



Freescale Semiconductor, Inc.

CodeWarrior® Debugger

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: <2/12/04>

For More Information: www.freescale.com



Freescale Semiconductor, Inc.

©Metrowerks, Inc., 1993, 2004; All Rights Reserved.

Documentation stored on the compact disks may be printed by licensee for personal use. Otherwise, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from Metrowerks, Inc.

Metrowerks, the Metrowerks logo, and CodeWarrior are registered trademarks of Metrowerks, Inc.

PowerPC® is a registered trademark of IBM, Inc.

All other trademarks belong to their respective owners.

How to Contact Metrowerks:

U.S.A.	Metrowerks Corporation 9801 Metric Blvd., Suite #100 Austin, TX 78758 - U.S.A.
Europe	Metrowerks Europe Riehenring 175 - CH-4058 Basel (Switzerland)
ASIA/PACIFIC	Metrowerks Japan - Shibuya Mitsuba Building 5F - Udagawa-cho 20-11, Shibuya-ku - Tokyo 150-0042 Japan
World Wide Web	http://www.metrowerks.com
Registration Information	http://www.metrowerks.com/register mailto:register@metrowerks.com
Desktop Technical Support	http://www.metrowerks.com/support/desktop/ mailto:cw_support@metrowerks.com
Embedded Technical Support	http://www.metrowerks.com/support/embedded/ U.S.A: mailto:cw_emb_support@metrowerks.com Europe: mailto:support_europe@metrowerks.com Asia/Pac mailto:j-emb-sup@metrowerks.com
Sales, Marketing, & Licensing	mailto:sales@metrowerks.com
Ordering	Voice: (800) 377-5416 Fax: (512) 873-4901

For More Information: www.freescale.com



Freescale Semiconductor, Inc.

For More Information: www.freescale.com



Freescale Semiconductor, Inc.

For More Information: www.freescale.com



Table of Contents

1 Important Notice	17
Copyrights	17
Trademarks	17
Warranty	18
2 Overview	19
About This Guide	19
Highlights	20
Read the Release Notes	20
Document Conventions	20
3 Introduction	23
What Is the Simulator/Debugger?	23
What Is a Simulator/Debugger Application?	24
What Is a Simulator/Debugger Execution Framework?.	25
Understanding the Simulator/Debugger Concept	26
The Simulator/Debugger Execution Framework	26
Objects and Services.	27
Framework Components	27
Demo Version Limitations Components	28
4 Simulator/Debugger User Interface	29
Introduction.	29
Application Programs	30
Start the Debugger	30
Start the debugger from the IDE	30
Starting the Debugger from a Command Line	31
Simulator/Debugger Main Menu Bar	33
Simulator/Debugger Simulator/Debugger Toolbar	33
Simulator/Debugger Status Bar	34
Object Info Bar of the Simulator/Debugger Components	34
Function of the Main Menu Bar	35
File Menu	36



View Menu	39
Run Menu	42
Target Menu	45
Simulator Menu	48
Component Menu	56
Window Menu	57
Help Menu	58
Component Associated Menus	61
Component Main Menu	61
Component Popup Menu	61
Highlights of the User Interface	62
Smart User Interface: Activating Services with Drag and Drop	62
To Drag and Drop an Object.	64
Drag and Drop Combinations	65
Selection Dialog Box	69

5 Framework Components 71

Component Introduction	71
CPU component	71
Window components.	71
Target components	72
Components Window	72
General Component	73
Adc_Dac component.	74
Assembly Component	80
Command Line Component	86
Coverage Component	91
DAC Component	96
Data Component	98
Memory Component.	111
IT_Keyboard.	122
Keyboard	126
LCD Display Component	130
Monitor components.	137
Push Buttons components.	141
MicroC Component	144
Module Component	149



Procedure Component	151
Profiler Component	154
Programmable IO_Ports	159
Recorder Component	162
Register Component	166
Seven segments display component	171
SoftTrace Component	175
Source Component	178
Stimulation Component	192
TestTerm Component	195
Terminal Component	201
Wagon Component	207
Visualization Utilities	210
Analog Meter Component	211
Inspector Component	213
IO LED Component	222
LED Component	224
The Phone Component	226
VisualizationTool	229
6 Control Points	247
Control points introduction	247
Breakpoints setting dialog	249
Breakpoint Symbols	249
Description of the Dialog	250
Multiple selections in the dialog	251
Checking condition in dialog	251
Saving Breakpoints	252
Define Breakpoints	254
Identify all Positions Where a Breakpoint Can Be Defined	254
Define a Temporary Breakpoint	255
Define a Permanent Breakpoint	256
Define a Counting Breakpoint	256
Define a Conditional Breakpoint	258
Delete a Breakpoint	259
Associate a Command with a Breakpoint	260
Watchpoints setting dialog	262



Description of the Dialog	262
Multiple selections in the dialog	263
Checking condition in the dialog	264
General Rules for Halting on a Control Point.	264
Define Watchpoints.	265
Defining a Read Watchpoint	265
Defining a Write Watchpoint	266
Defining a Read/Write Watchpoint	267
Defining a Counting Watchpoint	267
Defining a Conditional Watchpoint	268
Deleting a Watchpoint	270
Associate a Command with a Watchpoint	271

7 Debugger Commands 272

Simulator/Debugger Commands.	272
List of Available Commands	273
Definitions of Terms Commonly Used in Command Syntaxes	283
A	285
ACTIVATE	286
ADDCHANNEL	286
ADCPOR	287
ADDXPR	287
ATTRIBUTES	287
AT	299
AUTOSIZE	300
BASE	300
BC	301
BCKCOLOR.	302
BD.	303
BS	303
CALL	306
CD.	306
CF	307
CLOCK.	310
CLOSE	310
COPYMEM	310
CMDFILE.	311



CPORT	311
CR	312
CYCLE	312
DASM	313
DB	314
DDEPROTOCOL	315
DEFINE	316
DELCHANNEL	317
DETAILS	318
DL	318
DUMP	319
DW	319
E	320
ELSE	321
ELSEIF	321
ENDFOCUS	322
ENDFOR	322
ENDIF	323
ENDWHILE	323
EXECUTE	324
EXIT	324
FILL	324
FILTER	325
FIND	325
FINDPROC	326
FOCUS	326
FOLD	327
FONT	328
FOR	328
FPRINTF	329
FRAMES	329
G	330
GO	330
GOTO	331
GOTOIF	331
GRAPHICS	332
HELP	332



IF	333
INSPECTOROUTPUT	334
INSPECTORUPDATE	334
ITPORT	335
ITVECT	335
KPORT	336
LCDPORT	336
LINKADDR	337
LF	337
LOAD	338
LOADCODE	340
LOADMEM	340
LOADSYMBOLS	341
LOG	341
LS	345
MEM.	346
MS.	347
NB.	348
NOCR	350
NOLF	350
OPEN	350
OPENFILE	351
OPENIO	351
OUTPUT	352
P	352
PAUSETEST.	354
PBPORT	354
PORT	355
PRINTF.	355
PTRARRAY	356
RD.	356
RECORD	357
REGBASE	358
REGFILE	358
REPEAT	358
RESET	359
RESETCYCLES	359



RESETMEM.	360
RESETRAM.	361
RESETSTAT.	361
RESTART.	362
RETURN	362
RS	363
S	363
SAVE.	364
SAVEBP	365
SEGPORT.	366
SET	366
SETCOLORS	366
SETCONTROL	367
SETCPU	368
SHOWCYCLES	368
SLAY	369
SLINE	369
SMEM	370
SMOD	370
SPC	371
SPROC	372
SREC	373
STEPINTO	373
STEPOUT.	374
STEPOVER	374
STOP.	375
T	376
TESTBOX	377
TUPDATE.	377
UNDEF.	378
UNFOLD	380
UNTIL	381
UPDATERATE.	381
VER	382
WAIT	382
WB	383
WHILE	384



WL	385
WPORT.	385
WW	386
ZOOM	386
8 True Time I/O Stimulation	388
Stimulation Program examples	388
Running an Example Program Without Stimulation	388
Example Program with Periodical Stimulation of a Variable	391
Example Program with Stimulated Interrupt	392
Example of a Larger Stimulation File	394
Stimulation Input File Syntax	397
9 Real Time Kernel Awareness	400
Real Time Kernel Awareness Introduction.	400
Inspecting the state of a task.	401
Task description language	402
Example of application	404
Inspecting data structures of the Kernel	405
Register assignments for the RTK awareness.	406
OSEK Kernel Awareness	406
OSEK ORTI	407
OSEK RTK Inspector component.	409
10 Environment	415
Debugger environment	415
The Current Directory	416
Global Initialization File (MCUTTOOLS.INI) (PC only)	417
Local Configuration File (usually project.ini)	418
Configuration of the Default Layout for the Simulator/Debugger: the PROJECT.INI File	419
Paths	423
Environment Variable Details	425
ABSPATH	426
ABSPATH: Absolute Path	426
DEFAULTDIR	427
DEFAULTDIR: Default Current Directory	427
ENVIRONMENT	428
ENVIRONMENT: Environment File Specification.	428



GENPATH	429
GENPATH: #include “File” Path	429
LIBRARYPATH	430
LIBRARYPATH: ‘include <File>’ Path	430
OBJPATH	431
OBJPATH: Object File Path	431
TMP	432
TMP: Temporary directory	432
USELIBPATH.	433
USELIBPATH: Using LIBPATH Environment Variable	433
Searching order for sources files	434
Searching Order in the Simulator/Debugger for C source files (*.c, *.cpp). 434	
Searching Order in the Simulator/Debugger for Assembly source files (*.dbg) 434	
Searching Order in the Simulator/Debugger for object files (HILOADER) 434	
Files of the Simulator/Debugger.	435
11 How To ...	438
How To Configure the Simulator/Debugger	438
How To Configure the Simulator/Debugger for Use from Desktop on Win 95, Win 98, Win NT4.0 or Win2000	439
How To Start the Simulator/Debugger	439
How To Start the Simulator/Debugger from WinEdit	439
Automating startup of the Simulator/Debugger	440
How To Load an Application	442
How To Start an Application	443
How To Stop an Application	443
How To Step in the Application	444
How to step on Source Level	444
How to Step on Assembly Level	446
How To Work on Variables.	446
How to Display Local Variable from a Function	447
How to Display Global Variable from a Module	447
How to Change the Format for the Display of Variable Value.	448
How to Modify a Variable Value	449
Modify a Variable Value	449
How to Get the Address Where a Variable is Allocated	450
How to Inspect Memory starting at a Variable Location Address	450



How to Load an Address Register with the Address of a variable	450
How To Work on Register	451
How to Change the Format of the Register display	451
How to Modify a Register Content	451
How to Get a Memory Dump starting at the Address where a Register is pointing	453
How to Modify the content of a Memory Address.	454
How to Consult Assembler Instructions Generated by a Source Statement	454
How To view Code	455
How to Communicate with the Application	456
About startup.cmd, reset.cmd, preload.cmd, postload.cmd	456
12 CodeWarrior Integration	458
Requirements	458
Debugger Configuration	458
13 Debugger DDE capabilities	460
Debugger DDE Server	460
DDE introduction	460
Debugger DDE implementation	460
14 Synchronized debugging through DA-C IDE	462
Requirements	462
Configuring DA-C IDE for Metrowerks Tool Kit	462
Creating a new project	463
Configure the working directories	463
Debugger Interface	474
Principle of Communication between DA-C IDE and Simulator/Debugger	475
Synchronized debugging.	480
Troubleshooting	480
15 Scripting	513
The Component Object Model Interface	513
Parameters:	513
Return Values:	514
Manual Registration	514
Scripting Example	514
Remote Scripting another HI-WAVE	515
COM_START	515



COM_EXIT	515
COM_EXE	516
16 Appendix	517
Messages in Status Bar	517
Status Messages	517
Stepping, Breakpoint and Watchpoints Messages	518
CPU Specific Messages	519
Target Specific Messages	520
More Simulator Peculiar Messages: Memory Access Messages	521
EBNF Notation	522
Introduction to EBNF	522
“Expression” Definition in EBNF	524
Constant Standard Notation	527
Register Description File	528
OSEK ORTI File Sample	531
Bug Reports	538
Technical Support	541
E-mail	541
FAX	541
Support by MAIL	541
Internet	542
Index	543



Important Notice

This chapter provides information about Copyright, Trademarks and warranty.

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

- [Copyrights](#)
- [Trademarks](#)
- [Warranty](#)

Copyrights

Metrowerks CodeWarrior copyright ©1993–2003 by Metrowerks, Inc. and its licensors.

All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks, Inc.

Trademarks

Metrowerks, the Metrowerks logo, CodeWarrior, PowerPlant, and Metrowerks University are registered trademarks of Metrowerks Inc. CodeWarrior Constructor, Geekware, PowerParts, and Discover Programming are trademarks of Metrowerks Inc.

PowerPC® is a registered trademark of IBM, Inc.



All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT.

Warranty

While every effort has been made to ensure the accuracy of all information in this document, Metrowerks assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in the Simulator/Debugger user's guide, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Metrowerks further assumes no liability arising out of the application or use of any product or system described here; nor any liability for incidental or consequential damages arising from the use of this document and the related product.

Metrowerks disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness for a particular purpose.

Metrowerks reserves the right to make changes without further notice to any products herein to improve reliability, function or design.

Overview

This chapter provides an overview of the structure from the debugger documentation.

About This Guide

This document includes information to become familiar with the Simulator/Debugger, to use all functions and help you understand how to use this environment. This document is divided into the following chapters:

- The [Introduction](#) chapter introduces the Simulator/Debugger concept.
- The [Simulator/Debugger User Interface](#) chapter provides all details about the Simulator/Debugger user interface environment i.e., menus, toolbars, status bars and drag and drop facilities.
- The [Framework Components](#) chapter contains descriptions of each basic component and visualization utility.
- The [Debugger Commands](#) chapter describes and provides examples of all Commands line Commands.
- The [True Time I/O Stimulation](#) chapter explains the principle and provides examples on the Stimulation component.
- The [Real Time Kernel Awareness](#) chapter contains descriptions of the Real Time concept and related applications.
- The [Environment](#) chapter contains information for defining the application environment.
- The [Control Points](#) chapter is dedicated to the control points and associated dialogs.
- The [How To ...](#) chapter provides answers for common questions and describes how to use advanced features of the Simulator/Debugger.
- The [CodeWarrior Integration](#) chapter explains how to configure the Simulator/Debugger for use with CodeWarrior.
- The [Debugger DDE capabilities](#) describe the debugger DDE features.
- The [Synchronized debugging through DA-C IDE](#) chapter explains the use of tools with the DA-C IDE from RistanCase

- The [Appendix](#) contains information about all the Simulator/Debugger messages, the EBNF notation and how to get Technical Support.
- The “Index” contains all keywords for the Simulator/Debugger.

Highlights

- 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 HIWARE and ELF/Dwarf Object File Format and Motorola S-Records

Read the Release Notes

Before you use a tool such as the Debugger, read the release notes. They contain important last-minute information about new features and technical issues or incompatibilities that may not be included in the documentation.

Document Conventions

In this section, you will find terms and styles used in this document.

General terms

- Choose.



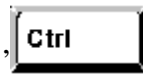


This term is used to select an item from a menu or a list/combo box.


- Check.

This term is used to select a check box item.


- Uncheck.

This term is used to deselect a selected check box item.

All keyboard keys are given as , , , , , etc.

Also the left mouse button is  and considered as a key.

- Key1 + Key2.

When you have to press two keys at the same time. The “+” sign means that Key1 is held down while Key2 is pressed. Example:  +

,  + .

Mouse operations

- Click




The word “click” means click the left mouse button once.

- Right-click

This “click” operation is done with the right mouse button.

- Double-click

This is a double “click” operation.


-  + Key, example:  + .

This means that you press and hold down the left mouse button while you press the specified key. When the key has been pressed, you can unclick.

- Drag.

This means that you press and hold down the left mouse button while you drag the mouse. If you perform this operation on an object that has been designed to be dragged, this object will move with the mouse arrow and drop when you unclick the mouse.

- Unclick.

When you release the left mouse button after a drag operation or when you have completed a “ + Key” operation.

Font styles

- **Bold**

Words in bold are menu items and entries.

- **Courier**

This font is used for filenames and pathnames, commands, command syntaxes and examples.

Examples

```
C:\HIWAVE\PROJECT.INI
```

```
in>Memory < ADR on
```

Menu Paths

When asked to follow specific selections/entries in menus and submenus, the following selections are given in a list of items separated by the “>” separator. Example: Choose **Window > Options > Autosize**. Here you click **Window** in the Simulator/Debugger main menu bar, drag the mouse to the **Options** submenu then check or uncheck **Autosize**.

Others

NOTE Notes provide important and helpful information on any subject.

Introduction

This chapter is an introduction to the Simulator/Debugger from Metrowerks used in 8/16 bit embedded applications.

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

- [What Is the Simulator/Debugger?](#)
- [What Is a Simulator/Debugger Application?](#)
- [What Is a Simulator/Debugger Execution Framework?](#)
- [Understanding the Simulator/Debugger Concept](#)

What Is the Simulator/Debugger?

The Simulator/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.



What Is a Simulator/Debugger Application?

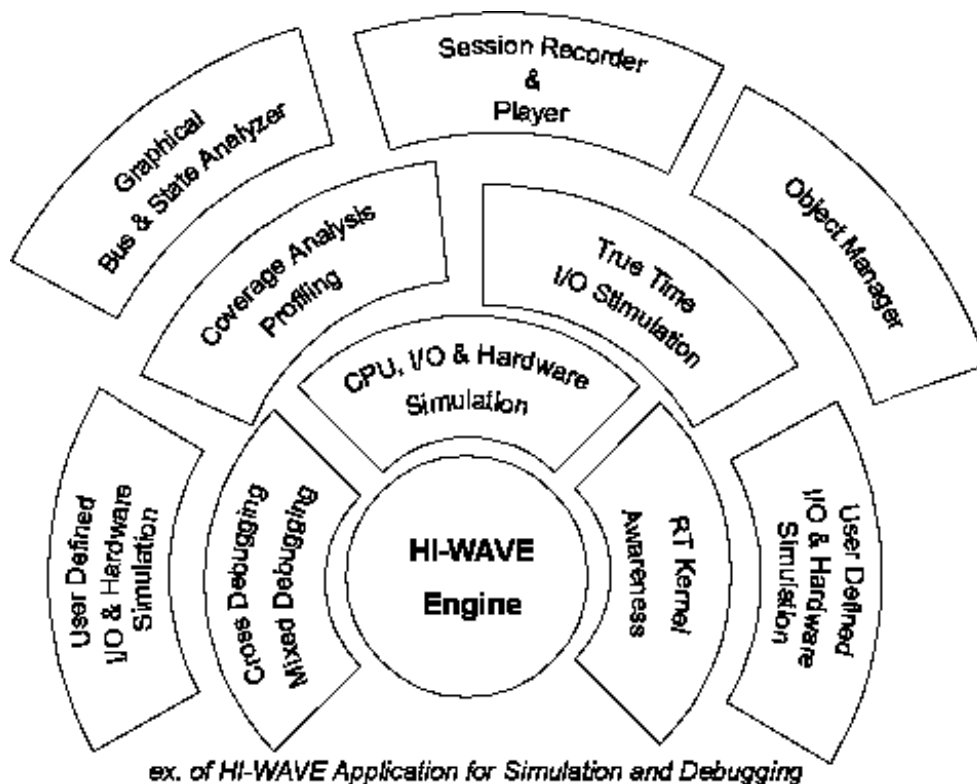
A Simulator/Debugger Simulator/Debugger Application contains the Simulator/Debugger Engine and a set of debugger components bound to the task that they should perform (for example a simulation and debugging session). The Simulator/Debugger Engine is the heart of the system. It monitors and coordinates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

You can adapt your Simulator/Debugger application to your specific needs. Integrating or removing the Simulator/Debugger Components is very easy. You can also choose a default configuration, refer to [Figure 3.1](#).

You can add additional Simulator/Debugger Components (for example, for simulation of a specific I/O peripheral chip) and integrate them with your Simulator/Debugger Application.

You can also open several components of the same type.

Figure 3.1 Example of Simulator/Debugger Application for Simulation and Debugging



What Is a Simulator/Debugger Execution Framework?

Since the Simulator/Debugger is a Multipurpose Tool you have to use the components according to the task you want to run. In other words, you either build a Simulator/Debugger Execution Framework or use a default one.

Each Execution Framework is built with selected components. Since the Simulator/Debugger is an open and extendable system, you can write and add your own debugger components if needed (for example a debugger component for a specific I/O simulation).

Understanding the Simulator/Debugger Concept

This section provides an overview of the Simulator/Debugger concept.

The Simulator/Debugger Execution Framework

Any Simulator/Debugger based task you create (for example: testing and debugging a target application, running a visualization application), has a specific debugger “Execution Framework”. A Simulator/Debugger Execution Framework is a set of user selected and configured [Framework Components](#), such as shown in [Figure 3.2](#) and [Figure 3.3](#). The debugger engine is always present.

Figure 3.2 Example of Execution Framework for Simulation.

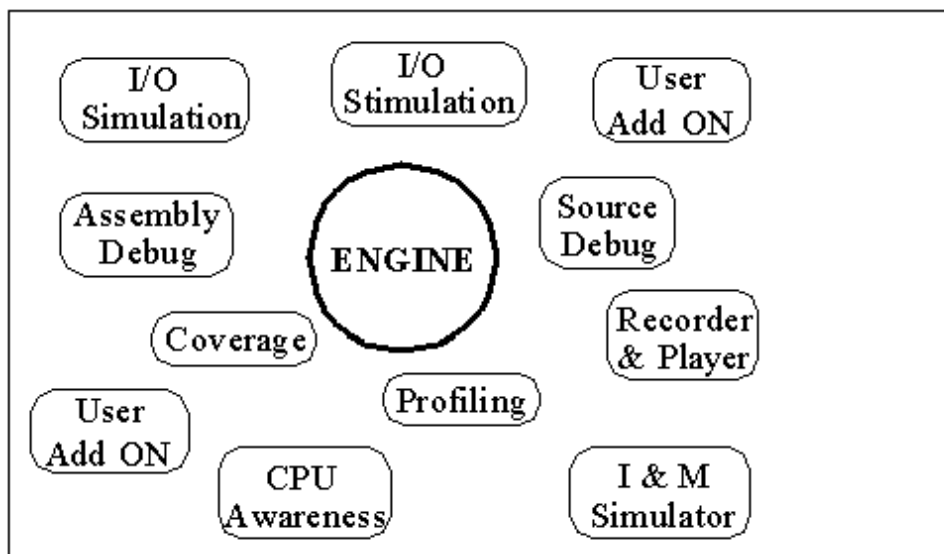
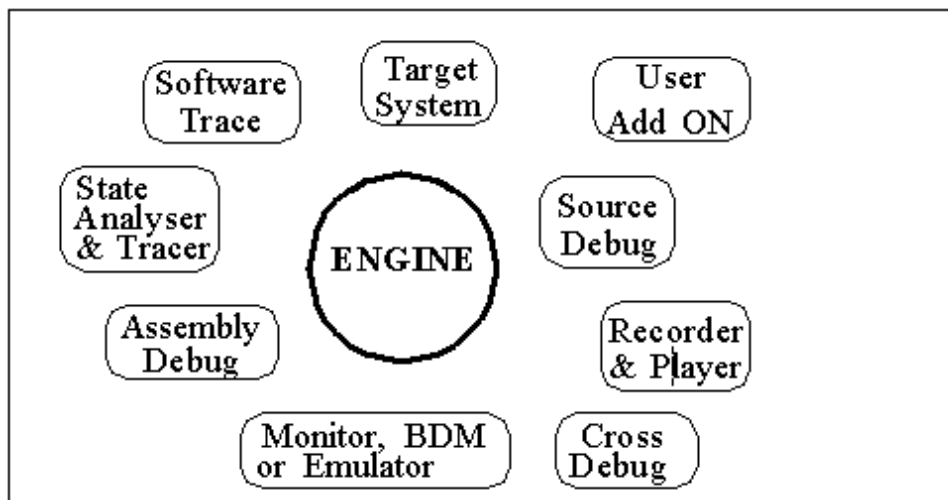


Figure 3.3 Example of Execution Framework for Cross Debugging.



Objects and Services

An object provides one or more services. For example an object of a variable type holds values in a specific range. An object like the Bus Analyzer component graphically displays the bus state. An I/O Simulation object provides the behavior of the corresponding hardware peripheral. Providing services is the ultimate goal of objects and that is why they are created and used. An object has a state, behavior and identity.

Framework Components

A Simulator/Debugger Framework Component is an object that you can integrate or remove from an Execution Framework. Each Framework component belongs to a service class.

Examples of Framework Components:

- Simulator/Debugger Engine
- CPU Simulator
- Source Level Debug Component
- Assembly Debug Component
- Profiler
- Bus Analyzer
- I/O Simulation Components
- True Time Stimulation Components



If any hardware component is present (e.g., target board, I/O peripheral, emulators), it is also considered to be a Framework component.

Demo Version Limitations Components

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

Simulator/Debugger User Interface

This chapter describes the Simulator/Debugger User Interface.

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

- [Introduction](#)
- [Application Programs](#)
- [Start the Debugger](#)
- [Simulator/Debugger Main Menu Bar](#)
- [Simulator/Debugger Status Bar](#)
- [Simulator/Debugger Status Bar](#)
- [Object Info Bar of the Simulator/Debugger Components](#)
- [Function of the Main Menu Bar](#)
- [Component Associated Menus](#)
- [Highlights of the User Interface](#)

Introduction

The Simulator/Debugger main window acts as a container for windows of all other components. Additionally, it provides a global menu bar, a tool bar, a status bar for status information, and object information bars for several components.

The main window manages the layout of the different component windows (**Window** menu of the Simulator/Debugger application). Component windows are organized as follows:

- Tiled arrangement

- Auto tiled, component windows are automatically resized when the main window is resized
- Overlapped
- Icon (windows that are currently minimized).

Application Programs

After installation, all executable programs are placed in the `prog` subdirectory, e.g. if you installed the software in `C:\Metrowerks` on a PC, all program files are located in C:\Metrowerks\PROG (for details refer to installation guide).

The following list provides an overview of the files used 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

Start the Debugger

This section explains how to start the debugger from the IDE or a command line.

Start the debugger from the IDE

You can start the debugger from the IDE by clicking the **Debug** button ([Figure 4.1](#)) from the project window.

Figure 4.1 IDE Debug button.



Starting the Debugger from a Command Line

You can start the HI-WAVE debugger from a (DOS) command line. The command syntax is shown below:

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

where **AbsFileName** is the name of the application to load in the debugger. Options may be introduced by a minus character.

Options are:

- **-T=<time>**: test mode. The debugger will terminate after the specified time (in seconds). The default value is 300 seconds, e.g.:

```
c:\Metrowerks\prog\hiwave.exe -T=10
```

The debugger will terminate after 10 seconds.

- **-Target=<targetname>** sets the specified target, e.g.:

```
C:\Metrowerks\prog\hiwave.exe  
c:\Metrowerks\demo\hc12\sim\fibonacci.abs -w -Target=sim
```

Starts the debugger, sets the simulator target, and loads fibonacci.abs file.

- **-W**: wait mode - will wait even when a <exeName> is specified, e.g. -W
- **-Instance=%currentTargetName**: defines a build instance name. When a build instance is defined, the same one will be used e.g.

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

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



Simulator/Debugger User Interface

Start the Debugger

- **-Prod**: specifies the project directory and/or project file to be used at start-up: **-Prod** = <fileName> e.g.

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

- **-Nodefaults**: will not load the default layout (see section 4 of the Project file Activation) e.g.

```
c:\Metrowerks\prog\hiwave.exe -nodefaults
```

- **-Cmd** specifies a command to be executed at start-up: **-cmd** = "" {characters} e.g.

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

- **-C**: specifies a command file to be executed at start-up: **-c** <cmdFile> e.g.

```
c:\Metrowerks\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 e.g.

