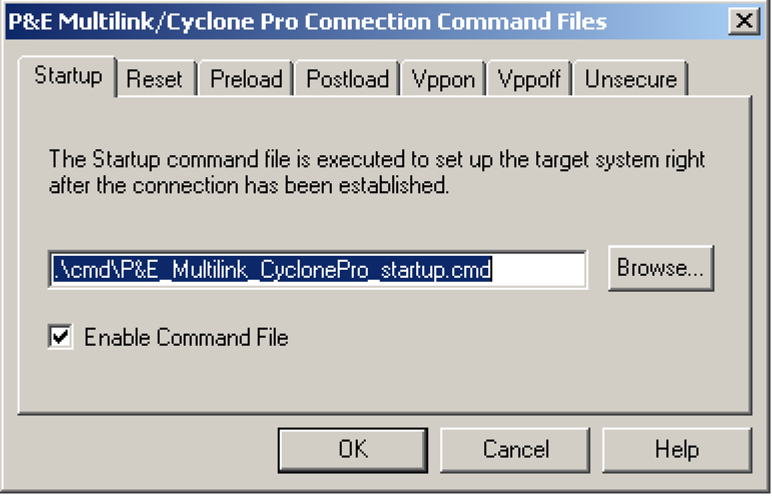
The menu selection *MultilinkCyclonePro* | *Set Communication Speed...*lets you control the various factors associated with communication speed for the ICD-12 connection.

Figure 10.19 Set Communication Speed Dialog Box



The menu selection *MultilinkCyclonePro* | *Command Files...* takes you to the Command Files window with its six tabs.

Figure 10.20 Command Files Window

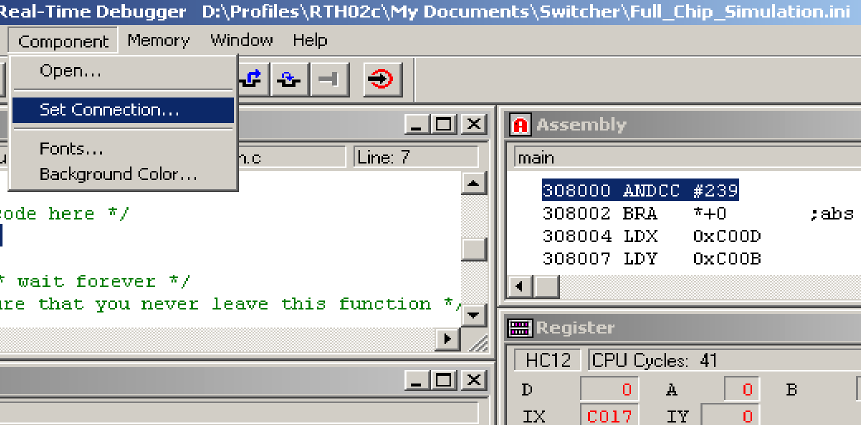


Switching to SofTec HC12

To take the first steps toward debugging with CodeWarrior and setting the SofTec HCS12 connection from within an existing debugging project, such as the Full Chip Simulation connection, take the following steps:

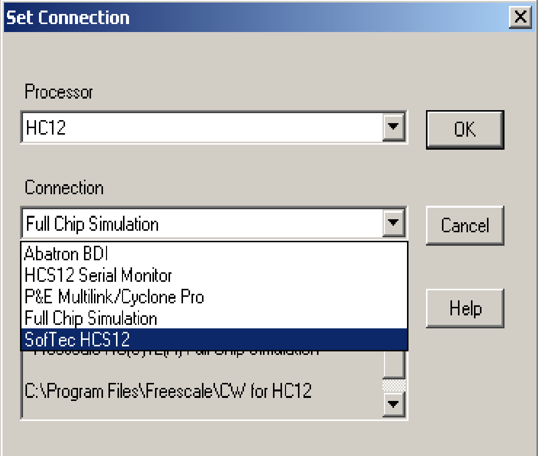
1. In the Debugger window menubar, display the Component menu

Figure 10.21 Component Menu



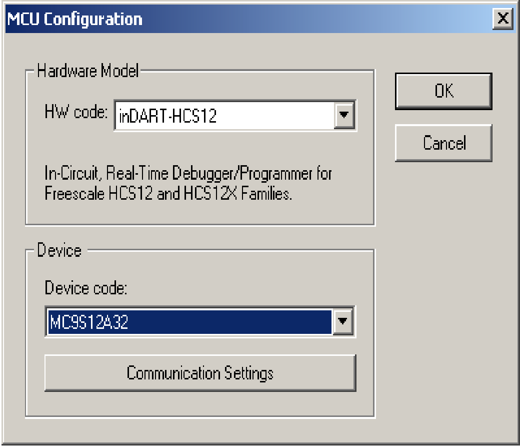
2. Choose **Component > Set Connection..** from this menu to select another connection in the Set Connection dialog box.

Figure 10.22 Set Connection Dialog Box - SofTec HCS12 Selection



3. Select ***HC12*** as Processor.
4. Select ***SofTec HCS12*** as connection.

Figure 10.23 MCU Configuration Dialog Box



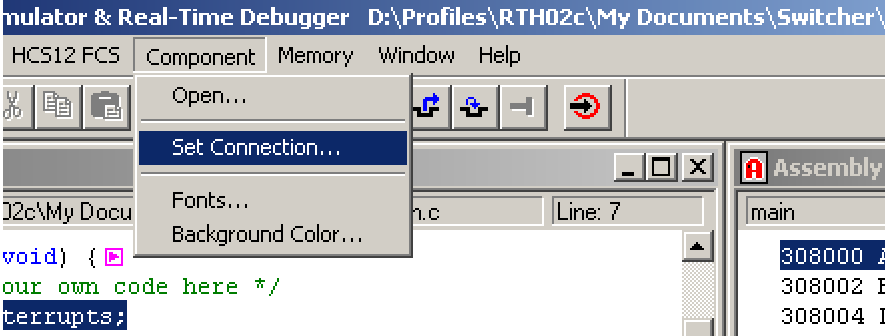
5. In the MCU Configuration dialog box, choose the correct target processor.
6. Press the OK button to start debugging.

Switching to HCS12 Serial Monitor Connection

To take the first steps toward debugging with CodeWarrior choosing the HCS12 Serial Monitor connection from within an existing debugging project that uses another conection, such as the Full Chip Simulation, take the following steps:

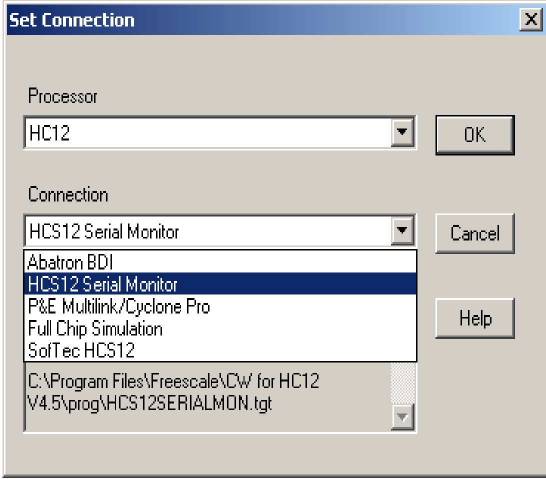
1. Within the Debugger Main window select the Component entry.

Figure 10.24 Debugger Main Window - Component Menu



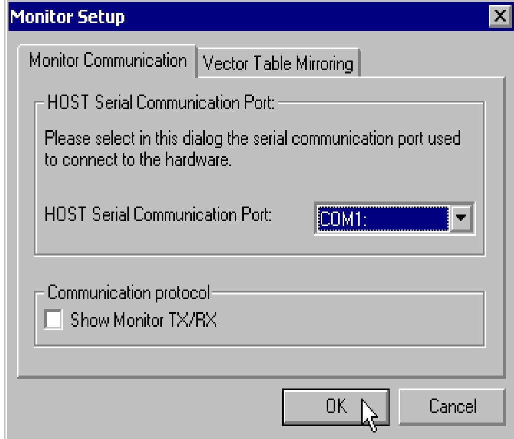
2. From the Component menu, choose **Component > Set Connection...** to select another connection.

Figure 10.25 Set Connection Dialog Box - HCS12 Serial Monitor Selection



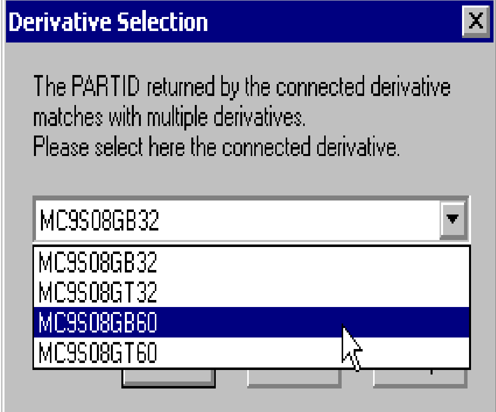
3. Select ***HC12*** as Processor then ***HCS12 Serial Monitor*** as the connection in the Set Connection dialog box and click the OK button.
4. Now in the Monitor Setup window, Monitor Communitation tab, choose the correct Host serial communication port if necessary.

Figure 10.26 Monitor Setup Window - Monitor Communication Tab



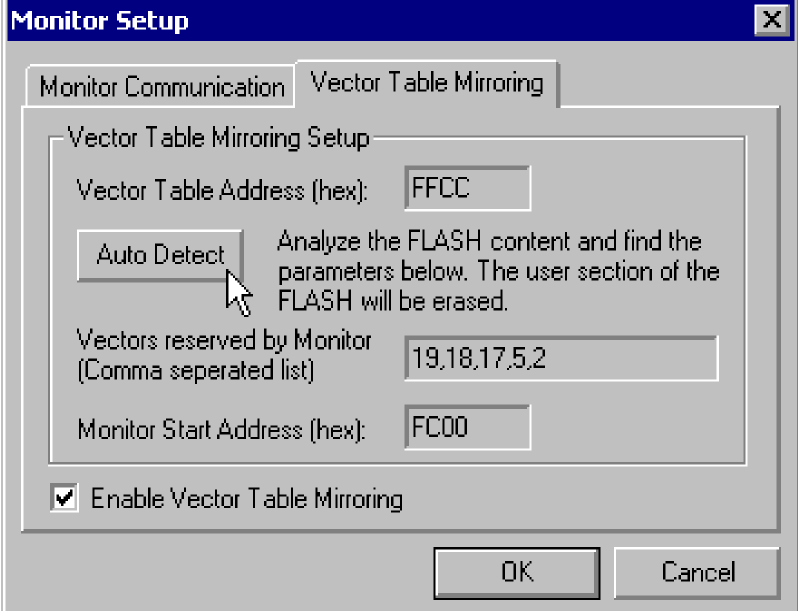
5. Press the OK button. The HCS12 Serial Monitor connection reads the device silicon ID. This ID can match several derivatives.
6. Set the correct derivative matching with your hardware in the Derivative Selection dialog box.

Figure 10.27 Derivative Selection Dialog Box



7. Press the OK button. The **Monitor Setup** window is opened again showing the Vector Table Mirroring Tab, to propose to use the “mirrored vector table” feature. We recommand 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.
8. To enable this specific feature, check the “Enable Vector Table Mirroring” checkbox. ¨

Figure 10.28 Monitor Setup Window - Vector Table Mirroring Tab



9. Press the “Auto Detect” button to make the debugger search for the vector table address and vectors reserved by the HCS12 Serial Monitor.
10. Once the autodetection succeeded, press the OK button to start debugging.

11

HC(S)12(X) Full Chip Simulation 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 11.1 on page 259 is associated with the Full Chip Simulation connection, and allows you to load an application in the Full Chip Simulation. Table 11.1 on page 260 describes the Full Chip Simulation menu entries.

Figure 11.1 HCS12 FCS Menu

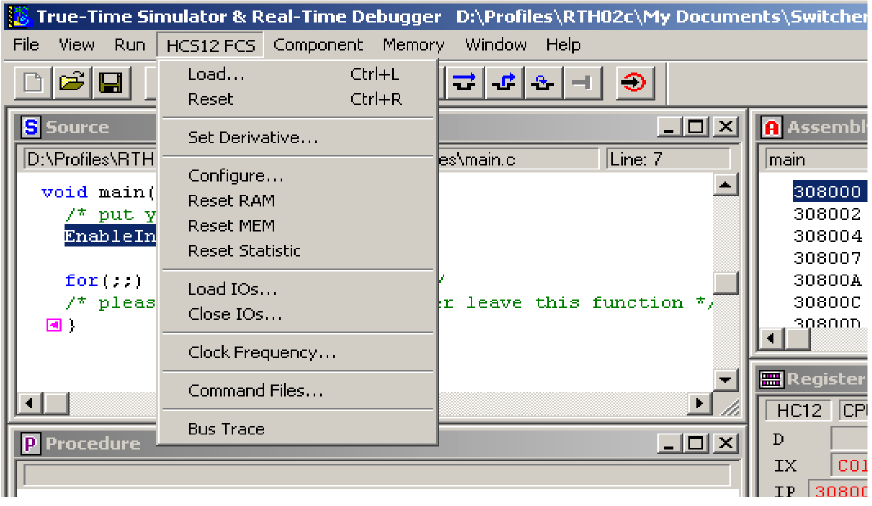


Table 11.1 Simulator Menu Entry Description

Menu Entry	Description
Load	Opens the Load Executable Window menu.
Reset	Resets the Full Chip Simulation.
Set Derivative	Selects the current simulated derivative.
Configure	Opens the Memory Configuration on page 263 Window.
Reset Ram	Resets the RAM to `undefined'
Reset Mem	Resets all configured memory to `undefined'
Reset Statistic	Resets the statistical data
Load I/Os	Opens I/O components
Close I/Os	Closes I/O components
Clock Frequency	Open the Clock Frequency Setup on page 269dialog box to set the Real Time clock of the Full Chip Simulation.
Command Files	Opens the Command File Window on page 262
Bus Trace	Opens the Bus Trace on page 270 dialog box to enable instructions and memory accesses recording and display recording captures.

Debugger Status Bar with Full Chip Simulation

The status bar (Figure 11.2 on page 260 and Figure 11.3 on page 260) 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 11.2 Debugger Status Bar with CPU Cycles

For Help, press F1 | HC12 | 8.000000 MHz | 11'007'656 | MC9S12XDP512 | HALTED

Figure 11.3 Debugger Status Bar with Elapsed Time

For Help, press F1 | HC12 | 8.000000 MHz | 1927:06'04"341.202ms | MC9S12XDP512 | HALTED

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 on page 269.

NOTE Double-clicking on the CPU cycles or true time resets the value.

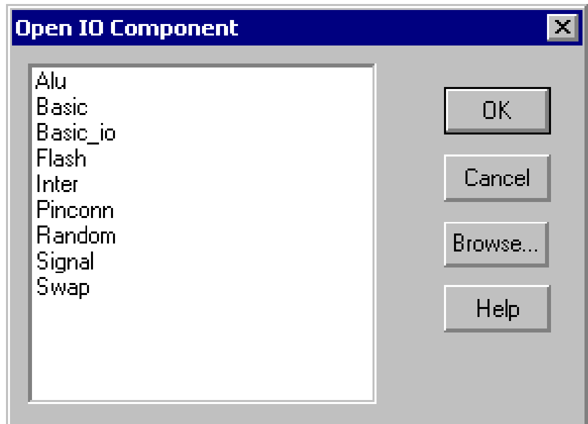
NOTE Clicking on the displayed derivative or “CORE” or core “SAMPLE” opens the Set Derivative dialog box.

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 11.4 on page 261 allows you to open an **I/O** device (peripheral) simulation. The **Browse** button allows you to specify a location for the I/O.

Figure 11.4 Open IO Component Dialog Box



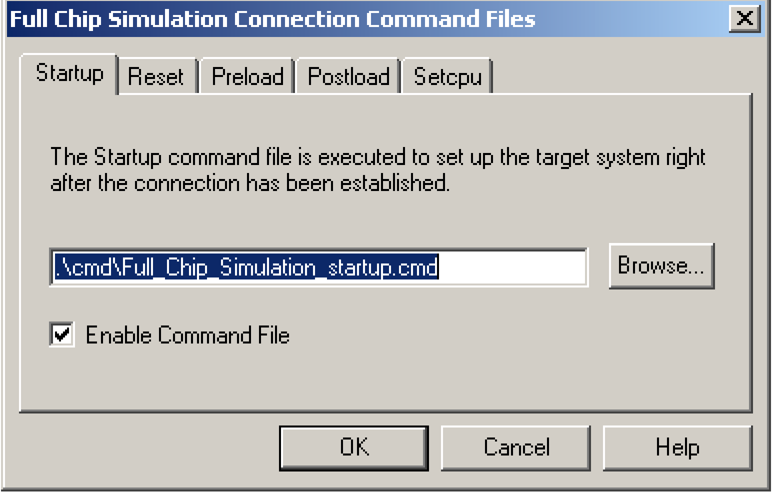
NOTE I/O simulation components are either designed by Freescale and delivered with the tool-kit installation or designed by the user with the Peripheral Builder (separate product).

Demo Version Limitations

Only 2 I/O components can be loaded at a time.

Command File Window

Figure 11.5 Full Chip Simulation Connection 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.

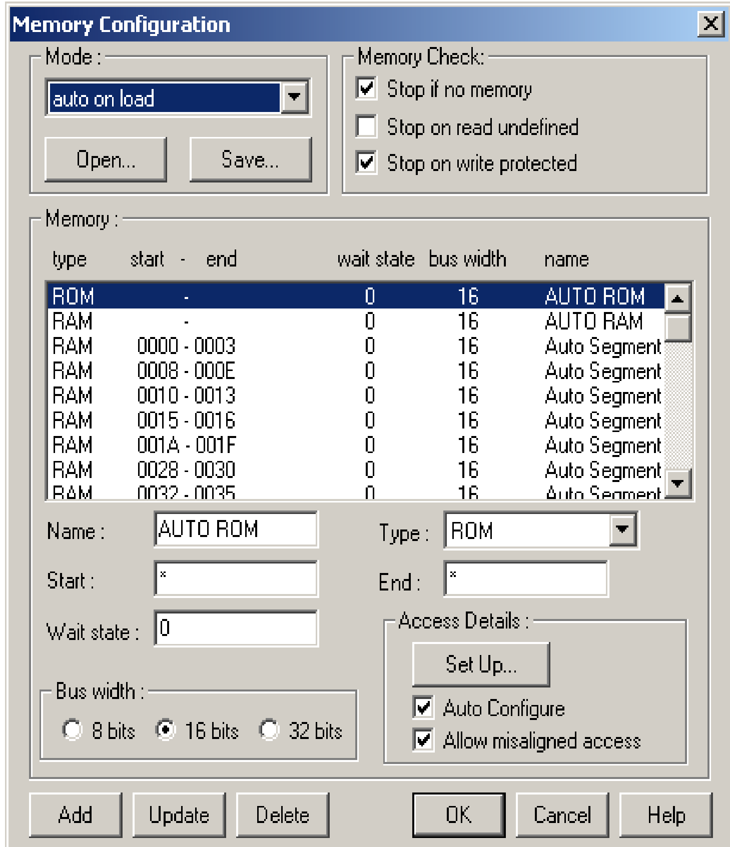
The memory configuration facility uses a binary file format to read and set the Full Chip Simulation configuration. The extension for binary files is `.mem`; the default memory file is `default.mem`. (The subsection “Format of the Default Memory Configuration File” includes Listing 11.2 on page 362, the EBNF-syntax definition of the file format.)

Memory Configuration Dialog Box Features

The memory configuration dialog box (Figure 11.6 on page 264) 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

Figure 11.6 Memory Configuration Dialog Box



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 11.2 on page 265 explains.

Table 11.2 Memory Configuration Modes

Mode	Description
Auto Configuration on Access (Standard Configuration)	Defines the Full Chip Simulation memory as RAM of unlimited size. The *Mode* combo box displays *auto on access*.
Auto Configuration on Load (default)	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 *not implemented*; access to not-implemented locations result in error messages.) The *Mode* combo box displays *auto on load*.
Manual Configuration: (User Defined)	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 *Mode* combo box displays *user defined*.

Memory Configuration Settings

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.

<u>Standard Configuration: Auto on Access:</u> 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.

<u>Auto Configuration on Load:</u> 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.

<u>Manual Configuration:</u> 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 Freescale 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 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 map 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 11.3 on page 267 lists the properties you may specify for a memory block:

Table 11.3 Memory Block Properties

Item	Description
name	Name of the memory block.
type	RAM, ROM, FLASH, EEPROM or I/O
start	Start address of the memory block
end	End address of the memory block
wait state	Time used for reading or writing a specific number of bytes
bus width	Width of the bus that accesses the memory
read access	Table that defines read-access details on Byte, Word, Word misaligned, Long, and Long misaligned
write access	Table that defines write-access details on Byte, Word, Word misaligned, Long, and Long misaligned
auto configure	Flag that directs automatic computation of read and write accesses
allow misaligned access	Flag that allows Word misaligned and Long misaligned
block type	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)

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

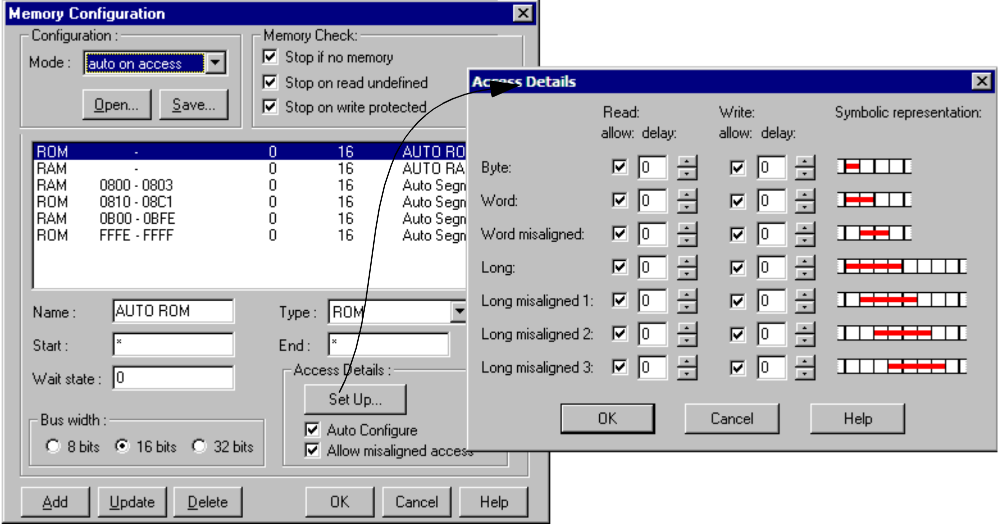
improper field value), the system generates a message box that informs you of the problem.

- **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 11.7 on page 268 shows the **Access Details** dialog box, which lets you change read and write access values for seven types.

Figure 11.7 Memory Configuration Dialog Box - Access Details Dialog Box



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.
- To abandon your changes, click the **Cancel** button. The **Access Details** dialog box disappears; the system discards your changes.
- To bring up appropriate help information, click the **Help** button.

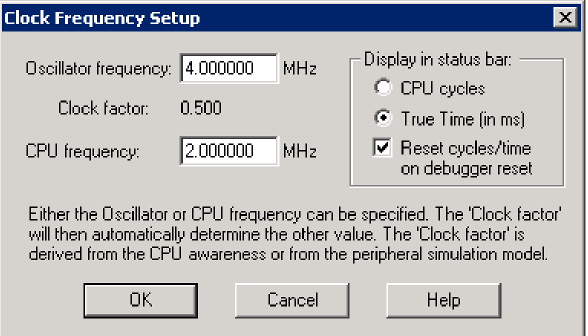
Output

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 11.8 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.

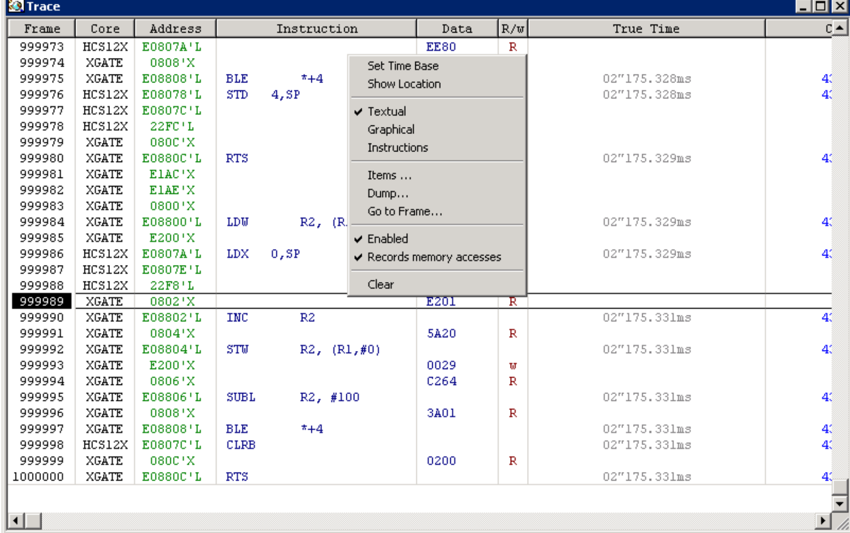
Unchecking "reset cycles/time..." will simply make the debugger cumulate cycles/time other CPU reset. The true time unit is the microsecond. The “TRUETIME” debugger command line command gives the time as a number in microseconds. The “OSCFREQUENCY” variable displays/sets the oscillator frequency.

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 11.9 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.