```
c:\Metrowerks\prog\hiwave.exe -EnvOBJPATH=c:\sources\obj
```

NOTE Options are not case sensitive.

Order of commands

Commands specified by options are executed in the following order:

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

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

WARNING! 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:\Metrowerks\PROG \DEMO\TEST.ABS -w -d

Simulator/Debugger Main Menu Bar

This Menu Bar, shown in [Figure 4.2](#) is associated with the main function of the debugger application, target, and selected windows.

Figure 4.2 Debugger Main Menu Bar



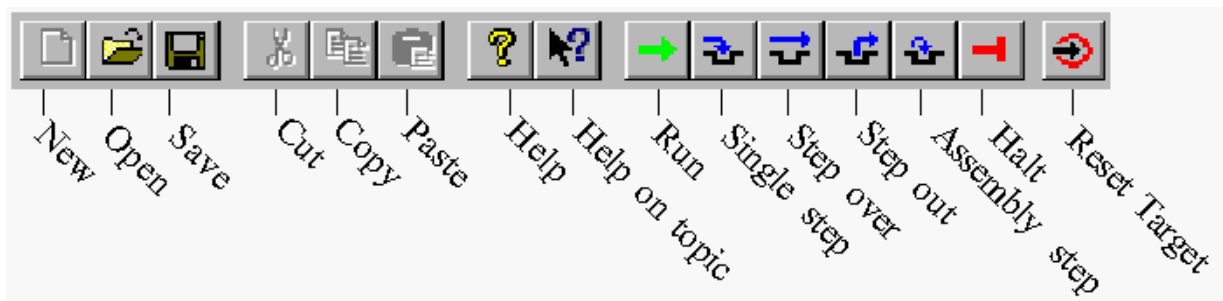
NOTE You can select menu commands by pressing the ALT key to select the menu bar and press the key corresponding to the underlined letter in the menu command.

Simulator/Debugger Simulator/Debugger Toolbar

This toolbar is the default toolbar. Most menu commands have a related shortcut icon on the debugger toolbar.

[Figure 4.3](#) identifies each default icon.

Figure 4.3 The Debugger Toolbar



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

Simulator/Debugger Status Bar

The status bar at the bottom of the debugger window, shown in [Figure 4.4](#) contains a context sensitive help line for target specific information, e.g., number of CPU cycles for the **Simulator** target and execution status. All messages that appear in the status bar are described in [Messages in Status Bar](#).

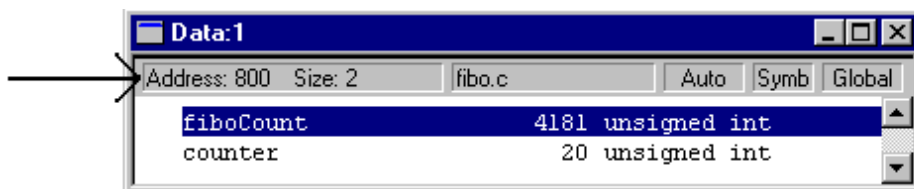
Figure 4.4 The Debugger Status Bar



Object Info Bar of the Simulator/Debugger Components

The object info bar of the debugger window, as shown in [Figure 4.5](#), provides information about the selected object.

Figure 4.5 Object Info Bar of Debugger Window Components



Function of the Main Menu Bar

[Table 4.7](#) describes menus entries available in the menu bar ([Figure 4.6](#)).

Figure 4.6 Debugger Main Menu

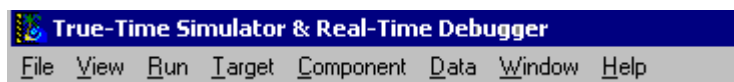


Table 4.1 Description of the Main Menu Entries

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.
Target	Contains entries to select the debugger target.
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.

Menu entry	Description
------------	-------------

Help	A standard Windows Help menu.
------	-------------------------------

File Menu

The **File** menu shown in [Table 4.8](#) is dedicated to the debugger project.

Figure 4.7 File Menu



[Table 4.2](#) describes File Menu entries.

Table 4.2 File Menu Description

Menu entry	Description
------------	-------------

New	Creates a new project.
-----	------------------------

Load Application	Loads an executable file (or debugger target if nothing is selected).
------------------	---

Menu entry	Description
...\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.), target name e.g., Simulator and the .ABS application file to load.
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 dialog to set environment variables for current project.
1.Project.ini 2.Test.ini 3...	Recent project file list
Exit	Quits the Simulator/Debugger.

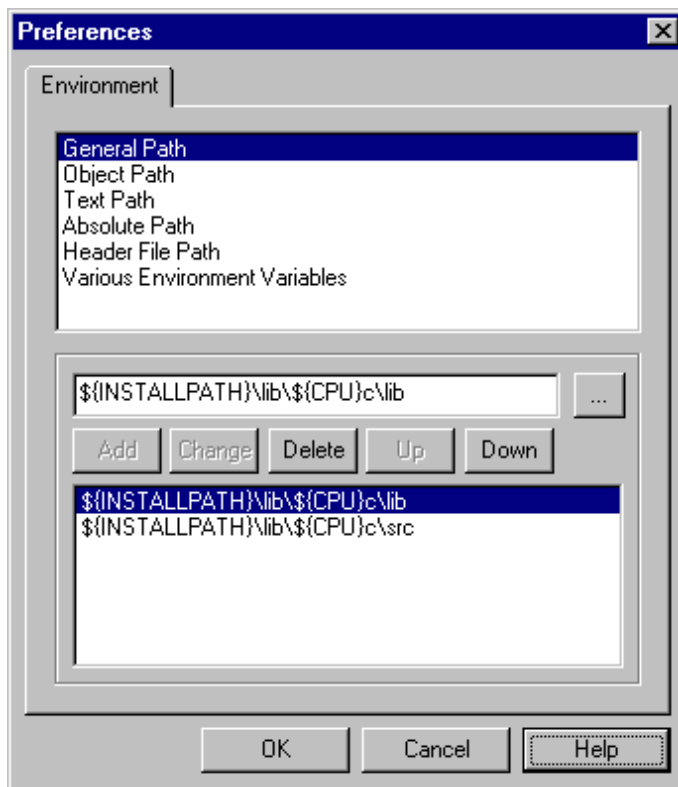
You can shortcut some of these functions by clicking toolbar icons (refer to the [Simulator/Debugger Simulator/Debugger Toolbar](#) section).

Preferences dialog

With this dialog ([Figure 4.8](#)) it is possible to set up environment variables for the current project. New variables will be saved in the current project file after clicking the **OK** button.

NOTE The corresponding menu entry (File>Configuration) is only enabled if a project file is loaded.

Figure 4.8 Preferences Dialog



The preference panel contains the following controls:

- A list box containing all environment variables, you can select a variable with the mouse or Up/Down buttons.
- Command Line Arguments: Command line options are displayed. You can add, delete, or modify options, and specify a directory with the browse button (...).
- A second list box containing all variables defined in the corresponding Environment section. Select a variable with the mouse or Up/Down 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 this menu ([Figure 4.9](#)) you can choose to show or hide the toolbar, status bar, window component titles and headlines (headlines are also called [Object Info Bar of the Simulator/Debugger Components](#) in this document). You can select smaller window borders and customize the toolbar. [Table 4.3](#) describes the View Menu entries.

Figure 4.9 View Menu

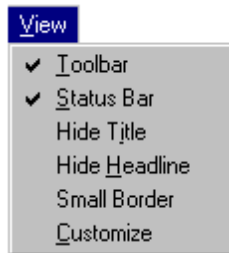


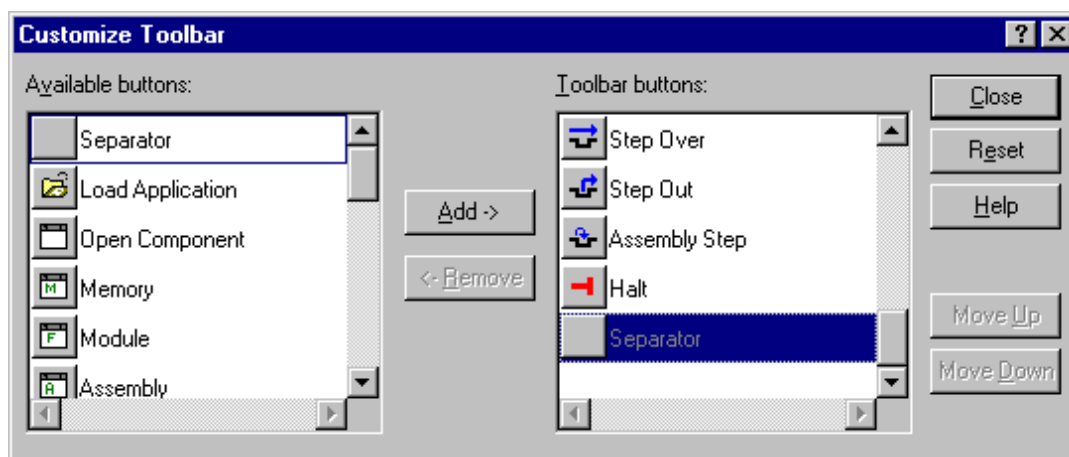
Table 4.3 View Menu Description

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

You can customize the toolbar of the Simulator/Debugger, adding and removing component shortcuts and action shortcuts. 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 4.10](#).

Figure 4.10 Customize Toolbar Dialog



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.

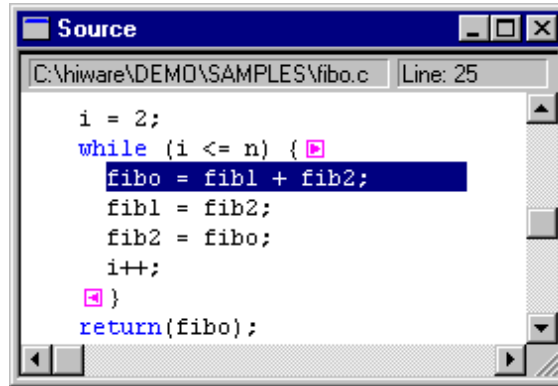
Demo Version Limitations

The default toolbar cannot be configured.

Examples of view menu options

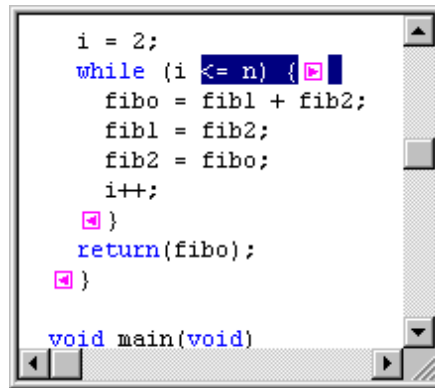
[Figure 4.11](#) shows a Typical component window display.

Figure 4.11 Typical component window display



[Figure 4.12](#) shows a component window without a title and headline.

Figure 4.12 Component window without title and headline



[Figure 4.13](#) shows a component window without a title and headline, and with a small border.

Figure 4.13 Component window without title and headline, and with small border

```

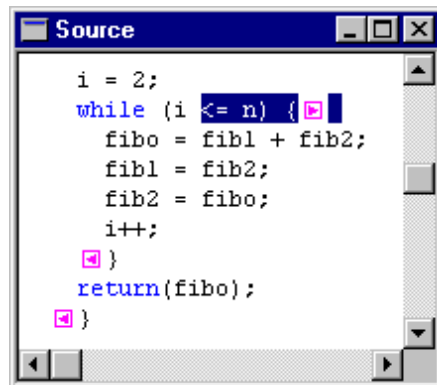
i = 2;
while (i <= n) {
    fibo = fib1 + fib2;
    fib1 = fib2;
    fib2 = fibo;
    i++;
}
return(fibo);
}

void main(void)

```

[Figure 4.14](#) shows a component window without headline and small border

Figure 4.14 Component window without headline and small border



Run Menu

This menu, shown in [Figure 4.15](#) is associated with the simulation or a debug session. You can monitor a simulation or debug session from this menu. Run menu entries are described in [Table 4.4](#).

Figure 4.15 Run Menu

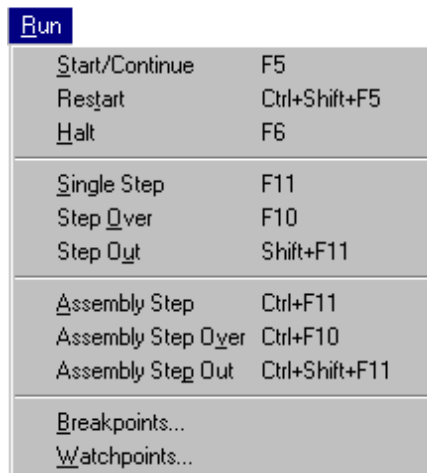













Table 4.4 Run Menu Description

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:
Restart	Starts execution of the loaded application from its entry point. Shortcut: + +
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:

Menu entry	Description
Single Step	<p>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.</p> <p>Shortcut: </p>
Step Over	<p>Similar to the Single Step command, but does not step into called functions. A function call is treated as one statement.</p> <p>Shortcut: </p>
Step Out	<p>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.</p> <p>Shortcut:  + </p>
Assembly Step	<p>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.</p> <p>Shortcut:  + </p>
Assembly Step Over	<p>Similar to the Step Over command, but steps over subroutine call instructions.</p> <p>Shortcut:  + </p>

Menu entry	Description
Assembly Step Out	<p>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.</p> <p>Shortcut:  +:  + </p>
Breakpoints...	<p>Opens the Breakpoints Setting dialog and displays the list of breakpoints defined in the application (refer to Control Points chapter).</p>
Watchpoints...	<p>Opens the Watchpoints Setting dialog box and displays the list of watchpoints defined in the application (refer to Control Points chapter).</p>

You can provide shortcuts for some of these functions using the toolbar (refer to [Simulator/Debugger Simulator/Debugger Toolbar](#) section and [Customizing the Toolbar](#) section).

You can set breakpoints and watchpoints in Source and Assembly component windows.

NOTE For more information about breakpoints and watchpoints, refer to the [Control Points](#) chapter.

Target Menu

This menu entry ([Figure 4.16](#)) appears between the **Run** and **Component** menus when no target is specified in the PROJECT.INI file and no target has been set. The **Target** name is replaced by an actual target name when the target is set. To set the target, select **Component>Set Target...** Refer to the [Component Menu](#) section.

Figure 4.16 Target Menu



[Table 4.5](#) describes the Target Menu entries.

Table 4.5 Target Menu Description

Menu entry	Description
Load	Loads a Simulator/Debugger target.
Reset	Resets the current Simulator/Debugger target.

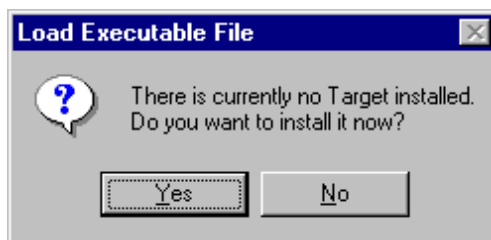
Loading a Target

Use the Target menu to load a debugger target.

1. Choose **Target>Load...**

The message shown in [Figure 4.17](#) is displayed:

Figure 4.17 Load Target Dialog

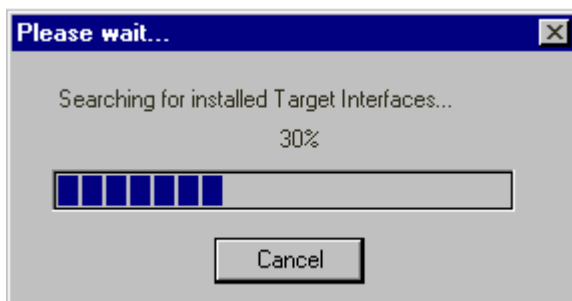


At this point, the target is not set and you cannot load any application (.ABS) file.

2. Click **Yes** to install a target in the debugger.

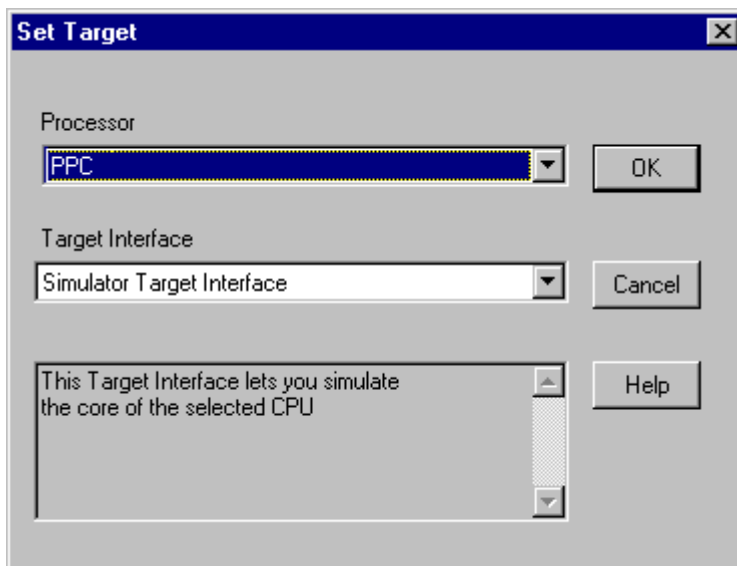
The debugger searches for all targets installed. The dialog shown in [Figure 4.18](#) is opened. Click **Cancel** to stop the process and skip target detection.

Figure 4.18 Scan Target Dialog



The **Set Target** dialog shown in [Figure 4.19](#) is opened.

Figure 4.19 Set Target Dialog



3. Use the **Processor** list popup to select the desired processor.
4. Use the **Target Interface** list popup to select the desired target.

A text panel displays information about the selected Target.

WARNING! When a target can not be loaded, the combo box displays the path where you should install missing dll.

5. Click **Ok** to load target in debugger.

NOTE For more information about which target to load and how to set/reset a target, refer to the Simulator/Debugger target manuals e.g., “SIMULATOR Target, CPU Awareness & True-Time Simulation”.

Targets file

All targets are associated with a window file with **.tgt** extension.

Example: The Simulator Target

The default target of the Simulator/Debugger is the Simulator (in [Configuration of the Default Layout for the Simulator/Debugger: the PROJECT.INI File: TARGET=SIM](#)). However, choose **Component>Set Target...** if you want to open the dialog to set a different target.

Simulator Menu

This menu, shown in [Figure 4.20](#) is associated with the simulator target, and allows you to load an application in the Simulator/Debugger. [Table 4.6](#) describes the Simulator menu entries.

Figure 4.20 Simulator Menu

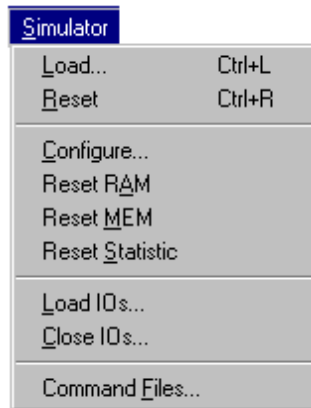


Table 4.6 Simulator Menu Description

Menu entry	Description
Load	Opens the Load Executable Window menu.

Menu entry	Description
Reset	Resets the simulator target.
Configure	Opens the Memory Configuration 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
Command Files	Opens the Command File Dialog

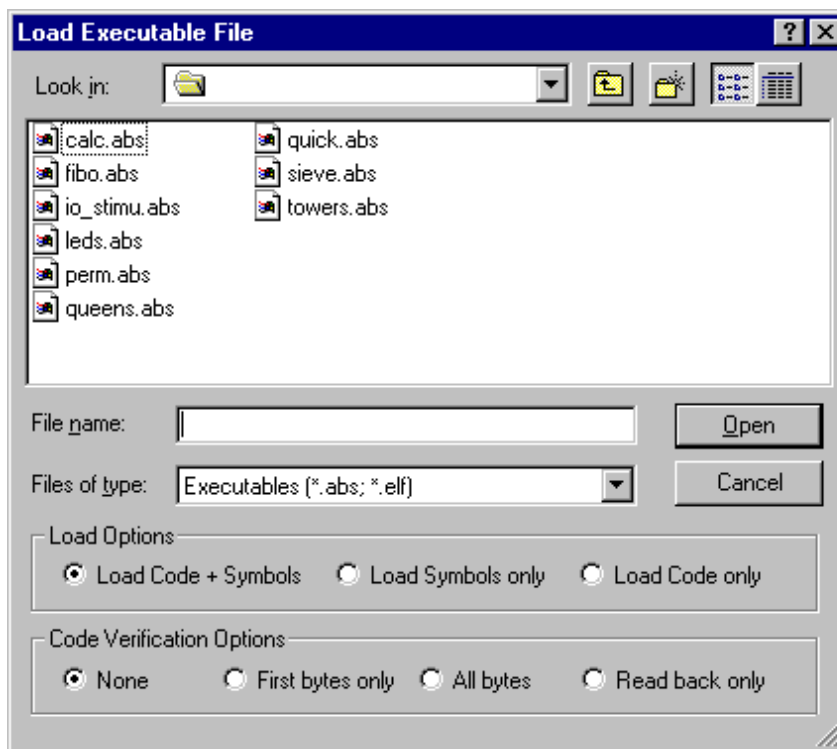
Simulators File

The simulator is associated with a window with a **.sim** extension.

Load Executable File dialog

Choose **Simulator>Load...** to open the Load Executable File window, shown in [Figure 4.21](#), then set the load options and choose a Simulation Execution Framework (a .ABS application file).

Figure 4.21 Load Executable File dialog



Description of the Load Options.

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

- **Load Code + Symbols.** This will load the application code followed by the debug information (symbols) to allow debugging of the application.
- **Load Symbols only.** If this option is selected, only debugging information is loaded. This can be used if the code is already loaded into the target system or programmed into a non-volatile memory device (ROM/FLASH).
- **Load Code only.** Only the application code will be loaded into the target system. This option can be used if no debugging is needed.

Description of the Code Verifications Options.

These four Radio buttons allow you to choose between four levels of code verification.

- **None.** The loader does not verify anything. The loader behaves the same as previous versions of the debugger.

- First bytes. The loader reads back a maximum of the first four bytes of a block that have been written to memory. This option is not as secure as the next option but is faster.
- All bytes. The loader reads back all bytes of a block that have just been written to memory. File loading is almost twice as long. However, verification is done on the whole file.
- Read back only. With this option the loader does not load data to memory. However, it reads back the current data matching the same areas from the target memory and compares all data with the data from the selected file.

NOTE If "Load Symbols only" is selected, verification radio buttons are grayed and NO verification is performed.

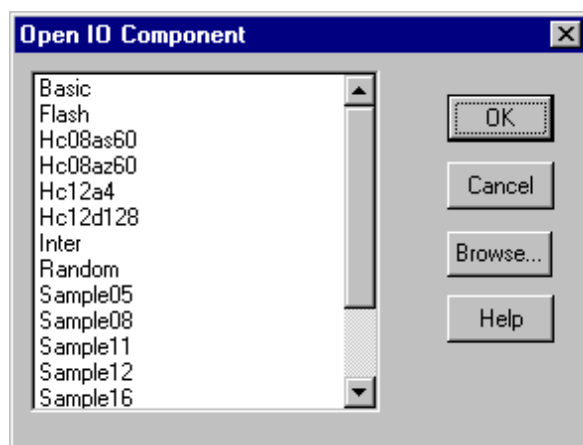
TIP If verification fails, a message is displayed, giving the address where a difference occurred.

For more details on the Simulator functions, consult the True Time Simulator Manual.

Dialog Load I/Os

This dialog box, shown in [Figure 4.22](#) allows you to open an I/O device (peripheral) simulation. The **Browse** button allows you to specify a location for the I/O.

Figure 4.22 Open IO Dialog

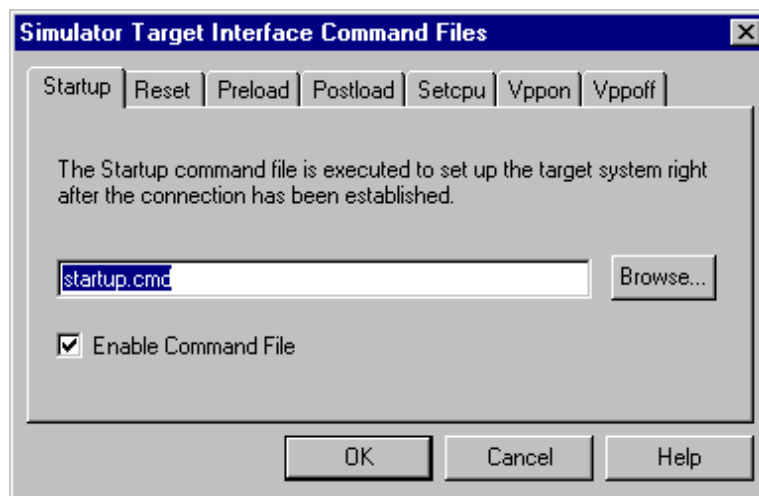


NOTE I/O simulation components are either designed by Metrowerks and delivered with the tool-kit installation or designed by the user with the Peripheral Builder.

Target Interface Command File Dialog

Each page of this property sheet dialog, shown in [Figure 4.23](#) corresponds to an event on which a command file (refer to [About startup.cmd](#), [reset.cmd](#), [preload.cmd](#), [postload.cmd](#)) can be automatically run from the Simulator/Debugger: [Startup Command File](#), [Reset Command File](#), [Preload Command File](#), [Postload Command File](#), [Setcpu Command File](#), [Vppon Command File](#) and [Vppoff Command File](#).

Figure 4.23 Target Interface Command File Dialog



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`.

- the default **Setcpu** command file is SETCPU .CMD.
- the default **Vppon** command file is VPPON .CMD.
- the default **Vppoff** command file is VPPOFF .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**.

TIP 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 Simulator/Debugger after the Target Interface 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 page of the [Target Interface Command File Dialog](#) dialog.

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 Simulator/Debugger 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 page of the [Target Interface Command File Dialog](#) dialog.

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 Simulator/Debugger before an application is loaded to the target system through the Target Interface.

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 page of the [Target Interface Command File Dialog](#) dialog.

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 Simulator/Debugger after an application has been loaded to the target system through the Target Interface.

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 page of the [Target Interface Command File Dialog](#) dialog.

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

Setcpu Command File

The **Setcpu** command file is executed by the Simulator/Debugger after a CPU has been set or modified in the simulator (this occurs when the **setcpu** command is used or when a file is loaded in the simulator 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 page of the [Target Interface Command File Dialog](#).

By default the `SETCPU.CMD` file located in the current project directory is enabled as the current **Setcpu** command file.

Vppon Command File

The **Vppon** command file is executed by the Simulator/Debugger before "Non Volatile Memory" is erased or before a file is programmed in "Non Volatile Memory" to the target system through the target interface Non Volatile Memory Control dialog (**Flash...** menu entry) or **FLASH PROGRAM/ERASE** commands from Flash Programming utilities.

The **Vppon** command file full name and status (enable/disable) can be specified either with the [CMDFILE VPPON](#) Command Line command or using the **Vppon** property page of the [Target Interface Command File Dialog](#) dialog.

By default the VPPON.CMD file located in the current project directory is enabled as the current Vppon command file.

This command file can be used, for example, to enable a programming voltage by software.

NOTE This command file is not available for all target interfaces.

Vppoff Command File

The **Vppoff** command file is executed by Simulator/Debugger after a "Non Volatile Memory" has been erased or after a file has been programmed in "Non Volatile Memory" to the target system through the target interface Non Volatile Memory Control dialog (**Flash...** menu entry) or **FLASH PROGRAM/ERASE** commands from Flash Programming utilities.

The **Vppoff** command file full name and status (enable/disable) can be specified either with the [CMDFILE VPPOFF](#) Command Line command or using the **Vppoff** property page of the [Target Interface Command File Dialog](#) dialog.

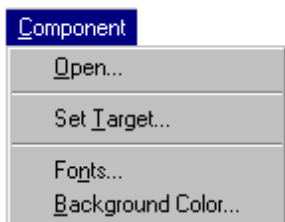
By default the VPPOFF.CMD file located in the current project directory is enabled as the current Vppoff command file.

NOTE This command file is not available for all target interfaces.

Component Menu

Select **Component>Open...** to load an extra component window, shown in [Figure 4.24](#), that has not been loaded by the Simulator/Debugger at startup. The popup dialog presents a set of different components that are introduced in [Framework Components](#).

Figure 4.24 Component Menu



[Table 4.7](#) describes the Component Menu entries.

Table 4.7 Component Menu Description

Menu entry	Description
Open	Loads an extra component window that has not been loaded by the Simulator/Debugger at startup. The popup dialog presents a set of different components that are introduced in Components Window .
Set Target	Sets the Simulator/Debugger target e.g., Simulator.
Fonts	Opens a standard Font Selection dialog, where you can set the font used by Simulator/Debugger components.
Background Color	Opens a standard Color Selection dialog, where you can set the background color used by the Simulator/Debugger component windows.

TIP For a readable display, we recommend using a proportional font (e.g., Courier, Terminal, etc.).

Demo Version Limitations

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

Window Menu

In this menu, shown in [Figure 4.25](#), you can set the component windows general arrangement. The Submenu **Window>Options** is shown in [Figure 4.26](#) and the Submenu **Window>Layout** in [Figure 4.27](#).

Figure 4.25 Window Menu

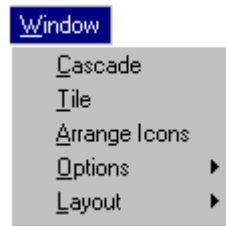


Figure 4.26 Window>Options SubMenu

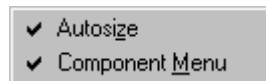
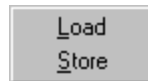


Figure 4.27 Window>Layout SubMenu



[Table 4.8](#) specifies the Window Menu entries.

Table 4.8 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 help menu ([Figure 4.28](#)). [Table 4.9](#) shows menu entries.

Figure 4.28 Help Menu

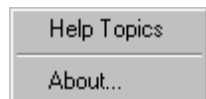


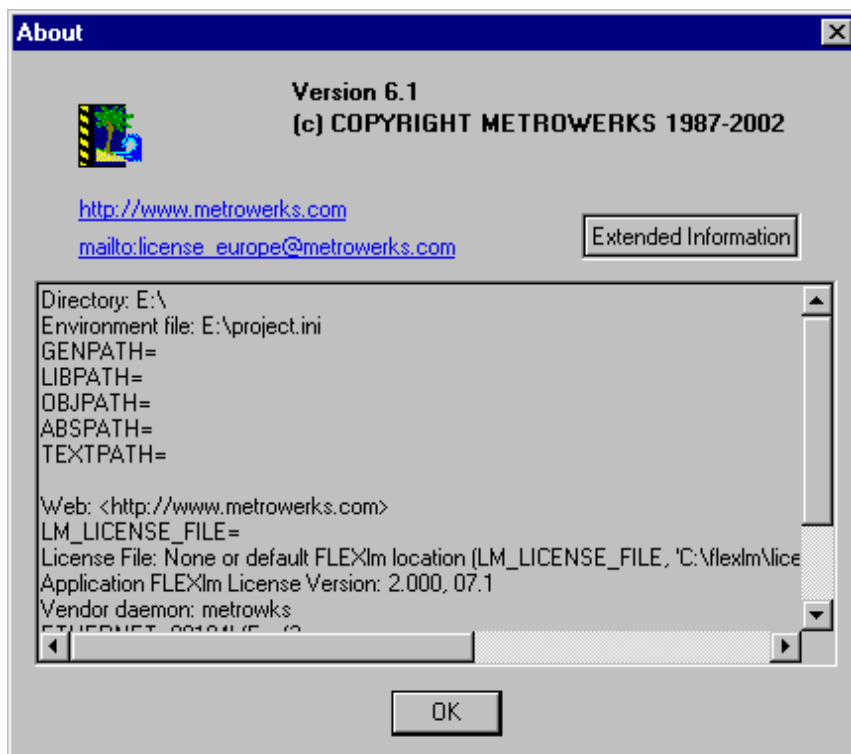
Table 4.9 Help Menu Description

Menu entry	Description
Help Topics	Choose Help Topics in the menu for online help or if you need specific information about a Simulator/Debugger topic.
About HI-WAVE	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 4.29](#). 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@metrowerks.com`.

Figure 4.29 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 Metrowerks internet home page.

Click on **OK** to close this dialog.

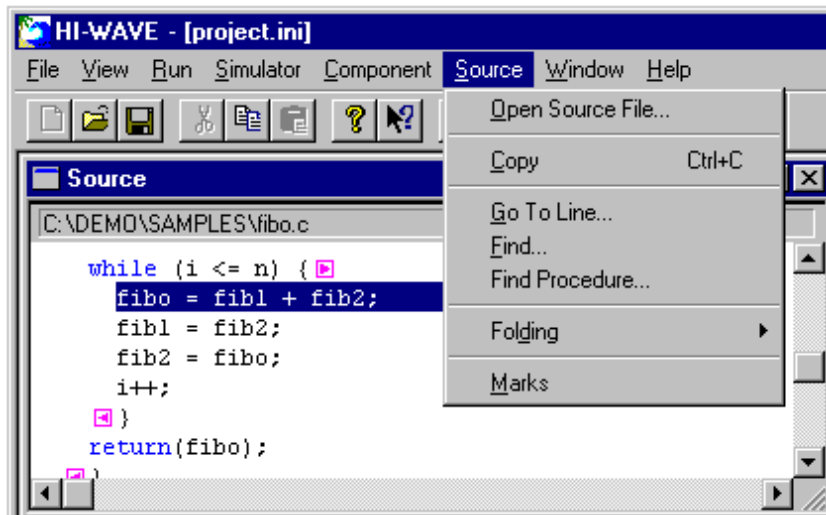
Component Associated Menus

Each component loaded by default or that you have loaded has two menus. One menu is in the Simulator/Debugger 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 4.30](#) is always between the Component entry and the Window entry of the Simulator/Debugger main menu. It contains general entries of the current active component. You can hide this menu by unchecking **Window>Options>Component Menu**.

Figure 4.30 Example of Component Main Menu



Components File

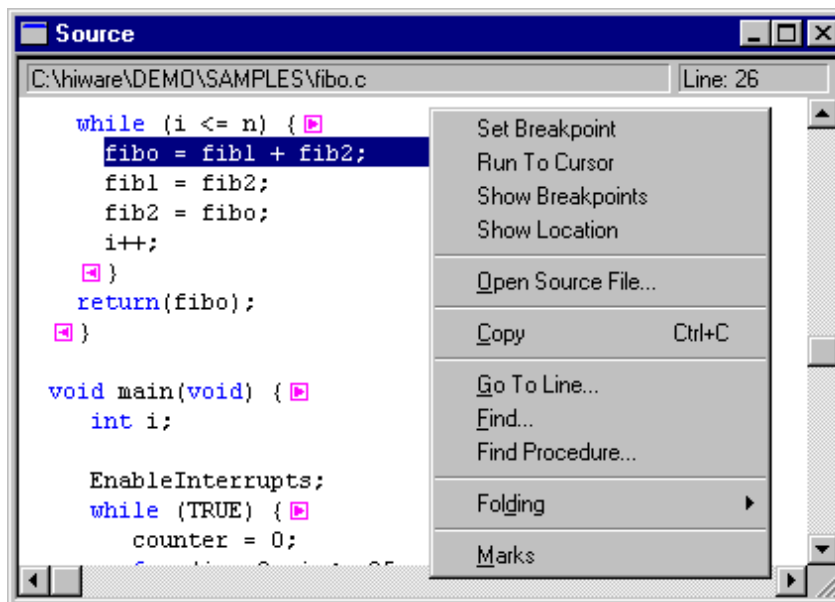
Each component is a windows file with a **.wnd** extension

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 4.31 Example of Component Popup Menu



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

However some entries are identical with entries in the main menu.

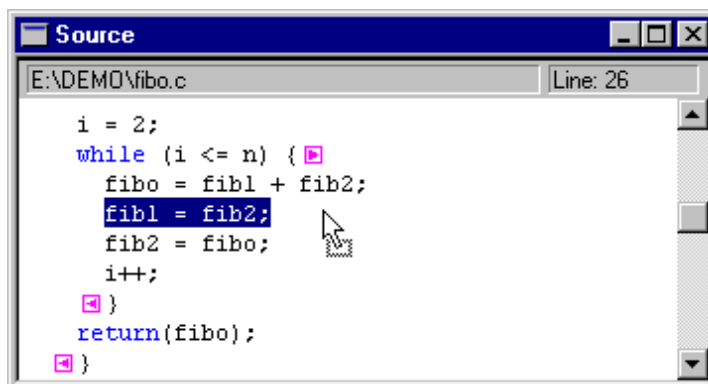
Highlights of the User Interface

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

Smart User Interface: Activating Services with Drag and Drop

You can activate services by dragging objects from one component to another. This is known as drag and drop, an example is shown in [Figure 4.32](#).

Figure 4.32 Drag and Drop Example

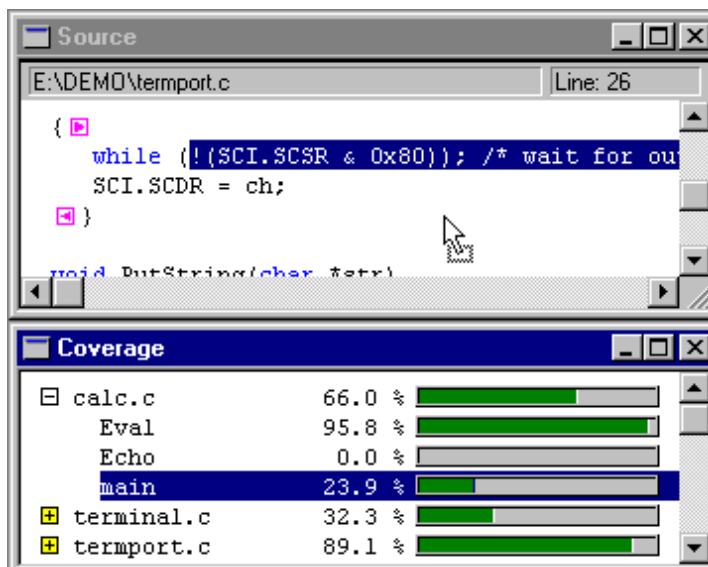


When the destination of a dragged item is not possible, the following cursor symbol is displayed:

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 4.33](#)).

Figure 4.33 Dragging the chosen procedure name from the “Coverage Component,” to the Source.



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

- 1. Select the component containing the object you want to drag.**
- 2. Make sure the destination component where you want to drag the object is visible.**
- 3. Select the object you want.**
- 4. Press and hold the left mouse button, drag the object onto the destination component 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 the Assembly Component

[Table 4.10](#) summarizes dragging from the Assembly Component.

Table 4.10 Dragging from the Assembly Component

Destination compo.	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 the Data Component

[Table 4.11](#) summarizes dragging from the Data Component.

Table 4.11 Dragging from the Data Component



Destination compo.	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 the Source component

[Table 4.12](#) summarizes dragging from the Source Component.

Table 4.12 Dragging from the Source component

Destination compo.	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.
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 and Expression Editor)

Dragging from the Memory component

[Table 4.13](#) summarizes dragging from the Memory Component.

Table 4.13 Dragging from the Memory component

Destination component.	Action
Assembly	Displays disassembled instructions starting at the first address selected. Instructions corresponding to the selected memory area are highlighted in the Assembly component.



Destination component.	Action
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 the Procedure component

[Table 4.14](#) summarizes dragging from the Procedure Component.

Table 4.14 Dragging from the Procedure component

Destination compo.	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 the Register component

[Table 4.15](#) summarizes dragging from the Register Component.

Table 4.15 Dragging from the Register component

Destination compo.	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 the Module component

[Table 4.16](#) summarizes dragging from the Register Component.

Table 4.16 Dragging from the Module component

Destination compo.	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 Simulator/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 Target](#)

[Open IO component](#)

[Open Source File](#)

[Open Module](#)

[Components Window](#)

Framework Components

This Chapter introduces the concepts of the Debugger Components.

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

- [Component Introduction](#)
- [Components Window](#)
- [General Component](#)
- [Visualization Utilities](#)

Component Introduction

The Simulator/Debugger kernel includes various components.

CPU component

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 simulator/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 Target 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 Simulator/Debugger automatically detects the MCU type and loads the appropriate CPU component, if available on your environment.

Window components

The Simulator/Debugger 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 target interface (to communicate with different targets), and the symbol table. The Simulator/Debugger window components are implemented as dynamic link libraries (DLLs) with extension **.WND**. These components are introduced in this section.

Target components

Different debugger targets are available. For example, you can set a CPU Simulator to simulate your .ABS application files, and also set a background debugger.

One target shall be loaded at any time. Either a simulator or a driver implements the link to the target system. The simulator implements the CPU and memory simulation and may be extended by I/O simulation. Different targets are available to connect the target system (hardware) to the Simulator/Debugger on the Host computer. For example, the target may be connected using an Emulator, a ROM monitor or any other supported device.

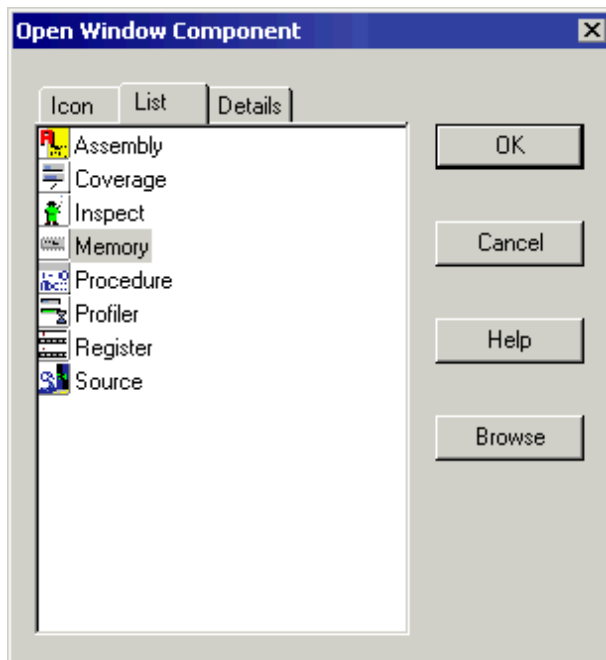
NOTE Target components are introduced in their respective manual.

Components Window

Use the Component menu to load all framework components.

1. Choose **Component>Open...**
2. In the dialog shown in [Figure 5.1](#), select the desired component.

Figure 5.1 Open Window Component Dialog



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

3. Click **OK** to open the selected component.

The **Icon** panel shows you components with large icons.

The **List** panel shows you components with small icons.

The **Details** panel shows you components with their description.

Demo Version Limitations

Maximum number of components opened at a time is limited to 8.

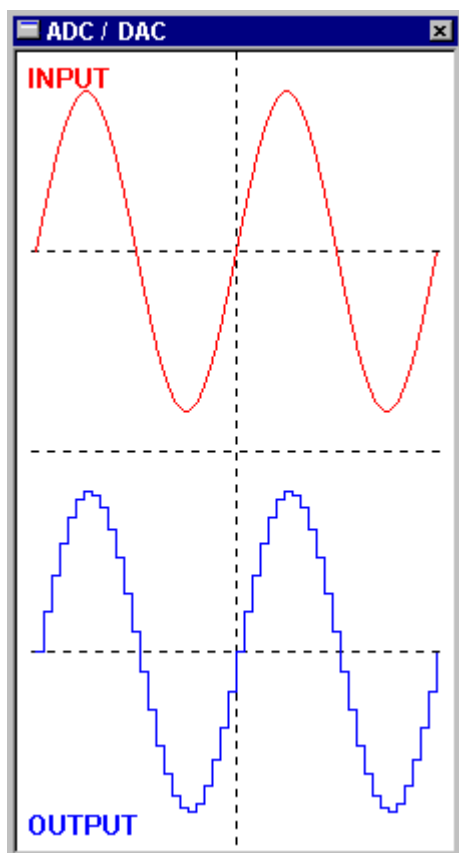
General Component

This chapter describes features of the debugger components.

Adc_Dac component

The Adc_Dac component window, shown in [Figure 5.2](#) consists of a Digital to Analog and an Analogic to Digital converter.

Figure 5.2 Adc_Dac Component

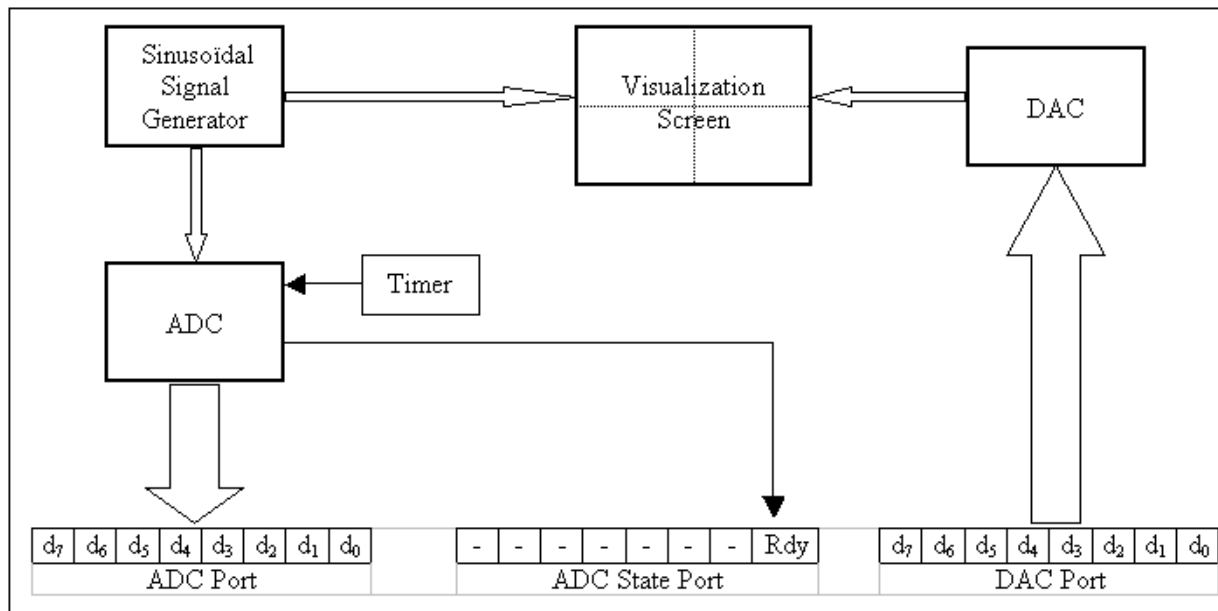


Description

This component is made of 4 units as shown in [Figure 5.3](#):

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

Figure 5.3 Internal converter module organization and coupler connections.