Figure 11.10 Bus Trace Data Access Symbolic Information

Trace

Data	R/w	True Time	Cycles	Symbol information
EE80	R			
3A01	R			
		02"175.328ms	4350657	SCI0Handler() @ xgate.cxgate
		02"175.328ms	4350657	Fibonacci() @ main.c
C720	R			
0002	w			
0200	R			
		02"175.329ms	4350659	SCI0Handler() @ xgate.cxgate
0800	R			
E200	R			
4A20	R			
		02"175.329ms	4350659	SCI0Handler() @ xgate.cxgate
0028	R			
		02"175.329ms	4350659	Fibonacci() @ main.c
0C18	R			
0007	R			
E201	R			
		02"175.331ms	4350662	SCI0Handler() @ xgate.cxgate
5A20	R			
		02"175.331ms	4350662	SCI0Handler() @ xgate.cxgate
0029	w			
C264	R			
		02"175.331ms	4350662	SCI0Handler() @ xgate.cxgate
3A01	R			
		02"175.331ms	4350662	SCI0Handler() @ xgate.cxgate
		02"175.331ms	4350662	Fibonacci() @ main.c
0200	R			
		02"175.331ms	4350663	SCI0Handler() @ xgate.cxgate

Full Chip Simulation Warnings

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
FCS Warning (ID 12): reading from unimplemented register at pc = 0x400a'L.
Value: 0x0, Memory Address: 0x106. FLASH CONTROL module not implemented
FCS Warning (ID 12): reading from unimplemented register at pc = 0x400a'L.
Value: 0x0, Memory Address: 0x106. FLASH CONTROL module not implemented
FCS Warning (ID 12): reading from unimplemented register at pc = 0x400a'L.
Value: 0x0, Memory Address: 0x106. FLASH CONTROL module not implemented
STOPPING
HALTED
```

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

```
WARNING_SETUP <HALT|CLMSG|MSGBOX|NONE|STATUS>
```

- **WARNING_SETUP STATUS**: displays the current warning setup status.

Example:

```
in>warning_setup status
 WARNING_SETUP STATUS: CLMSG
```

- WARNING_SETUP HALT: The FCS simply stops/halts the debugger when a warning message occurs.

Example:

```
in>warning_setup none
in>warning_setup halt
in>warning_setup status
 WARNING_SETUP STATUS: HALT
```

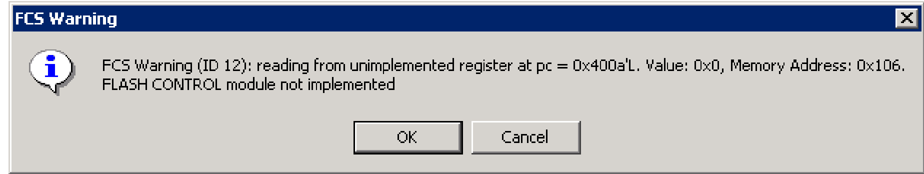
- **WARNING_SETUP CLMSG**: Warning messages are displayed in the Command window (debugger default).

Example:

```
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 11.11 FCS Warning Message Box



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.
```

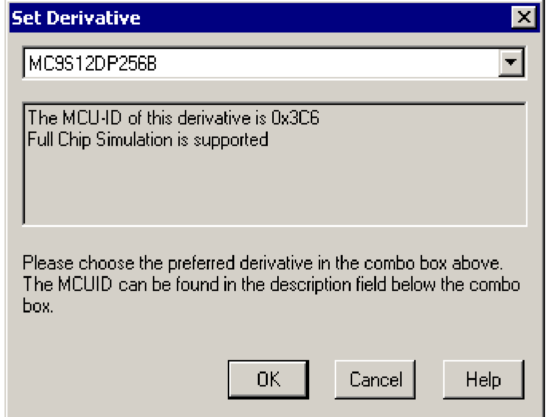
FCS and Silicon On-chip Peripherals Simulation

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 on page 276 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 simulated 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 11.12 '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 on page 260.

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 11.4 Supported Derivatives

Derivative Name	Modules
MC9S12A32	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290
MC9S12A64	Analog to Digital Converter on page 288 J1850 Bus (BLCD) on page 285 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290

Table 11.4 Supported Derivatives (*continued*)

Derivative Name	Modules
MC9S12C32	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Flash (FTS) on page 290 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Timer Module (TIM) on page 301 Voltage Regulator (VREG) on page 290
MC9S12D32	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290

Table 11.4 Supported Derivatives (*continued*)

Derivative Name	Modules
MC9S12D64	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290
MC9S12DB128A	Analog to Digital Converter on page 288 Byteflight (BF) on page 285 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290

Table 11.4 Supported Derivatives (*continued*)

Derivative Name	Modules
MC9S12DB128B	Analog to Digital Converter on page 288 Byteflight (BF) on page 285 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290
MC9S12DG128B	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290

Table 11.4 Supported Derivatives (*continued*)

Derivative Name	Modules
MC9S12DG256B	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290
MC9S12DJ128B	Analog to Digital Converter on page 288 J1850 Bus (BLCD) on page 285 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290

Table 11.4 Supported Derivatives (*continued*)

Derivative Name	Modules
MC9S12DJ256B	Analog to Digital Converter on page 288 J1850 Bus (BLCD) on page 285 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290
MC9S12DJ64	Analog to Digital Converter on page 288 J1850 Bus (BLCD) on page 285 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290

Table 11.4 Supported Derivatives (*continued*)

Derivative Name	Modules
MC9S12DP256B	Analog to Digital Converter on page 288 J1850 Bus (BLCD) on page 285 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290
MC9S12DP512	Analog to Digital Converter on page 288 J1850 Bus (BLCD) on page 285 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290

Table 11.4 Supported Derivatives (*continued*)

Derivative Name	Modules
MC9S12DT128B	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290
MC9S12DT256B	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Multiplexed External Bus Interface (MEBI) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Pulse Width Modulator (PWM) on page 299 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290

Table 11.4 Supported Derivatives (*continued*)

Derivative Name	Modules
MC9S12XDP512	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Debug Module (DBG) on page 291 External Bus Interface (EBI) on page 292 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Module Mapping Control (MMC) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Periodic Interrupt Timer (PIT) on page 299 Pulse Width Modulator (PWM) on page 299 S12X_INT on page 291 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290 XGATE on page 291

Table 11.4 Supported Derivatives (*continued*)

Derivative Name	Modules
MC9S12XDT512	Analog to Digital Converter on page 288 Clock and Reset Generator (CRG) on page 294 Debug Module (DBG) on page 291 External Bus Interface (EBI) on page 292 Enhanced Capture Timer (ECT) on page 296 EEPROM (EETS) on page 290 Flash (FTS) on page 290 Inter-IC Bus (IIC) on page 286 Module Mapping Control (MMC) on page 293 Motorola Scalable CAN (MSCAN) on page 285 Port Integration Module (PIM) on page 294 Periodic Interrupt Timer (PIT) on page 299 Pulse Width Modulator (PWM) on page 299 S12X_INT on page 291 Serial Communication Interface on page 286 Serial Peripheral Interface on page 288 Voltage Regulator (VREG) on page 290 XGATE on page 291

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.

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

SCIInput

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 SCIInputH register. A read access to SCIInput has no specified meaning.

Bit 7..0 character send to the SCI.

SCIInputH

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 SCIInput register. A read access to SCIInputH 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 a alias for the SCIInput 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 a 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:

SPICR1 (SPI Control Register 1)

SPIE, SPE, MSTR, CPOL, CPHA and LSBFE are simulated.

SPICR2 (SPI Control Register 2)

SPISWAI and SPC0 are simulated.

SPIBR (SPI Baud Rate Register)

SPPR2, SPPR1, SPPR0, SPR2, SPR1 and SPR0 are simulated.

SPISR (SPI Status Register)

SPIF, SPTEF and MODF are simulated.

SPIDR (SPI Data Register)

Bit 7..0 are simulated.

Not-memory-mapped Registers

SPIValue

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_CFADDR (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 a 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 (PLLCLK) = 2 OSCCLK ? (SYNR + 1)/(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 theses event 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, MCEN, 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) on page 296.

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 11.5 on page 302 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 11.5 MC68HC12A4 Register Block

Register Name	Register Address	Initial Value	Remarks
INITRG	0x0011	0x00	

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 11.6 on page 303 includes restrictions relative to special registers and single bits of registers.

LIM Simulated Registers

Table 11.6 on page 303 shows the LIM Simulated Registers.

Table 11.6 LIM Simulated Registers

Register Name	Register Address	Initial Value	Remarks
CLKCTL	0x0047	0x00	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.
RTICTL	0x0014	0x00	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.
RTIFLG	0x0015	0x00	
COPCTL	0x0016	0x0F	CME, FCME, FCM: The Full Chip Simulation does not support these COPCTL bits; writing to these bits has no effect.
COPRST	0x0017	0x00	
INTCR	0x001E	0x60	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.
HPRIO	0x001F	0xF2	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.

Standard Timer Module (TIM)

All functions of the timer module TIM are simulated.

General restrictions:

- The HPRIO register [$001F] 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 11.7 on page 304.

TIM Simulated Registers

Table 11.7 on page 304 shows all TIM Simulated Registers

Table 11.7 TIM Simulated Registers

Register Name	Register Address	Initial Value	Remarks
TIOS	0x0080	0x00	
CFORC	0x0081	0x00	
OC7M	0x0082	0x00	
OC7D	0x0083	0x00	
TCNT_H	0x0084	0x00	
TCNT_L	0x0085	0x00	
TSCR	0x0086	0x00	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.
TQCR	0x0087	0x00	
TCTL1	0x0088	0x00	
TCTL2	0x0089	0x00	
TCTL3	0x008A	0x00	
TCTL4	0x008B	0x00	
TMSK1	0x008C	0x00	

Table 11.7 TIM Simulated Registers

Register Name	Register Address	Initial Value	Remarks
TMSK2	0x008D	0x30	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.
TFLG1	0x008E	0x00	
TFLG2	0x008F	0x00	
TC0_H	0x0090	0x00	
TC0_L	0x0091	0x00	
TC1_H	0x0092	0x00	
TC1_L	0x0093	0x00	
TC2_H	0x0094	0x00	
TC2_L	0x0095	0x00	
TC3_H	0x0096	0x00	
TC3_L	0x0097	0x00	
TC4_H	0x0098	0x00	
TC4_L	0x0099	0x00	
TC5_H	0x009A	0x00	
TC5_L	0x009B	0x00	
TC6_H	0x009C	0x00	
TC6_L	0x009D	0x00	
TC7_H	0x009E	0x00	
TC7_L	0x009F	0x00	
PACTL	0x00A0	0x00	

Table 11.7 TIM Simulated Registers

Register Name	Register Address	Initial Value	Remarks
PAFLG	0x00A1	0x00	
PACNT_H	0x00A2	0x00	
PACNT_L	0x00A3	0x00	
TIMTST	0x00AD	0x00	TCBYP, PCBYP: The Full Chip Simulation does not support these TIMTST bits; writing to them has no effect. (These bits have meaning only for chip testing in special mode.)
PORTT	0x00AE	0x00	
DDRT	0x00AF	0x00	

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 11.8 on page 306 shows the SCI supported features.

Table 11.8 SCI Supported Features

Abbr	Full Name	Implemented Meaning
	Baud Rate Control	
SBRx	Baud Rate	Bit transmittal follows current baud rate settings
BTST	Reserved for internal tests	Ignored
BSPL	Reserved for internal tests	Ignored
BRLD	Reserved for internal tests	Ignored
	Control Register	

Table 11.8 SCI Supported Features

Abbr	Full Name	Implemented Meaning
LOOP	LOOP Mode	The LOOP mode determines SCI connection to the outer world. As this SCI is simulated, there is no connection to simulate.
WOMS	Wired Or Mode	Special feature of LOOP mode, not simulated
RSRC	Receiver Source	Special feature of LOOP mode, not simulated
M Mode	8 or 9 data bits	Supported (different timing, 9th bit)
WAKE	Wakeup by Address Mark/ Idle	Not supported
ILT	Idle Line Type	Considered in the Idle Line Detection
PE	Parity Enabled	Not simulated
PT	Parity Type	Not simulated
TIE	Transmit Interrupt Enable	Supported
TCIE	Transmit Complete Interrupt Enable	Supported
RIE	Receive Interrupt Enable	Supported
ILIE	Idle Line Interrupt Enable	Supported
TE	Transmitter Enable	Transmission process stops if this bit is clear
RE	Receiver Enable	Receive process stops if this bit is clear. As the input register is not part of the simulation, it still receives stimuli.
RWU	Receiver Wake Up Control	Not supported

Table 11.8 SCI Supported Features

Abbr	Full Name	Implemented Meaning
SBK	Send Break	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.
	Status Registers	
TDRE	Transmit Data Register Empty Flag	The system sets this flag upon the move of the value to be transmitted from the transmit data register to the serial shift register.
TC	Transmit Complete Flag	The system sets this flag if the transmission of one value ends, but no other value is yet in the transmit data register.
RDRF	Receive Data Register Full Flag	The system sets this flag upon the complete read of a value and the clearing of RDRF.
IDLE	Idle Line Detection Flag	The system sets this flag after a period without any input as stated in [3]. The system considers the ILT flag.
OR	Overrun Error Flag	The system sets this flag if the receipt of value ends, but the processor has not yet read the value.
NF	Noise Error Flag	Not supported, as no physical transmission takes place.
FE	Framing Error Flag	Not supported, as no physical transmission takes place.
PF	Parity Error Flag	Not supported, as no physical transmission takes place.
RAF	Receiver Active Flag	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.
	Data Register	
R8	Receive Bit 8	Supported

Table 11.8 SCI Supported Features

Abbr	Full Name	Implemented Meaning
T8	Transmit Bit 8	Supported
Rx/Tx	Receive/ Transmit Bit x	Supported, with autoclear feature

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 11.9 on page 309 lists. Other modules can subscribe to these registers to receive the sent values.

Table 11.9 Input, Output, Serial Output Registers

Name	Meaning	Comment
Input	Adds a value to be received. The system takes the 9th bit from the last value written to InputH. Read has no specified meaning	
InputH	9th Input bit; must be written before Input. Read has no specified meaning	
Output	Contains the last value sent. A notification is sent every time a new value is written. Write has no specified meaning	
OutputH	9th Output bit. Must be read immediately after Output. Write has no specified meaning	

Table 11.9 Input, Output, Serial Output Registers

Name	Meaning	Comment
SerialInput	Alias for Input for SCI 0; connects SCI 0 to terminal window. Only supports 8 bits.	Only available in SCI 0.
SerialOutput	Alias for Output for SCI 0; connects SCI 0 to terminal window. Only support 8 bits.	Only available in SCI 0

Serial Peripheral Interface (SPI)

Table 11.10 on page 310 describes the SPI interface.

Table 11.10 SPI interface

Abbr.	Full Name	Implemented Meaning
	Control Register 1	
SPIE	Interrupt Enable	Implemented
SPE	System Enable	If set, the Full Chip Simulation supports SPI functions
SWOM	Port S Wired-OR Mode	Not simulated, as no physical transmission takes place.
MSTR	Master Slave Mode Select	Master or Slave mode select
CPOL	Clock Polarity	Not simulated, as no physical transmission takes place.
CPHA	Clock Phase	Not simulated, as no physical transmission takes place.
SSOE	Slave Select Output Enable	Not simulated, as no physical transmission takes place.
LSBF	LSB First Enable	Not simulated, as no physical transmission takes place.
	Control Register 2	
PUPS	Pull Up Port S Enable	Not simulated, as no physical transmission takes place.

Table 11.10 SPI interface

Abbr.	Full Name	Implemented Meaning
RDS	Reduce Drive of Port S	Not simulated, as no physical transmission takes place.
SPC0	Serial Pin Control 0	Selects Normal or Bidirectional transmission mode
	Baud Rate Register	
SPRx	Baud Rate Register	Baud rate of the SPI transmission
	Status Register	
SPIF	Interrupt Request	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.
WCOL	Write Collision Status Register	System sets this flag upon the writing of new data to the Data Register, during a serial data transfer.
MODF	Mode Error Interrupt Status Flag	Not simulated, as no physical transmission takes place.
	Data Register	
SP0DR		8-bit Data Register for SPI data.
	Port S	
PORTS	Port S Data Register	Not simulated, as no physical transmission takes place.
	Data Direction Register	
DDRSx	Data Direction for Port S Bit x	Direction of Data. Only bits 4 and 5 have any effect.

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 11.11 on page 312 defines the Key Wakeups.

Table 11.11 Key Wakeups

Abbr.	Full Name	Implemented Meaning
	Key Wakeups Registers	
PORTD	Port D Register	Implemented
DDRD	Port D Data Direction Register	Implemented
KWIED	Port D Interrupt Enable Register	Implemented
KWIFD	Port D Flag Register	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.
PORTH	Port H Register	Implemented
DDRH	Port H Data Direction Register	Implemented
KWIEH	Port H Interrupt Enable Register	Implemented
KWIFH	Port H Flag Register	A falling edge on the associated pin sets each flag, provided that the corresponding DDRH Register bit is reset. To clear the flag, write one to the corresponding bit of the KWIFH register.
PORTJ	Port J Register	Implemented
DDRJ	Port J Data Direction Register	Implemented
KWIEJ	Port J Interrupt Enable Register	Implemented
KWIFJ	Port J Flag Register	A falling edge on the associated pin sets each flag, provided that the corresponding DDRJ Register bit is reset. To clear the flag, write one to the corresponding bit of the KWIFJ register.
KPOLJ	Port J Polarity Register	Implemented

Table 11.11 Key Wakeups

Abbr.	Full Name	Implemented Meaning
PUPSJ	Port J Pull-Up/ Pulldown Select Register	Not simulated, as there are no physical outputs.
PULEJ	Port J Pull-Up/ Pulldown Enable Register	Not simulated, as there are no physical outputs.

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 11.12 on page 313 describes the Memory-Mapped Page Registers.

Table 11.12 Memory Mapped Page Registers

Abbr.	Full Name	Implemented Meaning
	Port F Register	
CS	Chip Select / General Purpose IO (Bit 0-6)	Not implemented, as there are no physical outputs.
	Port G Register	
ADDR	Memory Expansion / General Purpose IO (Bit 0-5)	Not implemented, as there are no physical outputs.
	Port F Data Direction Register	
DDRF	Data Direction Register Port F (Bit 0-6)	Not implemented, as there are no physical outputs.
	Port G Data Direction Register	

Table 11.12 Memory Mapped Page Registers

Abbr.	Full Name	Implemented Meaning
DDRG	Data Direction Register Port G (Bit 0-5)	Not implemented, as there are no physical outputs.
	Data Page Register	
PDA	Data Page	Selects the data page
	Program Page Register	
PPA	Program Page	Selects the program page
	Extra Page Register	
PEA	Extra Page	Selects the extra page
	Window Defition Register	
DWEN	Data Window Enable	Enables paging of data space
PWEN	Program Window Enable	Enables paging of program space
EWEN	Extra Window Enable	Enables paging of extra space
	Memory Expansion Assignment Register	
A21E-A16E	Memory Expansion Assignment/ General Purpose IO	Not simulated, as there are no physical outputs.

Current Non-Supported Modules

Non-Supported Modules

- A/D Converter Device

Register Block Address Map

Table 11.13 shows the mapping of the Register Block Address.

Table 11.13 Register Block Address Map

Register Block Address	Description	Remarks
$0000-$000D	Port access	Not simulated: memory configuration controls correct timing of memory accesses
$000E-$000F	Reserved	
$0010	Internal RAM mapping	Register not simulated. Use the memory configuration dialog box to specify simulated memory configuration.
0x0011	Register Block mapping	Completely simulated
$0012-$0013	ROM/ EEPROM mapping	Registers not simulated. Use the memory configuration dialog box to specify simulated memory configuration.
$0014-$0017	Clock Function Control	Completely simulated
$001E-$001F	Interrupt Control & Highest Priority I Interrupt	Completely simulated
$0020-$002E	Key Wakeup Control	Completely simulated
$002F	Reserved	
$0030-$0033	Port Registers	Currently not simulated
$0034-$0038	PAGE & memory configuration Registers	Page Registers are simulated
$0039-$003B	Reserved	
$003C-$003F	Chip select control registers	Currently not simulated

Table 11.13 Register Block Address Map

Register Block Address	Description	Remarks
$0040-$0043	PLL divider registers	Currently not simulated
$0044-$0046	reserved	
$0047	Clock Control Register	Completely simulated
$0048-$005F	Reserved	
$0060-$0069	Analog to Digital Converter	Currently not simulated
$006A-$006E	Reserved	
$006F	PORTAD	Currently not simulated
$0070-$007F	ADRxH/ reserved	Currently not simulated
$0080-$009F	Timer Registers	Completely simulated
$00A0-$00A3	Pulse Accumulator Control Registers	Completely simulated
$00A4-$00AC	Reserved	
$00AD-$00AF	Timer Test, Timer Port	Completely simulated
$00B0-$00BF	reserved	
$00C0-$00C7	SCI0	Completely simulated
$00C8-$00CF	SCI1	Completely simulated
$00D0-$00D3	SPI	Completely simulated
$00D4	Reserved	
$00D5-$00D7	SPI, PORTS	Completely simulated

Table 11.13 Register Block Address Map

Register Block Address	Description	Remarks
$00D8-$00EF	Reserved	
$00F0-$00F3	EEPROM Control	Currently not simulated
$00F3-$01FF	Reserved	

Related Documentation

The following documents are available from Motorola:

- MOTOROLA SEMICONDUCTOR TECHNICAL DATA, MC68HC812A4, Technical Summary 16-Bit Microcontroller 1996
- CPU12 Reference Manual, Preliminary draft 15 July 95, AMCU Division, 1995, MOTOROLA

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 transparen, behavin 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.

```
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”,¶meter, 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.

Capture / Compare Timer

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.

bit 7..1 unused

bit 0 ninth bit send from the SCI

SerialInput:

This non-memory-mapped register is an alias for the SCIInput 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 SCIOutput 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.

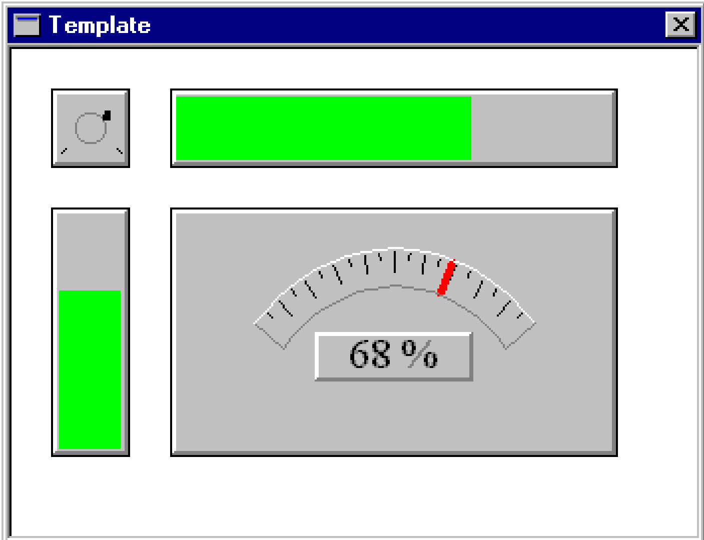
WARNING! The following visualization components can only be used with the Full Chip Simulation connection.

Analog Meter Component

The Analog Meter window shown in Figure 11.13 on page 324 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 11.13 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.

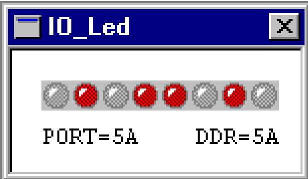
Analog Meter Menu

The Analog Meter does not have a menu.

IO_LED Component

The IO_LED window shown in Figure 11.14 on page 325 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 11.14 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.

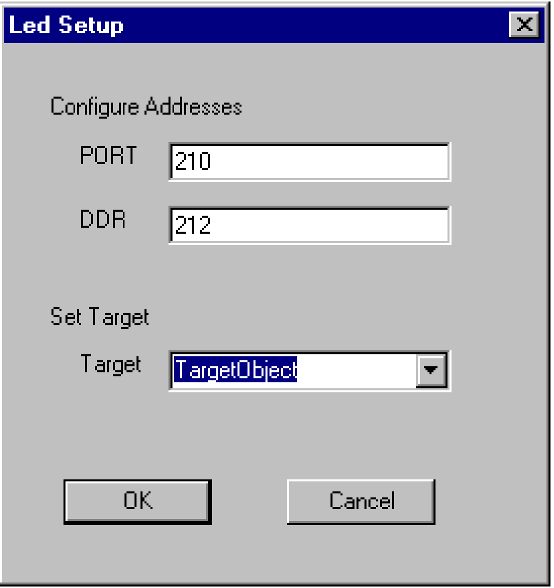
If you want to change the color of the leds, you must change the value of the byte at the PORT address in the Memory Component window.

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 11.15 on page 326 , which allows you to specify the PORT and DDR addresses.

Figure 11.15 ILed Setup Dialog Box



Associated Popup Menu

Identical to menu.

Drag Out:

Nothing can be drag out.

Drop Into:

Nothing can be dropped into.

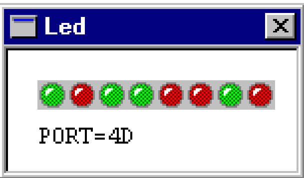
Demo Version Limitations

No limitation

LED Component

The LED window shown in Figure 11.16 on page 327 is a visual utility that displays an arbitrary 8 bit value by means of an LED bar.

Figure 11.16 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.

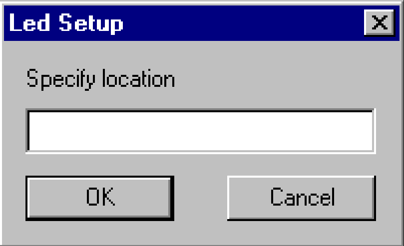
LED Operations

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 11.17 on page 327.

Figure 11.17 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 11.18 on page 329 is an input utility that provides a graphical keyboard pad that allows you to interactively modify the value of a memory cell.

Figure 11.18 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

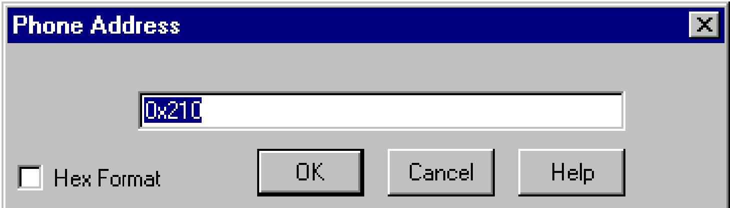
Click a phone key and the matching ASCII character of the label on the key is stored at the configured memory cell.

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 11.19 on page 330. 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 11.19 Phone Address Dialog Box



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.

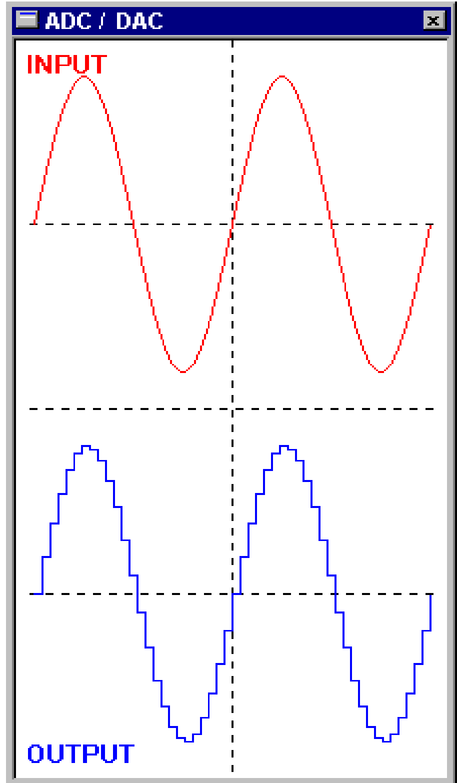
Demo Version Limitations

No limitation

ADC/DAC Component

The ADC_DAC window, shown in Figure 11.20 on page 331 consists of a Digital to Analog and an Analog to Digital converter.

Figure 11.20 ADC/DAC Window

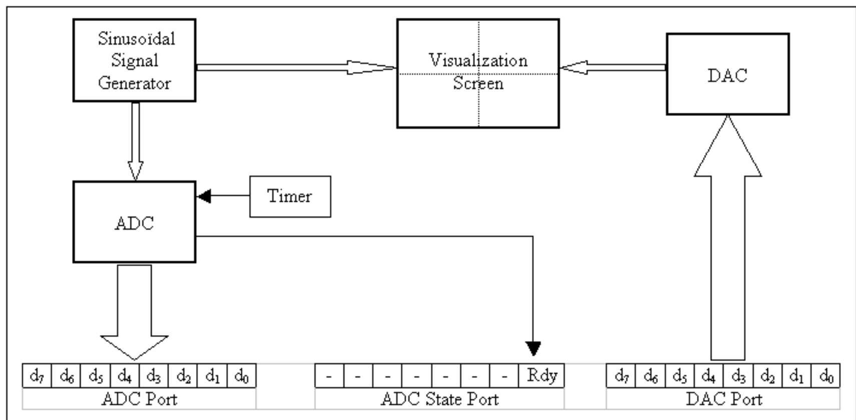


Description

The ADC/DAC component is made of 4 units as shown in Figure 11.21 on page 332:

- A signal generator
- An analog to digital converter (ADC)
- A digital to analog converter (DAC)
- A visualization unit

Figure 11.21 Internal Converter Module Organization and Coupler Connections.



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 11.20 on page 331, 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 11.21 on page 332, 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 11.22 on page 333 contains all functions associated with the Adc-Dac component. These entries are described in Table 11.14 on page 333.

Figure 11.22 ADC/DAC Menu

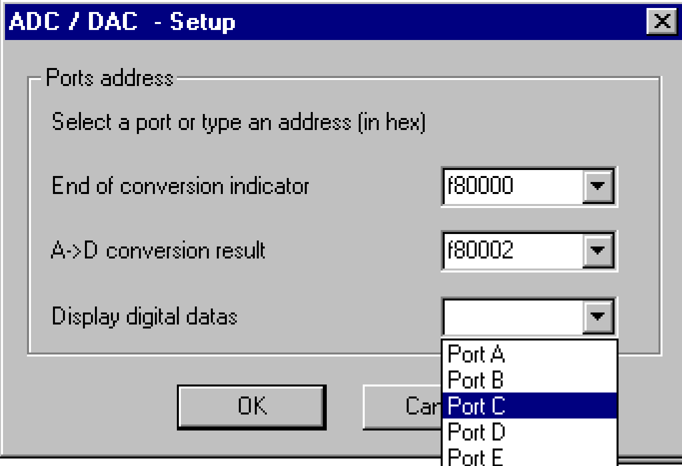
Table 11.14 ADC/DAC Menu Description

Menu Entry	Description
Setup	Opens the ADC/DAC - Setup dialog box, allowing you to set the port addresses.
Reset	This function erases the visualization screen and re-initializes the display properties.
Conversion Parameters	Opens the Conversion Parameters dialog box, allowing you to set the signal frequency
Start Conversion	Runs the conversion process
Display Properties	Opens the Display Properties dialog box allowing you to set the display properties

ADC/DAC - Setup Dialog Box

This dialog box shown in Figure 11.23 on page 334 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 on page 352 component.