The 4th unit shows the value of the initial analogic signal and value of the DAC output analogic 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

The signal generator

It only generates a sinus signal. The generator output is connected to the ADC visualization screen.

The visualization screen

A 200 point horizontal resolution screen. The sinus signal period is deployed by default in red, in the upper part of the screen, and the signal generated by the DAC is displayed in blue in the lower part.

The ADC

An 8 bit resolution converter generating unsigned values. As we can see in the figure below, 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.

The 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.

Menu

The Adc-Dac menu shown in [Figure 5.4](#) contains all functions associated with the Adc-Dac component. These entries are described in [Table 5.1](#).

Figure 5.4 Adc-Dac menu

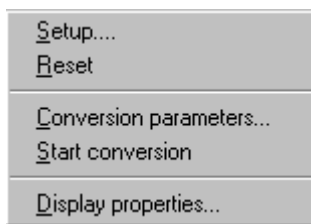


Table 5.1 Adc-Dac Menu Description

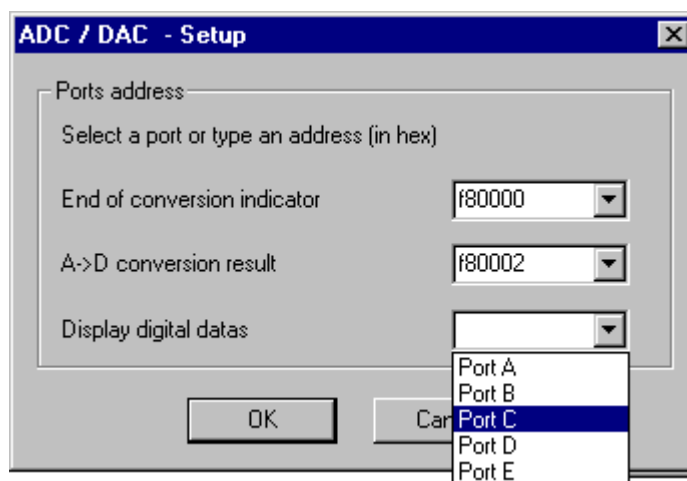
Menu entry	Description
Setup	Opens the dialog box allowing you to set the port addresses.

Menu entry	Description
Reset	This function erases the visualization screen and re-initializes the display properties.
Conversion parameters	Opens the dialog box allowing you to set the signal frequency
Start conversion	Runs the conversion process
Display properties	Opens the dialog box allowing you to set the display properties

Adc_Dac Setup dialog

This dialog shown in [Figure 5.5](#) 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.

Figure 5.5 Adc-Dac Setup dialog

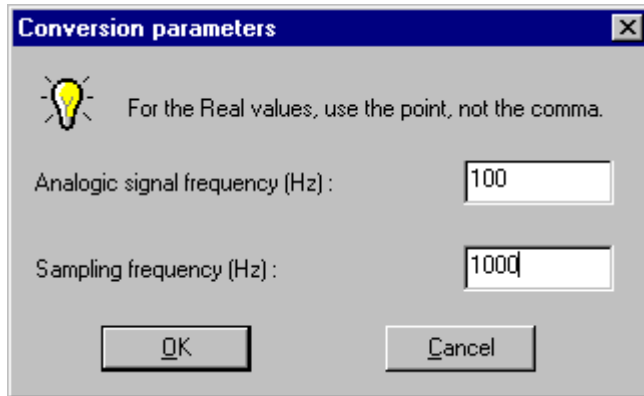


Adc_Dac Conversion parameters dialog

This dialog box shown in [Figure 5.6](#) allows you to choose the analogic signal frequency generated by the sinusoidal generator and the sampling frequency.

The choice of these two frequencies will internally initialize the timer, which will give the conversion orders.

Figure 5.6 Adc_Dac Conversion parameters dialog

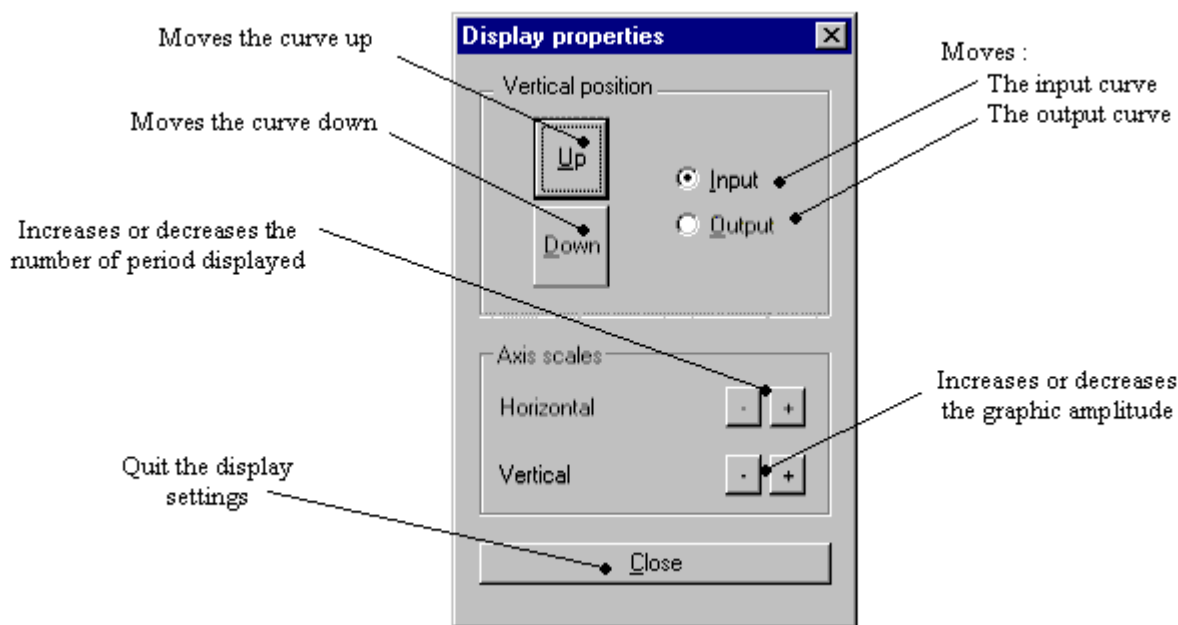


Now you can start the conversion with Start conversion menu entries.

Adc_Dac Display properties dialog

This dialog box shown in [Figure 5.7](#) 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 5.7 Adc_Dac Display properties dialog



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.

Demo Version Limitations

No limitations

Associated Commands

Following commands are associated with the Adc_Dac component:

[ADCPORT](#), [LINKADDR](#)

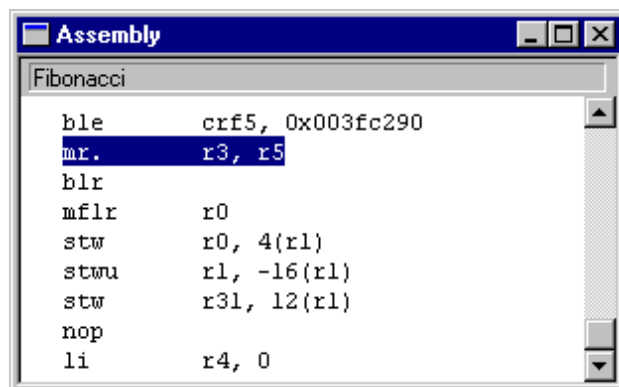
Assembly Component

The Assembly component window, shown in [Figure 5.8](#) displays program code in disassembled form.

Description

The Assembly component 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 5.8 Assembly Component



The 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 Simulator/Debugger Components](#) 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.

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., as explained below.

NOTE For information on using breakpoints, see [Define Breakpoints](#) chapter.

Menu

The Assembly menu shown in [Figure 5.9](#) contains all functions associated with the assembly component. These entries are described in [Table 5.2](#).

Figure 5.9 Assembly Menu

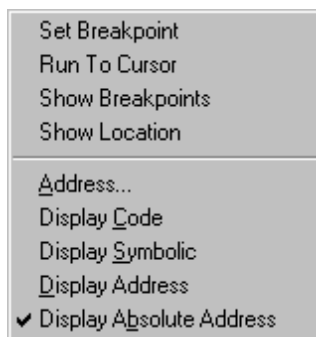


Table 5.2 Assembly Menu Description

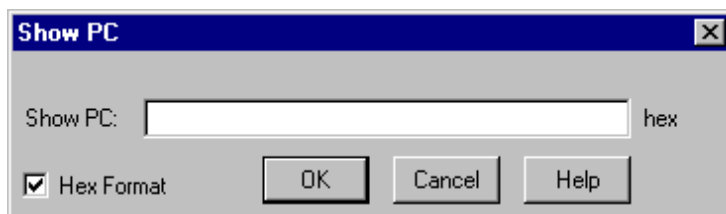
Menu entry	Description
Address...	Opens a dialog box prompting for an address: Show PC.

Menu entry	Description
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.

Show PC Dialog

If an hexadecimal address is entered in the Show PC Dialog shown in [Figure 5.10](#), memory contents are interpreted and displayed as assembler instructions starting at the specified address.

Figure 5.10 Show PC Dialog



Associated Popup Menu



To open the popup menu right-click in the text area. The popup menu contains default menu entries for Assembly component (see above). It also contains some context dependent menu entries described in [Table 5.3](#); depending on the current state of the simulator/debugger.

Table 5.3 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 Breakpoints setting dialog box and displays list of breakpoints defined in the application (refer to Control Points).

Menu entry	Description
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.

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.
-  + : 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 5.4](#) shows the drag and drop actions possible from the Assembly component.

Table 5.4 Drag and Drop possible from the Assembly Component.

Destination component	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 5.5](#) shows the drag and drop actions possible in the Assembly component

Table 5.5 Drop Into Assembly Component

Source component	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.

Source component

Action

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](#), [SMEM](#), [SPC](#).

Command Line Component

The Command component shown in [Figure 5.11](#) interprets and executes all Simulator/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 5.11 Command Line Component






Description

This section explains functions of the Command component.





Command key in.

You can type Simulator/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 , ,  (to retype the previous command).


Scrolling the Command Component Window Content

Use  and  to move the cursor on the line,  to move the cursor to the beginning of the line,  to move the cursor to the end of the line.



NOTE To scroll a page, use the PgDn (scroll down a page) or PgUp keys (scroll up a page).

Clear the line or a character of the Command Line

Selected text can be deleted by pressing .

To clear the current line type .

Command interpretation

The component executes the command entered, displays results or error messages, if any. Ten previous commands can be recalled using  to scroll up or  to scroll down. Commands are displayed in blue.

Prompts and command responses are displayed in black. Error messages are displayed in red.

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

"Command Component is busy. Closing will be delayed"

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

Variable checking in the Command Line

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

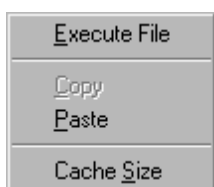
Closing the Command Line during an execution

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

Menu and popup menu




[Figure 5.12](#) shows the Command component menu and popup menu.


Figure 5.12 Command Component Menu and popup menu






Clicking **Execute File** opens a dialog where you can select a file containing Simulator/Debugger commands to be executed. These 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  + .
- 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.

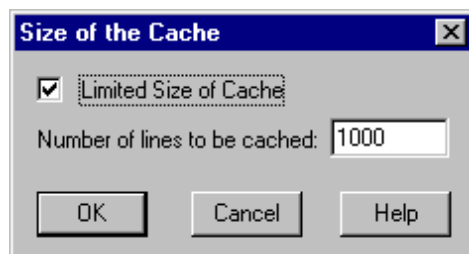
The first line of text contained in the clipboard can be pasted where the caret is blinking (end of current line) by:

- selecting the menu entry **Command>Paste**
- pressing  + .
- clicking the  button in the toolbar.

Cache Size

Select **Cache Size** in the menu to set the cache size in lines for the Command Line window, as shown in [Figure 5.13](#).

Figure 5.13 Cache Size Dialog



This Cache Size dialog 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 5.6](#). The command line component appends corresponding items of the current command.

Table 5.6 Drop Into the Command Component

Source component	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 Simulator/Debugger session.

Command files with more than 20 commands cannot be executed.

Associated Commands

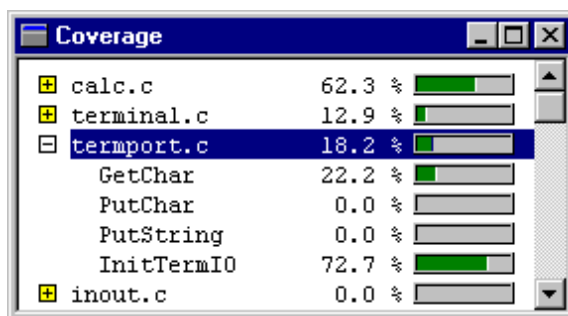
[BD](#), [CF](#), [E](#), [HELP](#), [NB](#), [LS](#), [SREC](#), [SAVE](#).

NOTE For more details about commands, refer to [Debugger Commands](#).

Coverage Component

The Coverage component window, shown in [Figure 5.14](#) contains source modules and procedure names as well as percentage values representing the proportion of executed code in a given source module or procedure.

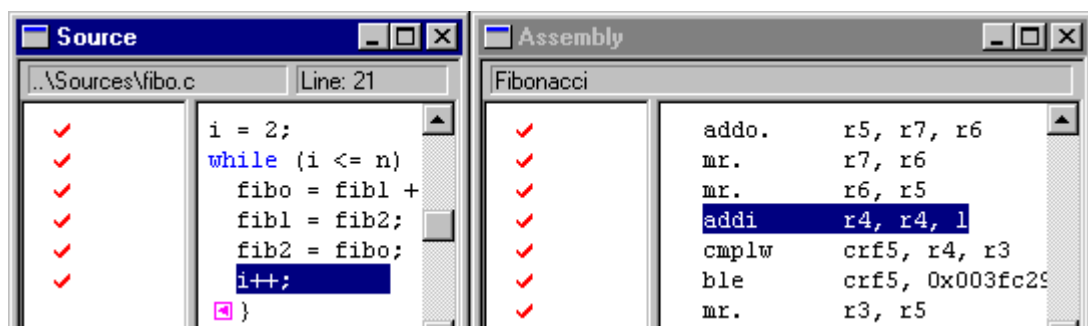
Figure 5.14 Coverage Component



Description

The Coverage component window contains percentage numbers and graphic bars. From this component, you can split views in the Source component window and Assembly component window, as shown in [Figure 5.15](#). A mark ✓ is displayed in front of each source or assembler instruction that has been executed. Split views are removed when the Coverage component is closed or selecting Delete in the split view popup menu.

Figure 5.15 Coverage Splitting views



Operations

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

Menu

The coverage menu is shown in [Figure 5.16](#) and submenus in [Figure 5.17](#) and [Figure 5.18](#).

Figure 5.16 Coverage Menu

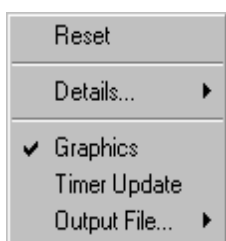


Figure 5.17 Coverage Details Submenu

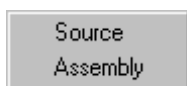


Figure 5.18 Coverage Output File Submenu

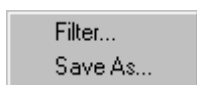


Table 5.7 Coverage Menu Description

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.

Menu entry	Description
Timer Update	Switches the periodic update on/off. If activated, statistics are updated each second.
Output File	Opens the Output File options.

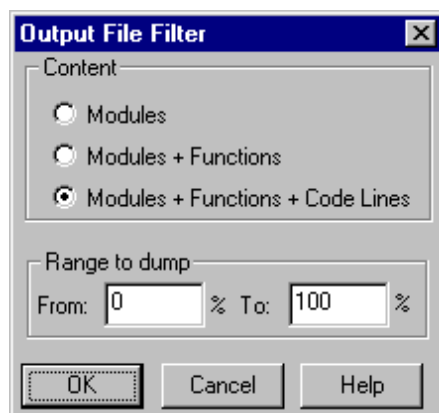
Output File

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 shown in [Figure 5.19](#). 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 5.19 Output File Filter



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 5.1](#).

Listing 5.1 Example of an output file with modules and functions:

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

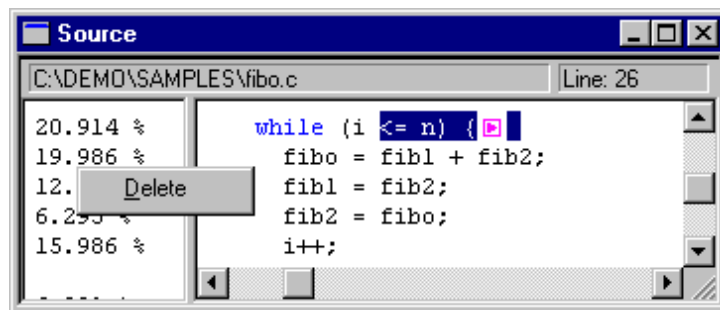
Associated Popup Menu

Identical to menu.

Split view associated Popup Menu

The popup menu for the split view ([Figure 5.20](#)) contains the **Delete** entry, which is used to remove the split view.

Figure 5.20 Coverage Split view associated Popup Menu



Drag Out

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

Drop Into

Nothing can be dropped into the Coverage Component window.



Demo Version Limitations

Only modules are displayed and the Save function is disabled.

Associated Commands

[DETAILS](#), [FILTER](#), [GRAPHICS](#), [OUTPUT](#), [RESET](#), [TUPDATE](#)

DAC Component

The DAC component shown in [Figure 5.21](#) is an interface module between the DA-C IDE.

Figure 5.21 DAC Component



Description

The DAC component is an interface module between the DA-C IDE (Development Assistant for C - from RistanCASE GmbH) allowing synchronized debugging features.

Operation

When the DAC component is loaded, 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](#).

Menu

Figure 5.22 DAC Menu



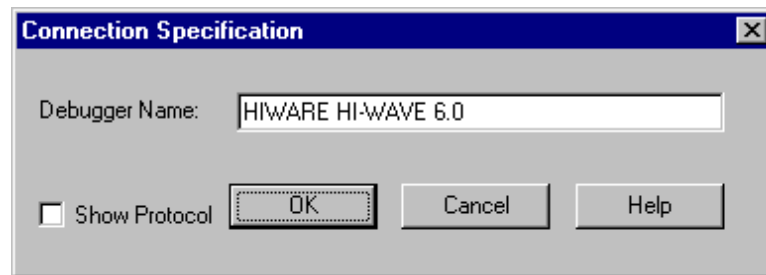
Table 5.8 DAC Menu Description

Menu entry	Description
Setup	Opens the connection setup Window.

Connection Specification

In the dialog shown in [DAC Connection Specification](#), you can set the DA-C debugger name.

Figure 5.23 DAC Connection Specification



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 Simulator/Debugger. To validate the settings, click the **OK** button. A new connection is established and the "Connection Specification" is saved in the current Project.ini file. The **HELP** button opens the help topic for this dialog.

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

Drag Out

Nothing can be dragged out.

Drop Into

Nothing can be dropped into the DAC Component window.

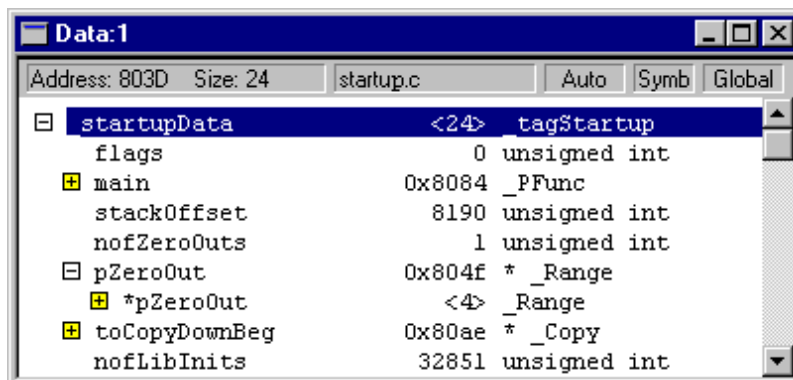
Demo Version Limitations

None.

Data Component

The Data Component window shown in [Figure 5.24](#) contains the names, values and types of global or local variables.

Figure 5.24 Data Component



Description

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

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

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













Structures can be expanded to display their member fields.

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

Watchpoints can be set in this component. Refer to [Control Points](#) chapter.

Operations

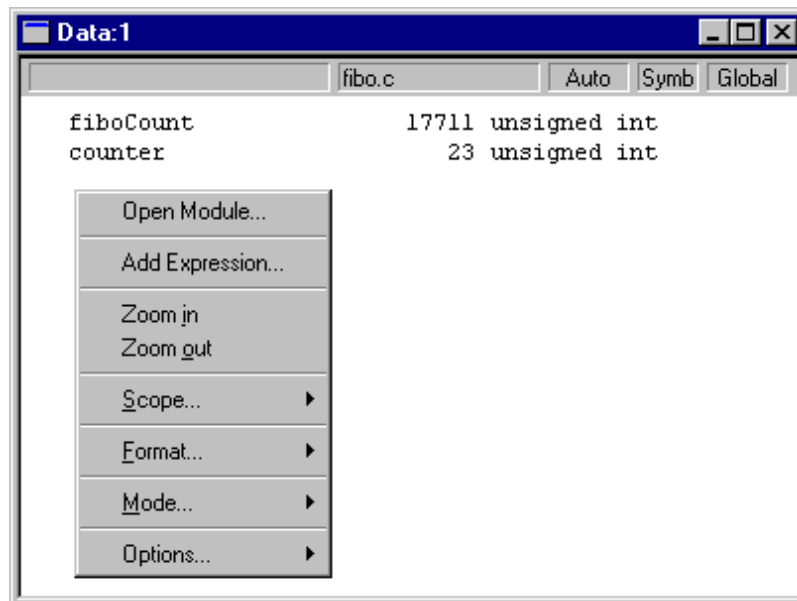
- Double-click a variable line to edit the value.

- Click the folded/unfolded bitmaps   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  +  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  +  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  +  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  +  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  +  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 ([EBNF Notation](#)) double-click a blank line in the Data component window to open the **Edit Expression** dialog shown in [Figure 5.26](#), or point to a blank line as shown below and right-click to select **Add Expression...** in the popup menu shown in [Figure 5.25](#).

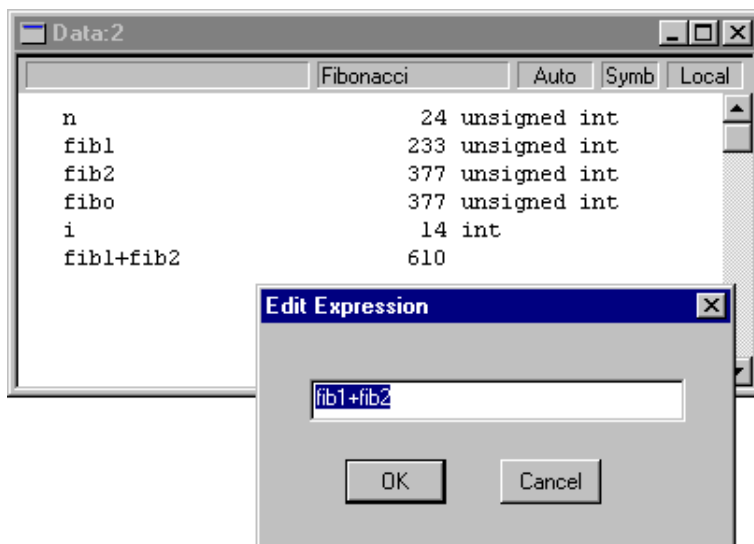
Figure 5.25 Expression Editor Dialog



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.

NOTE The definition of **expression** and examples are in the Appendix [EBNF Notation](#)

Figure 5.26 Edit Expression Dialog



Example: with 2 variables **variable_1**, **variable_2**;

expression entered: **(variable_1<<variable_2)+ 0xFF) <= 0x1000** will result 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

This file is automatically generated when a new application is loaded or exiting from the Simulator/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 **fib0.abs**, the debugger executes **Fib0.xpr**

Menu

[Figure 5.27](#) shows the Data component menu, the Data Scope submenu is shown in [Figure 5.28](#), the Data Format submenu in [Figure 5.29](#), the Data

Mode submenu in [Figure 5.30](#) and the Data Options submenu in [Figure 5.31](#). Menu entries are described in [Table 5.9](#).

Figure 5.27 Data Menu



Figure 5.28 Data Scope Submenu



Figure 5.29 Data Format Submenu

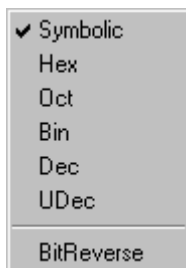


Figure 5.30 Data Mode Submenu

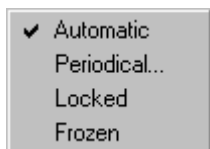


Figure 5.31 Data Options Submenu

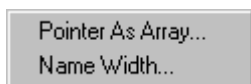


Table 5.9 Data Menu Description

Menu entry	Description
Zoom in	Zooms in the selected structure. The member field of the structure replaces the variable list.
Zoom out	Returns to the previous level of development.
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.

Scope Submenu

The [Table 5.10](#) describes the Scope submenu entries.

Table 5.10 Data Scope Submenu

Menu entry	Description
Global	Switches to Global variable display in the Data component.
Local	Switches to Local variable display in the Data component.

Menu entry	Description
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

[Table 5.11](#) describes the Data Format submenu entries.

Table 5.11 Data Format 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 Sub Menu

The [Table 5.12](#) describes the Data Mode Sub Menu entries.

Table 5.12 Data Mode Sub Menu

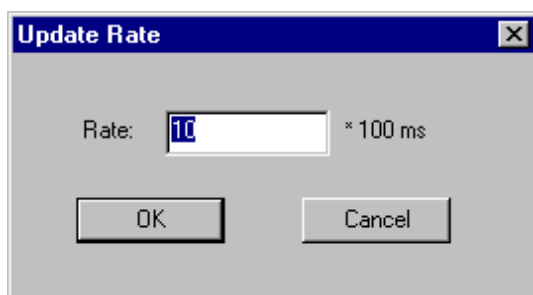
Menu entry	Description
Automatic	Switches to Automatic mode (default), variables are updated when the target is stopped. Variables from the currently executed module or procedure are displayed in the data component.
Periodical	Switches to Periodical mode: variables are updated at regular time intervals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

This dialog box shown in [Figure 5.32](#) allows you to modify the default update rate by steps of 100 ms.

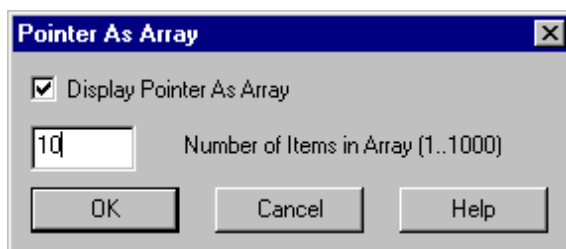
Figure 5.32 Update Rate Dialog



Pointer as Array option

In the Data component menu or popup menu, choose **Options...>Pointer as Array...** to open the dialog shown in [Figure 5.33](#).

Figure 5.33 Pointer as Array Dialog

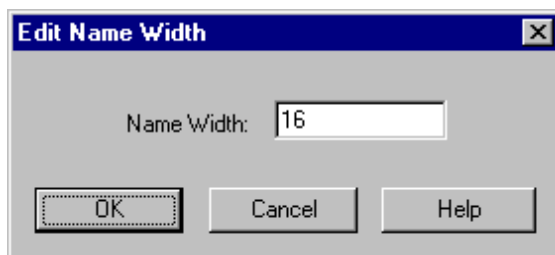


Within this dialog, you can display pointers as arrays, assuming that the pointer points to the first item (**pointer[0]**). Note that this setup is valid for all pointers displayed in the Data window. Check the **Display Pointer as Array** checkbox and set the number of items that you want to be displayed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

Name Width Option

Choose **Options... > Name Width...** to open the window shown in [Figure 5.34](#).

Figure 5.34 Edit Name Width Dialog



This dialog 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.

Associated Popup Menu

[Table 5.13](#) specifies the Data Popup Menu entries.

Table 5.13 Data Popup Menu

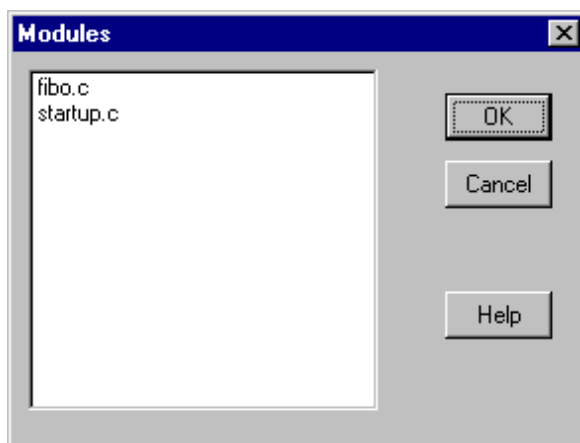
Menu entry	Description
Open Module...	Opens the dialog Open Module .
Set Watchpoint	Appears only in the popup menu if no watchpoint is set or disabled on the pointed to variable. When selected, sets a read/write watchpoint on this variable. A yellow vertical bar is displayed for the variables on which 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.

Menu entry	Description
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).
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 5.35](#) 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 5.35 Open Modules Dialog



Drag Out

[Table 5.14](#) describes the Drag and Drop actions possible from the Data component.

Table 5.14 Dragging Data Out

Destination compo.	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.

WARNING! 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 5.15](#) describes the Drag and Drop actions possible in the Data component.

Table 5.15 Drop Into Data

Source component	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 , Expression Editor .
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.

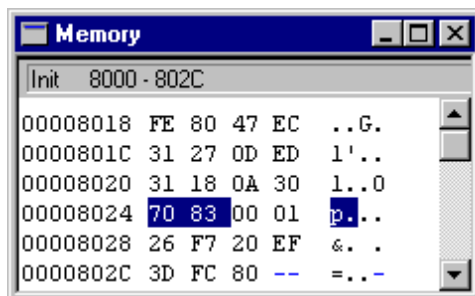
Associated Commands

[ADDXPR](#), [ATTRIBUTES](#), [DUMP](#), [PTRARRAY](#), [SMOD](#), [SPROC](#), [UPDATERATE](#), [ZOOM](#).

Memory Component

The Memory Component window shown in [Figure 5.36](#) displays unstructured memory content or memory dump, i.e. continuous memory words without distinction between variables.

Figure 5.36 Memory Component



Description

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 [Watchpoints setting dialog](#) for more information about watchpoints.

A memory area can be initialized with a fill pattern using the [Fill Memory Dialog](#) 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 [Object Info Bar of the Simulator/Debugger Components](#) 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)

TIP 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.

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.
-  + **A** 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  + **R** 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  + **W** 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  + **B** 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 + to delete the watchpoint on the selected memory area. The memory area is no longer underlined.
- Select a memory range, and + 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.

Menus

The Memory Menu shown in [Figure 5.37](#) provides access to memory commands. [Table 5.16](#) describes the menu entries.

Figure 5.37 Memory Menu

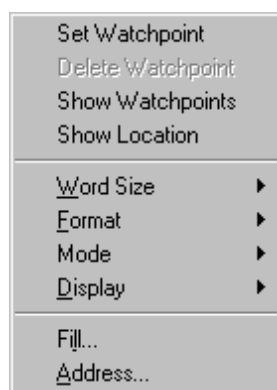


Table 5.16 Memory Menu Description

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.

Menu entry	Description
Fill...	Opens the Fill Memory Dialog to fill a memory range with a bit pattern.
Address...	Opens the memory dialog and prompts for an address.
CopyMem	Opens the CopyMem dialog that allows you to copy memory range values to a specific location.

Word Size Submenu

With the Word Size submenu shown in [Figure 5.38](#), you can set the memory display unit. [Table 5.17](#) describes the menu entries.

Figure 5.38 Word Size Submenu

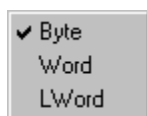


Table 5.17 Word Size Submenu Description

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 5.39](#), you can set the memory display format. [Table 5.18](#) describes the menu entries.

Figure 5.39 *Format Submenu*

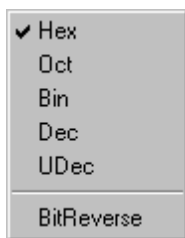


Table 5.18 *Format Submenu Description*

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
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 5.40](#), you can set the memory mode format. [Table 5.19](#) describes the menu entries.

Figure 5.40 *Mode Submenu*

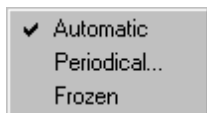


Table 5.19 Mode Submenu Description

Menu entry	Description
Automatic	Selects Automatic mode (default), memory dump is updated when the target is stopped.
Periodical	Selects the Periodical mode, memory dump is 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 (see below).
Frozen	Selects the Frozen mode, memory dump displayed in the memory component is not updated when the target is stopped.

Displays Submenu

With the Displays submenu shown in [Figure 5.41](#), you can set the memory display (address/ascii). [Table 5.20](#) describes the menu entries.

Figure 5.41 Displays Submenu

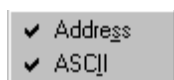


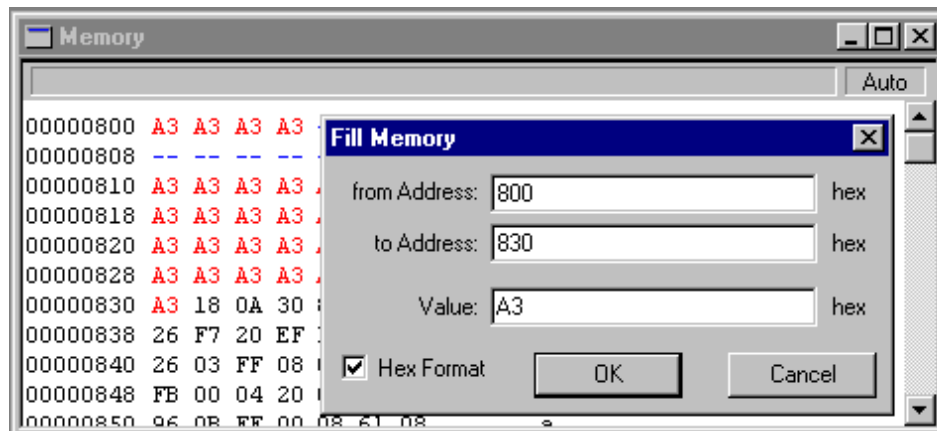
Table 5.20 Displays Submenu Description

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 Dialog

This dialog shown in [Figure 5.42](#) allows you to fill a memory range (**from Address** edit box and **to Address** edit box) with a bit pattern (**value** edit box).

Figure 5.42 Fill Memory Dialog

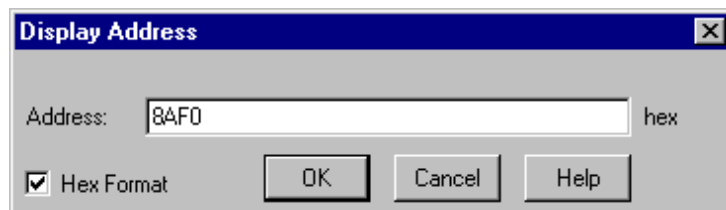


NOTE If “Hex Format” is checked, numbers and letters are interpreted as hexadecimal numbers. Otherwise, expressions can be typed and Hex numbers should be prefixed with “Ox” or “\$”. Refer to [Constant Standard Notation](#).

Display Address Dialog

With the dialog shown in [Figure 5.43](#), the memory component dumps memory starting at the specified address.

Figure 5.43 Display Address Dialog

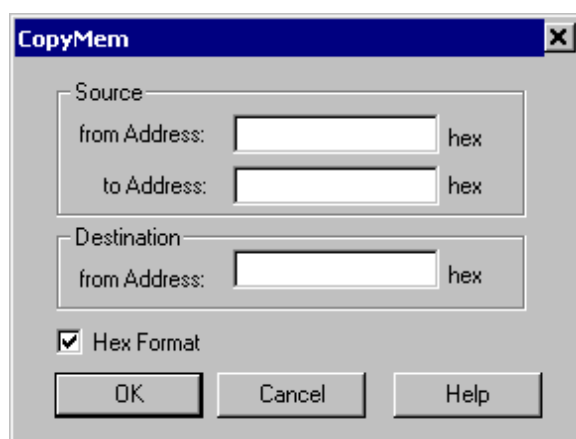


NOTE The **Show PC** dialog box is the same as the Display Address dialog box. In this dialog, the Assembly component dumps assembly code starting at the specified address.

CopyMem Dialog

The dialog shown in [Figure 5.44](#) allows you to copy a memory range to a specific address.

Figure 5.44 CopyMem Dialog



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.

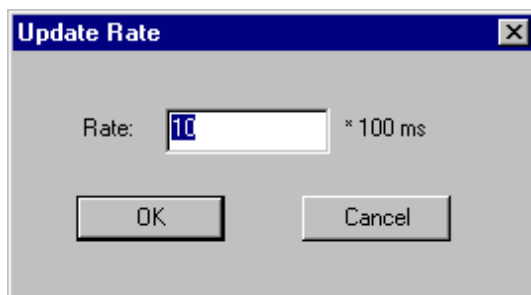
If "**Hex Format**" is checked, all given values are in Hexadecimal Format. You don't need to add "0x". For instance type 1000 instead of 0x1000.

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

Update Mode

This dialog box shown in [Figure 5.45](#) allows you to modify the update rate in steps of 100ms.

Figure 5.45 Update Mode



NOTE Periodical mode is not available for all hardware targets or some additional configuration may be required in order to make it work.

Associated Popup Menu

The memory popup menu shown in [Table 5.21](#) allows you to execute memory associated commands.

Table 5.21 Memory Associated Popup Menu Description

Menu entry	Description
Set/Delete Watchpoint	Appears only in the Popup Menu 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 if a watchpoint is set or disabled on the selected memory range. When selected, deletes this watchpoint.
Enable Watchpoint	Appears in the Popup Menu if a watchpoint is disabled on the selected memory range. When selected, enables this watchpoint.

Menu entry	Description
Disable Breakpoint	Appears in the Popup Menu if a breakpoint is set on the selected memory range. 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 and modify their properties (See “Control Points” chapter).
Show location	Forces all opened windows to display information about the selected memory area.

Drag Out

[Table 5.22](#) Drag and Drop describes the actions possible from the Memory component.

Table 5.22 Drag and Drop possible from the Memory component.

Destination compo.	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 5.22](#) shows the Drag and Drop actions possible in the Memory component.

Table 5.23 Drag and Drop into the Memory component.

Source comp.	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](#), [FILL](#), [SMEM](#), [SMOD](#), [SPC](#), [UPDATERATE](#).

IT_Keyboard

The IT_Keyboard component shown in [Figure 5.46](#) is a 20 key keyboard that generates an interruption when a key is pressed.

Figure 5.46 IT_Keyboard Component

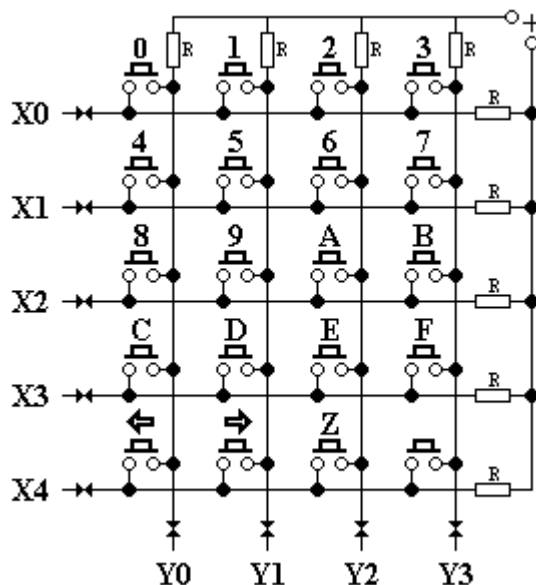


Description

The IT_Keyboard consists of a 20 key keyboard, as shown in [Figure 5.47](#). 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 5.47 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.

Menu

[Figure 5.48](#) shows the IT_Keyboard menu and described in [Table 5.24](#).

Figure 5.48 IT_Keyboard menu

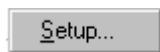


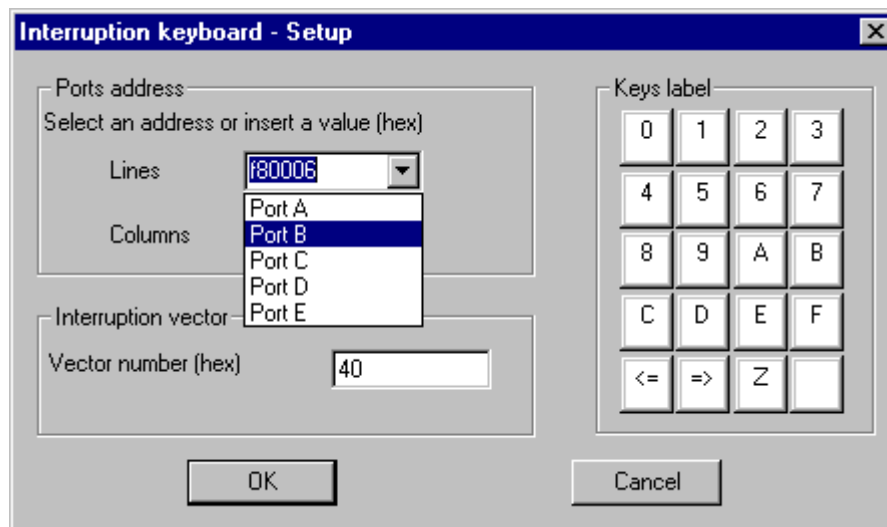
Table 5.24 IT_Keyboard Menu Description

Menu entry	Description
Setup	Opens the Interrupt keyboard setup dialog.

Interrupt keyboard setup dialog

This dialog shown in [Figure 5.49](#) allows you to set the address of the lines port, the columns port and the number of the interruption vector.

Figure 5.49 IT_Keyboard Setup



In the **Port address** section, for each two ports you can insert an address (in hexadecimal) in the **Lines** field or select one of the five ports listed in the **Columns** field. These are used when the component works with the [Programmable IO Ports](#) component.

The **Vector number 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

Associated Commands

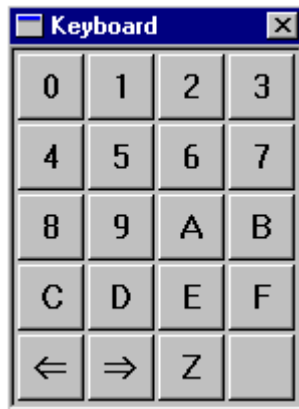
Following commands are associated with the IT_Keyboard component:

[ITPORT](#), [ITVECT](#), [LINKADDR](#)

Keyboard

The Keyboard component shown in [Figure 5.50](#) is a 20 key keyboard.

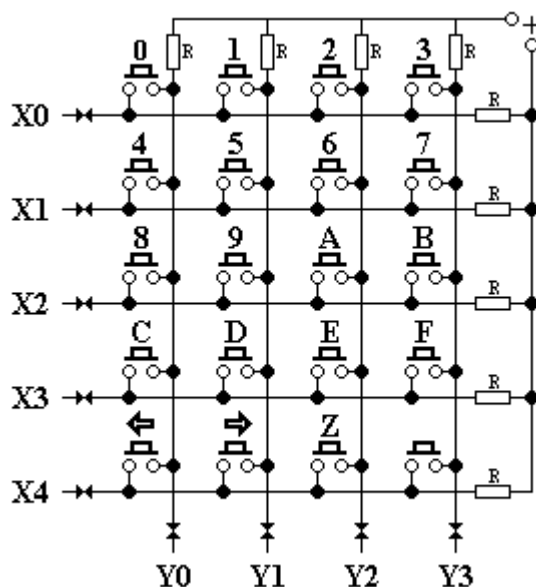
Figure 5.50 Keyboard Component



Description

The Keyboard consists of a 20 key keyboard, as shown in [Figure 5.47](#). 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 have the effect of placing the column (or line) corresponding with 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.

Figure 5.51 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 the line X4 (X3, X2, X1, X0 being at 1).
- Read the column successively, from Y3 to Y0.
- Put a 0 at the line X3 (X4, X2, X1, X0 being at 1).
- Read again the column 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 for scanning. This one could work only when a key is activated and not continually.

Menu

[Figure 5.52](#) shows the Keyboard menu and its entry is described in [Table 5.25](#).

Figure 5.52 Keyboard menu



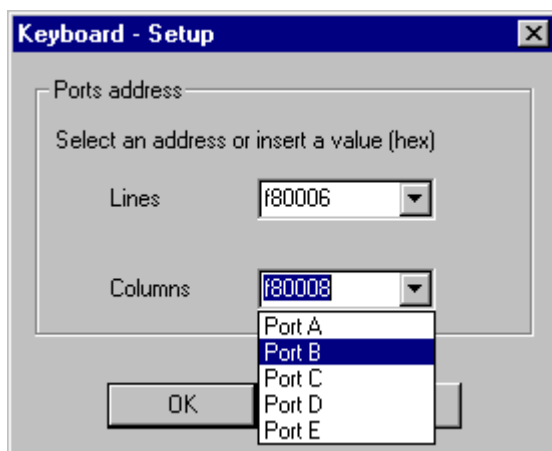
Table 5.25 Keyboard Menu Description

Menu entry	Description
Setup	Opens the Keyboard setup dialog.

keyboard setup dialog

This dialog shown in [Figure 5.49](#) allows you to set the address of the lines port and columns port.

Figure 5.53 Keyboard 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 the programmable [Programmable IO Ports](#) component.

Drag out

Nothing can be dragged out.



Drop Into

Nothing can be dropped into the Keyboard Component window.

Demo Version Limitations

No limitations

Associated Commands

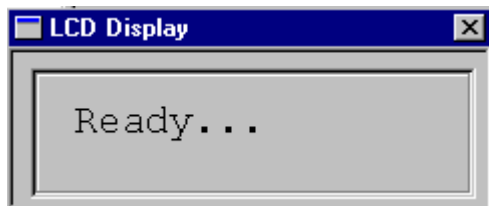
Following commands are associated with the Keyboard component:

[KPORT](#), [LINKADDR](#)

LCD Display Component

The LCD Display component shown in [Figure 5.54](#) is the LCD display utility, which can display 1 or 2 lines of 16 characters and show or hide the position cursor.

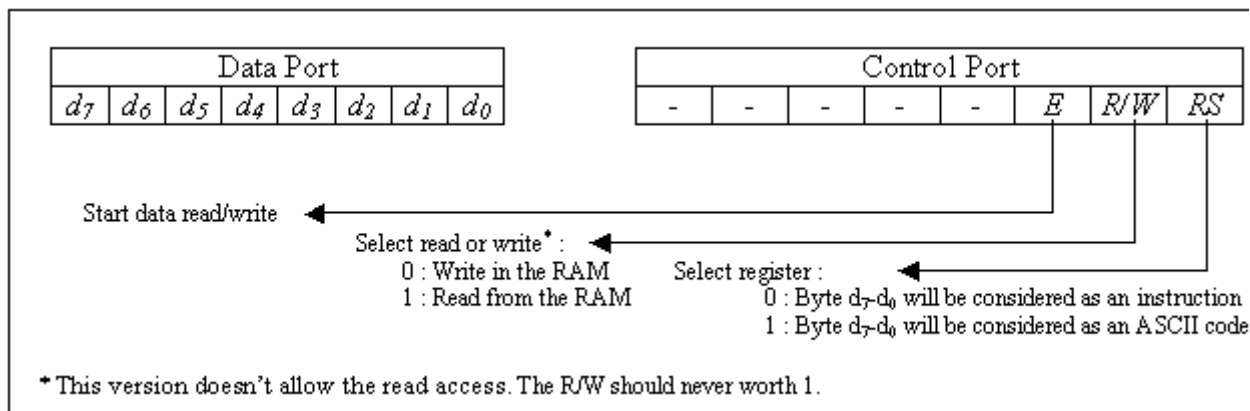
Figure 5.54 LCD Display Component



Description

The display module consists of 2 eight-bit-width parallel couplers: a data port and a control port, as shown in [Figure 5.55](#). These ports communicate with the mainframe.

Figure 5.55 The LCD display module ports



The bits d_7-d_0 represent an ASCII code to display characters or an instruction code. The RS bit defines the status of bits d_7-d_0 .

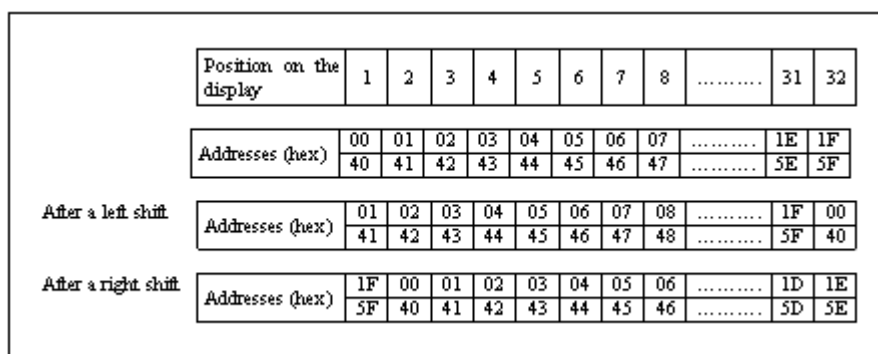
Operation

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 5.56](#) illustrates this arrangement.

Figure 5.56 .The DDRAM controller



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 5.57](#) lists the instructions available for the LCD Display Component.

Figure 5.57 LCD Component 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.

Instruction description

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 5.58](#) describes how to choose the moving direction.

Figure 5.58 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.

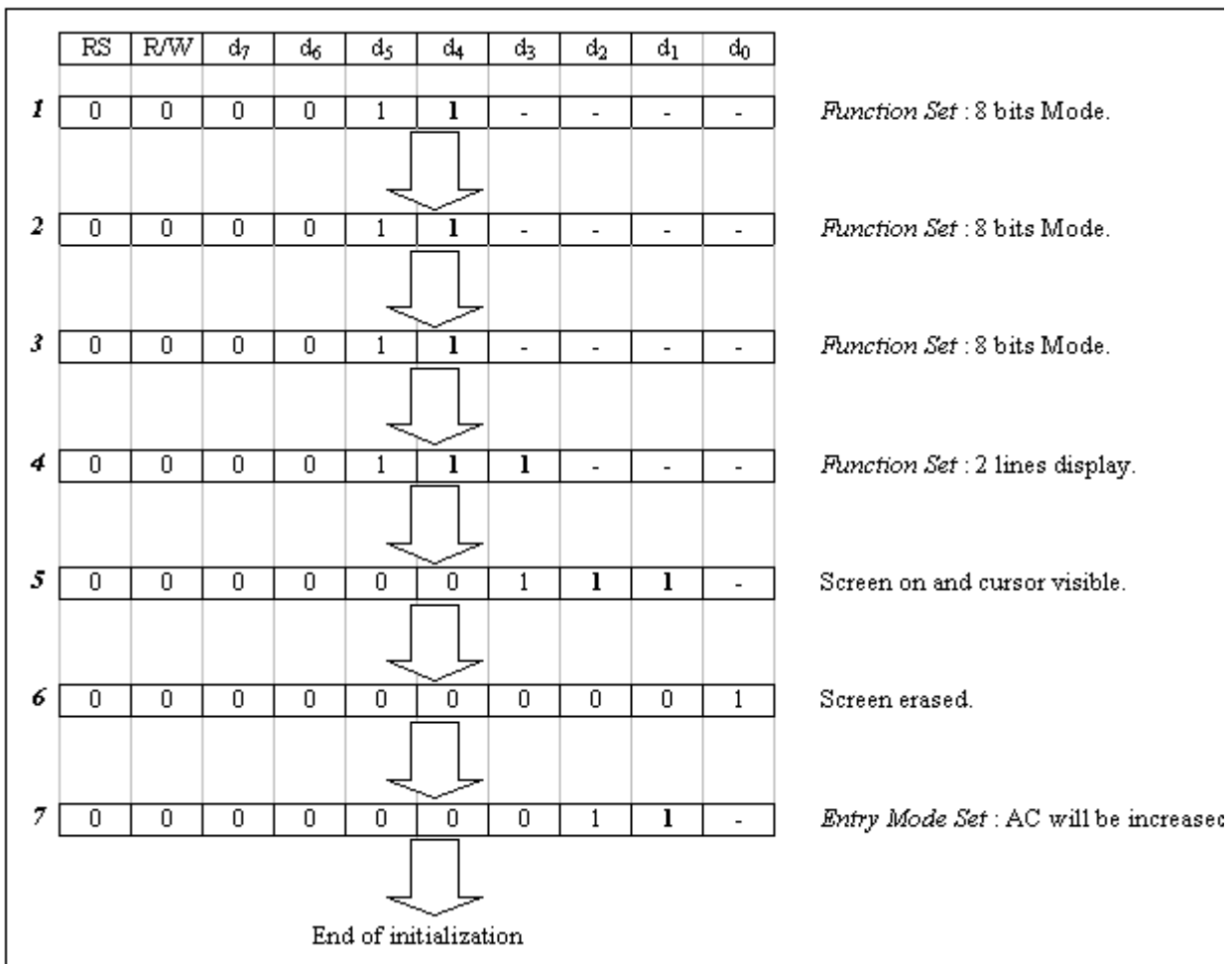
The initialization step

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 5.59](#) 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 5.59 The LCD display initialization



Menu

[Figure 5.60](#) shows the LCD Display menu and its entry is described in [Table 5.29](#).

Figure 5.60 The LCD display menu



The 7-segments display menu contains the Setup function to launch the 7-Segments Display dialog box.

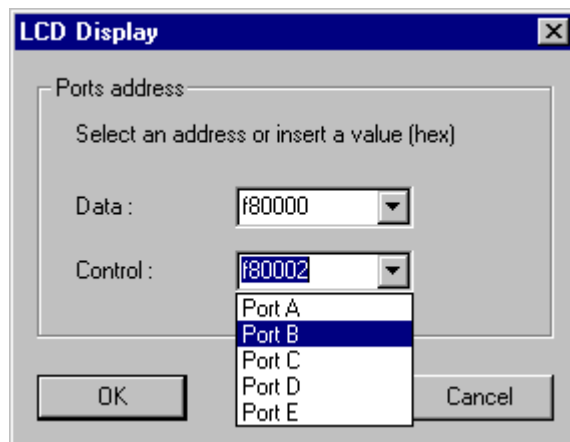
Table 5.26 LCD display Menu Description

Menu entry	Description
Setup	Opens the Lcd display dialog.

Lcd display dialog

This dialog shown in [Figure 5.49](#) allows you to set the address of the lines port and columns port.

Figure 5.61 LCD 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](#).

Drag out

Nothing can be dragged out.

Drop Into

Nothing can be dropped into the Lcd display Component.

Demo Version Limitations

No limitations

Associated Commands

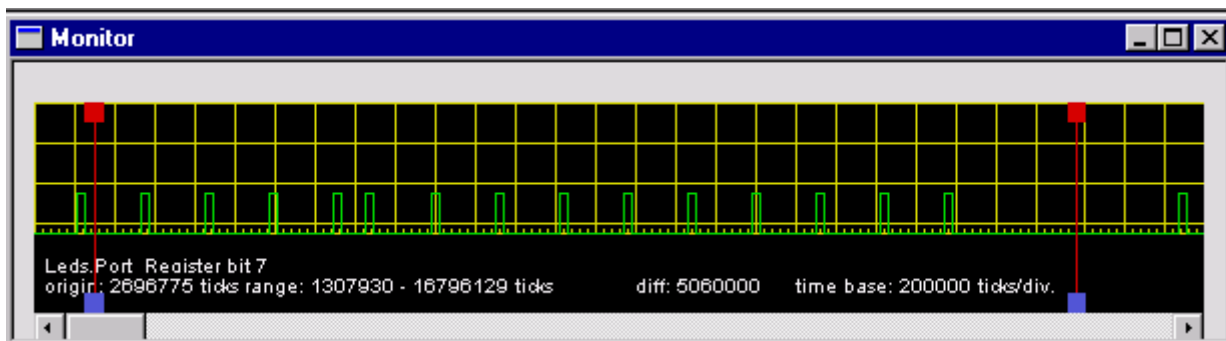
Following commands are associated with the Lcd display component:

[LCDPORT](#), [LINKADDR](#)

Monitor components

The Monitor component shown in [Figure 5.67](#) is a basis oscilloscope that can display the result of debugger objects.

Figure 5.62 Monitor Component



Description

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.

Menu

[Figure 5.63](#) shows the Monitor menu and its entry is described in [Table 5.29](#).

Figure 5.63 The monitor menu

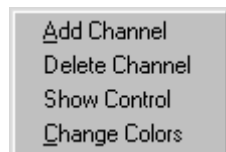


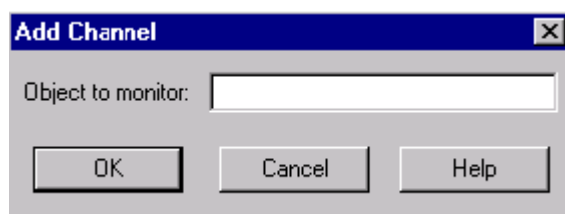
Table 5.27 Monitor Menu Description

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 dialog

This dialog shown in [Figure 5.64](#) allows you to create a new Channel in the monitor.

Figure 5.64 Add Channel dialog



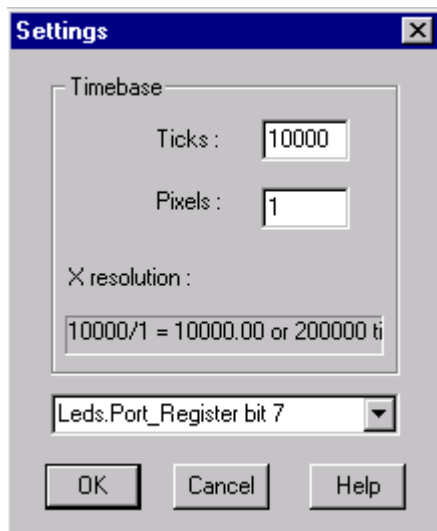
In the text area **Object to monitor**, enter the object name and bit e.g TIM12.PORTT bit 0 and click **OK** to validate or **Cancel** to exit.

Monitor Settings dialog

This dialog shown in [Figure 5.65](#) 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.

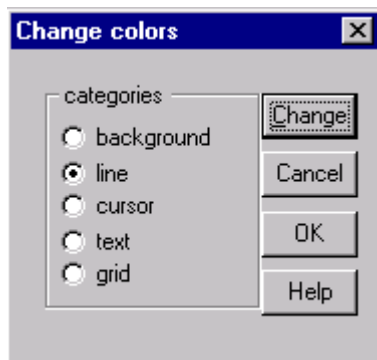
Figure 5.65 Settings dialog



Change colors dialog

This dialog shown in [Figure 5.66](#) allows you to change the colors from the selected Channel.

Figure 5.66 Change colors dialog



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 into the Monitor Component.

Demo Version Limitations

No limitations

Associated Commands

Following commands are associated with the Monitor component:

[ADDCHANNEL](#), [DELCHANNEL](#), [SETCOLORS](#), [SETCONTROL](#)

Push Buttons components

The Push Buttons component shown in [Figure 5.67](#) is a basis input device.

Figure 5.67 Push Buttons Component



Menu

[Figure 5.68](#) shows the LCD Display menu and its entry is described in [Table 5.29](#).

Figure 5.68 The Push Buttons menu

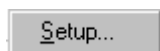


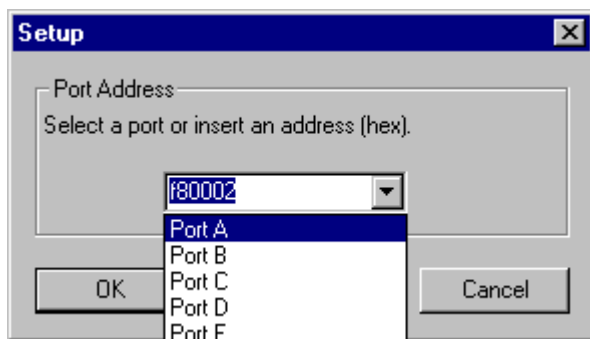
Table 5.28 The Push Buttons Menu Description

Menu entry	Description
Setup	Opens the Push Buttons Setup dialog.

Push Buttons Setup dialog

This dialog shown in [Figure 5.69](#) allows you to specify (in hexadecimal format) the port address or select the port in the list.

Figure 5.69 Push Buttons Setup dialog



NOTE The port should be an output port for the LEDs component.

Use with the IO_Ports

The address defined in the Push Buttons Setup dialog is used when the component works with the [Programmable IO Ports](#).

Use with the Leds component

The Bytes sent to the LEDs component coming from the Push Button component are described in [Figure 5.70](#).

Figure 5.70 Push Buttons Input port

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

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 into the Push Buttons Component.

Demo Version Limitations

No limitations

Associated Commands

Following commands are associated with the Push Buttons Component.

[PBPORT](#), [LINKADDR](#)

MicroC Component

The MicroC component shown in [Figure 5.71](#) is an interface module for RHAPSODY in MicroC, the analysis, design and implementation tool for embedded systems and software developers from I-LOGIX.

Figure 5.71 MicroC Component



Operations

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 Simulator/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.

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

Menu

[Figure 5.72](#) shows the MicroC menu and its entries are described in [Table 5.29](#).

Figure 5.72 MicroC Menu



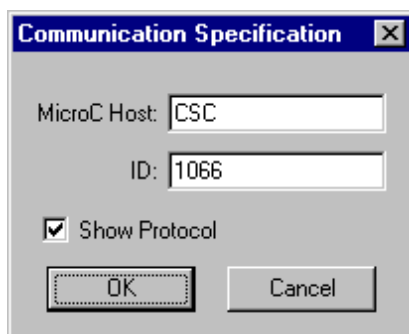
Table 5.29 MicroC Menu Description

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

Communication Specification

Within this dialog shown in [Figure 5.73](#), you can set the MicroC Host and ID for communication between the Simulator/Debugger and RHAPSODY in MicroC. A checkbox allows you to see the communication protocol.

Figure 5.73 MicroC Communication Specification



Drag Out

Nothing can be dragged out.

Drop Into

Nothing can be dropped into the MicroC Component window.

Demo Version Limitations

The MicroC Component is not available in demo mode.

MicroC Component 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/simulator (HI-WAVE) from Metrowerks (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

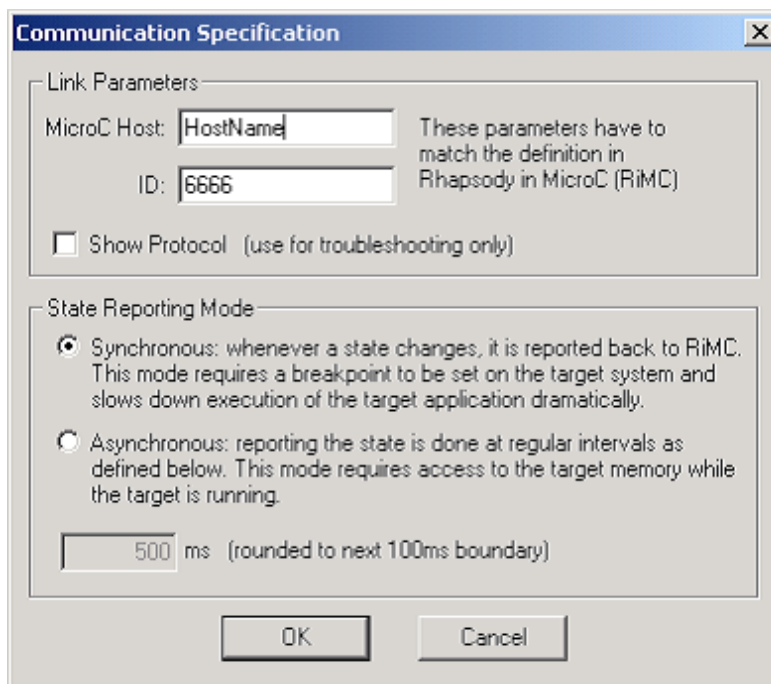
application is changed. When hit, the state is sent to RiMC and the application is resumed immediately. By concept, this procedure will slow down execution of the target application dramatically. Compared to the previous releases, only one single breakpoint is required for this mode.

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 targets that support this mode are the simulator and any Host Target Interface (HTI) that uses the BDM of features dual-ported RAM.

- The Setup dialog was extended to reflect that additional modes:

Figure 5.74 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 specified. If not specified, the previous value will be used.

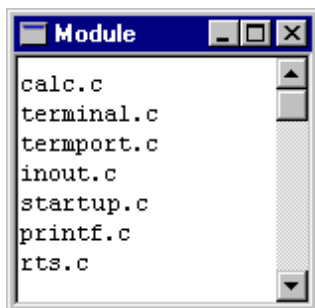
Each of these commands will close any pending communication and re-establish communication with the new parameters.

- in the 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 target.

Module Component

The module component shown in [Figure 5.75](#) gives an overview of source modules building the application.

Figure 5.75 Module Component



Description

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

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.

Menu

The Module Component window has no menu.

Drag Out

[Table 5.30](#) shows the Drag and Drop actions possible from the Module component.

Table 5.30 Drag and Drop possible from the Module component.

Destination compo.	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.

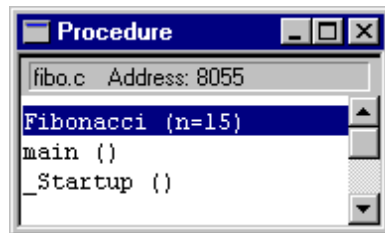
Demo Version Limitations

Only 2 modules are displayed

Procedure Component

The Procedure Component shown in [Table 5.43](#) displays the list of procedure or function calls that have been made so far (up to the moment the program was halted). This list is known as the procedure chain or the call chain.

Figure 5.76 Procedure Component



Description

In the Procedure Component, 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 Simulator/Debugger Components](#) contains the source module and address of the selected 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 Component.

Menu

[Figure 5.77](#) shows the Procedure menu and its entries are described in [Table 5.31](#).

Figure 5.77 Module Menu

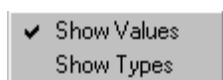


Table 5.31 Module Menu Description

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.

Associated Popup Menu

Identical to menu.

Drag Out

[Table 5.32](#) shows Drag and Drop actions possible from the Procedure component.

Table 5.32 Drag and Drop possible from the Procedure component.

Destination component	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.



**Destination
component**

Action

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.

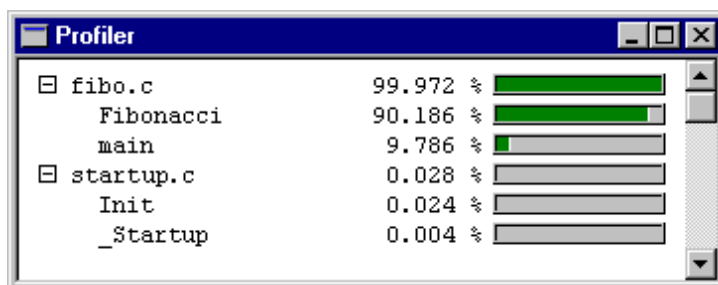
Associated Commands

[ATTRIBUTES](#), [FINDPROC](#)

Profiler Component

The Profiler Component shown in [Figure 5.78](#) provides information on application profile.

Figure 5.78 Profiler Component

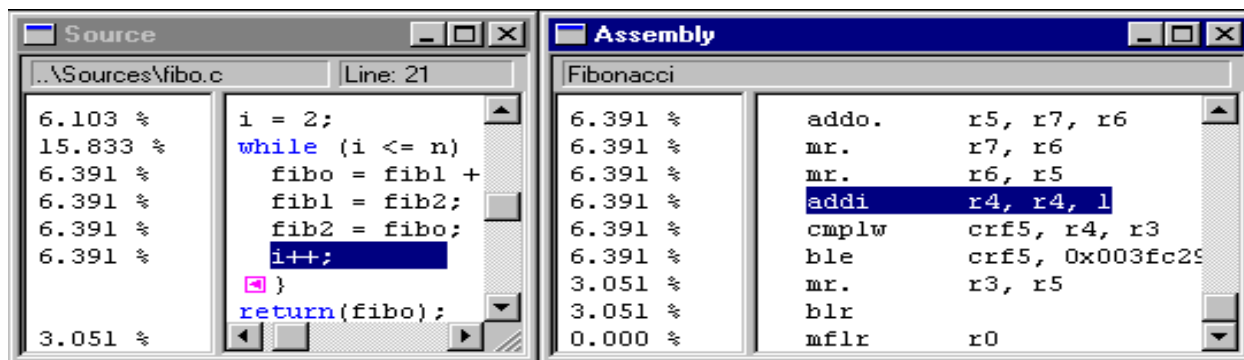


Description

The Profiler component 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 component window can set a split view in the Source and Assembly components ([Figure 5.79](#)).

Figure 5.79 The Profiler split view in the Source and Assembly components



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.

Operations

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

Menu

[Figure 5.80](#) shows the Profiler Menu entries, [Figure 5.81](#) shows the Profiler Details submenu, [Figure 5.82](#) the Profiler Base submenu, and [Figure 5.83](#) the Profiler Output File submenu, which are described in [Table 5.33](#).

Figure 5.80 Profiler Menu



Figure 5.81 Profiler Details Submenu

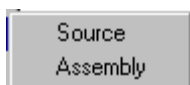


Figure 5.82 Profiler Base Submenu

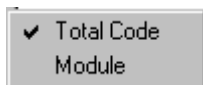


Figure 5.83 Profiler Output File Submenu

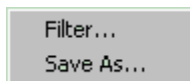


Table 5.33 Profiler menu entries Description

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.
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 .

Split view associated Popup Menu

[Figure 5.84](#) shows the Profiler popup menu, the **Delete** and **Graphics** menu entries are described in [Table 5.34](#).

Figure 5.84 Profiler Split view associated Popup Menu



Table 5.34 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.

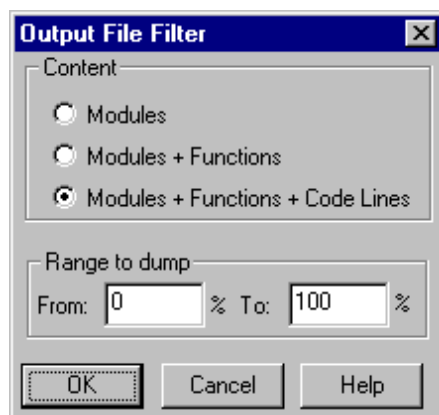
Profiler Output File functions

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 shown in [Figure 5.85](#) 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 5.85 Output File Filter



Output File Save

The **Save As...** entry opens a **Save As** dialog where you can specify the output file name and location.

Associated Popup Menu

Identical to menu.

Drag Out

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

Drop Into

Nothing can be dropped into the Profiler Component window.



Demo Version Limitations

Only modules are displayed and the Save function is disabled.

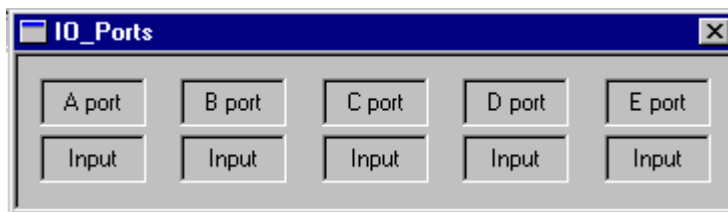
Associated Commands:

[GRAPHICS](#), [TUPDATE](#), [DETAILS](#), [RESET](#), [BASE](#).

Programmable IO_Ports

The Programmable IO_Ports component shown in [Figure 5.86](#) 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 5.86 Programmable IO_Ports Component



Description

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.

Menu

[Figure 5.60](#) shows the Programmable IO_Ports menu and its entry is described in [Table 5.29](#).

Figure 5.87 The Programmable IO_Ports menu

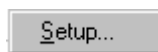


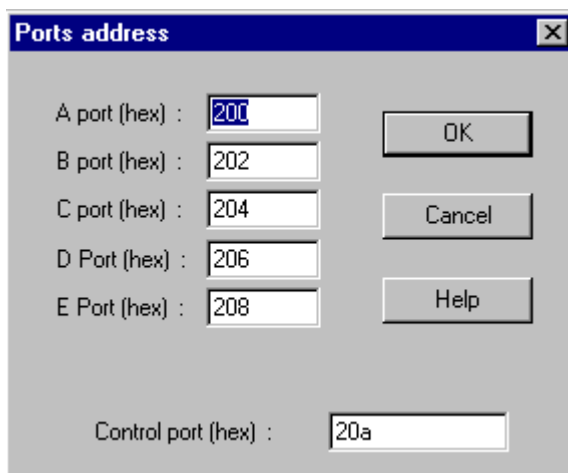
Table 5.35 Programmable IO_Ports menu Description

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

Programmable IO_Ports Port Address dialog

This dialog shown in [Figure 5.88](#) allows you to set the port address and control port address.

Figure 5.88 Programmable IO_Ports Port Address dialog



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 5.89](#).

Figure 5.89 Programmable IO_Ports Address dialog

Control register									Way	
Bits	b7	b6	b5	b4	b3	b2	b1	b0	Input	0
Ports	-	-	-	E	D	C	B	A	Output	1

Drag Out

Nothing can be dragged out.

Drop Into

Nothing can be dropped into the Programmable IO_Ports Component.

Demo Version Limitations

No limitations

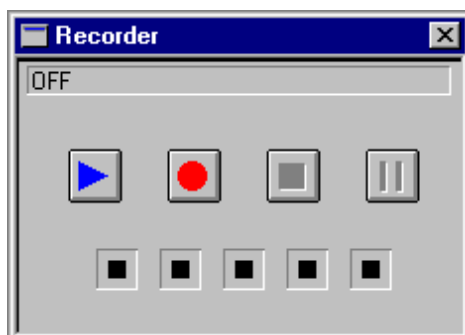
Associated Commands:

[CPORT](#), [LINKADDR](#)

Recorder Component

The Recorder component shown in [Figure 5.90](#) provides record and replay facilities for debug sessions.

Figure 5.90 Recorder Component



Description

The Recorder Component 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 to record, play, pause and stop.



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 Info Bar of the Simulator/Debugger Components](#).

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).

Menu

The recorder menu shown in [Figure 5.91](#) changes according to the current session. The menu items are described in [Table 5.36](#).

Figure 5.91 Recorder Menu



Table 5.36 Recorder Menu Description

Menu entry	Description
Record	Starts recording from a debug session.
Replay	Starts replaying from a debug session.
Pause Replay	Suspends replay in a debug session.
End Replay	Stops replay in a debug session.
End Record	Stops recording 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 5.2](#), a .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 5.2 Record File example