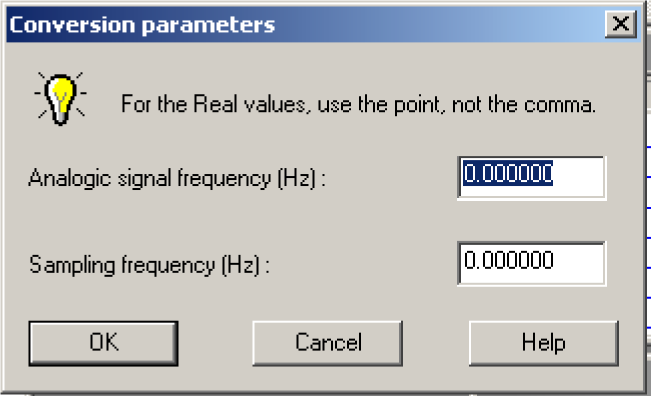
Figure 11.23 ADC/DAC - Setup Dialog Box



Conversion Parameters Dialog Box

This dialog box shown in Figure 11.24 on page 334 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 11.24 Conversion Parameters Dialog Box

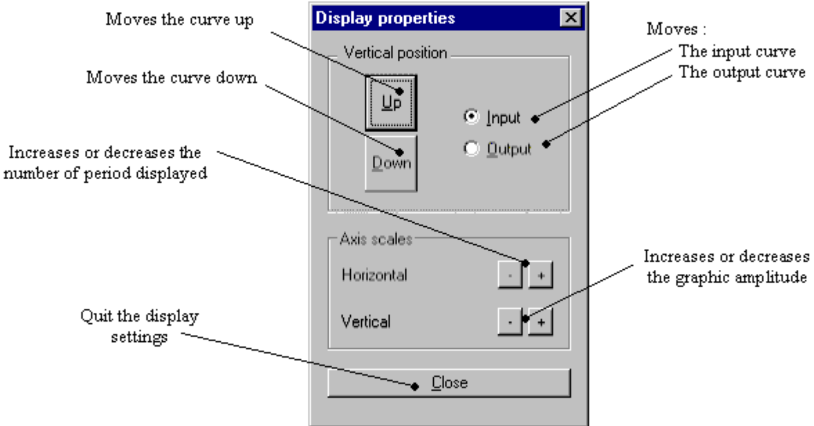


Now you can start the conversion with Start conversion menu entries.

Display Properties Dialog Box

This dialog box shown in Figure 11.25 on page 335 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 11.25 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.

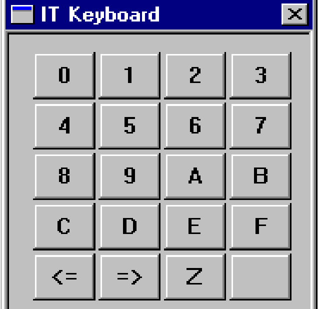
Drag Into:

Nothing can be dragged in.

IT_Keyboard Component

The IT_Keyboard window shown in Figure 11.26 on page 336 is a 20 key keyboard that generates an interruption when a key is pressed.

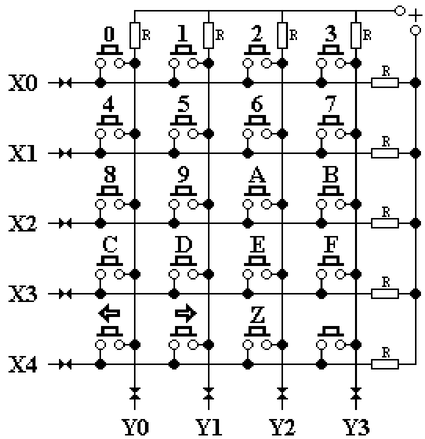
Figure 11.26 IT_Keyboard Window



The IT_Keyboard consists of a 20 key keyboard, as shown in Figure 11.26 on page 336. These 20 keys are positioned at the intersection of the five lines X0 to X4 and the 4 columns Y0 to Y3. The resistor R connected to the positive supply gives a logical level 1 when there is no connection (key not pressed). The activation of a line (or column) will give a logical level 0, and a key pressed on this line (or column) will place the column (or the line) corresponding on the low level. For example, if line X2 is activated, column Y3 will decrease from logical level 1 to logical level 0 when the « B » key is pressed.

An interruption is raised when an active key (line or column activated) is pressed.

Figure 11.27 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 11.28 on page 337 shows the IT_Keyboard menu and its entries are described in Table 11.15 on page 337.

Figure 11.28 IT_Keyboard menu

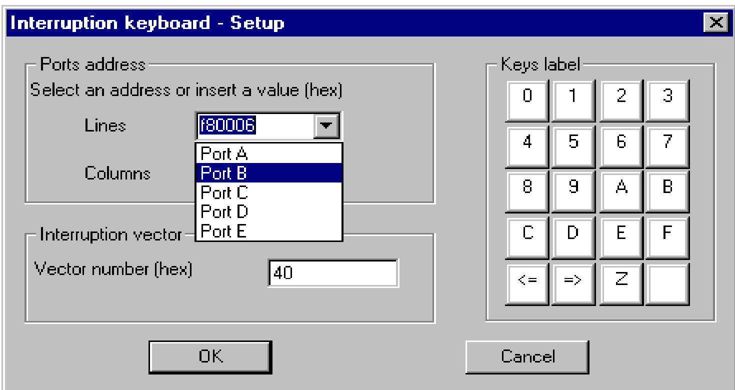
Table 11.15 IT_Keyboard Menu Description

Menu Entry	Description
Setup	Opens the Interrupt keyboard setup dialog.

Interruption Keyboard Setup

The Interrupton Keyboard Setup dialog box shown in Figure 11.29 on page 338 allows you to set the address of the lines port, the columns port and the number of the interruption vector.

Figure 11.29 Interruption Keyboard - Setup Dialog Box



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 on page 352.

The **Vector number filed** 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.

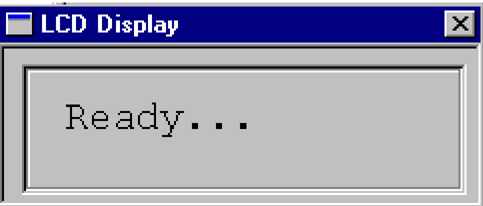
Demo Version Limitations

No limitations

LCD Component

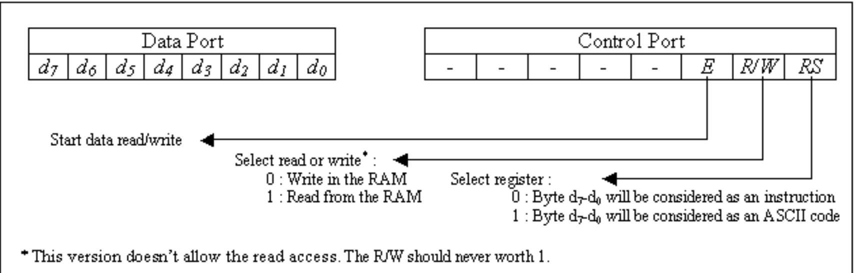
The LCD Display component message box shown in Figure 11.30 on page 339 is the LCD utility, which can display 1 or 2 lines of 16 characters and show or hide the position cursor.

Figure 11.30 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 11.31 on page 339. These ports communicate with the mainframe.

Figure 11.31 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.

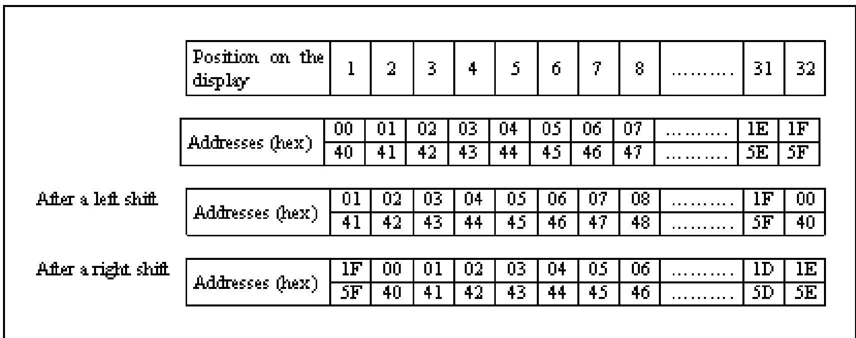
LCD Operation

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 11.32 on page 340 illustrates this arrangement.

Figure 11.32 .The DDRAM Controller



	Position on the display	1	2	3	4	5	6	7	8	………	31	32
	Addresses (hex)	00	01	02	03	04	05	06	07	………	1E	1F
		40	41	42	43	44	45	46	47	………	5E	5F
After a left shift	Addresses (hex)	01	02	03	04	05	06	07	08	………	1F	00
		41	42	43	44	45	46	47	48	………	5F	40
After a right shift	Addresses (hex)	1F	00	01	02	03	04	05	06	………	1D	1E
		5F	40	41	42	43	44	45	46	………	5D	5E

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 11.33 on page 341 lists the instructions available for the LCD component.

Figure 11.33 LCD DisplayComponent Instruction Listing

Instruction	Code								Description
	d_7	d_6	D_5	d_4	d_3	d_2	d_1	d_0	
Clear Display	0	0	0	0	0	0	0	1	Erases the display and put AC at 0.
Return Home	0	0	0	0	0	0	1	-	Puts the address 00h into AC and re-init the display.
Entry Mode Set	0	0	0	0	0	1	I/D	-	Fixes the moving direction of the cursor
Display On/Off Control	0	0	0	0	1	D	C	-	Lights on or off the display and shows or not the cursor.
Cursor or Display Shift	0	0	0	1	S/C	R/L	-	-	Moves the cursor and shifts the display.
Set DDRAM Address	1	a_6	A_5	a_4	a_3	a_2	a_1	a_0	Fixes the AC value.
Function Set	0	0	1	DL*	N	-	-	-	Fixes the data exchange width and the line number to display.

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-initialize 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.

- 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 11.34 on page 342 describes how to choose the moving direction.

Figure 11.34 Left Right Choice

S/C	R/L	
0	0	Moves the cursor to the left (decreases AC).
0	1	Moves the cursor to the right (increases AC).
1	0	Moves the full screen to the left. The cursor follows this move.
1	1	Moves the full screen to the right. The cursor follows this move.

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

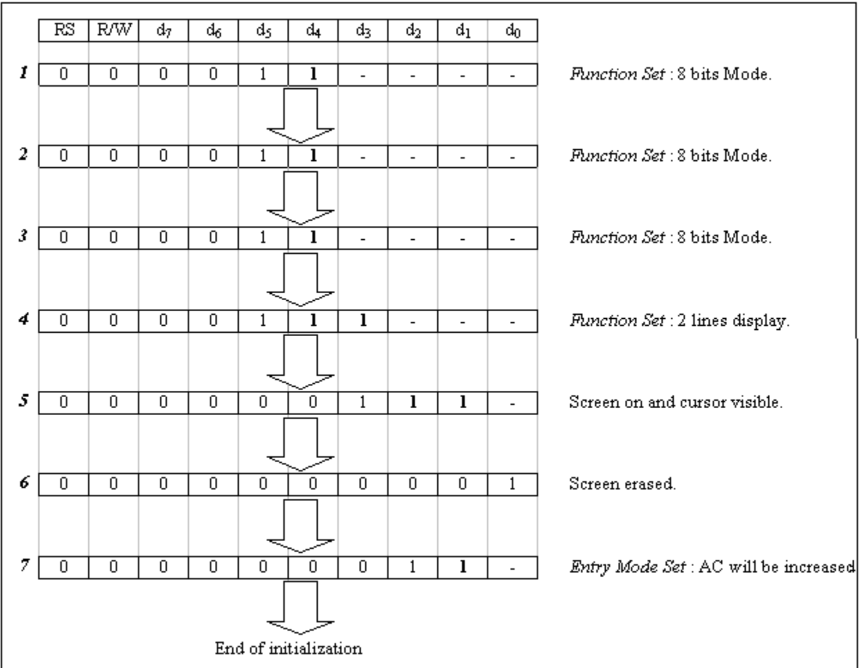
- If DL = 1, the data exchange is 8 bits wide.
- If N = 0, the display will take place on one line. If N = 1, the display will take place on two lines.

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 11.35 on page 343 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 11.35 The LCD Display Initialization



LCD Menus

Figure 11.36 on page 343 shows the LCD menu, which is identical to the popup menu. Its entry is described in Table 11.16 on page 343.

Figure 11.36 LCD Menu

The LCD menu contains the Setup function to launch the LCD Display dialog box.

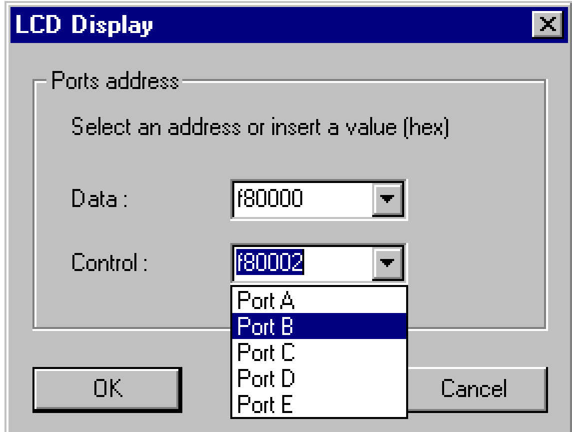
Table 11.16 LCD Display Menu Description

Menu Entry	Description
Setup	Opens the LCD Display dialog box (Setup)

LCD Display

The LCD Display dialog box shown in Figure 11.37 on page 344 allows you to set the address of the lines port and columns port.

Figure 11.37 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 on page 352.

Drag Out:

Nothing can be dragged out.

Drop Into:

Nothing can be dropped into the Lcd display Component.

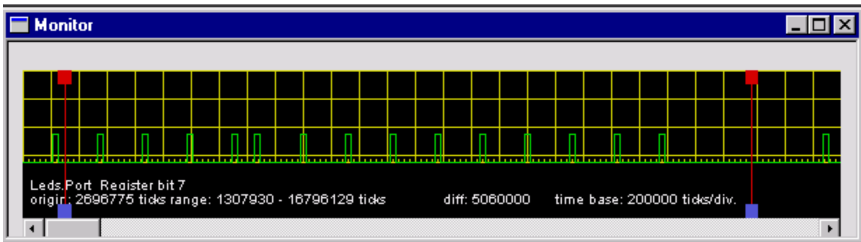
Demo Version Limitations

No limitations

Monitor Component

The Monitor window shown in Figure 11.38 on page 345 is a basis oscilloscope that can display the result of debugger objects.

Figure 11.38 Monitor Window



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 11.39 on page 345 shows the Monitor menu and its entries are described in Table 11.17 on page 345.

Figure 11.39 Monitor Menu

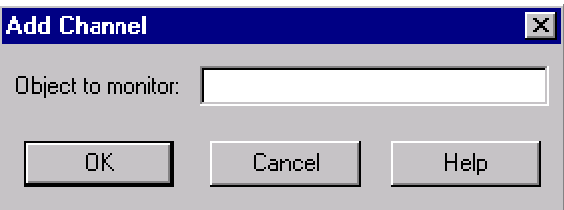
Table 11.17 Monitor Menu Description

Menu Entry	Description
Add Channel	Opens the dialog box to create a new Channel in the Monitor.
Delete Channel	Deletes the Selected Monitor Channel (click on it in the monitor view)
Show Control	Opens the Settings dialog box to change the time base.
Change Colors	Changes colors from the selected Channel.

Add Channel

The Add Channel dialog box shown in Figure 11.40 on page 346 allows you to create a new Channel in the monitor.

Figure 11.40 Add Channel Dialog Box



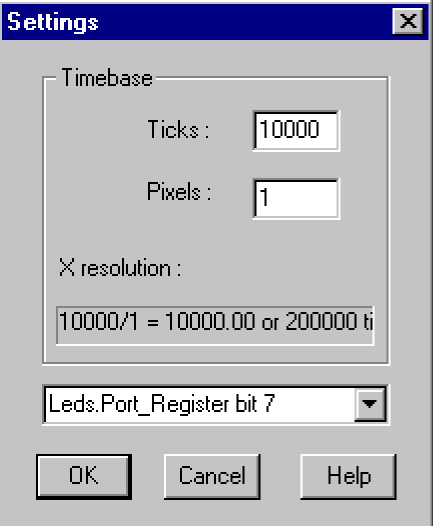
In the text area **Object to monitor**, enter the object name and bit e.g TIM12.PORTT bit 0 and click **OK** to validate or **Cancel** to exit.

Monitor Settings

The Monitor Settings dialog box shown in Figure 11.41 on page 347 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.

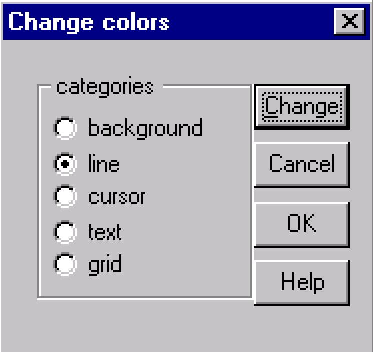
Figure 11.41 Settings Dialog Box



Change Colors

The Change Colors dialog sbox shown in Figure 11.42 on page 347 allows you to change the colors from the selected Channel.

Figure 11.42 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.

Drag Out:

Nothing can be dragged out.

Drop Into:

Nothing can be dropped in.

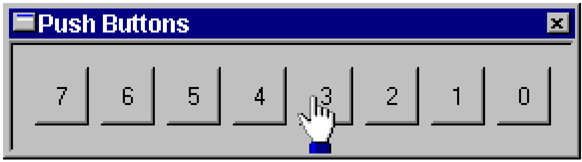
Demo Version Limitations

No limitations

Push Buttons Component

The Push Buttons window shown in Figure 11.43 on page 349 is a basis input device.

Figure 11.43 Push Buttons Window



Push Buttons Menu

Figure 11.44 on page 349 shows the Push Buttons menu and its entry is described in Table 11.18 on page 349.

Figure 11.44 Push Buttons Menu

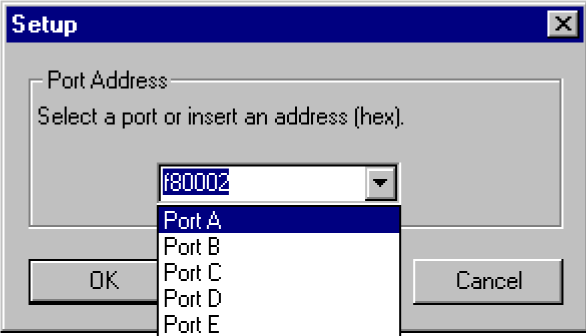
Table 11.18 Push Buttons Menu Description

Menu Entry	Description
Setup	Opens the Push Buttons Setup dialog box.

Push Buttons Setup

The Setup dialog box shown in Figure 11.45 on page 350 allows you to specify (in hexadecimal format) the port address or select the port in the list.

Figure 11.45 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 on page 352.

Use with LEDs Component

The Bytes sent to the LEDs component coming from the Push Button component are described in Figure 11.46 on page 350.

Figure 11.46 Push Buttons Input port

Push Buttons Input Port							
b7	b6	b5	b4	b3	b2	b1	b0
PB7	*PB6*	*PB5*	*PB4*	*PB3*	*PB2*	*PB1*	*PB0*

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.

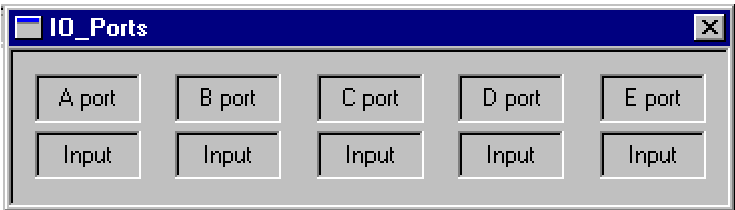
Demo Version Limitations

No limitations

Programmable IO_Ports Component

The Programmable IO_Ports window shown in Figure 11.47 on page 352 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 11.47 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 11.48 on page 352 shows the Programmable IO_Ports menu and its entry is described in Table 11.19 on page 352.

Figure 11.48 The Programmable IO_Ports Menu

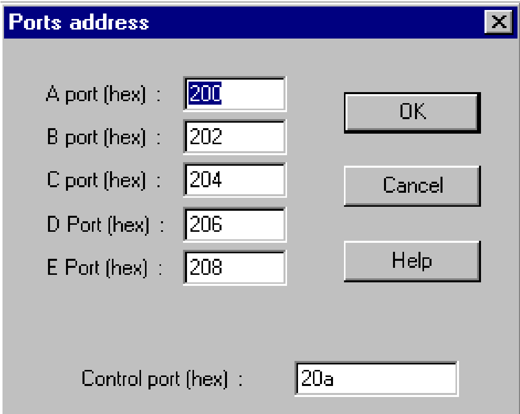
Table 11.19 Programmable IO_Ports Mmenu Description

Menu entry	Description
Setup	Opens the Programmable IO_Ports Port Address dialog.

Port Address

The Port Address dialog box shown in Figure 11.49 on page 353 allows you to set the port address and control port address.

Figure 11.49 Port Address Dialog Box (Setup)



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 11.50 on page 353.

Figure 11.50 Coupler Control Register

Control register								
Bits	b7	b6	b5	b4	b3	b2	b1	b0
Ports	-	-	-	E	D	C	B	A

Way	
Input	0
Output	1

Drag Out:

Nothing can be dragged out.

Drop Into:

Nothing can be dropped in.

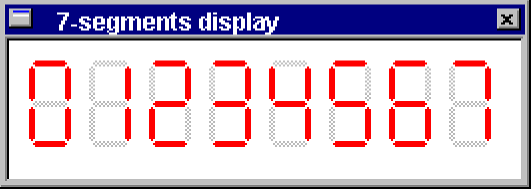
Demo Version Limitations

No limitations

7-Segments Display Component

The 7-Segments Display window shown in Figure 11.51 on page 354 consists of 8 "7-segment" display systems.

Figure 11.51 7-Segments Display Window

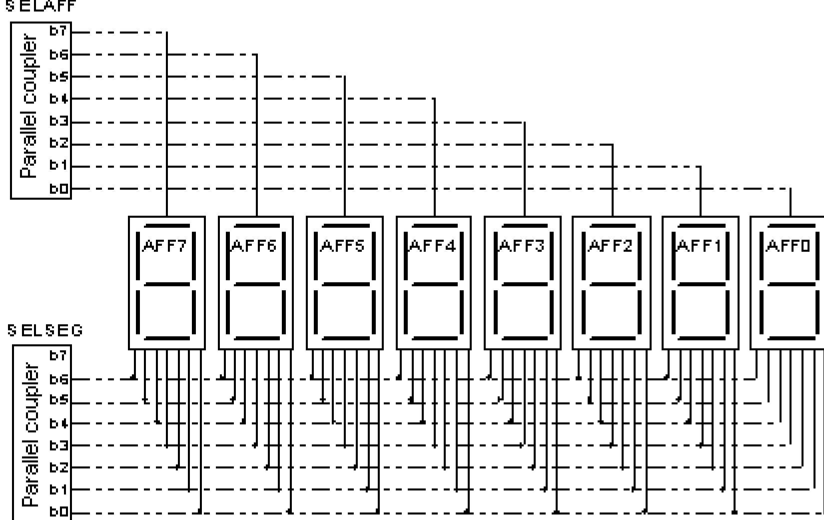


Operation of the Seven segments display component is based on the display scanning principle. Only one display can be activated simultaneously for the purpose of limiting consumption of the set.

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 11.52 on page 354.

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 11.52 7-Segments Display Component Constitution



7-Segments Display Menu

Figure 11.53 on page 355 shows the Seven segments display component menu and the menu entry is described in Table 11.20 on page 355.

Figure 11.53 7-Segments Display Menu

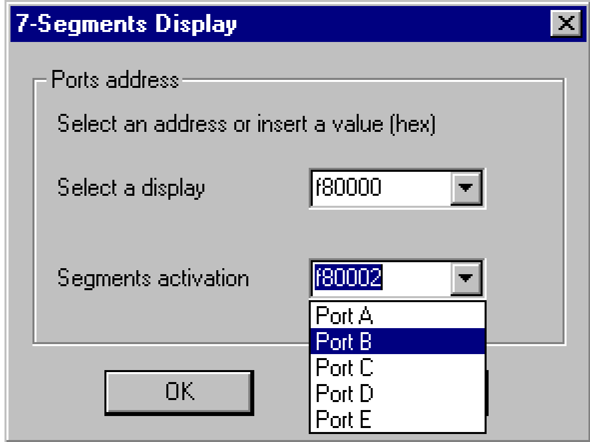
Table 11.20 7-Segments Display Menu Description

Menu Entry	Description
Setup	Opens the Seven segments display component setup dialog.

7-Segments Display Setup

The 7-Sements Display dialog box shown in Figure 11.54 on page 355 allows you to select the display and related value.

Figure 11.54 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 on page 352.

Control Bits Configuration

The 2 bytes sent to the 7 segments must be composed as shown in Figure 11.55 on page 356.

Figure 11.55 Seven Segments Display Control Bits

SELAFF Select of display							
b7	b6	b5	b4	b3	b2	b1	b0
Aff7	Aff6	Aff5	Aff4	Aff3	Aff2	Aff1	Aff0

SELSEG Select of segments							
b7	b6	b5	b4	B3	b2	b1	b0
-	g	F	e	d	c	b	a

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 on page 372 section describes in detail and with example how to take advantage of this component.

The Stimulation window shown in Figure 11.56 on page 357 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 11.56 Stimulation Window

```
Stimulation

def a = TargetObject.#1FA0;
def b = TargetObject.#1FB0;
def c = TargetObject.#1FC0.B[7:3];
10000
```

Stimulation Popup Menu

Figure 11.57 on page 357 shows functions associated with the Source component. Table 11.21 on page 357 describes these functions.

Figure 11.57 Stimulation Popup Menu

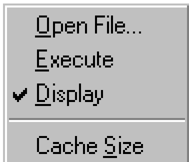


Table 11.21 Stimulation Popup Menu Description

Menu Entry	Description
Open File	Opens a dialog box to load a stimulation file.
Execute	Starts execution of the input file.

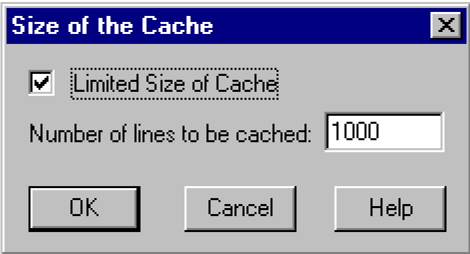
Table 11.21 Stimulation Popup Menu (*continued*)Description

Menu Entry	Description
Display	Switches display of stimulated file on or off.
Cache size	Opens the 'Size of the Cache' dialog box.

Cache Size

The Size of the Cache dialog box shown in Figure 11.58 on page 358, 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.

Figure 11.58 Size of the Cache Dialog Box



NOTE The bigger the cache size, the slower new lines are logged.

Example of a Stimulation File

Using an editor, open the file named `IO_VAR.TXT` located in the project directory. Listing 11.1 on page 358 is an example file.

Listing 11.1 Stimulation File Example

```
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 on page 372 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.

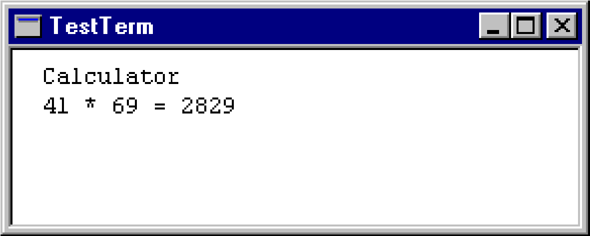
Demo Version Limitations

Only 15 interrupts and memory access will be generated.

TestTerm Component

The TestTerm window shown in Figure 11.59 on page 360 is a user-friendly terminal input/output. It provides a simple SCI (Serial Communication Interface) interface, which is "Full Chip Simulation independent".

Figure 11.59 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 11.22 on page 360.

Table 11.22 TestTerm Simulated Memory Mapped Registers

Register Name	Function	Register Address
BAUD	Baud Rate Control	0x0200
SCCR1	Serial Communication Control Register	0x0201
SCCR2	Serial Communication Control Register	0x0202
SCSR	Serial Communication Status Register	0x0203
SCDR	Serial Communication Data Register	0x0204

In the Serial Communication Status Register, the bits used are described in Table 11.23 on page 360.

Table 11.23 TestTerm Serial Communication Status Register

Bit Name (flag)	Function	Bit Mask Value
TDRE	Transmit Data Register Empty	0x80
RDRF	Receive Data Register Full	0x20

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 11.24 on page 361 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 11.24 Redirections and Associated Escape Sequences

Escape Sequence	Function
ESC “h” “1”	Output to Terminal window only.
ESC “h” “2” filename	Output to both Terminal window and file.
ESC “h” “3” filename	Output to file only.
ESC “h” “4”	Read from keyboard
ESC “h” “5” filename	Read input from file 'fileName'
ESC “h” “6” filename	Output to Terminal window and append to file
ESC “h” “7” filename	Append to file only

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 11.2 on page 362.

Listing 11.2 Term_Direct Source Code

```
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 11.3 on page 362 shows the functions defined in `termport.h` that can be called to access the TestTerm component:

Listing 11.3 Functions to Access the TestTerm Component

```
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 11.4 on page 363.

Listing 11.4 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 */
}
```

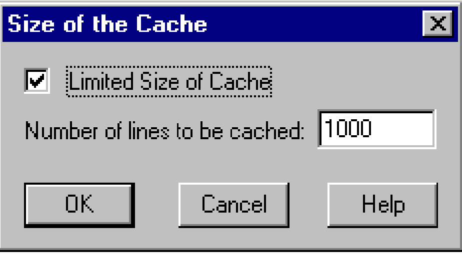
TestTerm Menu

The TestTerm component menu shown in Figure 11.60 on page 364 lets you set the Cache Size in lines of the TestTerm window in the dialog box shown in Figure 11.61 on page 364.