```
at 4537 load C:\Metrowerks\DEMO\fibonacci.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 into the Recorder Component window.



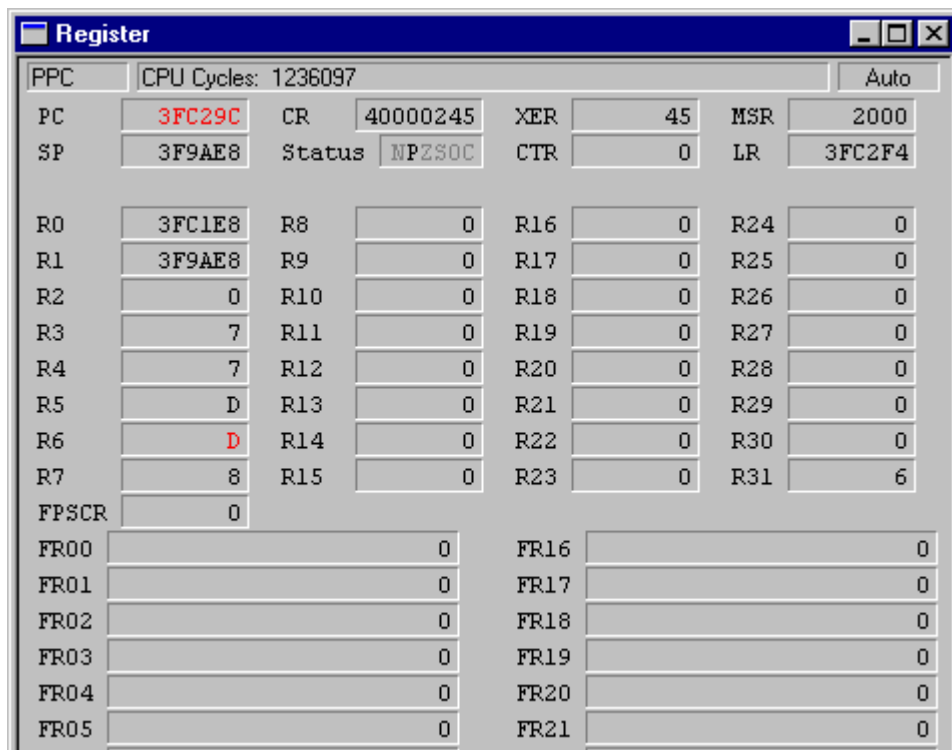
Demo Version Limitations

Only 20 commands will be recorded and replayed.

Register Component

The Register Component window shown in [Figure 5.92](#) displays the content of registers and status register bits of the target processor.

Figure 5.92 Register Component



Description

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 the bit.


During program execution, contents of registers that have changed since the last refresh are displayed in red, except for status register bits.

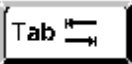
The [Object Info Bar of the Simulator/Debugger Components](#) contains the number of CPU cycles as well as the processor's name.

Editing Registers



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

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

If  is pressed or clicking outside the edited register, the new value is validated and the register content is changed.

If  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.

 + : Contents of Source, Assembly and Memory components change. The Source component shows the source code located at the address stored in the register. The Assembly component shows the disassembled code starting at the address stored in the register. The Memory component dumps memory starting at the address stored in the register.

Right-click: Opens the Register component Popup Menu.

Menu

The register menu contains the items shown in [Figure 5.93](#). [Table 5.37](#) describes the menu entries.

Figure 5.93 Register Menu

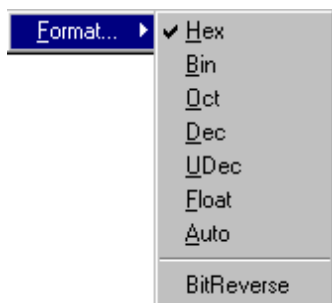


Table 5.37 Register Menu Description

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).

Associated Popup Menu

Identical to menu.

Drag Out

[Table 5.38](#) contains the Drag and Drop actions possible from the Register component.

Table 5.38 Drag and Drop possible from the Register component.

Destination component	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 5.39](#) shows the Drag and Drop actions possible in the Register component.

Table 5.39 Drag and Drop possible in the Register component.

Source component	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.



**Source
component**

Action

Memory

Loads the destination register with the start address of the selected memory block.

Demo Version Limitations

No limitation

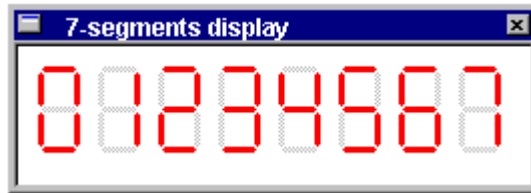
Associated Commands

[ATTRIBUTES.](#)

Seven segments display component

The Seven segments display component shown in [Figure 5.94](#) consists of 8 "7-segment" display systems.

Figure 5.94 Seven segments display component



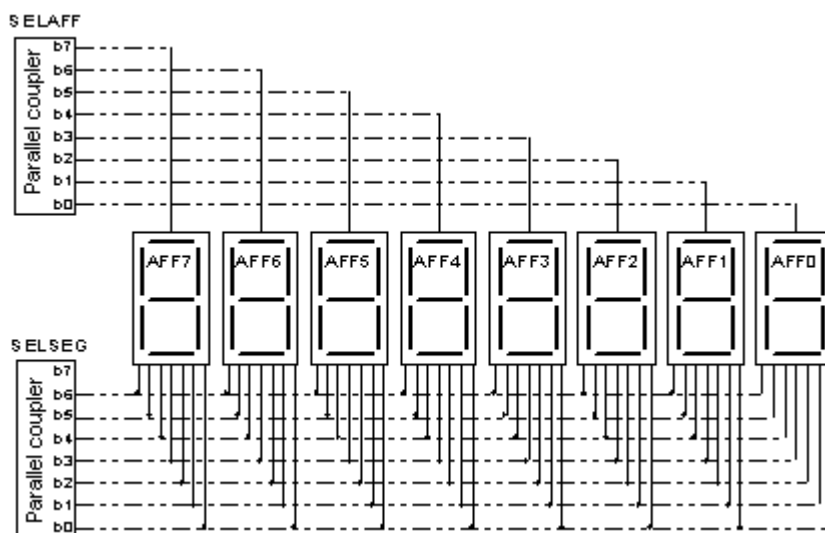
Description

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 5.95](#).

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 5.95 Seven segments display component constitution



Menu

[Figure 5.96](#) shows the Seven segments display component menu and the menu entry is described in [Table 5.40](#).

Figure 5.96 Seven segments display component menu



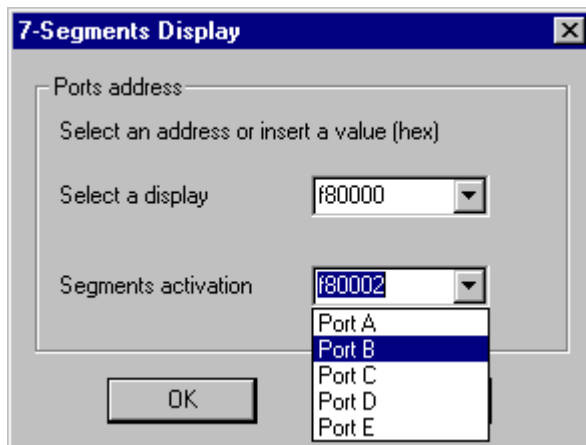
Table 5.40 Seven segments display component Menu Description

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

Seven segments display component setup dialog

This dialog shown in [Figure 5.97](#) allows you to select the display and related value.

Figure 5.97 Seven segments display component setup dialog



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 the component works with the [Programmable IO Ports](#).

Control bits configuration

The 2 bytes sent to the 7 segments must be composed as shown in [Figure 5.98](#).

Figure 5.98 Seven segments display control bits

SELAFF Select of display								SELSEG Select of segments							
b7	b6	b5	b4	b3	b2	b1	b0	b7	b6	b5	b4	B3	b2	b1	b0
Aff7	Aff6	Aff5	Aff4	Aff3	Aff2	Aff1	Aff0	-	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 into the Seven segments display Component window.

Demo Version Limitations

No limitations

Associated Commands

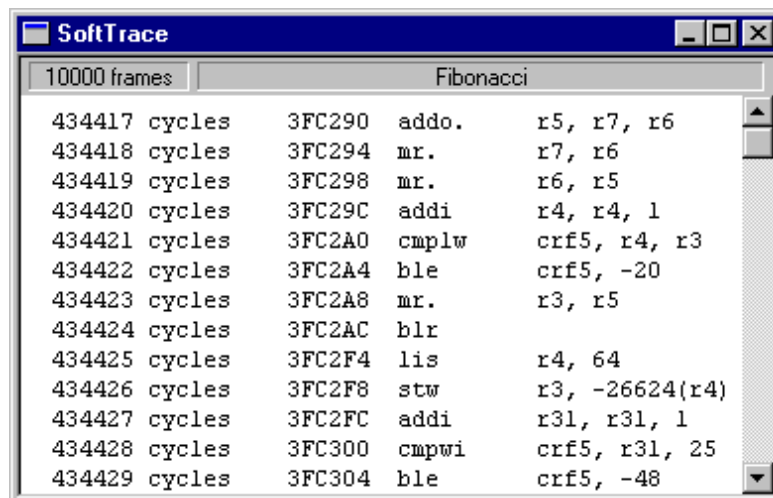
Following commands are associated with the Seven segments display Component:

[SEGPORT](#), [LINKADDR](#)

SoftTrace Component

The SoftTrace Component window shown in [Figure 5.99](#) records and displays instruction frames and time or cycles.

Figure 5.99 SoftTrace Component







Description

The [Object Info Bar of the Simulator/Debugger Components](#) displays the number of recorded frames and the name of the function where the selected frame is located.

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.

 +  sets the zero base frame to the pointed frame.

 +  forces all open component windows to show the code matching the pointed to frame.

Menu

The SoftTrace Menu shown in [Figure 5.100](#) contains the functions described in [Table 5.41](#).

Figure 5.100 SoftTrace Menu



Table 5.41 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 5.101](#) contains functions (described in [Figure 5.101](#)) associated with the pointed to frame.

Figure 5.101 SoftTrace Associated Popup Menu



Table 5.42 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 into the SoftTrace component window.

Demo Version Limitations

The number of frames is limited to 50.

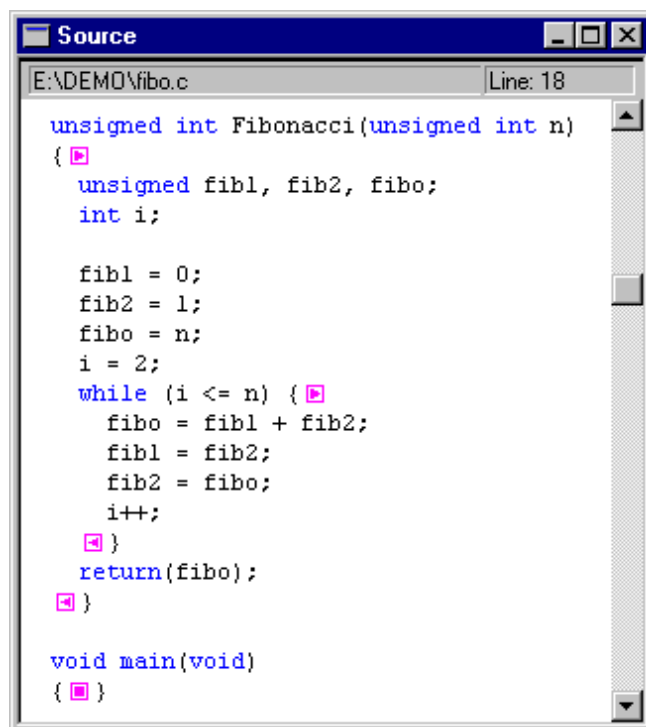
Associated Commands

[CLOCK](#), [CYCLE](#), [FRAMES](#), [RECORD](#), [RESET](#).

Source Component

The Source Component window shown in [Figure 5.102](#) displays the source code of your program, i.e. your application file.

Figure 5.102 Source Component




```
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)
{
}
```

Description


The Source Component 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  + 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.


The source statement matching the current PC is selected (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.

A temporary breakpoint has the following symbol: 

A permanent breakpoint has the following symbol: 

A disabled breakpoint looks like: 

A counting breakpoint has the following symbol: 

A conditional breakpoint has the following symbol: 

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

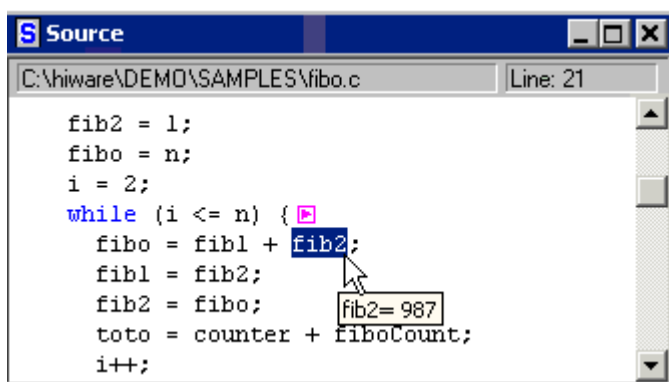
The complete path of the displayed source file is written in the [Object Info Bar of the Simulator/Debugger Components](#).

NOTE You cannot edit the visible text in the Source component 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 5.103](#)) or the parameter value and address of the selected procedure. A parameter or procedure can be selected by double-clicking it.

Figure 5.103 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 disassembling, refer to section [How to Consult Assembler Instructions Generated by a Source Statement](#).



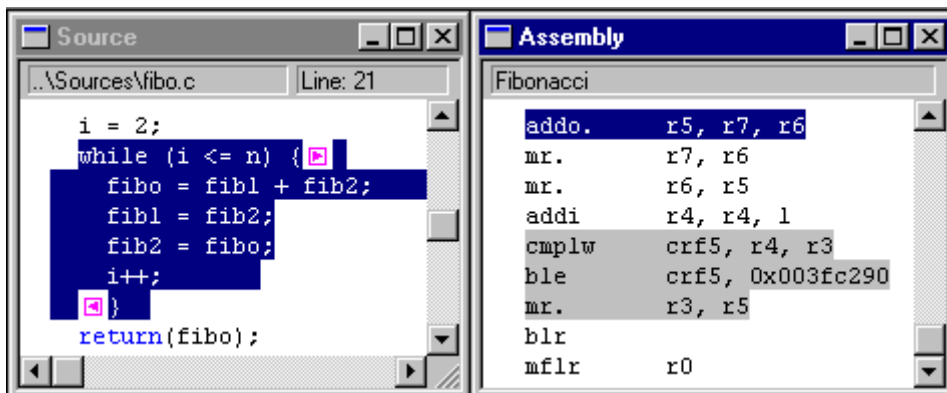
- Select a range of instructions in the source component and drag it into the assembly component. The corresponding range of code is highlighted in the Assembly component window, as shown in [Figure 5.104](#).
-  + : Highlights a code range in the Assembly component window corresponding to the first line of code selected in the Source component window where the operation is performed. This line or code range is also highlighted.

Figure 5.104 On Line Disassembling



Setting Temporary Breakpoints

For information on how to set breakpoints refer to sections in the [Control Points](#) chapter.



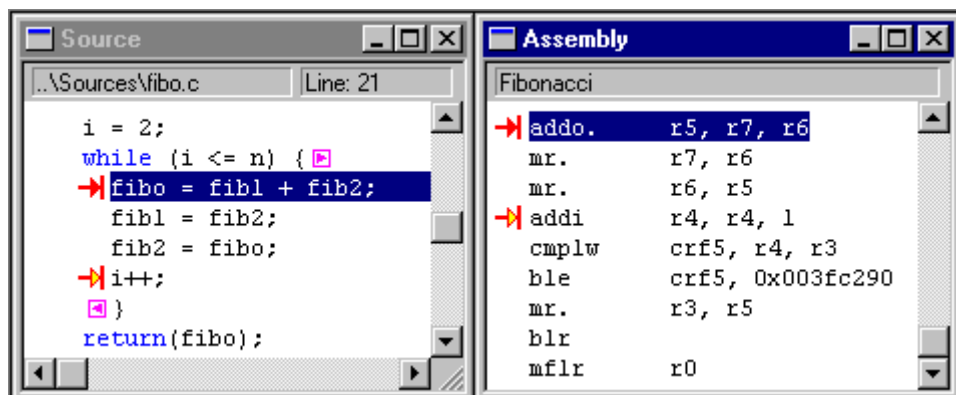




- Point to an instruction in the Source component Window and click the right mouse button. The Source Component popup menu is displayed. Select **Run To Cursor** from the popup menu. The application continues execution and stops at this location.
-  + : Sets a temporary breakpoint at the nearest code position (visible with marks) thereafter the program runs and breaks at this location, as shown in [Figure 5.105](#).

Figure 5.105 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 pointed to source statement.
-  + : Sets a permanent breakpoint at the nearest code position (visible with marks). The permanent breakpoint icon  is displayed in front of the pointed to source statement.

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.

[Figure 5.106](#) and [Figure 5.107](#) shows the functions associated with the Source component. [Table 5.43](#) describes these functions.

Figure 5.106 Source Associated Pop - Up Menu

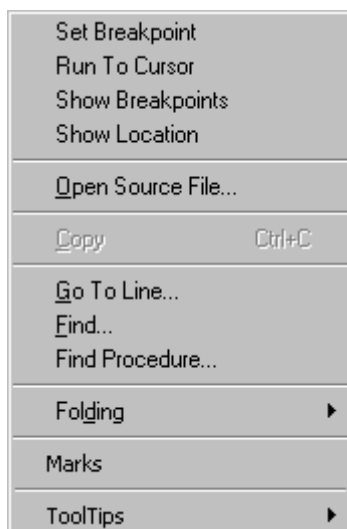


Figure 5.107 Second Source Associated Pop - Up Menu

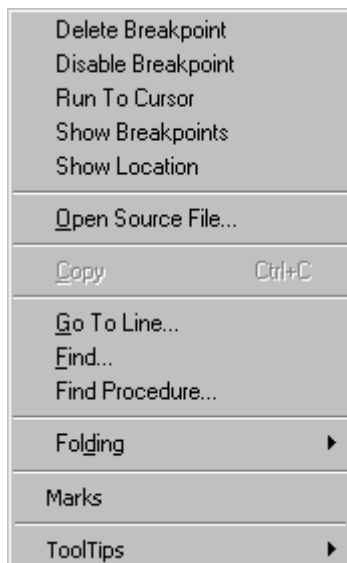


Table 5.43 Associated Pop - Up Menu Description

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.
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 Breakpoints Setting dialog box and allows you to view the list of breakpoints defined in the application and modify their properties (See Control Points chapter).

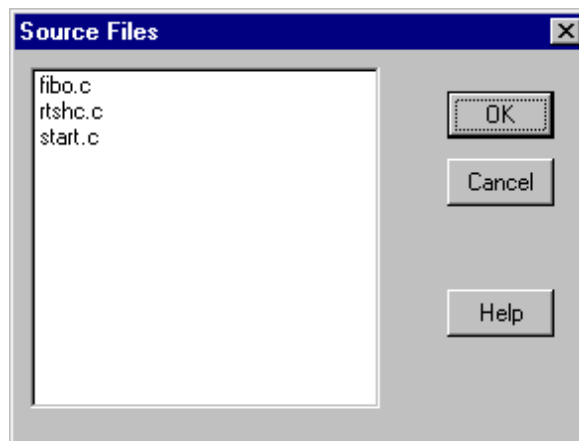
Menu entry	Description
Show Location	Highlights a code range in the Assembly component window matching the line or selected source code. The line or the source code range are highlighted as well.
Open Source File	Opens the Source File Dialog if a CPU is loaded (see chapter below).
Copy (CTRL+C)	Copies 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	Opens a dialog box to scroll the window to a number line (see chapter below).
Find...	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	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 Dialog

The Open Source File dialog shown in [Figure 5.108](#) allows you to open 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 5.108 Open Source File Dialog

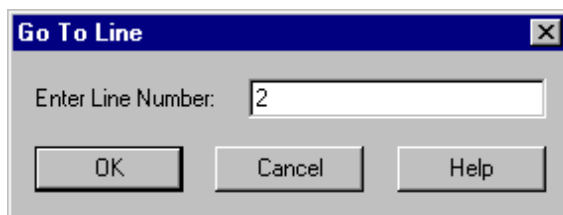


Go to Line Dialog

This menu entry is only enabled if a source file is loaded. It opens the dialog shown in [Figure 5.109](#).