Figure 11.60 TestTerm Menu

Select **Cache Size** in the menu.

Figure 11.61 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 11.62 on page 365 can be used to simulate input and output. It can receive characters from several input devices and send them to other devices.

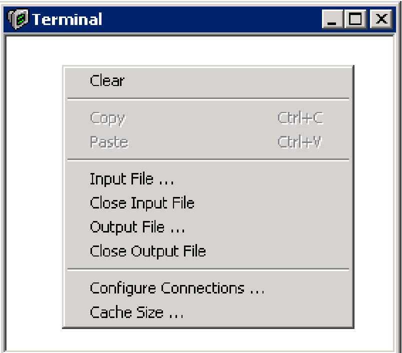
Figure 11.62 Terminal Window

```
Terminal
--------------------------------------------------
ZH_06 test (file: ZH_06.C)
Date: Oct 04 1996, 15:28:10
--------------------------------------------------
```

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 11.63 on page 365.

Figure 11.63 Terminal Menu and Popup Menu

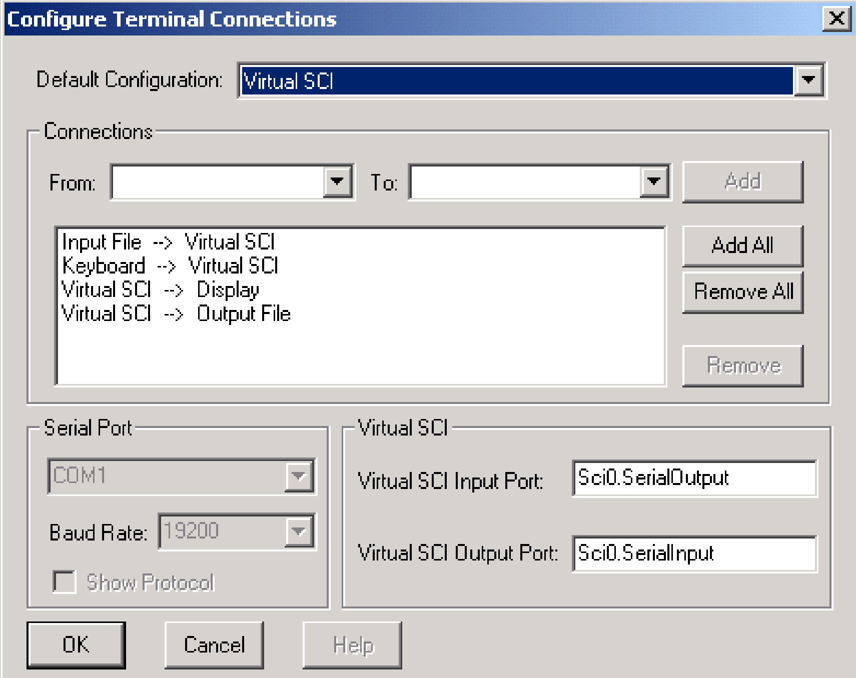


To open the popup menu just right click in the terminal window.

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 11.64 on page 366.

Figure 11.64 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 11.25 on page 367 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 11.25 Terminal File Control Commands

Escape Sequence	Function
ESC “h” “1”	Close output file.
ESC “h” “2” filename	Open output file.
ESC “h” “3” filename	Open output file and suppress output to terminal display.
ESC “h” “4”	Close input file

Table 11.25 Terminal File Control Commands (*continued*)

Escape Sequence	Function
ESC “h” “5” filename	Open input file.
ESC “h” “6” filename	Append to existing output file.
ESC “h” “7” filename	Append to existing output file and suppress output to terminal display.

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 11.5 on page 368,

Listing 11.5 TERM_Direct Source Code

```
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 "**OP_SetValue**" 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.

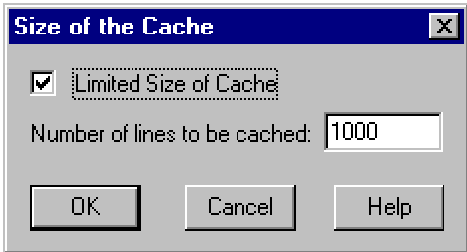
For implementations of this register with help of the "**IOBase**" class, the flag "**IOB_NotifyAnyChanges**" should be used. Otherwise only the first of two identical characters are received.

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 on page 366 on page 369for 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 11.65 on page 369.

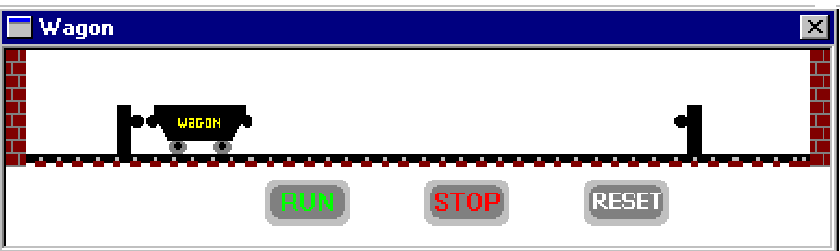
Figure 11.65 Size of the Cache Dialog Box



Wagon Component

The Wagon window shown in Figure 11.66 on page 370 simulates a tool machine wagon functionality.

Figure 11.66 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 11.67 on page 370 shows the Wagon menu and is described in Table 11.26 on page 370.

Figure 11.67 Wagon Menu

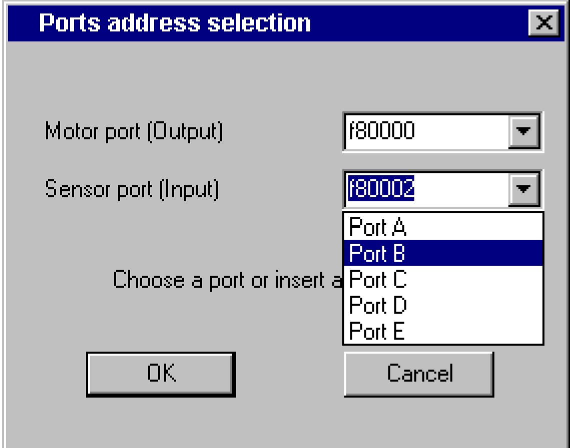
Table 11.26 Wagon Menu Description

Menu Entry	Description
Setup	Opens the Wagon setup dialog box shown in Figure 11.68 on page 371.

Wagon Setup

When you select Setup from the Wagon Menu, the Ports Address Selection dialog box appears. This is the Wagon component Setup window.

Figure 11.68 Ports Address Selection Dialog Box



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 on page 352.

Control Bits Configuration

The 2 bytes sent to the 7 segments must be composed as shown in Figure 11.69 on page 371.

Figure 11.69 Wagon Control Bits Description

Motor port								Sensor port							
b7	b6	b5	b4	b3	b2	b1	b0	b7	b6	b5	b4	b3	b2	b1	b0
l	-	-	-	-	-	-	*r*	*bl*	-	-	*st*	*stp*	-	-	*br*

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 on page 372
- Stimulation Input File Syntax on page 380

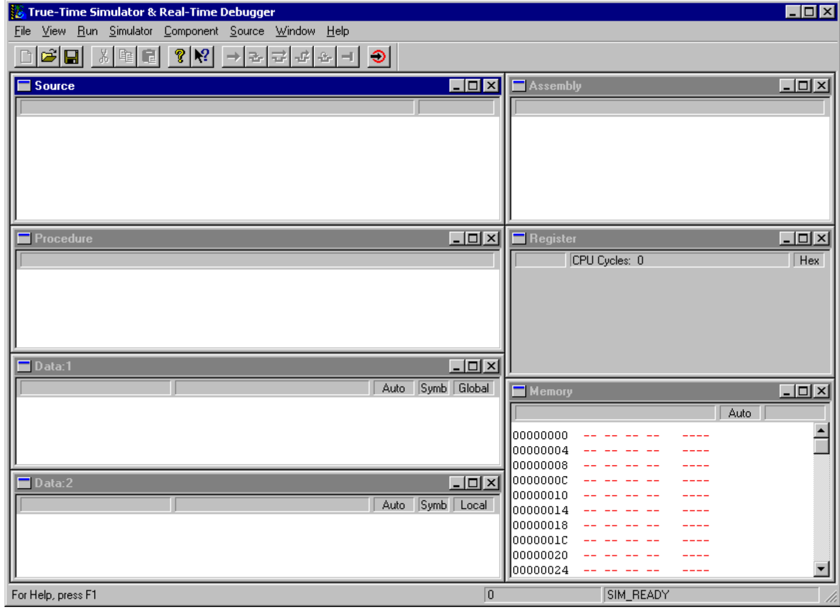
Stimulation Program Examples

Running an Example Program Without Stimulation

1. Run the debugger with the Full Chip Simulation connection.

The Main Window is shown in Figure 11.70 on page 372.

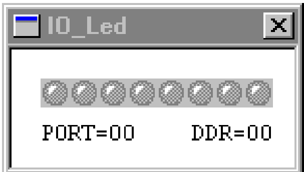
Figure 11.70 FCS I/O-Simulation Main Window



2. Choose Simulator > Set > Sim.
3. Choose Component > Open > Io_led.

The IO_Led component is shown in Figure 11.71 on page 373.

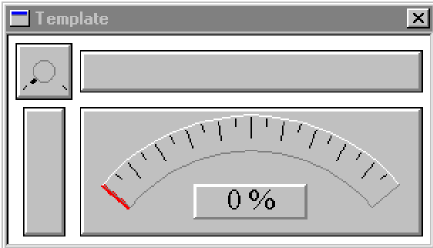
Figure 11.71 IO_Led Component Window



4. Choose Component > Open >Template.

The Template component is shown in Figure 11.72 on page 373.

Figure 11.72 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:

```
#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 **Hypertextparanum**, 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 11.6 IO_Show Procedure in io_demo.c

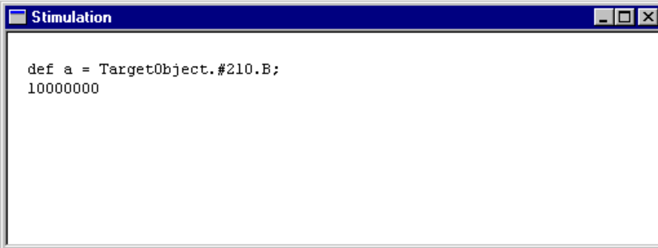
```
static void IO_Show(void) {
   for (;;) {                      // endless loop
      dir = 1;
      do {
         Delay();
         PORT_DATA++;
      } while ((dir == 1) && (PORT_DATA != 255));
      dir = -1;
      do {
         Delay();
         PORT_DATA--;
      } while ((dir == -1) && (PORT_DATA != 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 11.73 on page 374.

Figure 11.73 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 11.7 io_var.txt

```
/* 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 11.74 on page 376.

Figure 11.74 Source Component Window

```
Source
N:\DEMO\io_demo.c                                   Line: 21
                                   // -1 => decreasing


interrupt void Interrupt_Routine(void) {
  dir *= -1;                       // operation to be executed on interrupt
}

static void Delay(void) {            // Wait for a short time
   volatile char cnt = 0;
   while (cnt < DELAY) cnt++;
}
```

8. Activate Stimulation Window by clicking on it.
9. Choose Stimulation>Execute.
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 11.8 io_int.txt

```
def a = TargetObject.#210.B;

PERIODICAL 200000, 10:
  100000 RAISE 7, 3, "test_interrupt";
END

10000000 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 11.9 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

```
def a = TargetObject.#210.B;
```

defines a as byte mapped at address **0x210** in TargetObject.

```
def x = TargetObject.#212;
```

defines **x** as byte mapped at address **0x212** in TargetObject. Size can be omitted, **.B** for byte is default.

```
def b = TargetObject.#216.W;
```

defines **b** as word (.W) mapped at address **0x216** in TargetObject.

```
def c = TargetObject.#220.L;
```

defines **c** as long (.L) mapped at address **0x220** in TargetObject.

```
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).

```
#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.

```
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 +.

```
+20000 b = pbits + 1;
```

reads pbits **(3 bits in Leds. Port_Register)**, increments this value and writes it to b. The + in front of the time marker refers to the time relative to the last encountered time value in the Stimulation file.

```
PERIODICAL 100000, 10:
```

executes the following block

```
   10000 a = 128;
```

```
   30000 RAISE 7, 3, "test_interrupt";
```

10 times. Starts execution 100000 cycles after the start of the simulation.

```
10000 a = 128;
```

assigns **128** to **a**, 10000 cycles after each start of the periodical event.

```
30000 RAISE 7, 3, "test_interrupt";
```

raises an interrupt with priority 3 with vector number 7, 40000 cycles (!) after each start of the periodical event. The time specification in the **PERIODICAL** loop is always relative. So **30000** means +30000. The raised interrupt has the name "**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.

```
1000000 RAISE 7, 3, "test_interrupt";
```

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.

TimedEvent=[ "+" | "#" ] Time AssignmentList.
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.

- 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 programmation, 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 independant in parameters and options from other blocks. The Signal component can be opened within Open I/O Component Dialog Box on page 261 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 on page 387 `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 resused 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 on page 261 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.

`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"
```

FCS Tutorials

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 on page 276
- PWM Channel 0 on page 397

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

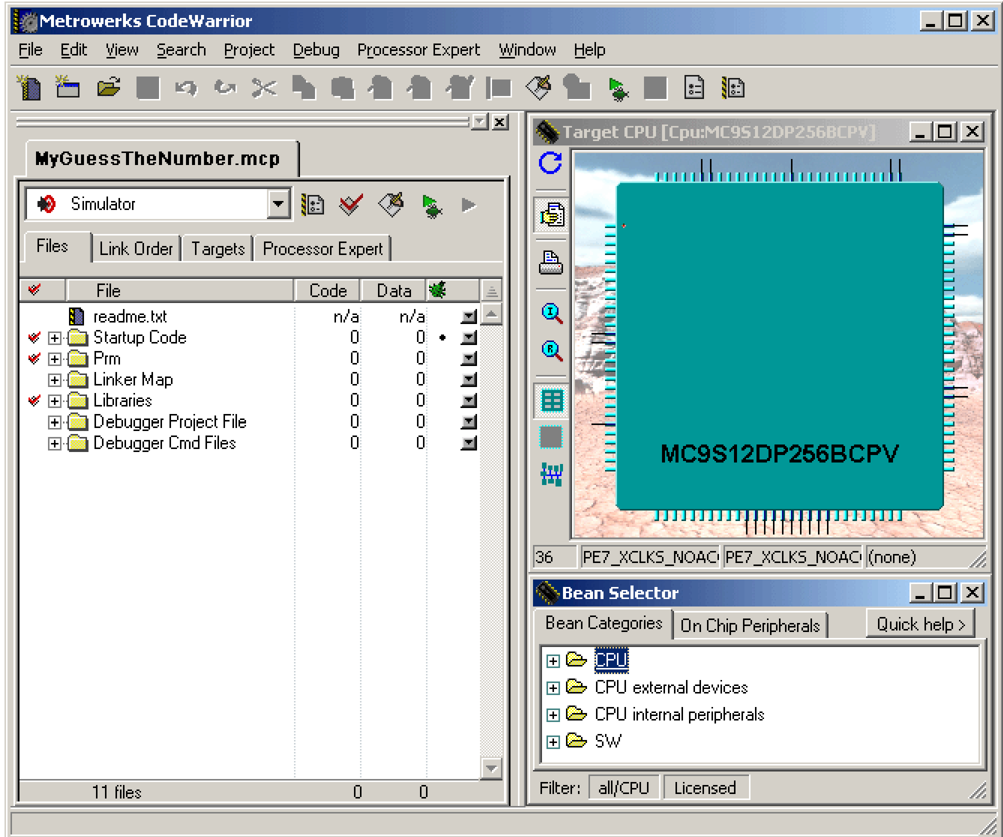
- The tutorial is using Processor Expert, you can get a free Processor Expert licence (Special Edition) from www.codewarrior.com.
- In order to run the produced example on real hardware, you will need a serial cable. This cable must connect COM1 (PC) with the SCI0 (Hardware Board).

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 11.75 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).

Figure 11.76 ‘Target CPU’ Window

Optional:

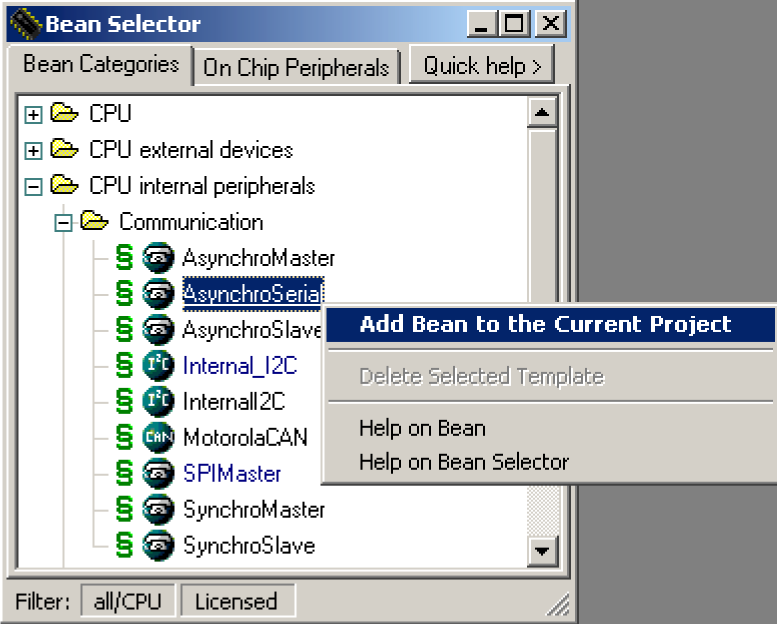
- Place the cursor of the mouse on the pins to see a description of their functions.
- Enlarge the ‘Target CPU’ window and you will see different on-chip modules.

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**'

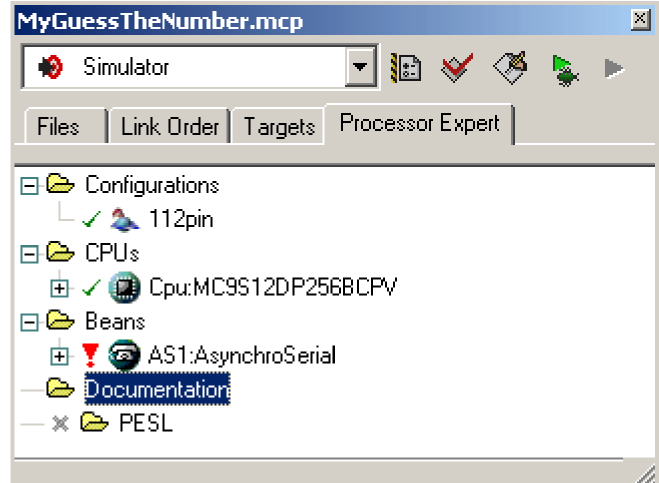
Figure 11.77 Bean Selector Window - Selection of 'AsynchroSerial' Bean



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.

Figure 11.78 Project Window - ‘Processor Expert’ Tab



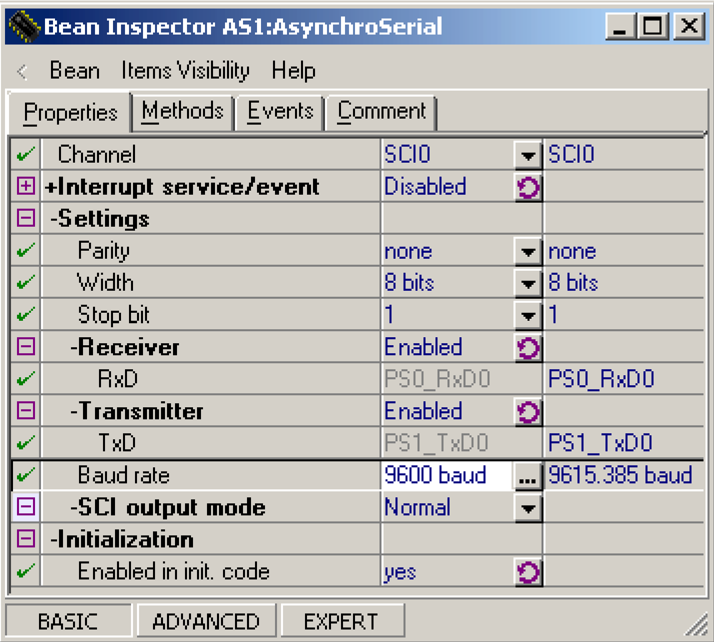
Under ‘Beans’ you should find the previously created bean with the name ‘**AS1:AsynchroSerial**’.

Step 6 - ‘Bean Inspector AS1:AsynchroSerial’ 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**’.

Figure 11.79 ‘Bean Inspector’ Window



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.

```
#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 if(c == ran + '0') {
        PutString("\nCongratulation! You have found the
number!");
        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.

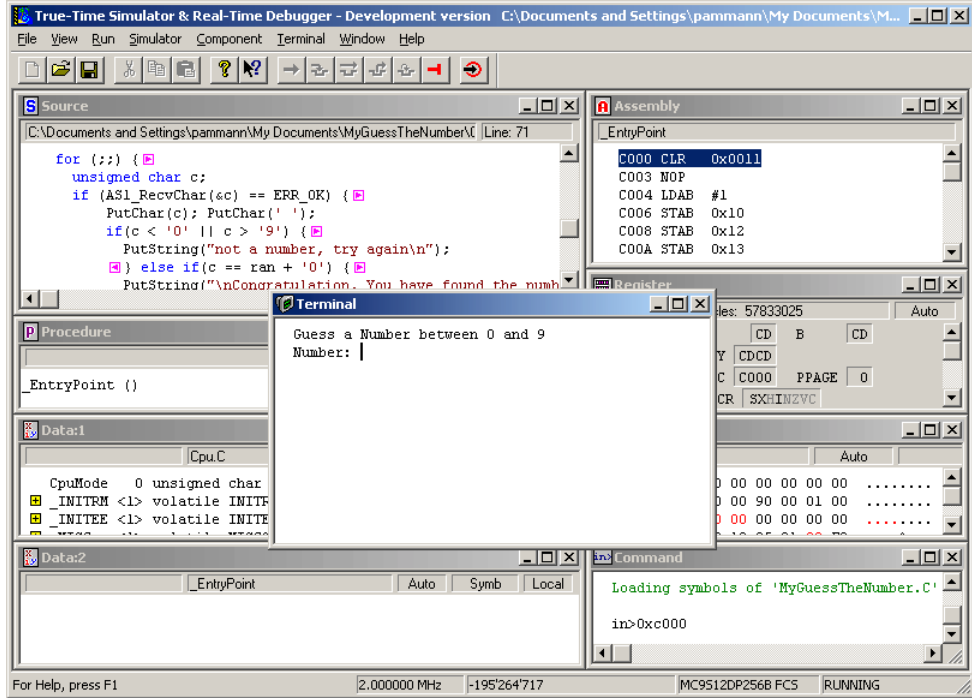
```
void main(void) {
  /*** Processor Expert internal initialization. DON'T
REMOVE THIS CODE!!! ***/
  PE_low_level_init();
 /*** End of Processor Expert internal initialization.
***/

   /*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 11.80 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

- The tutorial is using Processor Expert, you can get a free Processor Expert licence (Special Edition) from www.codewarrior.com.

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**:

```
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.
***/

  /*Write your code here*/
  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 follwing 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 11.81 Debugger Main WIndow - Final Application

12

P&E Multilink/Cyclone Pro Connection

Introduction

An advanced feature of the Freescale debugger for the embedded system development world is the ability to load different connections (debugging targets). This document introduces the P&E Multilink/Cyclone Pro Connection.

This connection (formerly ICD-12) is an interface developed by P&E Microcomputer Systems. The Freescale debugger uses the Multilink/Cyclone Pro to communicate with an external system (also called a *connection*).

With this interface, you can download an executable program from the Freescale debugging environment. The destination of this program is an external connection (target), based on a Freescale MCU, that executes the program. The Freescale debugger receives feedback of real target-system behavior.

The Freescale debugger fully supervises and monitors the connection (target-system) MCU. That is, it 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 When creating a new ICD-12 debugging project, you must select the P&E Multilink/Cyclone Pro connection in the stationery Wizard.

Windows NT Installation Notice

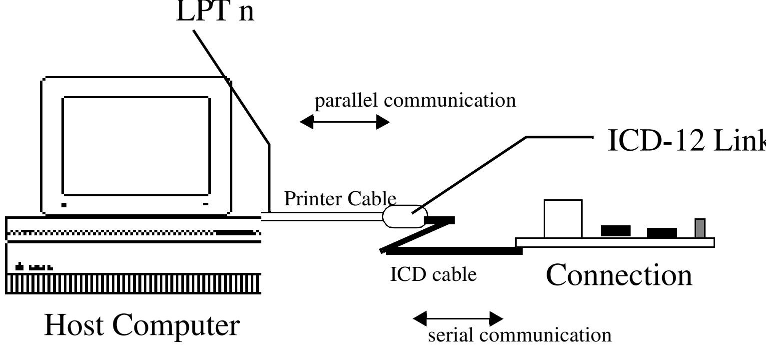
For access to the parallel port of a host computer running Windows NT, you must install a special driver. To install this driver, first install the Freescale Development Kit on the Host, then run the setup program. This program is in the directory: `icd_dr.bat`. `prog\drivers\Nt\ICD`. Freescale installation 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 Start menu: **Start > Programs > Freescale > ICD driver setup > Win NT driver**.

Interfacing Your System and P&E Multilink/ Cyclone Pro

Use a parallel (printer) cable for communication between the P&E Multilink/Cyclone Pro connection and the host computer. The connection hardware must be equipped with a BDM connector that the cable will link with the host. Communication to the connection is serial, as shown in the following figure.

Figure 12.1 System - P&E Multilink/Cyclone Pro Connection Interface



Creating an ICD-12 project in the IDE stationery Wizard includes loading the P&E Multilink/Cyclone Pro connection driver. The P&E Multilink/Cyclone Pro connection driver fully handles the communication protocol between the ICD-12 and the host computer.

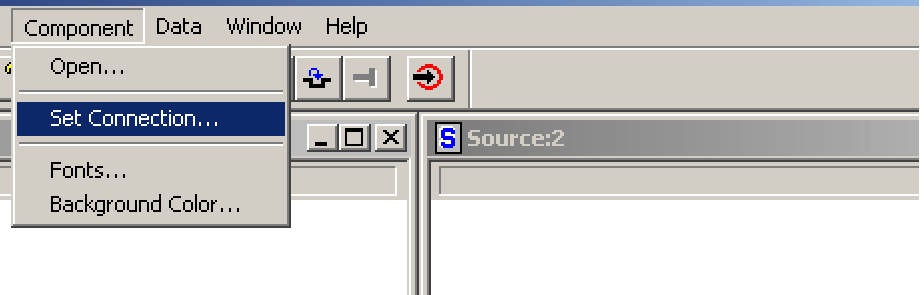
NOTE To prevent communication problems, use the shortest possible cable between the host and the connection hardware.

Loading the P&E Multilink/Cyclone Pro Connection

Usually, the `PROJECT.INI` file specifies the connection. `Target=Icd`. The P&E Multilink/Cyclone Pro driver automatically detects the ICD-12 connection to your system. However, if the driver detects nothing, an error message informs you that the connection hardware is not connected or that the target is connected to a different port.

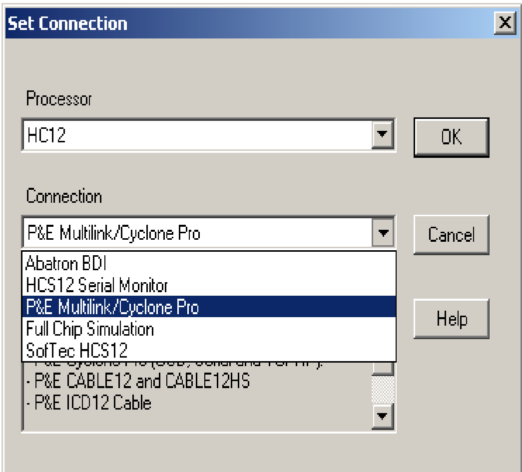
If the `PROJECT.INI` file connection setting is incorrect, load the P&E Multilink/ Cyclone Pro driver. From the Debugger main menu, select *Component | Set Connection...*, as shown below.

Figure 12.2 Component Menu



The Set Connection dialog box now appears.

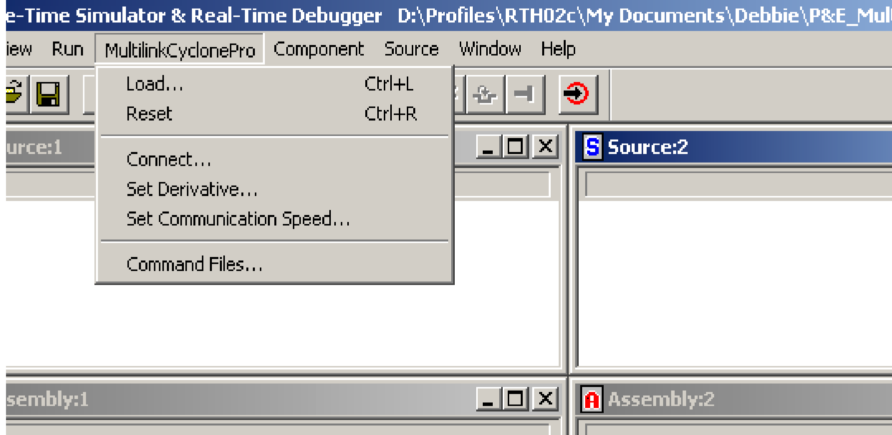
Figure 12.3 Set Connection Dialog Box - Connection Menu



Press the Down Arrow button to display the list of available connections. The Connection menu selection *P&E Multilink/Cyclone Pro* loads the proper drivers, etc. for the P&E Multilink/Cyclone Pro connection.

In the Debugger Main window, the Connection heading has been renamed **MultilinkCyclone Pro**. Click on this heading to display its menu with the list of options.

Figure 12.4 MultilinkCyclone Pro (ICD-12) Menu

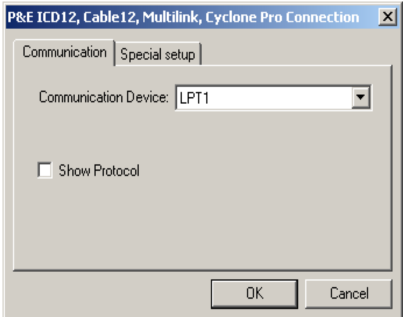


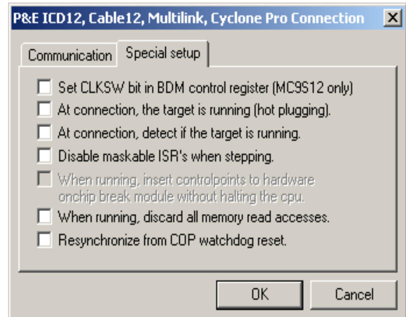
The menu selection *MultilinkCyclonePro* | *Load...* loads an executable ("`.abs`") file into connection memory. The file's program counter points to the first instruction of the startup section.

The menu selection *MultilinkCyclonePro*| *Reset* triggers a reset of the connection and executes the command file "`reset.cmd`".

The menu selection *MultilinkCyclonePro* | *Connect...* takes you to the P&E ICD-12, Multilink, Cyclone Pro dialog box. The two tabs of this dialog box allow you to set the Communications and Special Setup parameters for the P&E Multilink/Cyclone Pro connection.

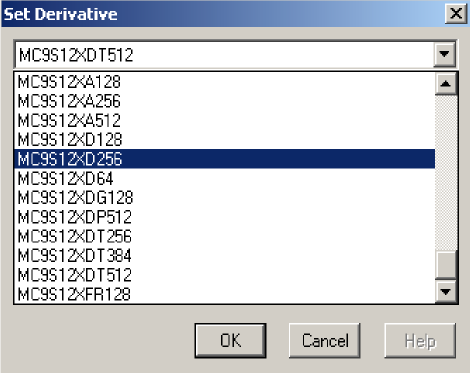
Figure 12.5 P&E ICDE-12 Multilink, Cyclone Pro Connection Dialog Box





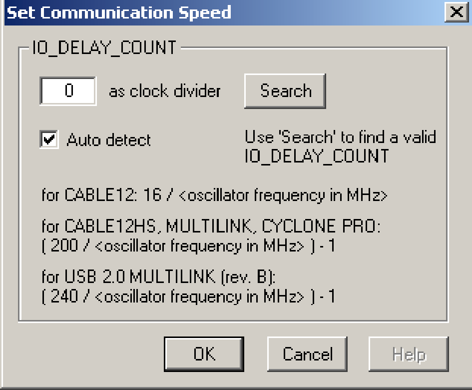
The menu selection *MultilinkCyclonePro | Set Derivative*...takes you to the Set Derivative dialog box. This dialog box allow you to choose the target MCU for the P&E Multilink/ Cyclone Pro connection.

Figure 12.6 Set Derivative Dialog Box



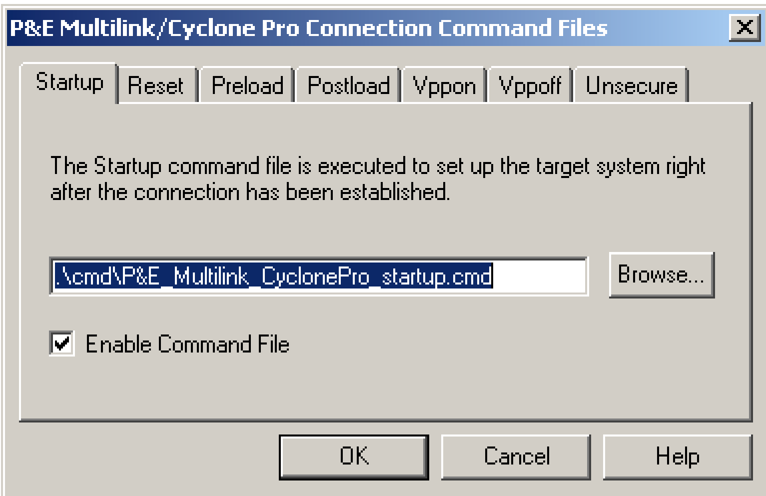
The menu selection *MultilinkCyclonePro | Set Communication Speed*...lets you control the various factors associated with communication speed for the P&E Multilink/Cyclone Pro connection.

Figure 12.7 Set Communication Speed Dialog Box



The menu selection *MultilinkCyclonePro | Command Files...* takes you to the Command Files window with its six tabs.

Figure 12.8 P&E Multilink/Cyclone Pro Connection Command Files Window



Default Connection Setup

As with any connection, you can use the *Connection* menu to load the P&E Multilink/ Cyclone Pro 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 12.1 Example of `PROJECT.INI` file:

```
Window0=Source      0   0  60  30
Window1=Assembly   60   0  40  30
Window2=Procedur    0  30  60  25
Window3=Register   60  30  40  30
Window4=Memory     60  60  40  40
Window5=Data        0  55  60  23
Window6=Command     0  78  60  22
Target=ICD
```

13

Softec HCS12 Connection

This section guides you through the *SofTec 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 throught the SofTec in-circuit debugger/programmer units, i.e:

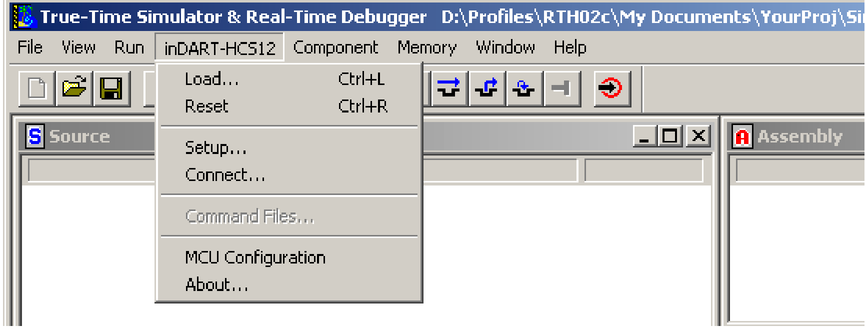
SofTec Microsystems HCS12 ISP Debuggers/Programmers (inDART Series) and Starter Kits (AK/SK/PK/ZK and newer Series).

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.

inDart-HCS12 Menu Options

Once the ***SofTec HCS12*** connection is set, the connection menu entry in the debugger main toolbar is "***inDART-HCS12***".

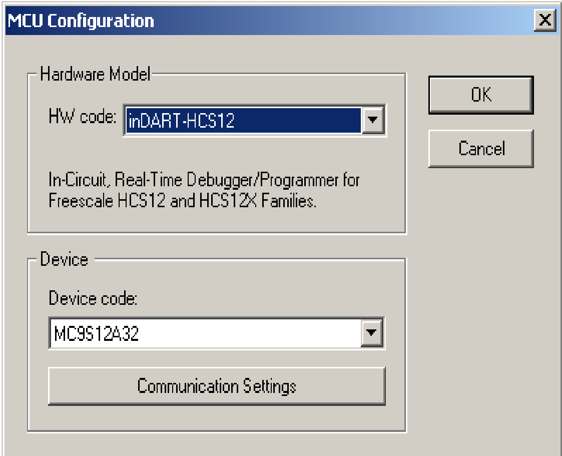
Figure 13.1 inDART-HCS12 Menu Options



MCU Configuration Option

Select the **inDART-HCS12 > MCU Configuration** *option* to display the MCU Configuration Dialog Box .

Figure 13.2 MCU Configuration 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.

Pressing the *Communication Settings* button opens the Communication Settings Dialog Box.

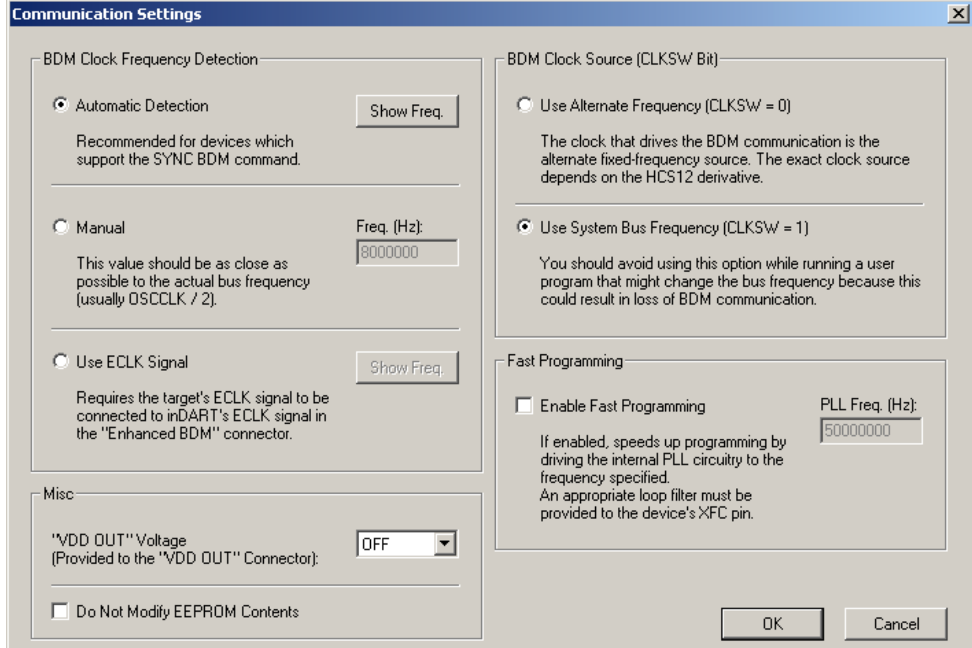
Communication Settings Dialog Box

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 13.3 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 forFreescale HC12 Family FLASH Devices User's Manual"*** from SofTec for further details.

User's Manual Option

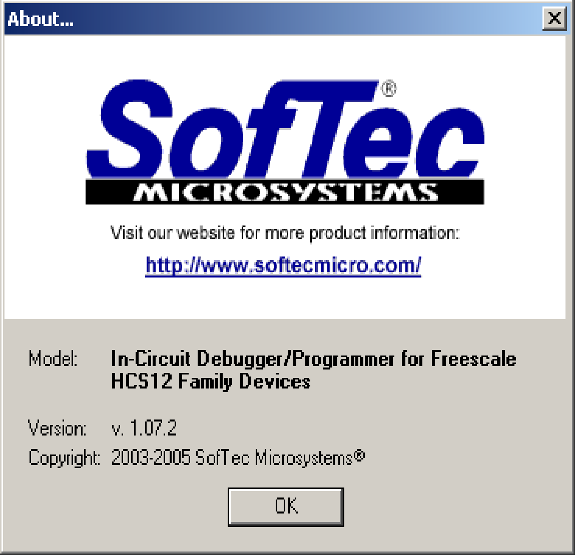
Select the **inDART-HCS12 > User's Manual** option to open the ***"inDART®-HCS12 In-Circuit Debugger/Programmer for Freescale HC12 Family FLASH Devices User's Manual"*** from SofTec.

About Option

Select the **inDART-HCS12 > About...** *option* to display the About Dialog Box .

This dialog box belongs to the SofTec GDI DLL and provides information about the ***inDART_HCS12.dll*** release and version.

Figure 13.4 About Dialog Box



14

HCS12 Serial Monitor Connection

This section guides you through 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 seperate 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

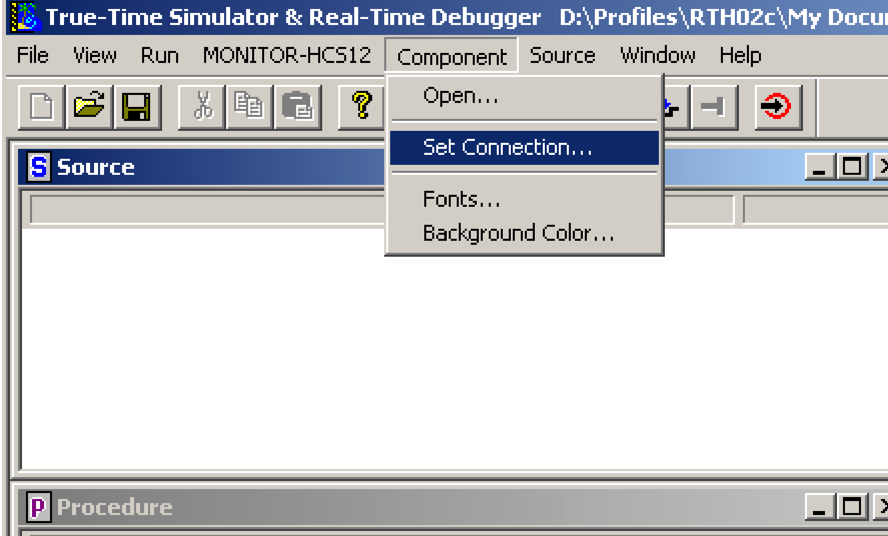
These paths are explained in the Getting Started chapter of this manual.

HCS12 Serial Monitor Interface

To utilize the HCS12 Serial Monitor connection take the following steps:

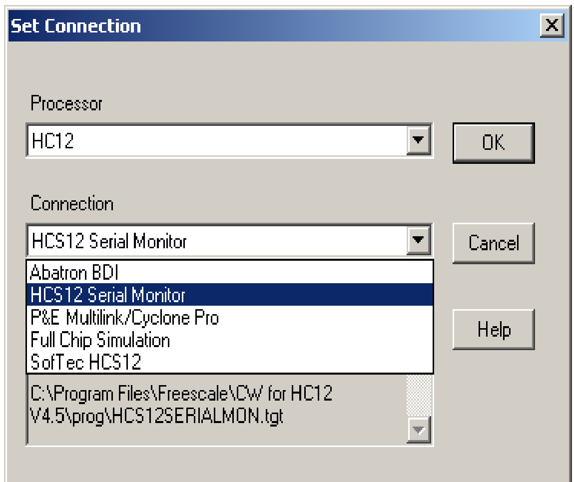
1. Run the *CodeWarrior IDE* with the shortcut created in the program group.
2. Create a project (see the Getting Started chapter of this manual).
3. Choose the menu **Project > Make** and **Project > Debug** to start the debugger - Debugger Main window opens.

Figure 14.1 Debugger Main Window - Component Menu



4. In the debugger main window, from the Component menu, you can choose **Component > Set Connection...** if you wish to select another connection.

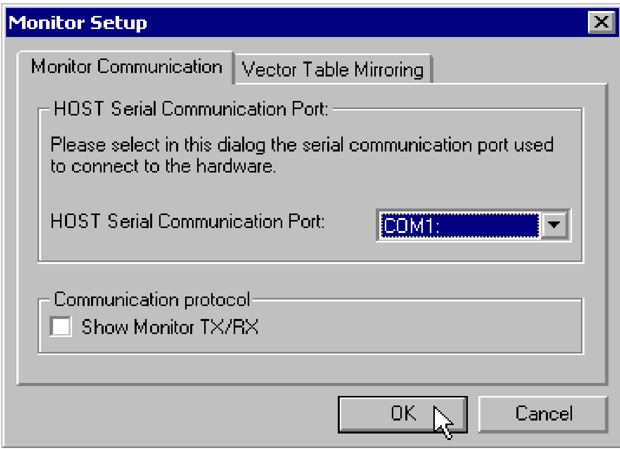
Figure 14.2 Set Connection Dialog Box - HCS12 Serial Monitor Selection



5. Select ***HC12*** as Processor then ***HCS12 Serial Monitor*** 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.

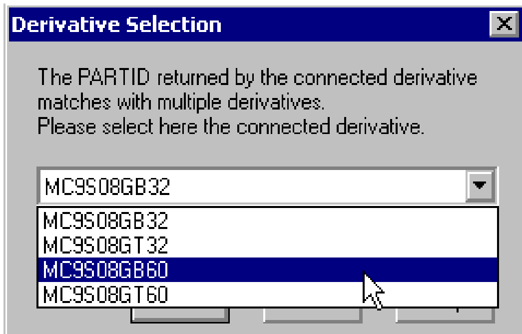
Figure 14.3 Monitor Setup Window - Monitor Communication Tab



7. Press the OK button. The HCS12 Serial Monitor connection reads the device silicon ID. This ID can match several derivatives.

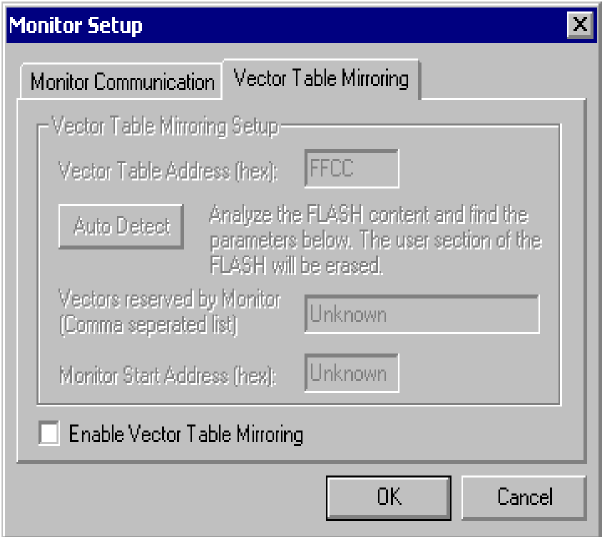
8. Set the correct derivative matching with your hardware in the Derivative Selection dialog box.

Figure 14.4 Derivative Selection Dialog Box



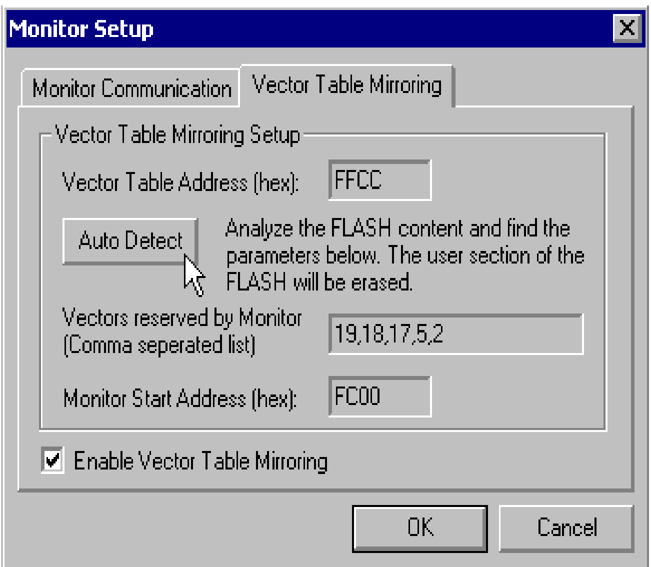
9. Press the OK button. The **Monitor Setup** window is opened again, to propose to use the “mirrored vector table” feature. We recommand that you use the Vector Table Mirroring feature. Otherwise, vectors cannot be programmed as captured and protected from erasing or overwriting by the HCS12 Serial Monitor.

Figure 14.5 Monitor Setup Window - Vector Table Mirroring Tab



10. To enable this specific feature, check the “Enable Vector Table Mirroring” checkbox. ¨

Figure 14.6 Monitor Setup Window - Vector Table Mirroring Tab



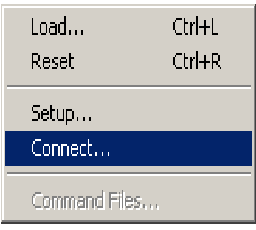
11. Press the "Auto Detect" button to make the debugger search for the vector table address and vectors reserved by the HCS12 Serial Monitor.

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 is set in the Debugger menu, as shown below.

Figure 14.7 MONITOR-HCS12 Menu Entries



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

Selectthe **MONITOR-HCS12> Bus Trace** toption o 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.

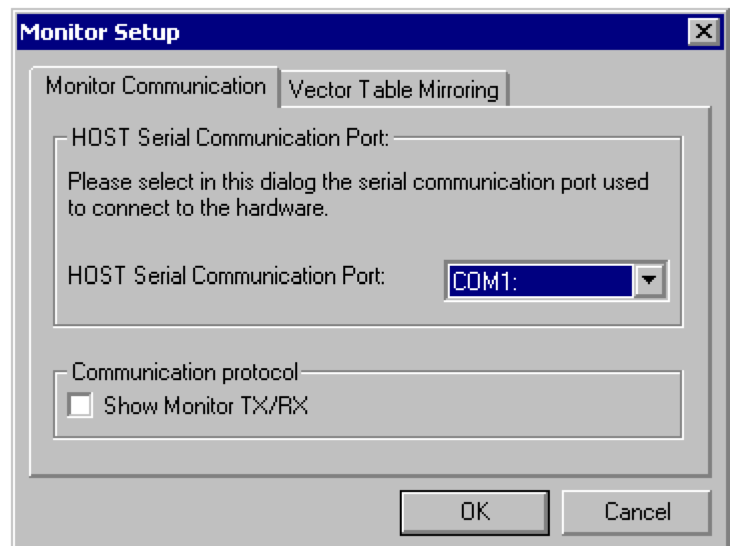
Select Derivative

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 14.8 Monitor Setup Window - Monitor Communication Tab

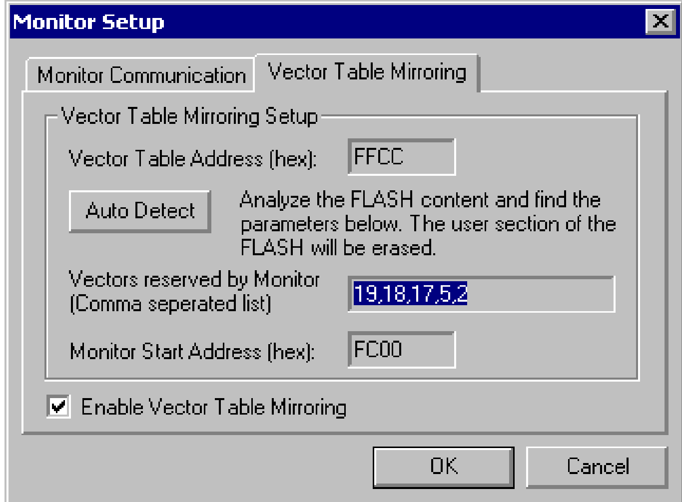


Monitor Communication Tab

Using the Monitor Communitaction 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 14.9 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.

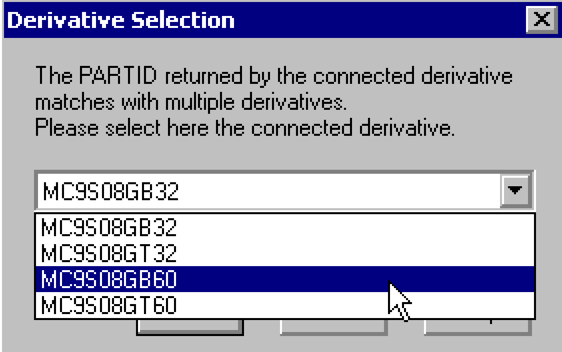
Vector table mirroring allows you to access chip vectors transparently. Indeed, the HCS12 Serial Monitor also uses some vectors, and the vector area is protected from erasing and overwriting. We recommend that you use this feature. Otherwise, user application vectors cannot be programmed as captured 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.

Figure 14.10 Derivative Selection Dialog Box



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.

15

Abatron BDI Connection

This document includes information on the *Abatron BDI* Connection and helps you understand how to use this debugger connection. This document is divided into following sections:

- The Abatron BDI Connection Introduction on page 428 section introduces the *Abatron BDI* Connection.
- The Interfacing Abatron BDI andYour System on page 429 section contains information about the connection between the BDI interface box and the debugger.
- The BDI Interface Software Setup on page 430 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 Abatron BDI Connection Menu Entries on page 437 section provides a description of the *Abatron BDI* Connection specific menu entries.
- The Abatron BDI Connection Windows, Edit and Dialog Boxes on page 438 section provides a description of the *Abatron BDI* Connection specific dialog boxes.
- The Abatron BDI Status Bar Information on page 441 section describes the status bar messages for the *Abatron BDI* Connection.
- The Terminal Emulation on page 443 section describes how to emulate a text terminal between **CPU12**, **CPU16** and **CPU32** derivatives and the debugger.
- The Flash Programming on page 445 section describes how to proceed to program on-chip non-volatile memory area.
- The HC12 and HCS12 Banked Memory support on page 447 section lists all the debugger commands specific to this connection.
- The Banked Memory Location Window on page 447 section describes how to use the banked memory model manager with related **CPU12** derivatives.

Abatron BDI Highlights

The *Abatron BDI* 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 on page 447 section.

Abatron BDI 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.

Abatron BDI 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 *Abatron BDI* Connection is introduced in this document.

This document describes the specific features of the Abatron BDI Connection.

With this interface, you can download an executable program from the debugger environment to an external target system based on a Freescale 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 Abatron BDI Connection. Profiling, Coverage analysing and I/O simulation are not available with the Abatron BDI Connection.

Interfacing Abatron BDI andYour System

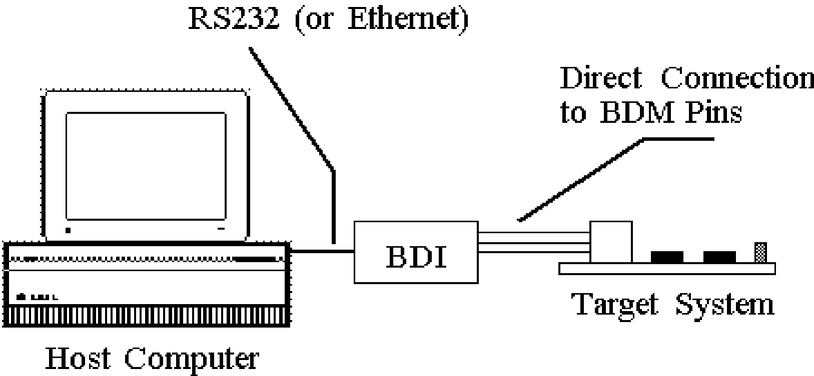
NOTE BDI Structure, Configuration, Connection to the Host, Connection to the Target, Configuration, and Working Modes are described in *ABATRON User Manuals*.