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 5.109 Go to Line Dialog



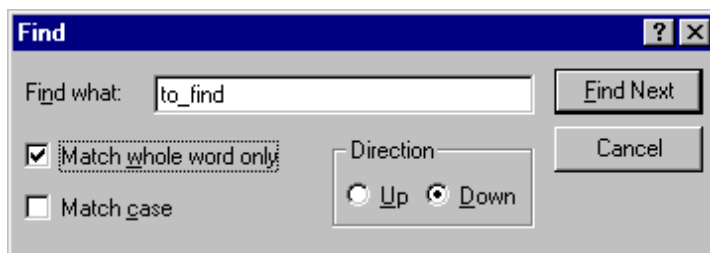
When this dialog is open, the line number of the first visible line in the source is displayed and selected in the **Enter Line Number** edit box.

Find Dialog

The Find Dialog, shown in [Figure 5.110](#) 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 5.110 Find Dialog



The dialog box allows you to specify the following options:

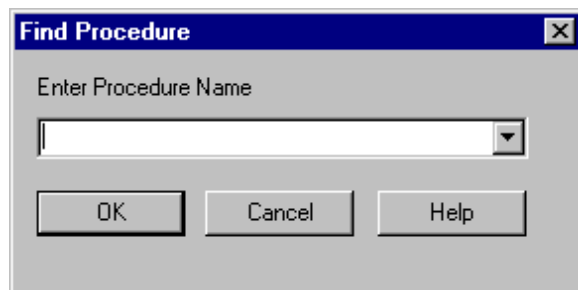
- **Match whole word only:** If this box is checked, only strings separated by special characters will be recognized.
- **Match case:** If this box is checked, the search is case sensitive.

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 Dialog

The Find Procedure dialog, shown in [Figure 5.111](#) 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 5.111 Find Procedure Dialog



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 5.112](#) allows you to select the Fold functions described in [Table 5.44](#).

Figure 5.112 Folding Menu

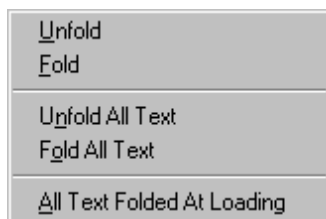


Table 5.44 Folding Menu Description

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 5.45](#) shows the Drag and Drop actions possible from the Source component.

Table 5.45 Drag and Drop possible from the Source component

Destination compo.	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



Destination compo.	Action
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 or Expression Editor)

Drop Into

[Table 5.46](#) shows the Drag and Drop actions possible into the Source component.

Table 5.46 Drag and Drop possible into the Source component.

Source compo.	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](#), [FIND](#), [FOLD](#), [FINDPROC](#), [SPROC](#), [SMOD](#), [SPC](#), [SMEM](#), [UNFOLD](#).

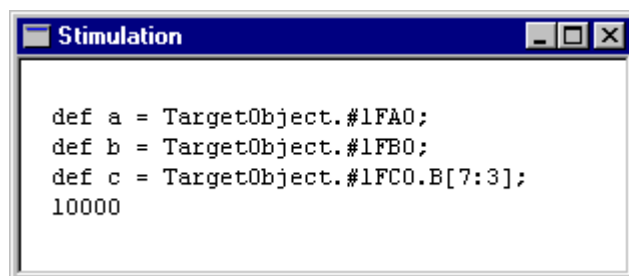
Stimulation Component

The Simulator/Debugger also supports **I/O Stimulation**. Using this feature you can generate (stimulate) interrupts or memory access generated by an external I/O device.

Description

The Stimulation component shown in [Figure 5.113](#) is a window component that provides the basic functionality of the simulator debugger. 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 5.113 Stimulation Component



Popup menu

[Figure 5.114](#) shows functions associated with the Source component. [Table 5.47](#) describes these functions.

Figure 5.114 Stimulation Popup menu

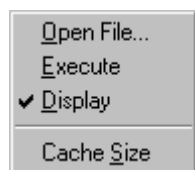


Table 5.47 Stimulation Popup menu

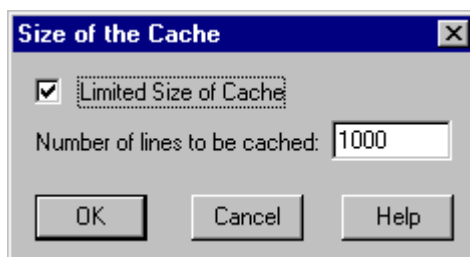
Menu entry	Description
Open File	Opens a dialog to load a stimulation file.

Menu entry	Description
Execute	Starts execution of the input file.
Display	Switches display of stimulated file on or off.
Cache size	Opens the 'Size of Cache' dialog.

Cache Size Dialog

This dialog shown in [Figure 5.115](#), 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 5.115 Cache Size Dialog



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 5.3](#) is an example file.

Listing 5.3 Stimulation File example

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

PERIODICAL 200000, 50:
    50000 a = 128;
```



Framework Components

General Component

```
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 later, it is set to 4.

NOTE For more information about Stimulation, refer to the True Time Stimulation document.

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.

Associated Commands

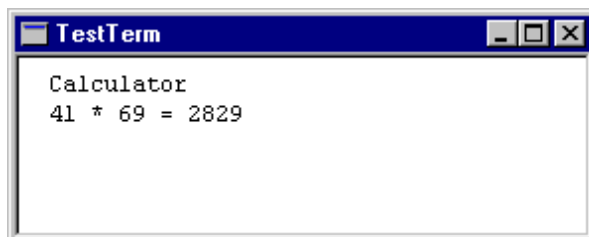
[ATTRIBUTES](#), [EXECUTE](#), [OPENFILE](#),

For more information about commands, refer to [Debugger Commands](#).

TestTerm Component

The TestTerm component shown in [Figure 5.116](#) is a user-friendly terminal input/output. It provides a simple SCI (Serial Communication Interface) interface, which is target independent.

Figure 5.116 TestTerm Component



The TestTerm component emulates a serial communication interface based at the address 200 hex, therefore providing 5 simulated memory mapped registers described in [Table 5.48](#).

Table 5.48 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 5.49](#).

Table 5.49 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.

NOTE See also [Terminal Component](#) section.

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 component 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 5.50](#) illustrates the different possible redirections and associated Escape sequences:

Table 5.50 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.

Escape Sequence	Function
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

where filename is a sequence of characters terminated by a control character (e.g., CR) and is a valid filename.

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 5.4](#).

Listing 5.4 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.



How to Use TestTerm

[Listing 5.5](#) shows the functions defined in `termport.h` that can be called to access the TestTerm component:

Listing 5.5 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 5.6](#).

Listing 5.6 Source code of the 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 */
}
```

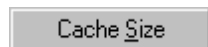
Example

The calc.abs example needs Terminal Component.

Menu

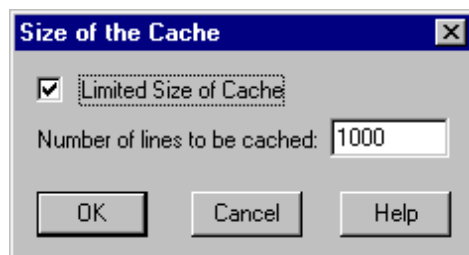
The TestTerm component menu and popup menu shown in [Figure 5.117](#) let you set the Cache Size in lines of the Testterm window in the dialog shown in [Figure 5.118](#).

Figure 5.117 TestTerm Menu



Select **Cache Size** in the menu.

Figure 5.118 TestTerm cache Size Dialog



Drag Out

Currently, nothing can be dragged out of the TestTerm component.



Drop Into

Currently, nothing can be dropped into the TestTerm component.

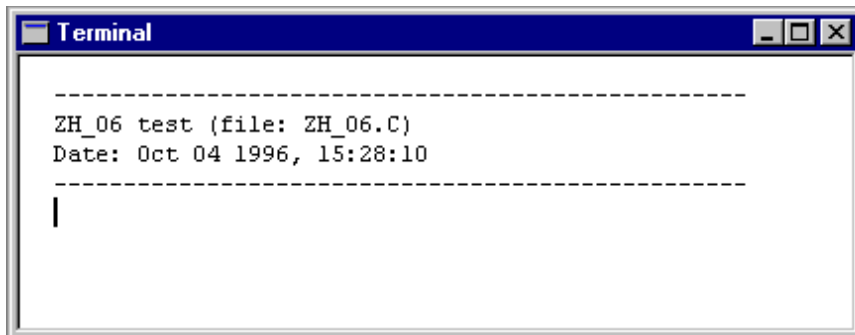
Demo Version Limitations

No limitation

Terminal Component

The Terminal component shown in [Figure 5.119](#) can be used to simulate input and output. It can receive characters from several input devices and send them to other devices.

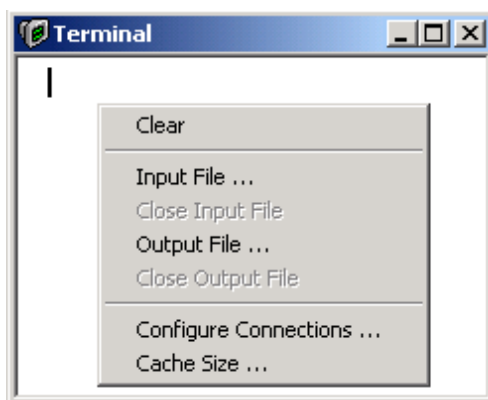
Figure 5.119 Terminal Component



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 context menu of the terminal shown in [Figure 5.120](#).

Figure 5.120 Terminal Context Menu

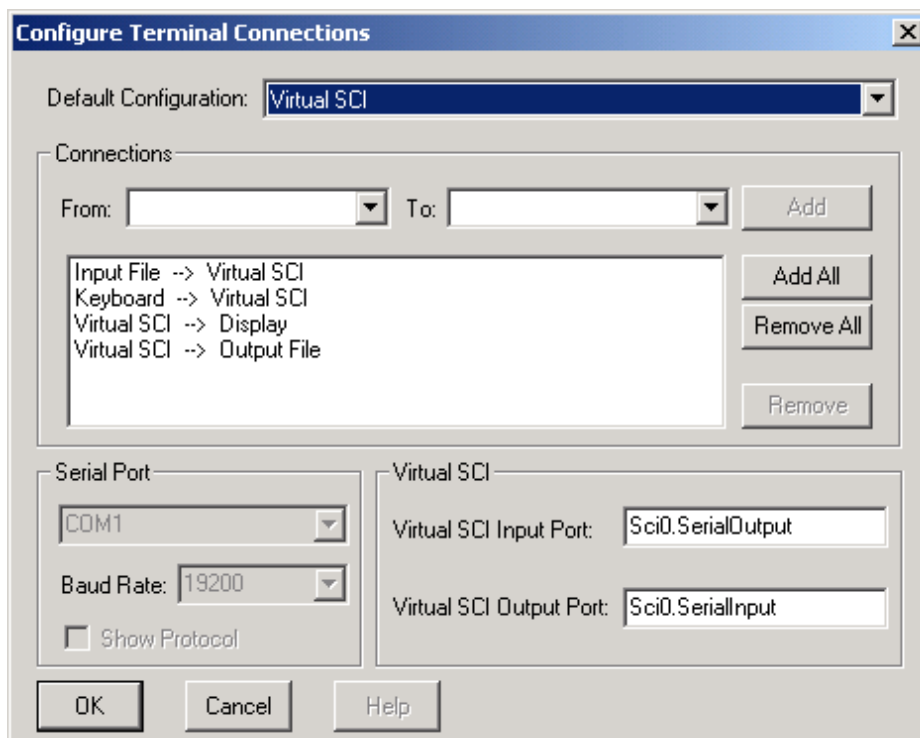


To open the menu just right click in the terminal window.

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 shown in [Figure 5.121](#).

Figure 5.121 Configure Terminal Connections Dialog



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 simulator 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 5.51](#) illustrates the different possible commands and associated Escape sequences:

Table 5.51 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
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.

where filename is a sequence of characters terminated by a control character (e.g. CR) and is a valid filename.

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 5.7](#),

Listing 5.7 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);
    }
}
```

where 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.

TIP Only some specific simulator target components currently have a **Sci0** object. For all other simulator target 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 [Connections](#) for informations about where you can do this.

Example

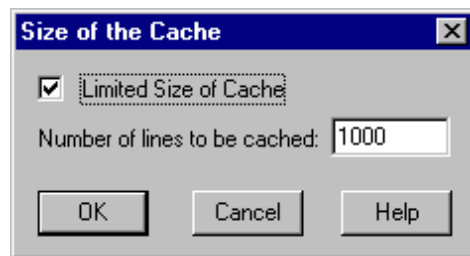
Please refer to the `Calc.abs` and `Term_demo.abs` examples installed with your Simulator/Debugger environment in the demo directory.

Other Information

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 5.122](#).

Figure 5.122 Terminal Cache Size Dialog



Drag Out

Currently, nothing can be dragged out of the Terminal component.

Drop Into

Currently, nothing can be dropped into the Terminal component.

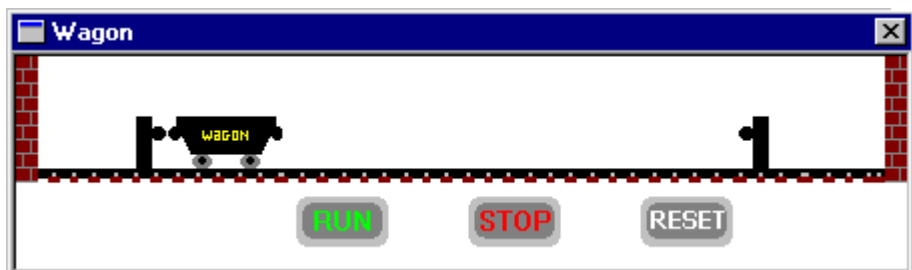
Demo Version Limitations

No limitations

Wagon Component

The Wagon component shown in [Figure 5.123](#) simulates a tool machine wagon functionality.

Figure 5.123 Wagon Component



Description

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.

Menu

[Figure 5.124](#) shows the Wagon menu and is described in [Table 5.52](#).

Figure 5.124 Wagon menu

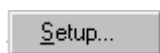
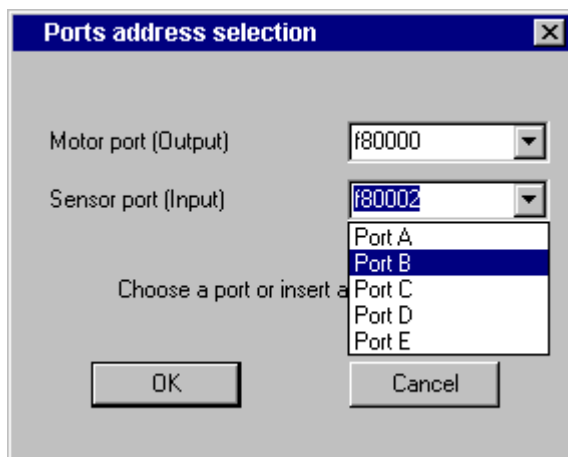


Table 5.52 Wagon Menu Description

Menu entry	Description
Setup	Opens the Wagon setup dialog shown in Figure 5.125 .

Wagon setup dialog

Figure 5.125 Wagon setup dialog



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](#).

Control bits configuration

The 2 bytes sent to the 7 segments must be composed as shown in [Figure 5.126](#).

Figure 5.126 Wagon Control bits Description

Motor port								Sensor port							
b7	b6	b5	b4	b3	b2	b1	b0	b7	b6	b5	b4	b3	b2	b1	b0
<i>l</i>	-	-	-	-	-	-	<i>r</i>	<i>bl</i>	-	-	<i>st</i>	<i>stp</i>	-	-	<i>br</i>

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.



Drag out

Nothing can be dragged out.

Drop Into

Nothing can be dropped into the Wagon Component.

Demo Version Limitations

No limitations

Associated Commands

Following commands are associated with the Wagon component:

[WPORT](#), [LINKADDR](#)



Visualization Utilities

Besides components that provide the Simulator/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.

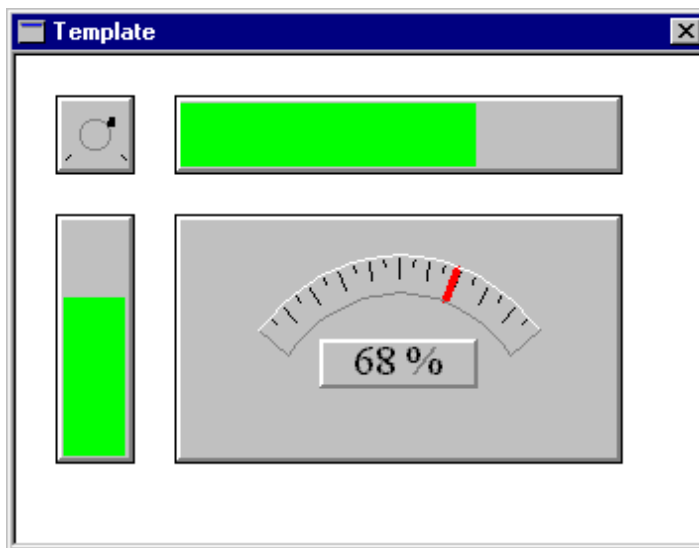
Among these components, there are visualization utilities that graphically display values, registers, memory cells, etc., or provide an advanced graphical user interface to simulated I/O devices, program variables, and so forth.

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

Analog Meter Component

The Analog Meter shown in [Figure 5.127](#) is a template 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.

Figure 5.127 Analog Meter Component



Description

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.

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.

Menu

The Analog Meter does not have a menu.



Drag Out

Nothing can be dragged out of the Analog Meter component.

Drop Into

Nothing can be dropped into the Analog Meter component.

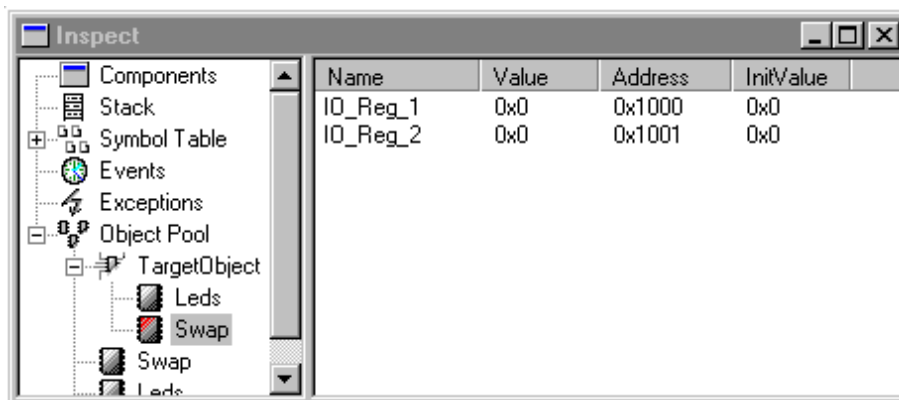
Demo Version Limitations

No limitation.

Inspector Component

The Inspector shown in [Figure 5.128](#) displays information about several topics. It displays loaded components, the visible stack, pending events, pending exceptions and loaded I/O devices.

Figure 5.128 Inspector Component



Description

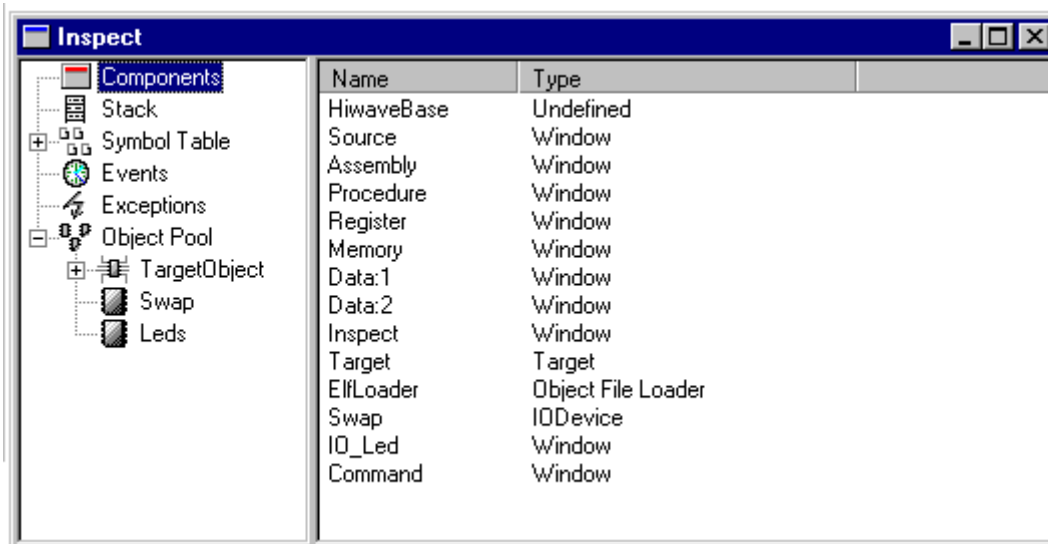
The hierarchical content of the items is displayed in a tree structure. If any item is selected on the left side, then additional information is displayed on the right side.

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

When the components icon is selected, as shown in [Figure 5.129](#), the right side displays various information about all loaded components. A Component is the “unit of dynamic loading”, therefore all windows, the CPU, the target and maybe the target-simulator are listed.

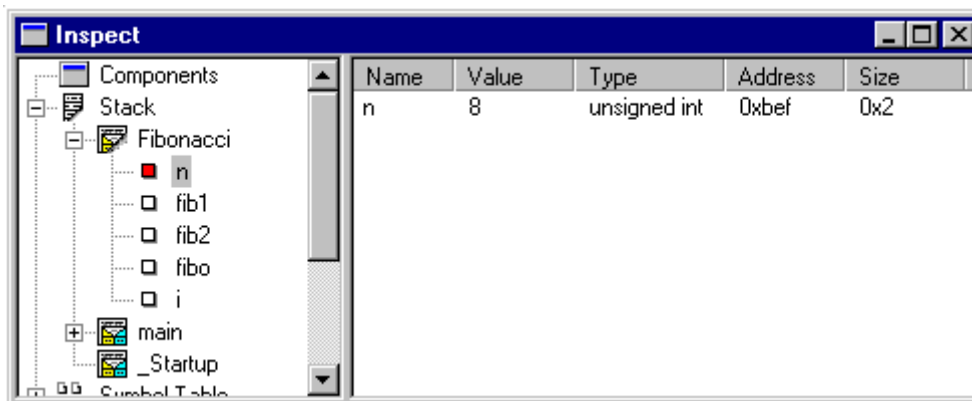
Figure 5.129 Inspector components icon



Stack

The Stack shown in [Figure 5.130](#) 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.

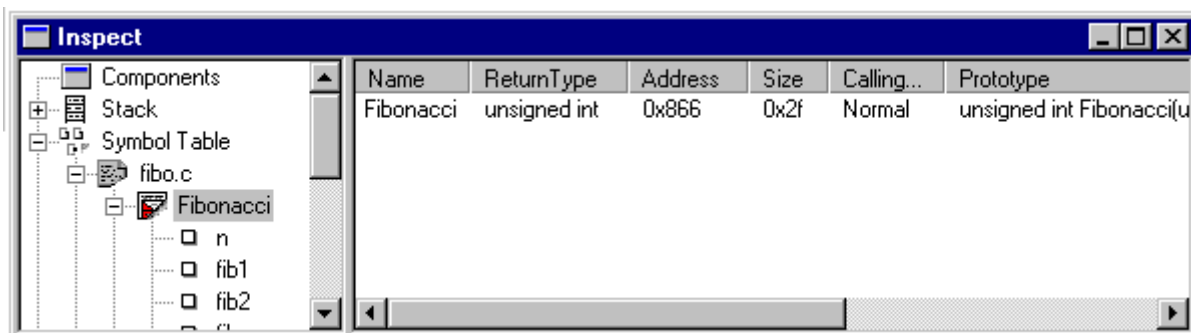
Figure 5.130 Inspector Stack



Symbol Table

The symbol table shown in [Figure 5.131](#) 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.

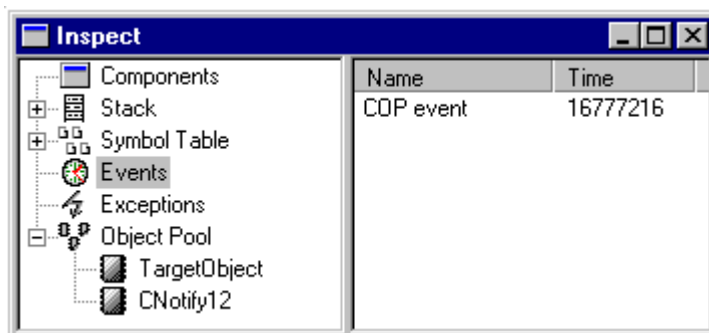
Figure 5.131 Inspector Symbol Table



Events

The events icon shown in [Figure 5.132](#) 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 5.132 Inspector Events



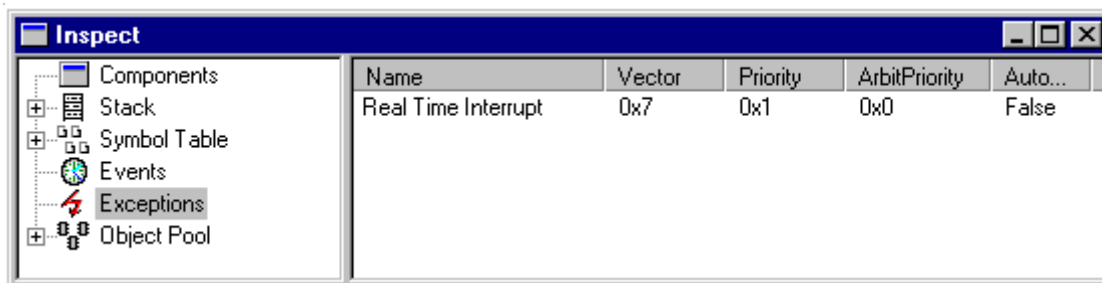
Events are only used in the Simulator. This information is used for simulation I/O device development.

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

Exceptions

The exception icon shown in [Figure 5.133](#) shows all currently raised exceptions. Exceptions are pending interrupts.

Figure 5.133 Inspector Exceptions



Events are only used in the Simulator. This information is used for simulation I/O device development.

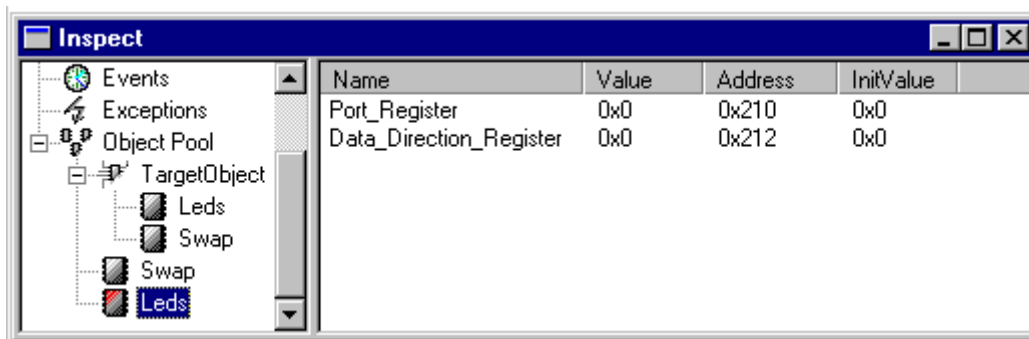
Since interrupts are usually simulated immediately when they are raised, the Exceptions are usually empty. Only when interrupts are disabled or an interrupt is handled, something is visible in this item.

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

Object Pool

The Object Pool shown in [Figure 5.134](#) is a pool of objects. It can contain any number of Objects, which can communicate together and also with other parts of the Simulator/Debugger.

Figure 5.134 Inspector Object Pool





The most common use of Objects is to simulate special hardware with the I/O development package, however, other targets 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 targets.

For the 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.