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 Abatron BDI 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 15.1 Your System/Abatron BDI Interface



BDI Interface Software Setup

The Abatron BDI connection is delivered during installation and contains all required files, demo projects to use the Abatron BDI 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.

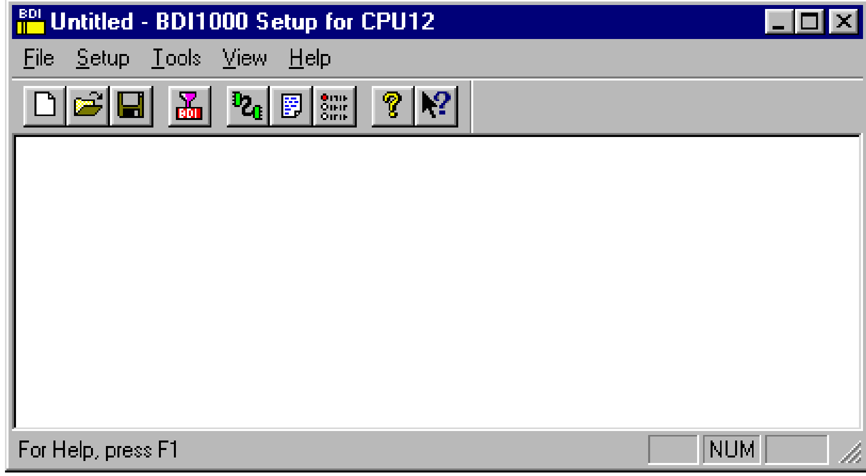
Running the ABATRON Configuration Tool

The configuration program (e.g. B10C12.EXE for CPU12 processor with BDI1000, B20MCORE.EXE for the M·CORE with BDI2000, BDIHSHCI.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 **Abatron BDI | Configure BDI Box...** or specify the tool path in the **Abatron 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.

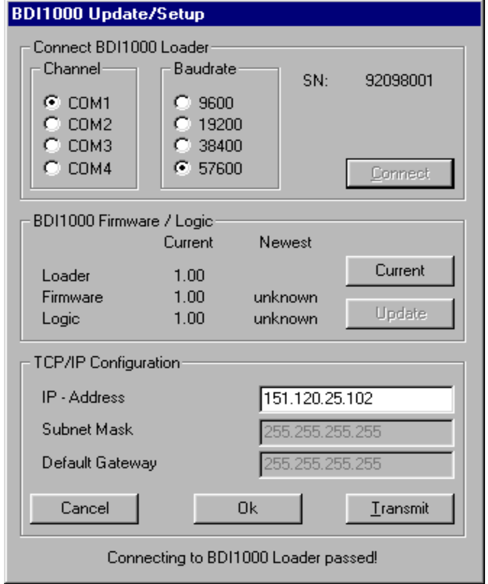
Figure 15.2 BDI1000 Setup for CPU12 Dialog Box



Firmware Loading

In the dialog box shown above, select **Setup | Firmware...** to open the firmware dialog.

Figure 15.3 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`).

Figure 15.4 File Menu

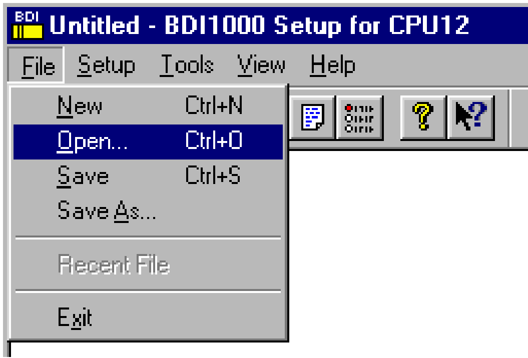
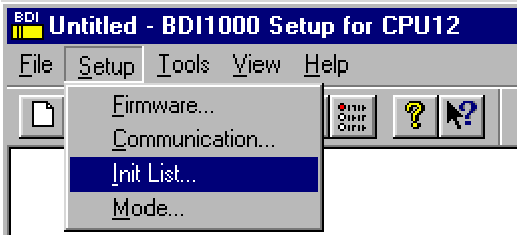


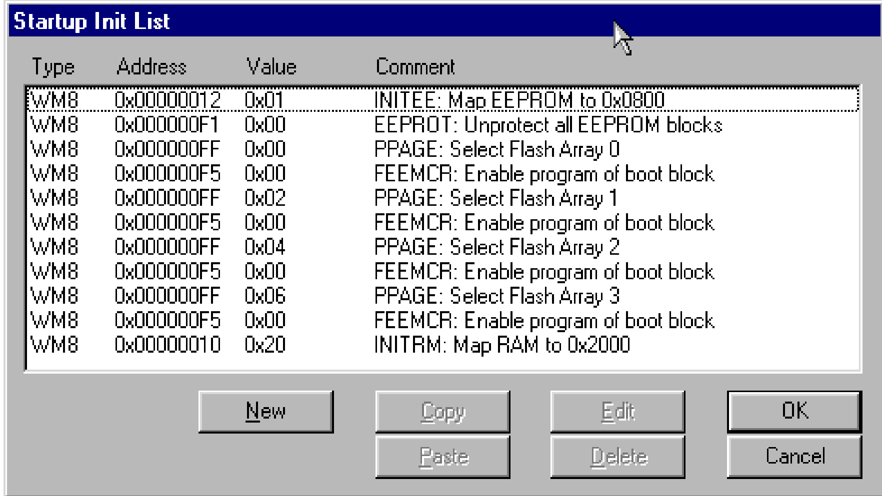
Figure 15.5 Setup Menu



Select **Setup | Init List...** tfrom the Setup menu o 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 15.6 Startup Init List Dialog Box

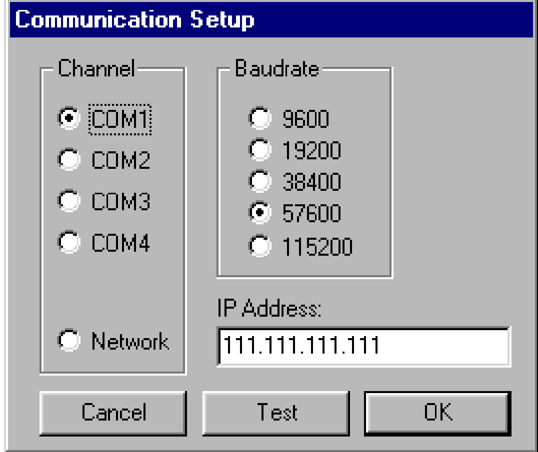


Quit this dialog box by clicking **OK** and save the settings if necessary.

Communication with the Debugger Setup

Select **Setup | Communication...** from the Setup menu to open the Communication Setup dialog box.

Figure 15.7 Communication Setup Dialog Box

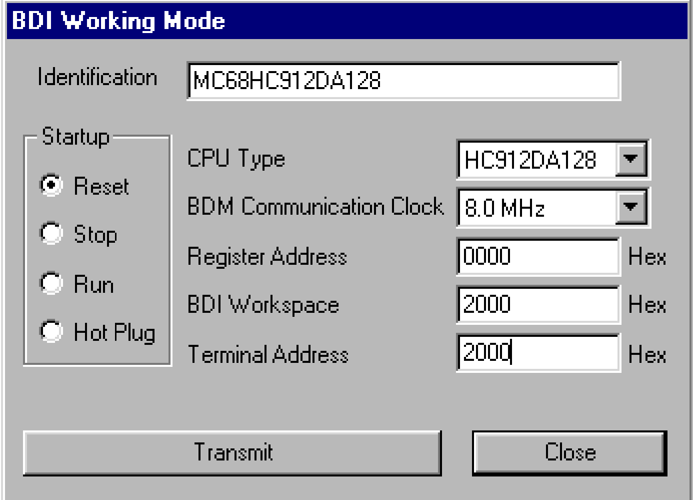


In this dialog box, set the communication for your future use of the BDI with the debugger. Settings made here should be identical to communication settings made in the debugger within the Communication Setup Dialog Box on page 433. Press the *Test* button to check the setup then click *OK* to quit this 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 15.8 BDI Working Mode Dialog Box



Loading the Abatron BDI Connection

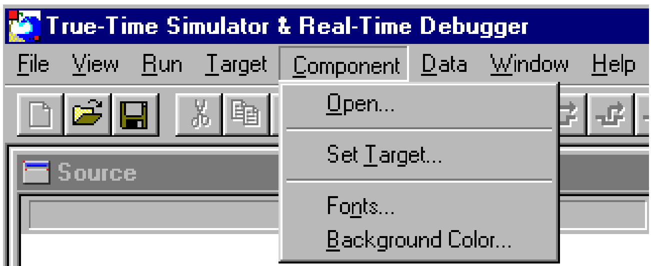
The target is set in the `[Environment Variables]` section of the project file, through the statement `Target=Abatron BDI`.

The *Abatron BDI* Connection automatically detects that the target is connected to your system. If nothing is detected, the Communication Device Specification Edit Box on page 439 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 *Abatron BDI* Connection as described below.

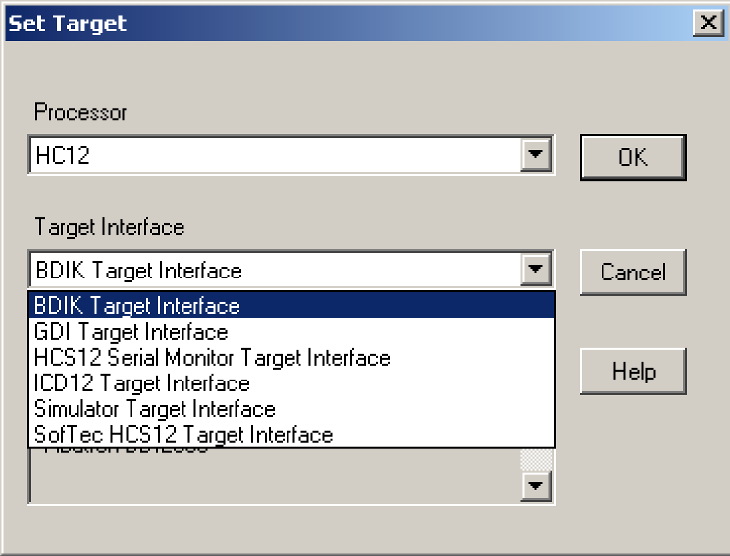
In the debugger, select **Component | Set Target...**in the component menu.

Figure 15.9 Debugger Component Menu



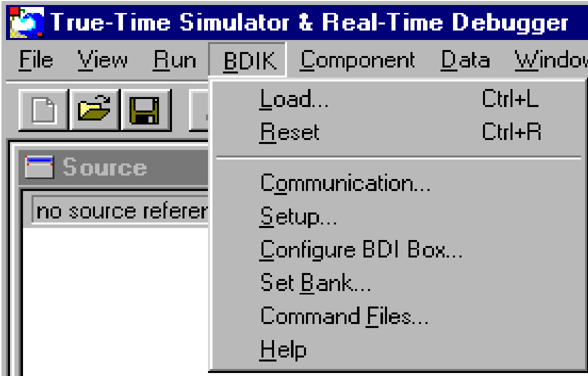
The Set Connection dialog box is displayed. Select *Abatron BDI Connection* in the list of proposed targets and click **OK**.

Figure 15.10 Set Connection Dialog Box



After a successful target loading, the Debugger Main Window **Target** menu item is replaced by *Abatron BDI*.

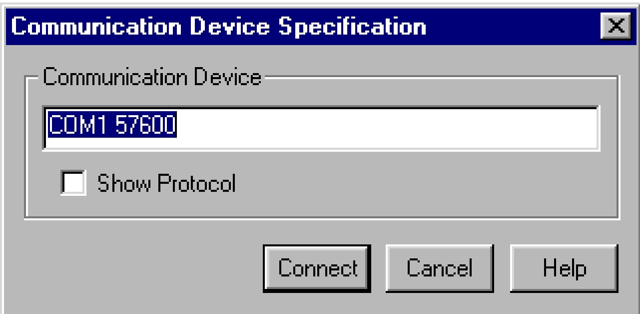
Figure 15.11 Abatron BDI Menu



You can change the communication parameter (baud rate and port) by selecting the menu entry *Abatron BDI | Connect...* .

If communication with the BDI Interface could not be established, an error message is displayed followed by the Communictaion Debice Specification dialog box.

Figure 15.12 Communication Device Specification Dialog Box

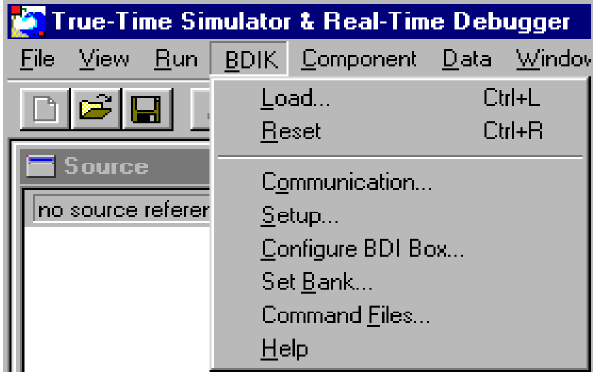


In this dialog box, you can modify the device specification (e.g. Communication Port and baud rate). These settings are saved in the current project and will be used again in future sessions.

Abatron BDI Connection Menu Entries

After loading the *Abatron BDI* Connection, the **Target** menu item is replaced by *Abatron BDI*.

Figure 15.13 Debugger Abatron BDI 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 *Abatron BDI* is replaced with **Connect...** .

The different entries of the *Abatron BDI* menu are described below:

Load...

Select **Abatron BDI | Load...** to load the application to debug, i.e. a `.ABS` file.

Reset

The menu entry **Abatron BDI | 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 **Abatron BDI | Communication...** or **Abatron BDI | Connect...** to display the Communication Device Specification Edit Box on page 439. If the connection to the target has failed, the entry **Communication...** of menu *Abatron BDI* is replaced with *Connect...* .

Setup...

Select **Abatron BDI | Setup...** to open the Setup Dialog Box on page 440 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 **Abatron BDI | 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 on page 440, 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 on page 440.

Set Bank...

This dialog is only available if the connected processor is a *Freescale CPU12/HC12* derivative. Select the entry **Abatron BDI | Set Bank...** to display the Banked Memory Location Window on page 447.

Command Files

Select the entry **Abatron BDI | Command Files** to display the Target Interface Command Files Window .

Help

Select the entry **Abatron BDI | Help** to open the *Abatron BDI* Connection Help File.

Abatron BDI Connection Windows, Edit and Dialog Boxes

This section describes the dialog boxes which are specific to the *Abatron BDI* Connection.

Those dialogs are:

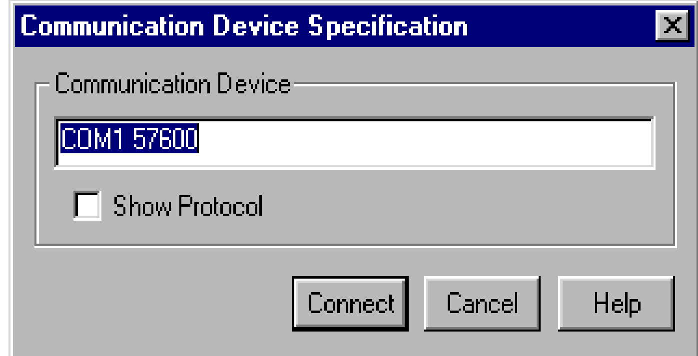
- The Communication Device Specification Edit Box on page 439.
- The Setup Dialog Box on page 440.
- The Banked Memory Location Window on page 447 (available only if the connected derivative is a Freescale HC12 that supports banking).

Communication Device Specification Edit Box

The Communication Device Specification edit box pops up automatically if the *Abatron BDI* Connection could not establish the communication with the BDI (box) interface. However, this dialog box can be opened by selecting the menu entry **Abatron BDI | Communication...** or **Abatron BDI | Connect...** .

If the connection to the target has been successfully achieved (dialog opened using menu entry **Abatron BDI | Communication...**), it is not possible to modify the **Communication Device** edit box. Only the **Show Protocol** check box can be modified.

Figure 15.14 Communication Device Specification Edit Box



If the connection to the BDI box has failed (dialog box automatically opened or using menu entry **Abatron BDI | 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:

"**COM***n 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".

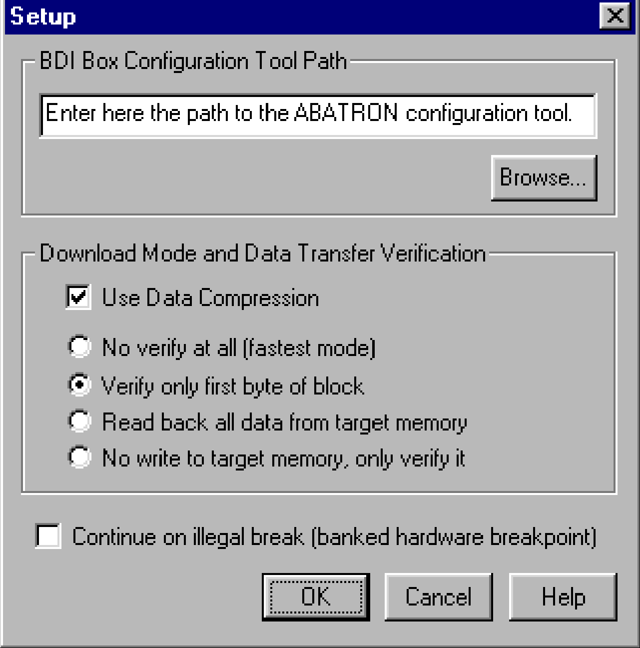
The **Show Protocol** check box allows you to switch on/off the displays of the messages sent between the debugger and the BDI interface. If the **Show Protocol** box is checked, all commands and responses sent and received are reported in the **Command Line** window.

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 `[Abatron BDI]` section of the project file.

Setup Dialog Box

The Setup dialog box is opened selecting the Abatron BDI menu entry **Abatron BDI | Setup...**.

Figure 15.15 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 *Abatron BDI | 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**.

Abatron BDI Status Bar Information

When the *Abatron BDI* 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 15.16 Abatron BDI Connection - Debugger Status Bar

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 **Abatron BDI | 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 Abatron BDI 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 on page 430* 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 15.1 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 ***Abatron 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):

```
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
```

Abatron BDI Connection Environment

Default Connection Setup

As any other debugger connection, the *Abatron BDI* Connection can be loaded from the **Connection** 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 *Abatron BDI* Connection interactively. Please refer to the Loading the Abatron BDI Connection on page 434 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.

HC12 and HCS12 Banked Memory support

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.

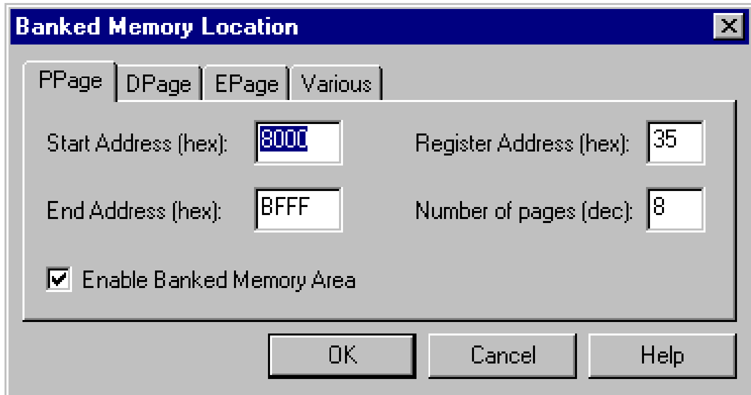
Banked Memory Location Window

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 Ban*k.... (In this section, ***Target Name*** is the name of the connection, like ***Abatron BDI***, ***ICD-12***, 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 15.17 Banked Memory Location Window - PPage Tab



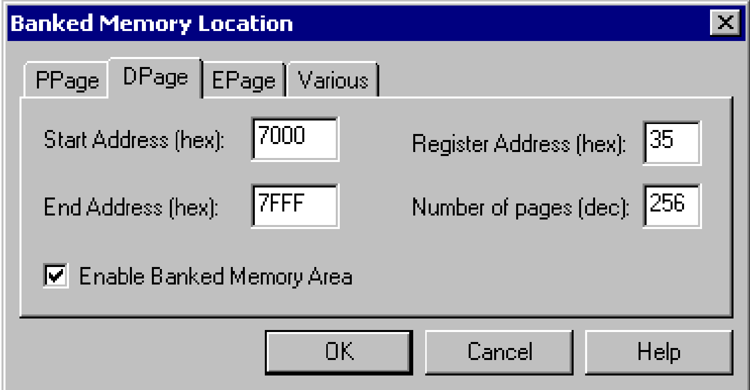
PPAGE Tab

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).

Figure 15.18 Banked Memory Location Window - DPage Tab



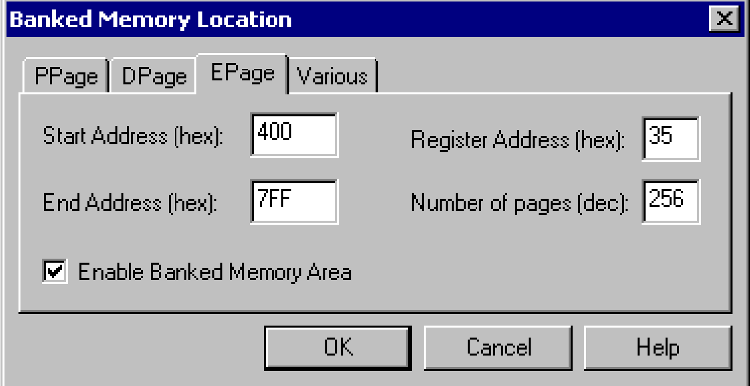
DPAGE Tab

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).

Figure 15.19 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).

Figure 15.20 Banked Memory Location Window - Various Tab

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.

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" on page 457
- Chapter 3.4 "HC(S)12(X) Flash Programming" on page 467

15

HCS12 On-chip DBG Module

Documentation for this chapter is now being updated and will be available on a special Web Release of this product on or before March 15, 2006.

16

HCS12X On-chip DBG Module

Documentation for this chapter is now being updated and will be available on a special Web Release of this product on or before March 15, 2006.

17

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.

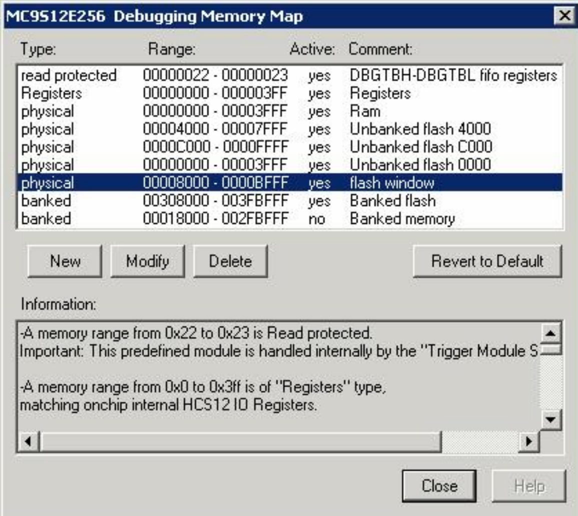
Indeed, the DMM has a GUI, therefore providing to the user a way to change anytime the way to access the memory.

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 the the edition mode diagnostic.

Pressing the "New" button will open the edition dialog box to create a new memory address range.

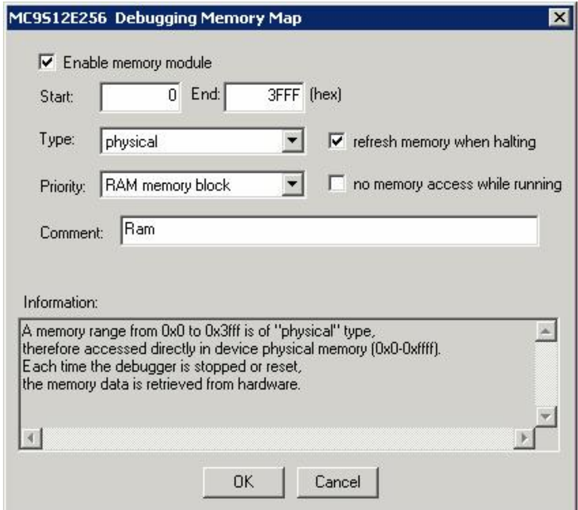
Pressing the "Modify" button opens the edition dialog of the selected memory address range to modify it. More memory range information is displayed in the edition dialog, and an enhanced diagnostic is also displayed.

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 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.

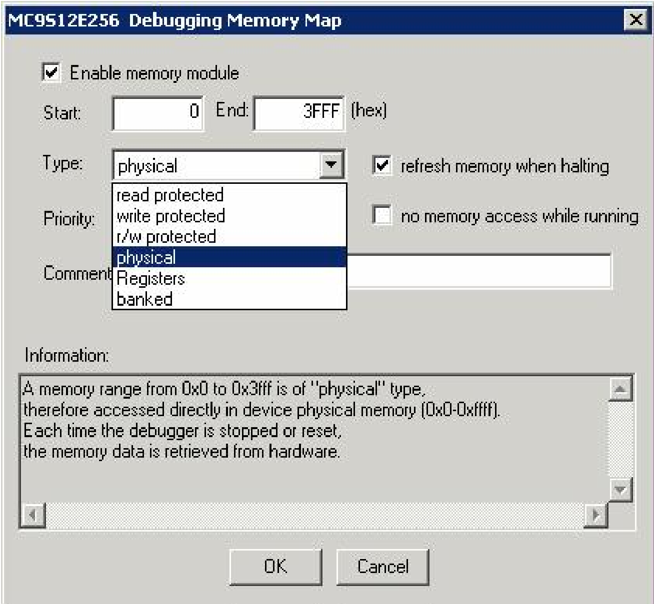
NOTE The "Start" "End" range is a range address for a “Type” and for a “Priority”. Internally, ranges can overlap only if they are of the same type and of the same priority. The debugger will always read with rules of the range with the highest priority.

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.

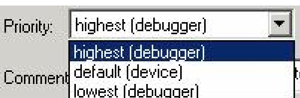
NOTE CPU core specific memory "types" and "Priorities" are listed at the end of this manual section.

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



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 leed to eroneous 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.

- "**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 leeds 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.

- "**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.
- "**eep 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 leeds 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, “**eep 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 on page 599 chapter of this manual.

18

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/unprotected) 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

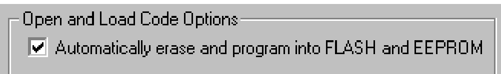
Pressing the OK button launches backgrounded flash commands described later in this section to arm programming, load/program an application file, then disarm programming.

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.

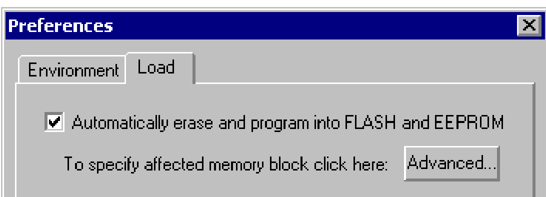
Figure 18.2 Open and Load Code Options Dialog Box



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).

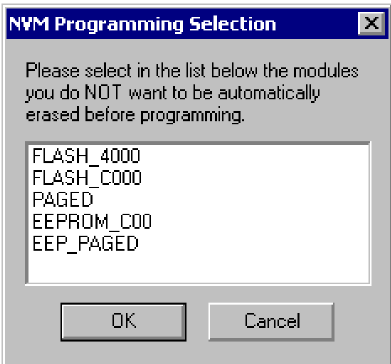
Figure 18.3 Preferences Window - Load Tab



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.

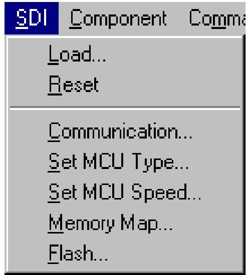
The “NVM Programming Selection” list box is tightly associated to the “FLASH AEFSKIPERASING” command of the debugger.

TIP When using this feature, make sure to also select modules that could cover/include all other modules listed, modules usually called PAGED, EEP_PAGED, ALL_PPAGES, ALL_EPAGES, ALL_xxx, etc.

NVMC Graphical User Interface

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.

Figure 18.5 SDI Connection Menu Options



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 indivual/selective module for these.

NOTE Please see Hardware Considerations on page 478 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 harware.

- ***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.

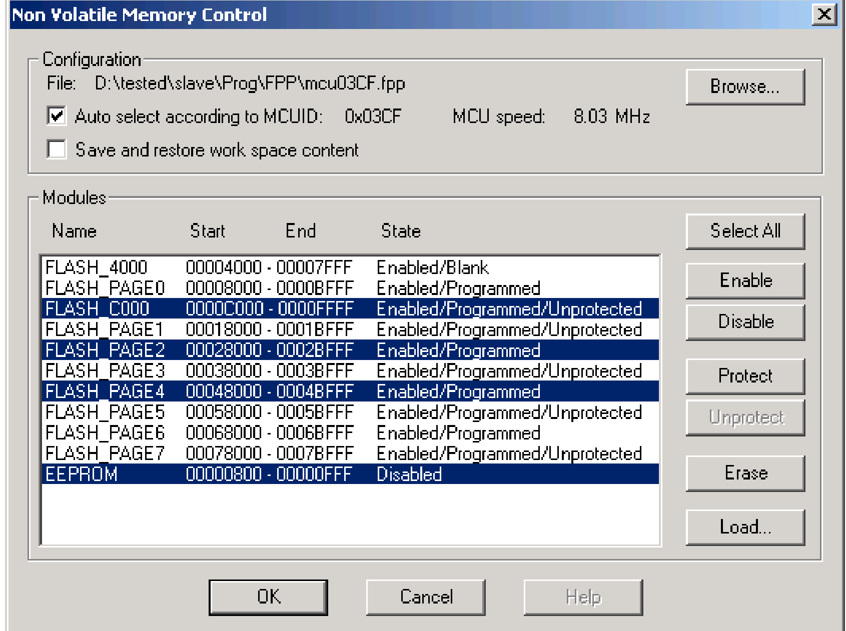
To select a module or other list item, left-click the module. To unselect a module, press the Ctrl key and left-click. For multiple selections or unselections, use the Shift key.

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 modules 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 on page 478 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 on page 599.)

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.

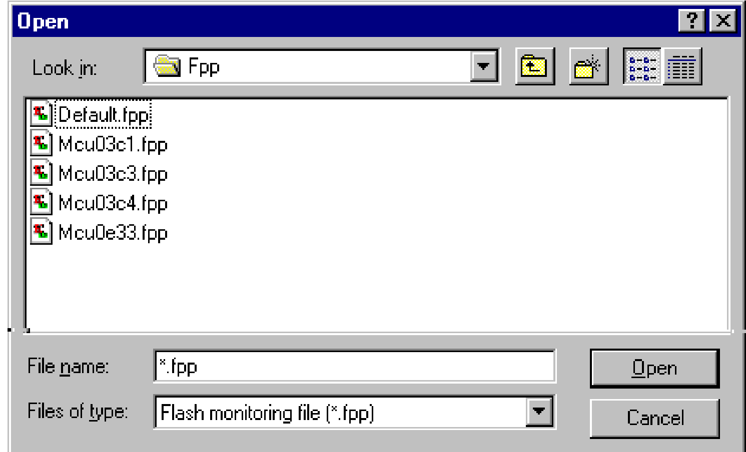
   Example:

   ```
   [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.

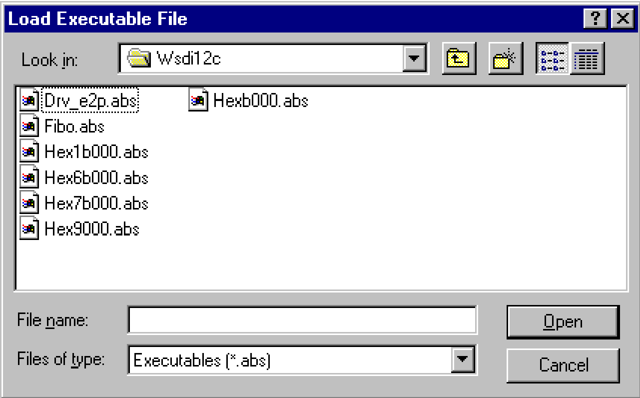
Click the **OK** button to close the NVMC dialog box. If the **Auto select according to MCUID:** checkbox is clear, the NVMC utility saves the name of the selected configuration file under the `NV_PARAMETER_FILE` entry of the `project.ini` file. If you check this checkbox, the utility dies not save the `.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.

Figure 18.8 Load Executable File Dialog Box

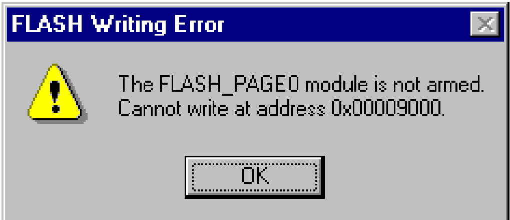


If a problem occurs during application loading into flash, the NVMC utility displays an error message.

Figure 18.9 FLASH Programming Error Message Box

If programming voltage is not available, the following error message is displayed

Figure 18.10 FLASH Writing Error Message Box



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 on page 599 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 on page 599 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).

-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_PAGE2 / module number: 4

- 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:

```
// 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.

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 on page 487
- Chapter 4.2 Debugger Connection-specific Commands on page 599

20

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.

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 on page 488 contains all available Kernel commands.

Table 20.1 List of Kernel Commands

Command, Syntax	Short Description
A on page 498	Affects a **value**
AT on page 509	Sets a time delay for command execution
CALL on page 515 fileName**[;C][;NL]**	Executes a command file
DEFINE on page 526 symbol [=] expression	Defines a user symbol
ELSE on page 531	Other operation associated with **IF** command
ELSEIF on page 531 condition	Other conditional operation associated with **IF** command
ENDFOCUS on page 532	Resets the current focus (refer to **FOCUS** command)

Table 20.1 List of Kernel Commands (*continued*)

Command, Syntax	Short Description
ENDFOR on page 533	Exits a **FOR** loop
ENDIF on page 533	Exits an **IF** condition
ENDWHILE on page 534	Exits a **WHILE** loop
FOCUS on page 539 component	Sets the focus on a specified component
FOR on page 541 [variable =]range [“,” step]	**FOR** loop instruction
FPRINTF on page 542 (fileName,format,parameters**)**	FPRINTF instruction
GOTO on page 545 label	Unconditional branch to a label in a command file
GOTOIF on page 546 condition Label	Conditional branch to a label in a command file
IF on page 548 condition	Conditional execution
PAUSETEST on page 566	Displays a modal message box
PRINTF on page 567 (“Text:,” value]**)**	PRINT instruction
REPEAT on page 569	REPEAT loop instruction
RETURN on page 571	Returns from a CALL command
TESTBOX on page 588	Displays a message box with a string
UNDEF on page 589 symbol | *	Undefines a userdefined symbol
UNTIL on page 592 condition	Condition of a REPEAT loop
WAIT on page 594 [time] [;s]	Command file execution pause
WHILE on page 596 condition	WHILE loop instruction

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 on page 490 contains all available Base commands.

Table 20.2 List of Base Commands

Command, Syntax	Short Description
BC on page 510 address|*	Deletes a breakpoint (breakpoint clear)
BS on page 513 address|function [P|T[state]]	Sets a breakpoint (breakpoint set)
CD on page 516 [path]	Changes the current working directory
CR on page 521 [fileName][;A]	Opens a record file (command records)
DASM on page 522 **[address|range][;OBJ]**	Disassembles
DB on page 523 [address|**range]**	Displays memory bytes
DL on page 528 [address|**range]**	Displays memory bytes as longwords
DW on page 529 [address|**range]**	Displays memory bytes as words
G on page 543 [address]	Starts execution of the application currently loaded
GO on page 544 [address]	Starts execution of the application currently loaded
LF on page 550 [fileName][;A]	Opens a log file
LOG on page 553 type [=] state {[,] type [=] state}	Enables or disables logging of a specified information type
MEM on page 558	Displays the memory map
MS on page 559 range list	Sets memory bytes
NOCR on page 562	Closes the record file
NOLF on page 562	Closes the log file
P on page 565 [address]	Single assembly steps into program
RESTART on page 570	Restarts the loaded application
RD on page 568 [list|*]	Displays the content of registers

Table 20.2 List of Base Commands (*continued*)

Command, Syntax	Short Description
RS on page 572 register[=]value{,register[=]value}	Sets a register
S on page 573	Stops execution of the loaded application
STEPINTO on page 583	Steps to the next source instruction of the loaded application
STEPOUT on page 584	Executes program out of a function call
STEPOVER on page 585	Steps over the next source instruction of the loaded application
STOP on page 586	Stops execution of the loaded application
SAVEBP on page 575 on|off	Saves breakpoints
T on page 587 [address][,count]	Traces program instructions at the specified address
WB on page 595 range list	Writes bytes
WL on page 597 range lis	Writes longwords
WW on page 597 range list	Writes words

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 on page 491 contains all available Environment commands.

Table 20.3 List of Environment Commands

Command, Syntax	Short Description
ACTIVATE on page 498 component	Activates a component window
AUTOSIZE on page 509 on|off	Autosizes windows in the main window layout
BCKCOLOR on page 511 color	Sets the background color
CLOSE on page 519 component | *	Closes a component

Table 20.3 List of Environment Commands (*continued*)

Command, Syntax	Short Description
DDEPROTOCOL on page 525 ON|OFF|SHOW|HIDE|STATUS	Configures the Debugger/Simulator DDE protocol
FONT on page 540 'fontName' [size][color]	Sets text font
LOAD on page 551 applicationName	Loads a framework application (code and debug information)
LOADCODE on page 552 applicationName	Loads the code of a framework application
LOADSYMBOLS on page 552 applicationName	Loads debugging information of a framework application
OPEN on page 563 component [[x y width height][;][i|max]]	Opens a Windows component
SET on page 576 targetName	Sets a new target
SLAY on page 577 fileName	Saves the general window layout

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 on page 492 contains all available Component commands.

Table 20.4 List of Component Commands

Command, Syntax	Short Description
CMDFILE on page 520	Specify a command file state and full name
EXIT on page 535	Terminates the application
HELP on page 547	Displays a list of available commands
RESET on page 570	Resets statistics
SMEM on page 578 range	Shows a memory range
SMOD on page 579 module	Shows module information in the destination component

Table 20.4 List of Component Commands

Command, Syntax	Short Description
SPC on page 580 address	Shows the specified address in a component window
SPROC on page 581 level	Shows information associated with the specified procedure
VER on page 593	Displays version number of components and engine

Component Specific Commands

Component specific commands are associated with specific components. Table 20.5 on page 493 contains all available Component Specific commands.

Table 20.5 List of Component Specific Commands

<table>
<tr><th>Command, Syntax</th><th>Short Description</th></tr>
<tr><td>ADDXPR on page 499 “expression”</td><td>Adds a new expression in the data component</td></tr>
<tr><td>ATTRIBUTES on page 499 list</td><td>Sets up the display inside a component window</td></tr>
<tr><td>BASE on page 510 code | module</td><td>Sets the Profiler base</td></tr>
<tr><td>BD on page 512</td><td>Displays a list of all breakpoints</td></tr>
<tr><td>CF on page 517 fileName [;C][;NL]</td><td>Executes a command file</td></tr>
<tr><td>CLOCK on page 519 frequency</td><td>Sets the clock speed</td></tr>
<tr><td>COPYMEM on page 520 <Source addr range> dest-addr</td><td>Copies memory</td></tr>
<tr><td>CYCLE on page 521 on|off</td><td>Switches cycles and milliseconds in SofTrace component.</td></tr>
<tr><td>DETAILS on page 527 assembly|source</td><td>Sets split view</td></tr>
<tr><td>DUMP on page 529</td><td>Displays data component content</td></tr>
</table>

Table 20.5 List of Component Specific Commands (*continued*)

Command, Syntax	Short Description
E on page 530 expression [;O|D|X|C|B]	Evaluates a given expression
EXECUTE on page 534 fileName	Executes a stimulation file
FILL on page 535 range value	Fills a memory range with a value
FILTER on page 536 Options [<range>]	Selects the output file filter options
FIND on page 537 “string” [;B] [;MC] [;WW]	Finds and highlights a pattern
FINDPROC ON PAGE 538 ProcedureName	Opens a procedure file
FOLD on page 540 [*]	Folds a source block
FRAMES on page 542 number	Sets the maximum number of frames
GRAPHICS on page 547 on|off	Switches graphic bars on/off
INSPECTOROUTPUT on page 549 [name {subname}]	Prints content of Inspector to Command window
INSPECTORUPDATE on page 549	Updates content of Inspector
LS on page 557 [symbol | *][;C|S]	Displays the list of symbols
NB on page 560 [base]	Sets the base of arithmetic operations
OUTPUT on page 564 fileName	Redirects the coverage component results
PTRARRAY on page 567 on|off	Switches on /off the pointer as array display
RECORD on page 569 on|off	Switches on/off the frame recorder
SLINE on page 577 linenumber	Shows the desired line number
SAVE on page 574 range fileName [offset][;A]	Saves a memory block in S-Record format
SETCOLORS on page 576 ("Name") (Background) (Cursor) (Grid) (Line) (Text)	Changes the colors attributes of the "Name" channel from the Monitor component

Table 20.5 List of Component Specific Commands (*continued*)

Command, Syntax	Short Description
SREC on page 582 fileName [offset]	Loads a memory block in S-Record format
TUPDATE on page 588 on|off	Switches on/off time update for statistics
UNFOLD on page 591 [*]	Unfolds a source block
UPDATERATE on page 592 rate	Sets the data and memory update mode
ZOOM on page 598 address in|out	Zooms in/out a variable

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.

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:

```
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 1000000.

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:

```
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 on page 578 and SPC on page 580 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 on page 35. 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 on page 578 SPC on page 580, SPROC on page 581 and SMOD on page 579 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=command{,command}**

command= **SPC address | SMEM range | SMOD module** (without extension) **| SPROC numberAssociatedToProcedure | MARKS (ON|OFF)**

Example:

```
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 on page 592 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 on page 581, UPDATERATE on page 592 and SMOD on page 579 command descriptions for more detail about these 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 on page 592 command).

NOTE Also refer to SMEM on page 578, SPC on page 580 and SMOD on page 579 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}**)

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.

Example1:

```
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.

Example2:

```
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.**

BD

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:

```
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"
```

Sets a breakpoint at the address of the variable **counter** + 5, where the condition is **fib1 > fib2** and the command is **"bckcolor red"**.

```
in>BS &Fibo.c:Fibonacci+13
```

Sets a breakpoint at the address of the **Fibonacci** procedure + 13, where **Fibonacci** is a function of the `FIBO.C` module.

CALL

Executes a command in the specified command file.

NOTE If no path is specified, the destination directory is the current project directory.

Usage

CALL FileName [;C][;NL]

Components

Debugger engine.

Example:

```
in>cf \util\config.cmd
```

Loads the config command file.

CD

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`

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 on page 550 command), even if the **CMDFILE** type is set to **ON** (see also the LOG on page 553 command).

Components

Debugger engine.

Examples:

```
in>CF commands.txt
```

The `COMMANDS.TXT` file is executed. It should contain debugger commands like those described in this chapter.

Example Without “;C” Option:

if a `command1.txt` file contains:

```
bckcolor green
cf command2.txt
```

```
bckcolor white
```

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!
```

```
1!
done command2.txt

done command1.txt
```

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\myposloadfile.cmd"
```

The myposloadfile 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 on page 562) 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
```

```
00F066 DECX
00F067 DECX
```

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 on page 529, DL on page 528 command of a session).

Components

Debugger engine.

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: 5A Z

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:

```
in>DDEPROTOCOL ON
in>DDEPROTOCOL SHOW
in>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 on page 211.

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:

```
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.

```
/* 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 on page 589.

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
```

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 on page 523, DW on page 529, DL on page 528 command of a session).

Components

Debugger engine.

Example:

```
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.

```
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:

```
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 **Esc** key.

NOTE Open the Command Line component before executing this command to see the dumped code.

Usage

DW [address | **range]**

When **address** is an address constant expression, the address of the first word is displayed.

When **address** and **range** are omitted, the first word displayed is taken from the address following the last word displayed by the most recent **DB**, **DW**, or **DL** command, or from address **0x0000** (for the first DB on page 523, DW on page 529, DL on page 528 command of a session).

Components

Debugger engine.

Example:

```
in>DW 0x8000,4
```

8000: FE80 45FD 8043 2710

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:

```
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 on page 550 and LOG on page 553 command descriptions).

ELSE

The **ELSE** keyword is associated with the LF on page 550 command.

Usage

ELSE

Components

Debugger engine.

Example:

```
if CUR_TARGET == 1000          /* Condition */
      set sim
else set bdi                   /* Other Condition */
```

ELSEIF

The **ELSEIF** keyword is associated with the IF on page 548 command.

Usage

ELSEIF condition

where **condition** is same as defined in “C” language.

Components

Debugger engine.

Example:

```
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 on page 541 command.

Usage

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 on page 548 command.

Usage

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 on page 596 command.

Usage

ENDWHILE

Components

Debugger engine.

Example:

```
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:

```
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 on page 532 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 on page 498 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 on page 533 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 on page 526 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.

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 1000000.

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.

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
    ....
```

IF

The conditional commands (IF on page 548, ELSEIF on page 531, ELSE on page 531 and ENDIF on page 533) 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:

```
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
```

```
      Swap
      *  Name       Value  Address  Init...
      -  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 on page 562) 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 on page 553) 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.

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

LOADSYMBOLS [applicationName]

Components

Debugger engine.

Example:

```
LOADSYMBOLS 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 on page 550 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 **state** is **on** or **off.**

state is the new state of **type**:
When **ON**, enables logging of the type.
When **OFF**, disables logging of the **type**.

Components

Debugger engine.

Example:

```
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 **IF** block are logged. That is, a command file executed with the **CF** or **CALL** command without the **NL** option and with **CMDFILE** flag of the **LOG** command set to **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:

```
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:

```
!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**.

Example2:

When executing the following file:

```
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**.

Example3:

When executing the following file:

```
define i = 1
WHILE i < 3
   define i = i + 1
ls
ENDWHILE
```

The following log file is generated:

```
!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**.

Example4:

When executing the following file:

```
define i = 1
REPEAT
   define i = i + 1
ls
UNTIL i == 4
```

The following log file is generated:

```
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 on page 526 command to define symbols, and the UNDEF on page 589 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
```

```
User Symbols:
j            0x2 (2)
Application Symbols:
counter      0x80 (128)
fiboCount    0x81 (129)
j            0x83 (131)
n            0x84 (132)
fib1         0x85 (133)
fib2         0x87 (135)
fibo         0x89 (137)
Fibonacci    0xF000 (61440)
Entry        0xF041 (61505)
```

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.

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
```

```
Type    Addresses   Comment
-------------------------------------------------------
IO          0..  3F  PRU or TOP   TOP board resource or the PRU
NONE       40..  4F  NONE
RAM        50.. 64F  RAM
NONE      650.. 7FF  NONE
EEPROM    800.. A7F  EEPROM
NONE      A80..3DFF  NONE
ROM      3E00..FDFF  ROM
IO       FE00..FE1F  PRU or TOP   TOP board resource or the PRU
NONE     FE20..FFDB  NONE
ROM      FFDC..FFFE  ROM
COP      FFFF..FFFF  special ram for cop
RT MEM      0.. 3FF  (enabled)
-------------------------------------------------------
```

MS

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 on page 595) 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:

```
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 on page 560.

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

Notation	Meaning
0x----	Hexadecimal constant
0----	Octal constant

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 on page 561.

Table 20.7 Assembler Notation for Constant

Notation	Meaning
$----	Hexadecimal constant
@----	Octal constant
%----	Binary constant

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 on page 561. Otherwise, the command line interpreter cannot detect if you are specifying a number or a symbol.

Table 20.8 Base is 16: Constants Starting with Letter A, B, C, D, E or F

Notation	Meaning
5AFD	Hexadecimal constant $5AFD
AFD	Hexadecimal constant $AFD

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 on page 521 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 on page 550 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 on page 565 and T on page 587 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
```

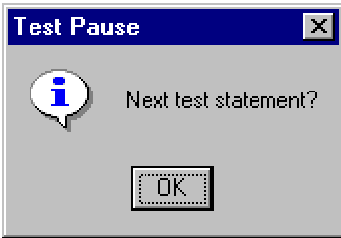
```
A=0x0 HX=0x450 SR=0x70 PC=0xF04E SP=0xFF

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 on page 566 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

PTRARRY 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.

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:

```
in>rd a hx
```

```
A=0x14
HX=0x2
```

Example2:

```
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 on page 515 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`:

```
...
CALL d:\demo\cmd2.txt
T
...
```

Iin file `d:\demo\cmd2.txt`

```
...
...
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 on page 587 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 on page 543 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 on page 582) 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 on page 513 command and to the Control Points on page 147 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
```

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:

```
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:

```
in>STEPINTO
```

```
STEP INTO
TRACED
```

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
```

```
STEP OVER
STARTED
RUNNING
STOPPED
```

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 on page 543 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.

T

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
```

```
TRACED
A=0x0 HX=0x7F02 SR=0x62 PC=0xF032 SP=0x44D

00F032 B787              STA    0x87
```

Contents of registers are displayed and current instruction is disassembled.

TESTBOX

Displays a modal message box shown in Figure 20.2 on page 588 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 on page 589 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 on page 526 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          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
```

```
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)
```

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 *
```

UNTIL

The **UNTIL** keyword is associated with the REPEAT on page 569 command.

Usage

UNTIL condition

Where **condition** is defined as in “C” language definition.

Components

Debugger engine.

Example:

```
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
```

```
HI-WAVE          6.0.27
HI-WAVE Engine   6.0.49
Source           6.0.20
Assembly         6.0.14
Procedure        6.0.10
Register         6.0.14
Memory           6.0.19
Data             6.0.27
Data             6.0.27
Simulator Target 6.0.17
Command Line     6.0.16
```

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.

WB

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 on page 559) 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 0x0FFFFF0F
```

This command fills up memory starting at address 0x2000 with the 0x0FFFFF0F longword value. The addresses 0x2000 to 0x2003 will be modified.

```
in>WL 0x2000, 2 0x0FFFFF0F
```

This command fills up the memory area 0x2000 to 0x2007 with the longword value 0x0FFFFF0F.

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.

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 on page 591 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

Abatron BDI Connection Commands

This section describes the *Abatron BDI* Connection-specific commands that are used when the *Abatron BDI* Connection is set.

The *Abatron BDI* Connection specific commands are:

- BANKREG on page 600
- BDI on page 600
- PROTOCOL on page 601
- RESET on page 602

Those commands are entered in the Abatron BDI Connection Command Files or in the **Command Line** component of the debugger.

This section describes each of the commands available for the *Abatron BDI* Connection. The commands are listed in alphabetical order. Each is divided into several topics.

Table 21.1 Command Description Parameters

Topic	Description
Short Description	Provides a short description of the command.
Syntax	Specifies the syntax of the command in a EBNF format.
Description	Provides a detailed description of the command and how to use it.
Example	Small example of how to use the command.

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 BANREG 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.

Example

```
RESET
```

Banked Memory Location-associated Commands

The following sections describe the Banked Memory Location Command Line commands which are used by the Abatron BDI Connection. These variables are:

BANKWINDOW on page 603

Those commands can be entered in the Abatron BDI 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 21.2 Command Descripton Parameters

Topic	Description
Short Description	Provides a short description of the command.
Syntax	Specifies the syntax of the command in a EBNF format.
Description	Provides a detailed description of the command and how to use it.
Example	Small example of how to use the command.

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 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 Banked Memory area** can be changed:

Listing 21.2 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


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.

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.

ICD-12 Memory Configuration Commands

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:

WB <address> <value> //write byte

WW <address> <value> //write word (16 bit)

WL <address> <value> //write long (32 bit)

NOTE Your ICD-12 connection includes command files that match the Freescale 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.

NVMC Commands

The following Flash Comands can be issued through the debugger Command component window, as shown in the figure below.

Figure 21.1 NVMC Commands In Command Window

```
Command
in>FLASH
FLASH parameters loaded for M68HC912DG128 from G:\HC12_D\PROG\FPP\mcu03C4.fpp

MCU clock speed: 8019000
Module Name      Address Range   Status
FLASH_4000        4000 -  7FFF   Enabled/Programmed - Unselected
FLASH_PAGE0       8000 -  BFFF   Enabled/Blank - Unselected
FLASH_C000        C000 -  FFFF   Enabled/Programmed/Protected - 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

in>
```

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|PROTOCOLON|PROTOCOLOFF|SKIPSTATUS
     ON|SKIPSTATUSOFF]

|[NVMFREQUENCY <frequency in Hz>]

|[NVMIF2RELOCATE <address>]

|[NVMIF2WORKSPACE <address> <address>]

|[INIT <fileName> | AUTOID]
```

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).

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 SAVECONTEX backs up current SRAM content into a buffer.

FLASH LOADCONTEX 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 follwos 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.

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 = {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: 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/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
```

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).

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 on page 617 contains all available Environment commands.

Table 21.3 Full Chip Simulator Commands

Command, Syntax	Short Description
SETCPU on page 629 ProcessorName	Sets a new cpu simulator
RESETCYCLES on page 625	Resets Simulator CPU cycles counter
RESETMEM on page 626	Resets all configured memory to 'undefined'
RESETRAM on page 627	Resets RAM to 'undefined'
RESETSTAT on page 627	Resets the statistical data
SHOWCYCLES on page 629	Returns executed Simulator CPU cycles

FCS-Associated Component-specific Commands

Component specific commands are associated with specific components supported by the Full Chip Simulator. Table 21.4 on page 618 contains all available Component Specific commands.

Table 21.4 List of Component-specific Commands

Command, Syntax	Short Description
ADCPORT on page 619 (address | ident) (address | ident) (address | ident)	Sets the ports addresses used by the Adc_Dac component.
ADDCHANNEL on page 619 ("Name")	Creates a new channel "Name" for the Monitor component.
CPORT on page 620 (address | ident) (address | ident) (address | ident) (address | ident)(address | ident)	Sets the 5 port addresses and control port address of the IO_Ports component
ITPORT on page 621 (address | ident) (address | ident)	Sets the line and column port addresses of the IT_Keyboard component
ITVECT on page 621 (address | ident)	Sets the interrupt vector port address of the IT_Keyboard component.
KPORT on page 622 (address | ident) (address | ident)	Sets the line and column port addresses of the Keyboard component
LCDPORT on page 622 (address | ident) (address | ident)	Sets the data port and the control port address of the Lcd component
LINKADDR on page 623 (address | ident) (address | ident) (address | ident)(address | ident)(address | ident)	Sets the components internal port addresses used with the IO_Ports as memory buffers
PBPORT on page 623 (address | ident)	Sets the port address of the Push_Buttons component
PORT on page 624 address	Sets the LED components port address
SEGPORT on page 628 (address | ident) (address | ident	Set the display selection port and the segment selection port addresses of the 7-Segments display component.
SETCONTROL on page 628 ("Name") (Ticks) (Pixels)	Changes the number of ticks and pixels for the "Name" channel from the Monitor component
WPORT on page 630 (address | ident) (address | ident	Sets the ports addresses of the Wagon component

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.

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_Buttons 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_Buttons component.

Example:

```
in>PBPORT 0x100 0x200
```

The ports of the Push_Buttons 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 */
```

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>SHOWCYCLES
 133801
```

```
in>RESETCYCLES
in>SHOWCYCLES
```

```
 0
```

```
in>RESETCYCLES 5500
in>SHOWCYCLES
```

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.

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.

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 on page 625 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>SHOWCYCLES
 133801
```

```
in>RESETCYCLES
in>SHOWCYCLES
```

```
 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.

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 on page 633
- Chapter 5.2 Connection-specific Environment Variables on page 655

22

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 on page 634
- Local Configuration File (usually project.ini) on page 637
- ABSPATH: Absolute Path on page 644
- DEFAULTDIR: Default Current Directory on page 645
- ENVIRONMENT=: Environment File Specification on page 646
- GENPATH: #include “File” Path on page 647
- LIBRARYPATH: ‘include <File>’ Path on page 648
- OBJPATH: Object File Path on page 649
- TMP: Temporary directory on page 650
- USELIBPATH: Using LIBPATH Environment Variable on page 651
- Search Order for Source Files on page 652
- Debugger Files on page 652

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/.hidefaults` 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 on page 645.

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 on page 645 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:

```
C:\WINDOWS\MCUTOOLS.INI
D:\INSTALL\PROG\MCUTOOLS.INI
```

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 on page 638. All default layout related parameters are stored in section **[DEFAULTS]**.

Listing 22.1 Example Content of PROJECT.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 on page 14 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.>

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:

Tooolbar = (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.

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:

```
OPTIONS=\
-W2 \
-Wpd
```

This is the same as:

`OPTIONS=-W2 -Wpd`

Be careful when using the line continuation character with paths. For example:

```
GENPATH=.\
TEXTFILE=.\txt
```

Will result in:

`GENPATH=.TEXTFILE=.\txt`

To avoid such problems, use a semicolon';' at the end of a path, if there is a '\' at the end:

```
GENPATH=.\;
TEXTFILE=.\txt
```

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 on page 643.

Table 22.1 Environment Variable Details

Topic	Description
Tools	Lists of other tools that are using this variable
Synonym	Fore 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.
Syntax	Specifies the syntax of the option in EBNF format.
Arguments	Describes and lists optional and required arguments for the variable.
Default	Shows the default setting for the variable or none.
Description	Provides a detailed description of the option and how to use it.
Example	Gives an example of usage and effects of the variable where possible. The examples show an entry in the **default.env** file for PC.
See also	Names related sections.

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 on page 635
Global Initialization File (MCUTOOLS.INI - PC Only) on page 636

ENVIRONMENT=: 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 on page 635 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 on page 648.

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 on page 647 and finally in directories given by LIBRARYPATH: ‘include <File>’ Path on page 648.

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 on page 647

Environment variable USELIBPATH: Using LIBPATH Environment Variable on page 651

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 on page 647 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 on page 635

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 on page 648 is used to look for system header files <*.h>.
"NO", "OFF": The environment variable LIBRARYPATH: ‘include <File>’ Path on page 648 is not used.

Default

ON

Description

This environment variable allows a flexible usage of the LIBRARYPATH: ‘include <File>’ Path on page 648 environment variable, because LIBRARYPATH: ‘include <File>’ Path on page 648 may be used by other software (for example, version management PVCS).

Example:

```
USELIBPATH=ON
```

See also:

Environment variable LIBRARYPATH: ‘include <File>’ Path on page 648

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 .pjt 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 .pjt 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 .pjt 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

File Extension	Description
*.ABS	Absolute framework application file e.g., fibo.abs
*.ASM	Assembler specific file e.g., macrodem.asm
*.BBL	Burner Batch Language file e.g, fibo.bbl
*.BPT	Debugger Breakpoint file e.g., fibo.bpt
*.C *.CPP	C and C++ source files
*.CHM	Compiled HTML help file
*.CMD	Command File Script, for example, Reset.cmd
*.CNF	Specific cpu configuration file
*.CNT	Help Contents File, for example, cxa.cnt
*.CPU	Central Processor Unit Awareness file
*.DBG	Debug listing files, for example, Fibo.dbg
DEFAULT.ENV	Debugger Default Environment file.
*.DLL	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.
*.H	Header file
HIWAVE.EXE	The Debugger for Windows executable program.
*.HWL	Debugger Layout file, for example, default.hwl
*.HWC	Debugger Configuration file (project.hwc)
*.EXE	Other Windows executable program, for example, LINKER.EXE
*.FPP	Flash Programming Parameters files (CPU specific) for example, mcu0e36.fpp
*.HLP	Application Help file, for example, Hiwave.hlp

Table 22.2 Debugger File Extensions

File Extension	Description
*.IO	I/O's simulation file, for example, sample11.io
*.ISU	Uninstall Application File
*.PJT	Debugger configuration Settings File, for example, Project.pjt
*.INI	Debugger configuration Settings File, for example, Project.ini
*.LST	Assembler Listing File, for example, fibo.lst
*.MCP	Freescale CodeWarrior IDE project file
*.MAK	Make file, for example, demo.mak
*.MAP	Mapping file, for example, macrodem.map
*.MEM	Memory Configuration file, for example, 000p4v01.mem
*.MON	Firmware loading, file for allowing to load a specified target, for example, Firm0508.mon
*.O	Object file code, for example, Fibo.o
*.PDF	Portable Document Format file.
*.PRM	Linker parameter file, for example, `fibo.prm`
Project.Ini	Debugger Project Initialization File
*.REC	Recorder File
*.REG	Register Entries files, for example, mcu081e.reg
*.SIM	CPU Awareness file, for example, st7.sim
*.SX	S-Record file, for example, fibo.sx
*.TXT	General Text Information file.
*:TGT	Target File for the Debugger, for example, xtend-g3.tgt
*.WND	Debugger Window Component File, for example,, recorder.wnd
*.XPR	Debugger User Expression file, for example, Fibo.xpr

23

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.

Abatron BDI Connection Environment Variables

This section describes the environment variables which are used by the *Abatron BDI* Connection.

The *Abatron BDI* Connection specific environment variables are:

- BDICONF on page 656
- COMDEV on page 657
- COMPRESS on page 658
- SHOWPROT on page 658
- SKIPILLEGALBREAK on page 659
- VERIFY on page 660

These variables are stored in the `[BDIK]` section from the project file.

Listing 23.1 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"
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 *Abatron BDI* Connection. The variables are listed in alphabetical order and are divided into several topics.

Table 23.1 Variable Description Parameters

Topic	Description
Short Description	Provides a short description of the variable.
Syntax	Specifies the syntax of the variable in a EBNF format.
Default	Shows the default setting for the variable.
Description	Provides a detailed description of the variable and how to use it.
Example	Small example of how to use the variable.

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 Box Configuration Tool Path** edit box of the **Setup Dialog Box**. The **BDI Box 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 **Abatron BDI | 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
```

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** edit box of the Communication Device Specification Dialog Box on page 436.

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 on page 440.

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 Dialog Box on page 436.

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

```
SKIPPILLEGALBREAK=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 on page 440.

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 on page 440.

Example

`VERIFY=1`

Banked Memory Location-associated Environment Variables

The following sections describe the Banked Memory Location environment variables which are used by the Abatron BDI connection. These variables are:

BANKWINDOWn on page 661

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 23.2 Variable Description Parameters

Topic	Description
Short Description	Provides a short description of the variable.
Syntax	Specifies the syntax of the variable in a EBNF format.
Default	Shows the default setting for the variable.
Description	Provides a detailed description of the variable and how to use it.
Example	Small example of how to use the variable.

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.

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 - PPage Tab on page 448.

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 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 437
.abs file 45
.cmd 56
.FPP file loading 474
.hidefaults 635, 645, 646, 650
.hwl 639
.HWP 15
.hwp 639
.INI 15
.PJT 15
.rec 102
.WND 45
.wnd 35
.xpr file 67

Numerics

16-Bit Modulus Down-Counter 318

A

A 498
ABATRON Configuration tool 430
About Box 34
About True Time Simulator and Real Time Debugger 33
ABSPATH 644
ACTIVATE 498
ADCPORT 619
Add New Instrument 133, 134
ADDCHANNEL 619
Address 80, 83
Address... 50
ADDXPR 499
Align 135
All Text Folded At Loading 122
Analog 136
Analog to Digital Converter 288
AND Mask 139, 140, 142
Application
    Assembly Step 199
    Embedded 5
    Loading 195
    Starting 196
    Step In 197
    Step Out 198
    Step Over 198
    Stopping 196
    Target 5
Application loading 475
Arrange Icons 33
ASCII 83
Assembly Step 23
Assembly Step Out 23
Assembly Step Over 23
AT 509
ATTRIBUTES 499
Auto 106
auto configure 263
Auto on Access 265
Auto on Load 265
Auto select according to MCU Id 475
Automatic 68, 82
AUTOSIZE 509

B

Background Color 30
Background Debug Mode 236, 428
Backgroundcolor 136, 137
Banked hardware breakpoint 441
Banked Memory Location dialog 447
Banked Memory Location Target commands 602
banked memory model 317
Banking on CPU12 438
BANKREG 600
Banks 318, 438
BANKWINDOW 603
BANKWINDOWn 661
Bar 136
Barcolor 138
Bardirection 138
BASE 510
BC 510
BCKCOLOR 511
BD 512
BDI 600
    Abatron setup 430
    Configuration 430
    Interface 429

Interface setup 430
BDI Firmware 431
BDI Initialization List 432
BDI Startup Init List 432
BDI Working Mode 434
BDI1000 427
BDICONF 656
BDI-HS 427
BDIK
BDIK Flash Programming with BDI 445
BDIK Target Interface Dialogs 438, 439, 440
BDIK Terminal Emulation with BDI 443
Menu Entries 437
BDIK | Command Files 438
BDIK | Communication... 437, 439
BDIK | Connect 437, 439
BDIK | Help 438
BDIK | Load... 437, 438
BDIK | Reset 437
BDIK | Set Bank... 438
BDIK | Setup... 440
BDIK Flash Programming with BDI 445
BDIK Target Interface Dialogs 438, 439, 440
BDIK Terminal Emulation with BDI 443
BDM connector/port 429
BDM port 236, 428
Bin 70, 81, 106, 200
Binary 200, 203
Bit Reverse 82, 106
Bitnumber to Display 141
blank 472, 473
BLCD 285
Blocks 470
Bottom 135
Bounding Box 137
Box configuration 430
BREAKPOINT 442
Breakpoint 49, 112
BREAKPOINT 442
Checking condition 151
Command 160
Conditional 158, 176
Counting 157, 176
Definition 147
Deleting 159
Multiple selection 151
Permanent 147
Position 154
Temporary 147, 155
breakpoint 441, 659
Breakpoint banked hardware 441
Breakpoint with Register Condition 158, 159
Breakpoints... 23
BRLD 320
BS 513
BSPL 320
BTST 320
Bus Trace 422
Byte 81
Byteflight 285

C

-C 10
CALL 317, 515
Call Chain 96
Capture 318
Capture / Compare Timer 319
Capture Stimulation 318
Cascade 33
CD 516
CF 517
CFORC 319
CLOCK 519
Clock and Reset Generator 294
Clone Attributes 135
CLOSE 519
-Cmd 10
CMDFILE 520
CodeWarrior Integration 209
Color if 142
Color if Bit 141
COMDEV 657
Command 145
Syntax 487
Command File Dialog 27, 262
Command File menu entry 27, 262
Command File Playing 56
Command Files 438
Command Line 9
Commands 599, 602, 608
Communication 429, 437
Communication Device Specification dialog 439
COMn 439

Compare 318
COMPLEMENT
- DATA Component 504
- Memory Component 505
- Register Component 501

Component
- Assembly 49, 195, 196
- Associated Menus 34
- Command Line 54
- Coverage 58
- CPU 45
- DAC 63
- Data 65, 195, 196, 199
- Framework 6
- Inspector 124
- IO_Led 325
- LED 327
- Main Menu 35
- Memory 78, 205
- MicroC 90
- Module 94
- Phone 329
- Pop Up Menu 36
- Profiler 98
- Recorder 102
- Register 105, 196, 203
- SoftTrace 108
- Source 111, 195, 196
- Stimulation 357
- Target 46
- Terminal 207
- VisualizationTool 131
- Window 45

Components File 35
COMPOPTIONS 637
COMPRESS 658
Compression 441, 658
Configuration 16
Configuration file 432
Configuration Tool 430
Configure BDI Box 430
Configure Box... 438
Connect 437
Continue on illegal break 438, 441, 659
Control Point
- Definition 147
- Dialogs 147

Copy 134
COPYMEM 520
CopyMem 80
Copyright 33
Coverage 428
CPORT 620
CPU
- Cycle 13
- cycle 105

CPU cycles (64 bits) 260
CPU12 427, 443
CR 521
Cross-debugging 5
Ctrl+E 133
Ctrl+L 134
Ctrl+S 134
CTRL-P 136
Current Directory 635, 645
Customize 18
Cut 135
CYCLE 521
Cycle 109

D

DAC
- Configure the file types 216
- Configuring 213
- Configuring the tools 221
- Database directory 215
- Debugger Interface 224
- Debugger name 230
- IDE 213
- library path 216
- Ndapi.dll 229
- new project 214
- Preprocessor | Header Directories 217
- Preprocessor | Preinclude file 218
- Project root directory 215
- Referential project root directory 215
- Requirements 213
- rue Time Simulator and Real Time Debugger project file 227
- Source 217
- Synchronized debugging 229
- Troubleshooting 229
- User help file 215
- working directories 214

DASM 522
Data compression 441, 658
DB 523
DDE
  HI-WAVE server 211
DDEPROTOCOL 525
Debug Module 291
Debugger Start Option -C 10
Debugger Start Option -Cmd 10
Debugger Start Option -ENVpath 11
Debugger Start Option -Instance=%currentTargetName 10
Debugger Start Option -Nodefaults 10
Debugger Start Option -Prod 10
Debugger Start Option -T 9
Debugger Start Option -Target 10
Debugger Start Option -W 10
Debugger, starting 8
Debugging 5
Debugging Memory Map 457
Dec 70, 81, 106, 200
Decimal 200
Decimalmode 142
Default target 447
DEFAULT.ENV 634, 635, 645, 646, 650
default.mem 263
DEFAULTDIR 645
DefaultDir 192
DEFINE 526
DELCHANNEL 620
Delete Breakpoint 52, 116
Demo Version Limitations 338
DETAILS 527
Dialog 471, 472
Direct commands 445
Disable Breakpoint 52, 116
disabled 472
Disabling 473
Display 80
Display Absolute Address 50
Display Adress 50
Display Adress Dialog 83
Display Bank Memory Location dialog at connection 450
Display Code 50
Display Headline 136
Display Scrollbars 136
Display Symbolic 50
Display Version 142
Displayfont 144
DL 528
Down-Counter 318
Download Mode and Data Transfer Verification 441
DPAGE 449
DPAGE Banked Memory Area 449
Drag Out 338
Dragging 37, 38
Driving True Time Simulator and Real Time Debugger trough DDE 211
Drop Into 338
DUMP 529
DW 529

E

E 530
Editing
  Memory 205
  Register 203
  Variable 201
Editmode 133, 134, 136
Editor 66
EEPPROM 290
ELSE 531
ELSEIF 531
Enable Breakpoint 52, 116
enabled 472
Enabling 473
End 472
ENDFOCUS 532
ENDFOR 533
ENDIF 533
ENDWHILE 534
Enhanced Capture Timer 296, 318
Environment
  ABSPATH 644
  DEFAULTDIR 645
  ENVIRONMENT 634
  File 634
  GENPATH 647, 649
  HIENVIRONMENT 646
  HIPATH 647, 649
  LIBPATH 648, 651
  LIBRARYPATH 648

OBJPATH 649
TMP 650
USELIBPATH 651
Variable 643
Environment variables 447
-ENVpath 11
EPAGE 449
EPAGE Banked Memory Area 449
EQUAL Mask 139, 142
Erase 445
Erase Flash 416, 422
Erasing 473
Ethernet 236, 428, 429
EXECUTE 534
EXIT 535
Exit 16
Explorer 635
Expression Command File 67
Expression Editor 66
External Bus Interface 292

F

FE 321
FEE28 478
FEE32 479
Field Description 144, 145
File
Environment 634
File Manager 635
Filename 139
FILL 535
Fill Memory Dialog 83
FILTER 536
FIND 537
Find 118, 120
Find Procedure 118, 121
FINDPROC 538
Firmware 431
FLASH 608
Commands 608
Disabling 473
Enabling 473
Loading 475
Module 471
Module selecting 473
Operations 473
Protecting 473
Select 471
Unprotecting 473
Unselect 471
Flash 290
Flash programming 445
FLASH SELECT 474
FLASH UNSELECT 474
Flash.Erase 445
Flash.Idle 445
Flash.Load 445
FLASH_4000 479
FLASH_B32 478
FLASH_C000 479
FLASH_PAGE0 479
FLASH_PAGE1 479
FLASH_PAGE2 479
FLASH_PAGE3 480
FLASH_PAGE4 480
FLASH_PAGE5 480
FLASH_PAGE6 480
FLASH_PAGE7 480
FLEXlm 6
Float 106
FOCUS 539
FOLD 540
Fold 122
Fold All Text 122
Folding 114
Mark 114
Folding Menu 121
Foldings 118
FONT 540
Fonts 30
FOR 541, 554
Format 80, 200, 203
Format mode 144
Format... 68
FPP Browse 475
FPP directory 474
FPRINTF 542
FRAMES 542
Frames 108
Frozen 68, 71, 82

G

G 543
GENPATH 647, 649

Global 69
Global Variable
Displaying 199
GO 544
Go to Line 117, 119, 120
Graphic bar 59, 98
GRAPHICS 547
Grid Color 136
Grid Mode 136
Grid Size 136
GUI Graphical User Interface 470

H

Halt 22
HALTED 442
Hardware 5, 478
hardware breakpoint 441, 659
HC08 Full Chip Simulation 259
HC12B32 478
HC12D60 478
HC12DG128 479
HCS08 Serial Monitor Connection 467
Height 136
HELP 547
Help 438
Help Topics 33
Hex 69, 70, 81, 106, 200, 203
Hexadecimal 200, 203, 206
Hide Headline 18
Hide Tile 18
HIENVIRONMENT 646
High Display Value 138, 140, 145
Highlights 427
HIPATH 647
HiTOP Ready 441
Horiz. Text Alignment 144
Horizontal Size 135
How To ... 191
How to Load 477

I

I/O 428, 443
ICLAT 318
IDF 211
IDLE 321
Idle 445
IF 548, 554
ILIE 320
I-LOGIX 90
ILT 320
inDART-HC08 > About 416
inDART-HC08 > MCU Configuration 414
inDART-HC08 > User's Manual 416
Indicatorcolor 138, 141
Indicatorlength 138
init.cmd 194
Initialization List 432
INITRG 317
Input 318
INSPECTORUPDATE 549
-Instance=%currentTargetName 10
Instruction Syntax 488
Inter-IC Bus 286
interruption 336
IPATH 649
ITPORT 621
ITVECT 621

J

J1850 Bus 285

K

keyword DAC
True Time Simulator and Real Time Debugger project file 227
Kind of Port 137
KPORT 622

L

Layout 7, 639
Layout - Load/Store 33
LCDPORT 622
Left 135
LF 550
LIBPATH 651
LIBRARYPATH 648
Line Continuation 642
LINKADDR 623
List Transmission 434
LOAD 551
Load 445
Load Application 15
Load Layout 134
Load Target 25, 260

Load... 437
LOADCODE 552
Loading an Application 195
Loading an application 437
Loading error 476
Loading problems 476
Loading the BDIK Target Interface 434
LOADSYMBOLS 552
Local 69
Local Variable
    Displaying 199
Locked 68, 71
LOG 553
LOOPS 320
Low Display Value 138, 140, 145
LS 557
Lword 81

M

M 320
MainFrame 639
Manual Configuration 265
Marks 118
MC9S12A32 276
MC9S12A64 276
MC9S12C32 277
MC9S12D32 277
MC9S12D64 278
MC9S12DB128A 278
MC9S12DB128B 279
MC9S12DG128B 279
MC9S12DG256B 280
MC9S12DJ128B 280
MC9S12DJ256B 281
MC9S12DJ64 281
MC9S12DP256B 282
MC9S12DP512 282
MC9S12DT128B 283
MC9S12DT256B 283
MC9S12XDP512 284
MC9S12XDT512 285
MCCNT 318
MCCTL 318
MCU Communication 414
mcu03c1.fpp 478
mcu03c3.fpp 478
mcu03c4.fpp 479
mcuId 474
MCUTOOLS.INI 192, 636
MEM 558
Memory
    Dump 78
    Word 78
Memory banking 428
Memory Configuration Modes 264
Memory Expansion Register 317
memory model 317
Menu
    Help 33
    Run 21
    Target 24, 29
    View 18
    Window 31
MicroC 90
misaligned access 263
Mode 80
Module 94
Modules 471
Modulus Down-Counter 318
Monitor Communication 422
MONITOR-HCS08 > Bus Trace 422
MONITOR-HCS08 > Erase Flash 422
MONITOR-HCS08 > Monitor Communications... 422
MONITOR-HCS08 > Select Derivative 422
MONITOR-HCS08 > Trigger Module Settings... 422
MONITOR-HCS08 > Vector Mirroring Setup... 422
Motorola Scalable CAN 285
MS 559
ms 109
Multiplexed External Bus Interface 293

N

Name 472
NB 560
NETWORK 439
New 15
NF 321
No Link To Target 441
NOCR 562
-Nodefaults 10
NOLF 562

NV_PARAMETER_FILE 474

O

Object Info Bar 35
OBJPATH 649
Oct 70, 81, 106, 200
Octal 200
OP_SetValue 318
OPEN 563
Open Component 30
Open Configuration 15
Open Memory Block 266
Open Source File 117
Options 192
    Pointer As Array. 68
Options - Autosize 33
Options - Component Menu 33
OR 321
OSEK Kernel Awareness 183
OSEK ORTI 183
OSEK RTK Inspector 184
OSPARAM.PRM 178
Outlinecolor 142
OUTPUT 564
Output 318

P

P 565
Parallel port 405
Paste 135
PATH 641
Pause 102
PAUSETEST 566
PBPORT 623
PE 320
Percentage 58, 98
Periodic Interrupt Timer 299
PERIODICAL 358
Periodical 68, 82
PF 321
pins 318
PIX0 318
PIX1 318
PIX2 318
Play 102
Pointer as Array 68, 72
PORT 624
Port 405
Port Integration Module 294
Port to Display 137
PORTT 318
PORTTBitx 318
Postload command file 29
postload.cmd 207
PPAGE 318, 448
PPAGE Banked Memory Area 448
PR0 319
PR1 319
PR2 319
Preference panel 17
Preferences dialog 16
Preload command file 28
preload.cmd 207
PRINTF 567
Procedure Chain 96
-Prod 10
Profiling 428
Program loading 475
programmed 472
Project 639
PROJECT.INI 24, 638
project.ini 474, 475, 637
Properties 134
protected 472
Protecting 473
PROTOCOL 601
PSMODE 624
PT 320
PTRARRAY 567
Pulse Width Modulator 299
PVCS 651

R

R8 321
RAF 321
RAMs 430
RD 568
RDRF 321
RE 320
real time 5
Real Time Kernel Awarness 177
Real Time Kernels 177
RECORD 569
Record 102

REGBASE 625
Register 105
Register Block 317
Register values 158, 159, 168
Registration 34
Relative Mode 145
Remove 134
REPEAT 554, 569
Replay 103
Requirements 428
RESET 442, 570, 602
Reset 437
Reset command file 28
Reset Target 25, 260
reset.cmd 207
RESETCYCLES 625
RESETMEM 626
RESETRAM 627
RESETSTAT 627
RESTART 570
Restart 22
RETURN 571
RHAPSODY 90
RIE 320
Right 135
ROMs 430
RS 572
RS-232 serial communication 236, 428
RSRC 320
RTC 317
Run To Cursor 52, 117
RUNNING 441
RWU 320

S

S 573
SAVE 574
Save Configuration 16
Save Configuration As 16
Save Layout 134
Save Memory Block 266
SAVEBP 575
SBK 320
SBR 320
SC0BDH 320
SC0BDL 320
SC0CR1 320
SC0CR2 320
SC0DRH 321
SC0DRL 321
SC0SR1 320
SC0SR2 321
SC1BDH 320
SC1BDL 320
SC1CR1 320
SC1CR2 320
SC1DRH 321
SC1DRL 321
SC1SR1 320
SC1SR2 321
SCIInput 320, 321
SCIInputH 320, 321
SCIOutput 321
SCIOutputH 321
Scope... 68
SDI 46
search order 652
Searching Order
    Assembly source files 652
    C source files 652
    Object files source files 652
SEGPORT 628
Select Derivative 422
Selecting 471, 473, 474
Send to Back 135
Send to Front 135
Serial Communication Interface 286, 320
Serial Peripheral Interface 288
SerialInput 320, 322
SerialOutput 322
SET 576
Set Bank... 438
Set Breakpoint 52, 116
Set Target 30
Set Zero Base 110, 357
SETCOLORS 576
SETCONTROL 628
SETCPU 629
Setcpu command file 262
Setup 134
Setup dialog 440
Setup... 438
Show Breakpoints 52, 117
Show Location 52, 117

Show Protocol 440
SHOWCYCLES 629
SHOWPROT 658
Simulation 5, 428
Single Step 23
Size 135
Size of Port 137
SKIPILLEGALBREAK 659
SLAY 577
SLINE 577
Sloping 142
Small Borders. 18
SMEM 578
SMOD 579
SofTec HC08 - Communication Settings Dialog Box 414
SofTec HC08 - First Steps From Within an Existing Project 253, 255
SofTec HC08 - First Steps Using the Stationery Wizard 237, 247
SofTec HC08 - inDart-HC08 Connection Menu Options 413
SofTec HC08 - MCU Configuration Dialog Box 254
SofTec HC08 - Set Connection Dialog Box 254
SofTec HC08 - Technical Considerations 235, 236, 413
Softec HC08 Connection 235, 413
Softec HCS08 Connection 457
SPC 580
Splitting View 59
SPROC 581
SREC 582
ST.SectionTitle 1, 3, 233, 235, 259, 413, 417, 451, 457, 467, 485, 487, 599, 631, 633, 655
ST1619-HDS
    Postload command file 29, 262
    Preload command file 28
    Reset command file 28
    Startup command file 28
Start 103, 472
Start/Continue 22
Starting an Application 196
startup 637
Startup command file 28
Startup Init List 432
startup.cmd 207
State 472
States 472
Statistics 99
Status Bar 13, 18
Status Bar Information for the BDIK Target Interface 441
Status Message 441
    BDI Ready 441
    HALTED 442
    No Link To Target 441
    Reset 442
    RUNNING 441
Status register bits 105
Step In 197
    Assembly Instruction 199
    Source Instruction 197
Step Out 23, 197
    Function Call 198
Step Over 23, 197, 198
STEPINTO 583
STEPOUT 584
STEPOVER 585
STEPPED 442
STEPPED OVER 442
Stepping and Breakpoints Messages 442
Stepping Message
    STEPPED 442
    STEPPED OVER 442
    STOPPED 442
    TRACED 442
Stimulation 318
STOP 586
STOPPED 442
Stopping an Application 196
Symbolic 69, 70, 200

T

-T 9
T 587
T8 321
-Target 10
Target commands 599
Target Interface Dialogs 447
TC 321
TCIE 320
TCNT 319
TCRE 318, 319

TCTL1 319
TCTL2 319
TCTL3 319
TCTL4 319
TCx 319
TDRE 320
TE 320
termbgnd.c 443
Terminal 443
Terminal Address 443
Terminal area 443
Terminal Emulation 443
Terminal work space 443
TESTBOX 588
Text 141
Text Mode 144
Textcolor 144
TFLG1 319
TFLG2 319
TIE 320
Tile 33
Timer 318, 319
Timer Module 301
Timer Update 60
TIOS 319
TMP 650
TMSK1 319
TMSK2 319
TOI 319
Toolbar 12, 18
  Customizing 18
ToolTips 118
ToolTips Activation 112
ToolTips format 112
ToolTips mode 112
Top 135
TRACED 442
Trigger Module Settings... 422
True Time Simulator and Real Time Debugger
  Configuration 192
  Default Layout Configuration 638
  Demo Version Limitations 6
  Drag and Drop 38
  Engine 5
  Layout 639
  Project 639
  project.ini 638
  Running from a command line 9
  Smart User Interface 37
  Tool tip 12
  User Interface 37
  Using on Windows 95 or Windows NT 4.0/ WIN2000 192
TUPDATE 588

U

UDec 70, 82, 106, 200
UNDEF 589
UNFOLD 591
Unfold 122
Unfold All Text 122
Unfolding 114
  Mark 114
unprotected 472
Unprotecting 473
Unselecting 471, 474
Unsigned Decimal 200
UNTIL 592
UPDATERATE 592
USELIBPATH 651
User 69
User's Manual 416

V

VA 598
Variable 447
  Address 202
  Displaying Global Variables 199
  Displaying Local Variables 199
  Editing Value 201
  Format 65
  Local and Global 65
  Mode 68
  Scope 65
  Showing Location 202
  Type 65
  Value 200
Vector Mirroring Setup... 422
VER 593
VERIFY 660
Version number 33
Vert. Text Alignment 144
Vertical Size 135
VisualizationTool

7 Segment Display 141
Analog 137
Bar 138
Bitmap 139, 140
Demo 146
Demo limitation 146
Demo Version Limitations 146
DILSwitch 140
Instrument 136
Knob 140
LED 141
Setup 136
Switch 142
Text 143
Voltage Regulator 290

W

-W 10
WAIT 594
WAKE 320
WATCHPOINT 443
Watchpoint
Checking condition 164
Command 169
Conditional 168, 176
Counting 167, 176
Definition 147
Deleting 169, 175
Read 165, 174
Read, Write 148
Read/Write 166, 175
WATCHPOINT 443
Write 166
WB 595
WHILE 554, 596
Width 137
Windows 635
WinEdit 635
WL 597
WOMS 320
Word 81
Word size 80
WorkDir 192
Working Mode 434
WorkingDirectory 192
WPORT 630
WW 597

X

X-Position 136

Y

Y-Position 136

Z

ZOOM 598
Zoom in 68