Operations

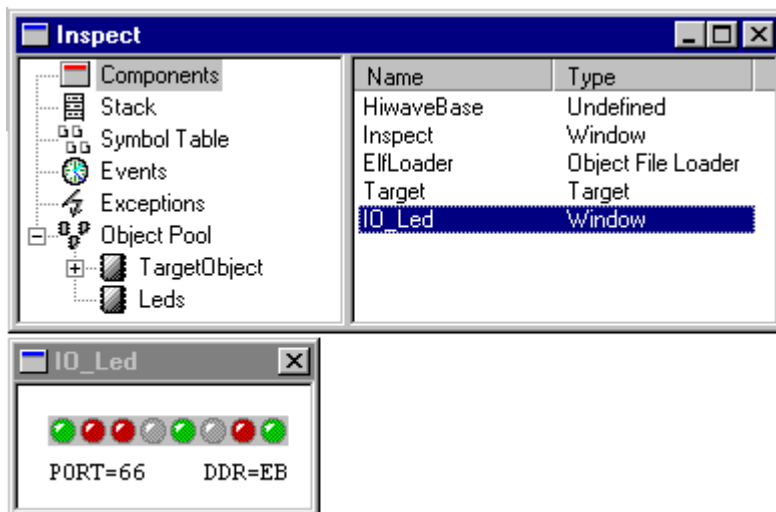
Click the folded/unfolded icons   to unfold/fold the tree and display/hide additional information.

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 **&**).

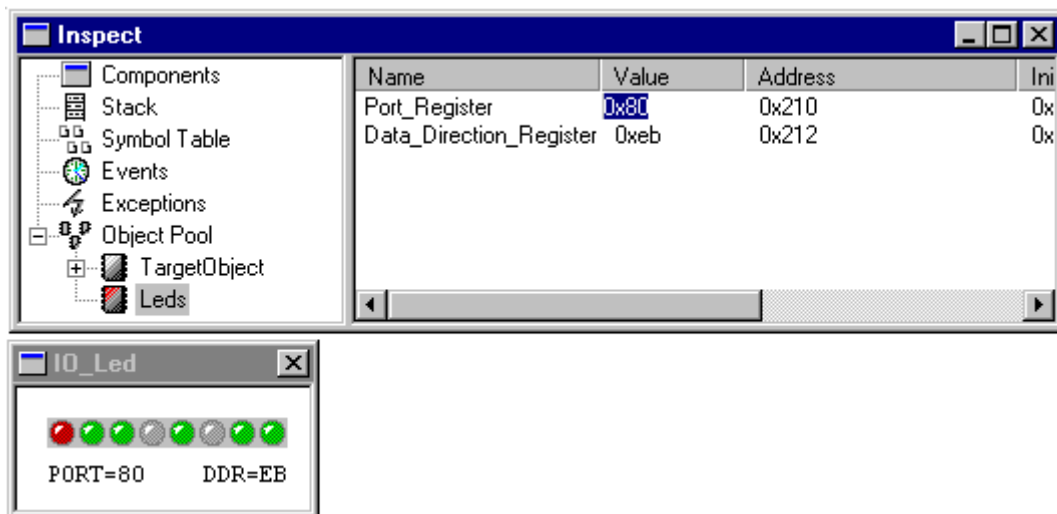
To see the IO_Led in the Inspector, as shown in [Figure 5.135](#), open the IO_Led with the context menu **Component-Open** and then open the Inspector. If the Inspector is already loaded, select **Update** from the context menu in the Inspector. Then click on the Components icon to see the Component list, which now includes the “IO_Led” component.

Figure 5.135 How to see the IO_Led in the Inspector



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 5.136](#)).

Figure 5.136 Changing “Data_Direction_Register” value



Menu

The Inspector menu contains entries described in [Table 5.53](#).

Table 5.53 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 5.54](#).

Table 5.54 Inspector 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 5.55](#) gives a brief description.

Table 5.55 Inspector Possible Drag Out

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	<p>The Symbol Table icon cannot be dragged out. Subitems can be dragged depending on their type:</p> <p>Modules: Modules can be dragged to the source and global data window to specify a specific module.</p> <p>Functions: Functions can be dragged to display the function or code range.</p> <p>Variables: Variables can be dragged to display their content in memory.</p> <p>Indirections: Indirections can be dragged to display their content in memory.</p>

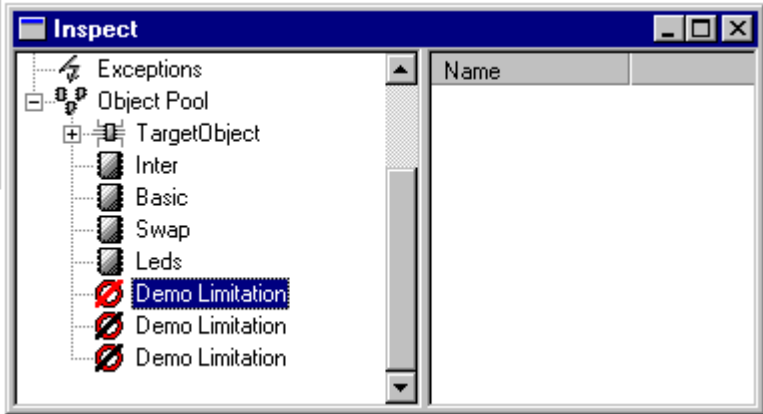
Drop Into

Nothing can be dropped into the Inspector Component window.

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 5.137](#).

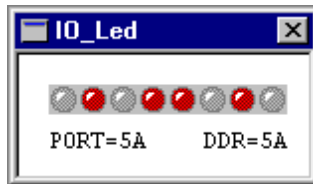
Figure 5.137 Inspector Demo Version Limitations



IO LED Component

The IO LED Component shown in [Figure 5.138](#) 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 leds states are switched on/off at the DDR address (symbolic values).

Figure 5.138 IO LED Component



Description

When you change the state of leds in this window, the value of the corresponding bit at the DDR address will change in the Memory Component window.

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.

Menu

The IO LED Menu shown in [Figure 5.139](#) contains the *Setup* command. This command opens the Led setup dialog shown in [Figure 5.140](#) and allows you to specify the PORT and DDR addresses.

Figure 5.139 IO LED Menu

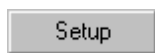
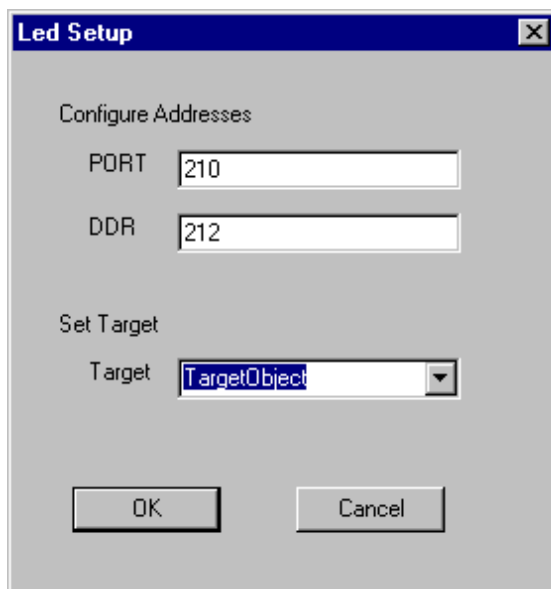


Figure 5.140 IO LED Setup Dialog**Associated Popup Menu**

Identical to menu.

Drag Out

Nothing can be drag out.

Drop Into

Nothing can be dropped into.

Associated Commands

.None.

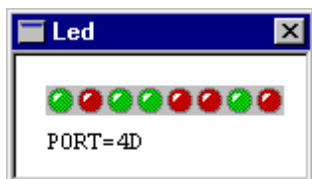
Demo Version Limitations

No limitation

LED Component

The LED component shown in [Figure 5.141](#) is a visual utility that displays an arbitrary 8 bit value by means of a LED bar.

Figure 5.141 LED Component



Description

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.

Operations

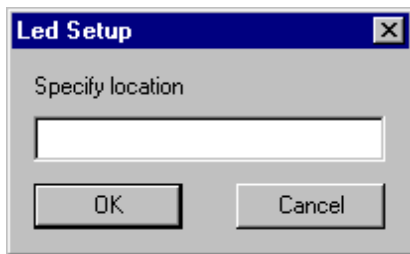
If you click a LED, its state toggles between green (0) and red (1). The corresponding bit in the underlying value is changed as well. Right-click within the component, a popup menu appears with the same menu entries as listed in the Led menu in the main menu bar.

Menu

The Led menu contains a single item **Setup...** that opens the [Led Setup Dialog](#) shown in [Figure 5.142](#).

Led Setup Dialog

Figure 5.142 Led Setup Dialog



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

Currently, nothing can be dragged out of the LED component.

Drop Into

Currently, nothing can be dropped into the LED component.

Demo Version Limitations

No limitation

Associated Command

[PORT](#)

The Phone Component

The phone component shown in [Figure 5.143](#) is an input utility that provides a graphical keypad pad that allows you to interactively modify the value of a memory cell.

Figure 5.143 The Phone Component



Features

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.

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.

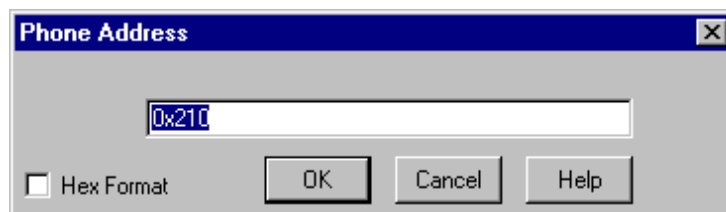
Menu

The Led menu contains the **Address...** command, which opens the Phone Address dialog shown in [Figure 5.144](#).

Phone Address Dialog

In the text field, 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 5.144 Phone Address Dialog



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

Currently, nothing can be dragged out of the Phone component.

Drop Into

Nothing can be dropped into the Phone component.



Demo Version Limitations

No limitation

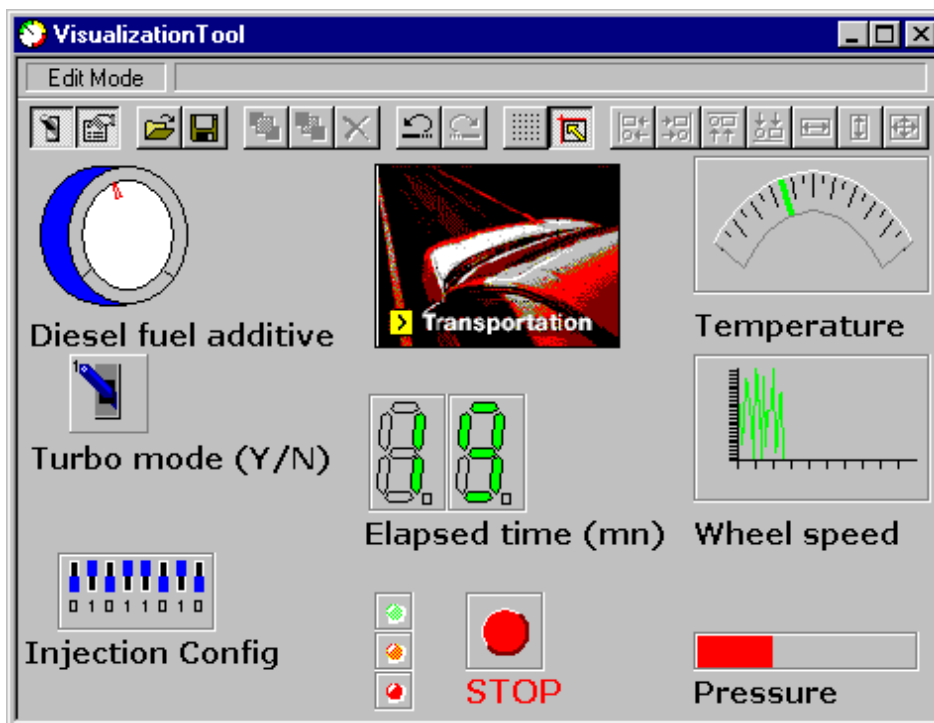
VisualizationTool

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

Not only is the VisualizationTool fully configurable, but it is also very easy to use. You can create your own visualization within a few minutes.

The tool consists of a plain workspace that can be equipped with many different instruments (See [Figure 5.145](#)).

Figure 5.145 VisualizationTool



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 (see [Menu](#)) to add a new instrument.

Selection

The VisualizationTool allows several ways to select instruments.

You can select a single instrument by left clicking 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

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

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.

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 5.56](#).

Table 5.56 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>
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 5.57](#).

Table 5.57 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.



Menu entry	Context	Description
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
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.

Menu entry	Context	Description
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
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.

Menu entry	Context	Description
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 5.58](#). To view the property dialog box of the VisualizationTool, use the shortcut <CTRL-P> or double click on the background.

Table 5.58 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.

Menu entry	Description
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 these common attributes shown in [Table 5.59](#).

Table 5.59 Instruments 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.
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.

Attribute	Description
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: <i>TargetObject.#210</i> Subscribe: <i>TargetObject.#210</i> Subscribe: <i>PORTB.PORTB</i> (check exact spelling using <i>Inspector</i>) Variable: <i>counter</i> Register: <i>SP</i> Memory: <i>0x210</i>
Size of Port	If you use the Memory Port, you can also specify the width of memory to display (up to 4 Bytes).

Analog

The Analog instrument (Figure 5.146) represents the classical pointer instrument, also known as speedometer, voltage meter...

Figure 5.146 Analog Instrument



Its attributes are shown in the [Table 5.60](#).

Table 5.60 Analog instruments 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

Values are displayed by a bar strip. This instrument (See [Figure 5.147](#)) may be used as a position state of a water tank.

Figure 5.147 Bar Instrument



Its attributes are shown in the [Table 5.61](#)

Table 5.61 Bar instruments 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

Use this instrument to give a special look to your visualization (Figure [5.148](#)), or to display a warning picture.

Figure 5.148 Bitmap Instrument



Additionally, it can also be used as a bitmap animation. Its attributes are shown in the [Table 5.62](#)

Table 5.62 Bitmap instruments 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:
 $(\text{port_memory} \& \text{ANDmask}) == \text{EQUALmask}$

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

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

- no light on bitmap: AND mask 3, EQUAL mask 0

- flasher on bitmap: AND mask 3, EQUAL mask 1
- brake light on bitmap: AND mask 3, EQUAL mask 2
- brake and flasher light on: AND mask 3, EQUAL mask 3

DILSwitch

This instrument is also known as Dual-in-Line Switch (Figure [5.149](#)). It is mainly used for configuration purpose.

You can use it for viewing or setting bits of one to four bytes.

Figure 5.149 DILSwitch Instrument



Its attributes are listed in the [Table 5.63](#).

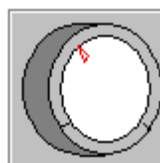
Table 5.63 DIL Switch instruments attributes

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

Normally known as an adjustment instrument, for example the volume control of a radio (Figure [5.150](#)).

Figure 5.150 Knob Instrument



Its attributes are shown in the [Table 5.64](#)

Table 5.64 Knob instruments 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%.
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

This instrument is used for observing one definite bit of one byte (Figure [5.151](#)). There are only two states: On and Off.

Figure 5.151 Led Instrument



Its attributes are shown in the [Table 5.65](#)

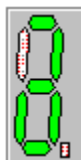
Table 5.65 LED instruments attributes

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

The well known display instrument for numbers and characters: it has seven segments and one point. These eight units represent eight bits of one byte (Figure [5.152](#)).

Figure 5.152 7 Segment Instrument



Its attributes are shown in the [Table 5.66](#)

Table 5.66 7 Segment Display instruments 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

Use this instrument to set or view a definite bit (Figure [5.153](#)). 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 5.153 Switch Instrument



Its attributes are shown in [Table 5.67](#).

Table 5.67 Switch instruments 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.
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 whether 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.

Attribute	Description
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

This instrument has several functions: Static Text, Value, Relative Value, and Command (Figure 5.154).

Figure 5.154 Text Instrument



Please use 'Text Mode' to switch between the five modes. Its common attributes are shown in the [Table 5.68](#)

Table 5.68 Text instruments 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 the [Table 5.69](#)

Table 5.69 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 the Table 5.70

Table 5.70 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 the Table 5.71

Table 5.71 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 you can specify a command that will be executed by clicking on this field. For more information about commands, read the chapter 'Simulator/Debugger Commands'. Its attributes are shown in the [Table 5.72](#)

Table 5.72 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' 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 the [Table 5.73](#)

Table 5.73 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.

Control Points

This chapter provides an overview of the debugger breakpoints and watchpoints.

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

- [Control points introduction](#)
- [Breakpoints setting dialog](#)
- [Define Breakpoints](#)
- [Watchpoints setting dialog](#)
- [General Rules for Halting on a Control Point](#)
- [Define Watchpoints](#)

Control points introduction

There are two kinds of control points: breakpoints and watchpoints (also called data breakpoints). Breakpoints are located at an address, watchpoints are located at a memory range. Watchpoints start from an address, have a range, and a read and/or write state. Breakpoints have an address and can be temporary or permanent. You can set or disable a control point, set a condition and an optional command, and set the current count and counting interval.

You can see and edit control point characteristics through two dialogs: The first one is the [“Breakpoints setting dialog”](#) and the second is the [“Watchpoints setting dialog”](#). These two dialogs 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/unchecking the related checkbox.



- Enabling/disabling multiple control points by checking/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 checkboxes.

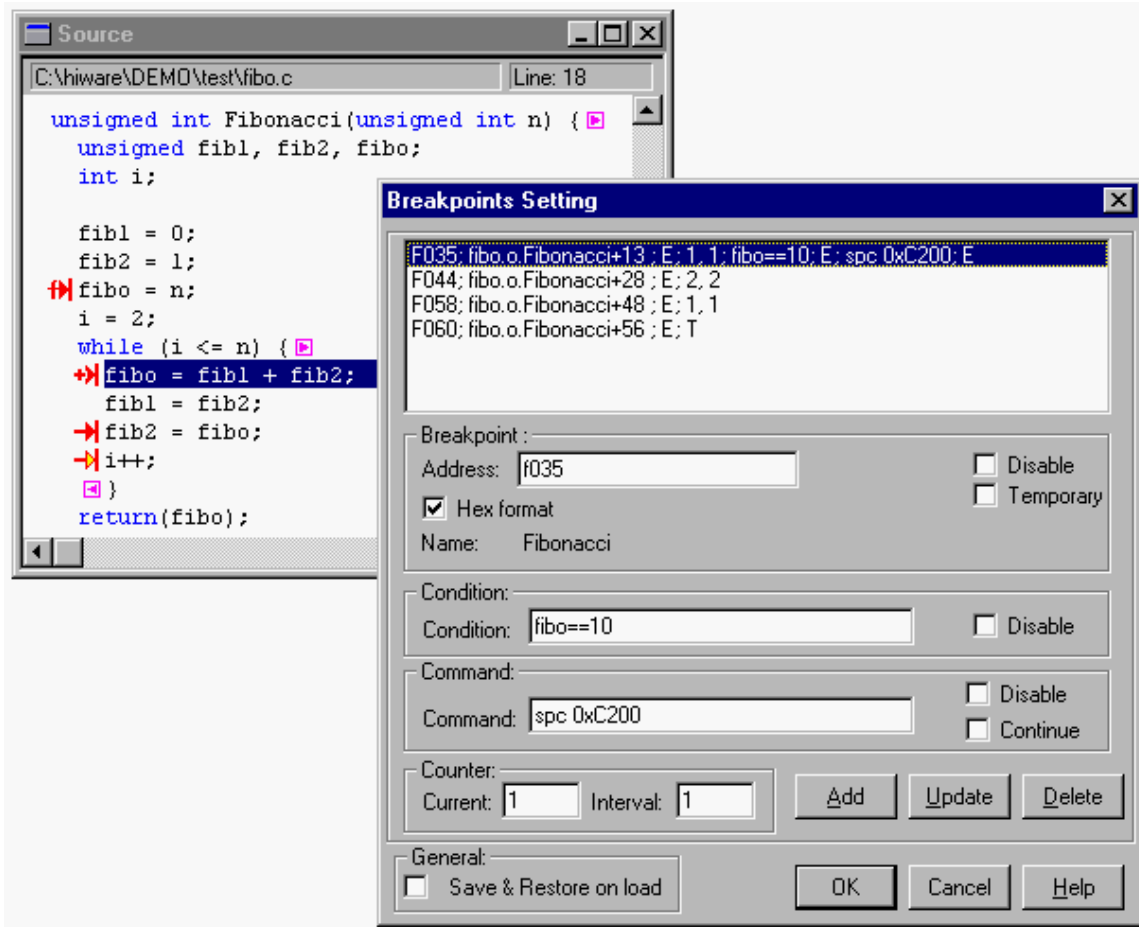
With watchpoints, the following operations are also available:

- 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 setting dialog

The Breakpoints setting dialog is shown in [Figure 6.1](#)

Figure 6.1 Breakpoints setting dialog



Breakpoint Symbols

- Temporary breakpoint symbol:
- Permanent breakpoint symbol:
- Disabled breakpoint symbol:
- A counting breakpoint symbol:

Conditional breakpoint symbol: 

Description of the Dialog

The [Breakpoints setting dialog](#) consists of:


- a list box that displays the list of currently defined breakpoints
- a “**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).
- a “**Condition:**” group box that displays the condition string to evaluate, and the state of the condition (disabled or not).
- a “**Command:**” group box that displays the command string to execute and the state of the command (disable or continue after command execution).
- a “**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.

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

- a “**Delete**” button to remove the currently selected breakpoint.
- an **Update** button to Update all modifications in the dialog.
- an **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).
- an **OK** button to validate all modifications.
- a **Cancel** button to ignore all modifications.
- a **Help** button to open related help information.

Multiple selections in the dialog

The list box allows you to select multiple consecutive breakpoints by clicking the first breakpoint then  + click 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  + click 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 condition in dialog

You can enter an expression in the condition edit box. The syntax of the expression will be 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 Simulator/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 [Breakpoints setting dialog](#)): the **SAVEBP** command is used: **SAVEBP on** when checked, **SAVEBP off** when unchecked.

NOTE See also [SAVEBP](#) command in Appendix.

- List of defined breakpoints: the **BS** command is used, as shown in [Listing 6.1](#).

Listing 6.1 .BPT File Syntax

```
BS address [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.

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 [Breakpoints setting dialog](#), for conditional breakpoint.

command is any debugger command. It matches the **Command** field in the [Breakpoints setting dialog](#), for associated commands.

current is an **expression**. It matches the **Current** field (**Counter**) in the [Breakpoints setting dialog](#), for counting breakpoints.

interval is an **expression**. It matches the **Interval** field (**Counter**) in the [Breakpoints setting dialog](#), 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 Simulator/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.

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 6.2](#).

Listing 6.2 Example of Breakpoint file with **Save & Restore on load** checked.

```
savebp on
BS &fibonacci.c:Fibonacci+19 P E; cond = "fibonacci > 10" E; cdSz = 47 srSz = 0
BS &fibonacci.c:Fibonacci+31 P E; cdSz = 47 srSz = 0
BS &fibonacci.c:main+12 P E; cdSz = 42 srSz = 0
BS &fibonacci.c:main+21 P E; cond = "fibonacciCount==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.

TIP If only one or few functions differ after a recompilation, not all **BP** will be lost. To achieve that, **BPs** 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 Simulator/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](#) and [SAVEBP](#) commands.

Define Breakpoints

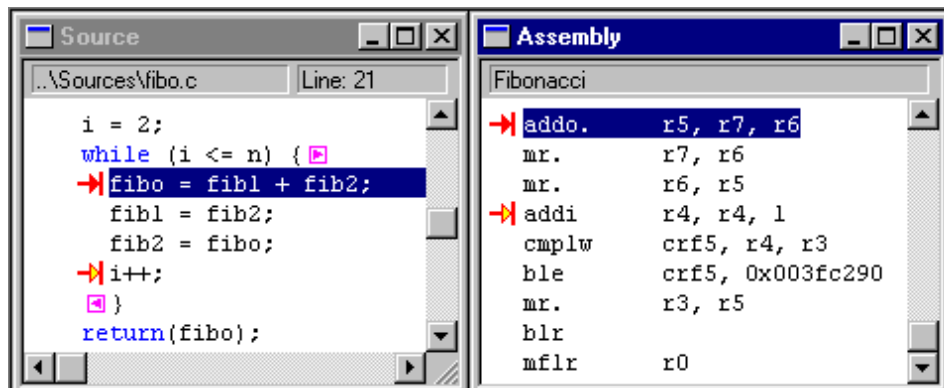
Breakpoints are control points associated with a PC value (i.e. program execution is stopped as soon as the PC reaches the value defined in a breakpoint). The Simulator/Debugger supports different types of Breakpoints:

- 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.

Identify all Positions Where a Breakpoint Can Be Defined

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




The Simulator/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 mark: **



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

Define a Temporary Breakpoint

A temporary breakpoint is recognized by the following icon: 

The Simulator/Debugger provides two ways to define a temporary breakpoint:

- Use Popup Menu
1. **Point at a C statement in the Source Component 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.**
- Use  + 

1. Point at a C statement in the Source Component Window, and  + .
2. A temporary breakpoint is defined, 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.



Define a Permanent Breakpoint

A permanent breakpoint is recognized by the following icon: 

The Simulator/Debugger provides two ways to define a permanent breakpoint:

- Use 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 permanent breakpoint mark is displayed in front of the selected statement.

- Use  + 

1. Point at a C statement in the Source Component window, and  + .
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.

Define a Counting Breakpoint

A Counting breakpoint is recognized by the following icon: 

Counting breakpoints can only be set using the [Breakpoints setting dialog](#). There are currently three ways to open this dialog:

- Use  + 

1. Point at a C statement in the Source Component Window, and  + .

2. The [Breakpoints setting dialog](#) box is opened and a new breakpoint is inserted in the list of breakpoints defined in the application.

- Use Source Popup Menu

1. Point at a C statement in the Source Component 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 Component window and right-click. The Source Popup Menu is displayed on the screen.
4. Choose Show Breakpoints from the Popup Menu. The [Breakpoints setting dialog](#) is displayed.
 - Choose Run>Breakpoints ...
1. Point at a C statement in the Source Component window and right-click. The Source Popup Menu is displayed on the screen.
2. Choose Set BreakPoint from the Popup Menu. A breakpoint is defined on the selected instruction.
3. Choose Run>Breakpoints The [Breakpoints setting dialog](#) is displayed.


Once the [Breakpoints setting dialog](#) is opened:

- You can select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints.
- You can specify the interval for the breakpoint detection in the **Interval** field.
- Then close the [Breakpoints setting dialog](#) box by clicking **OK**.

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.

Define a Conditional Breakpoint

A conditional breakpoint is recognized by the following icon: 

Conditional breakpoints can only be set using the [Breakpoints setting dialog](#). There are three ways to open this dialog:

- Use  + 

1. Point at a C statement in the Source Component window, and  + .

2. The [Breakpoints setting dialog](#) box is opened and a new breakpoint is inserted in the list of breakpoints defined in the application.

- Use Source 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 Breakpoints Setting dialog is displayed.
 - Choose Run>Breakpoints...
1. Point at a C statement in the Source Component 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. Choose Run>Breakpoints... The Breakpoints Setting dialog is displayed.





Once the [Breakpoints setting dialog](#) is opened:

- You can select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints.
- You can specify the condition for breakpoint activation in the field **Condition**. 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**)
- Then you can close the [Breakpoints setting dialog](#) box by clicking **OK**.

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

Delete a Breakpoint

The Simulator/Debugger provides four ways to delete a breakpoint:

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

- Choose **Run>Breakpoints...**
1. Choose **Run>Breakpoints...** The **Breakpoints Setting dialog** is displayed.
 2. Select the breakpoint to delete in the list of defined breakpoints.
 3. Click **Delete**. The selected breakpoint is removed from the list of defined breakpoints.
 4. 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 setting dialog](#) box. There are two ways to open this dialog box:

- Choose **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.
 - Choose **Run>Breakpoints...**
1. **Choose Run>Breakpoints...** The [Breakpoints setting dialog](#) is displayed.
 2. **Once the Breakpoints Setting dialog is opened:**
 - You can select the breakpoint to modify by clicking on the corresponding entry in the list of defined breakpoints.
 - 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**).
 - Click **OK** to close the [Breakpoints setting dialog](#) box.

When the breakpoint is detected, the command is executed and the application will stop.



The **Continue** check button allows the application to continue after the command is executed.

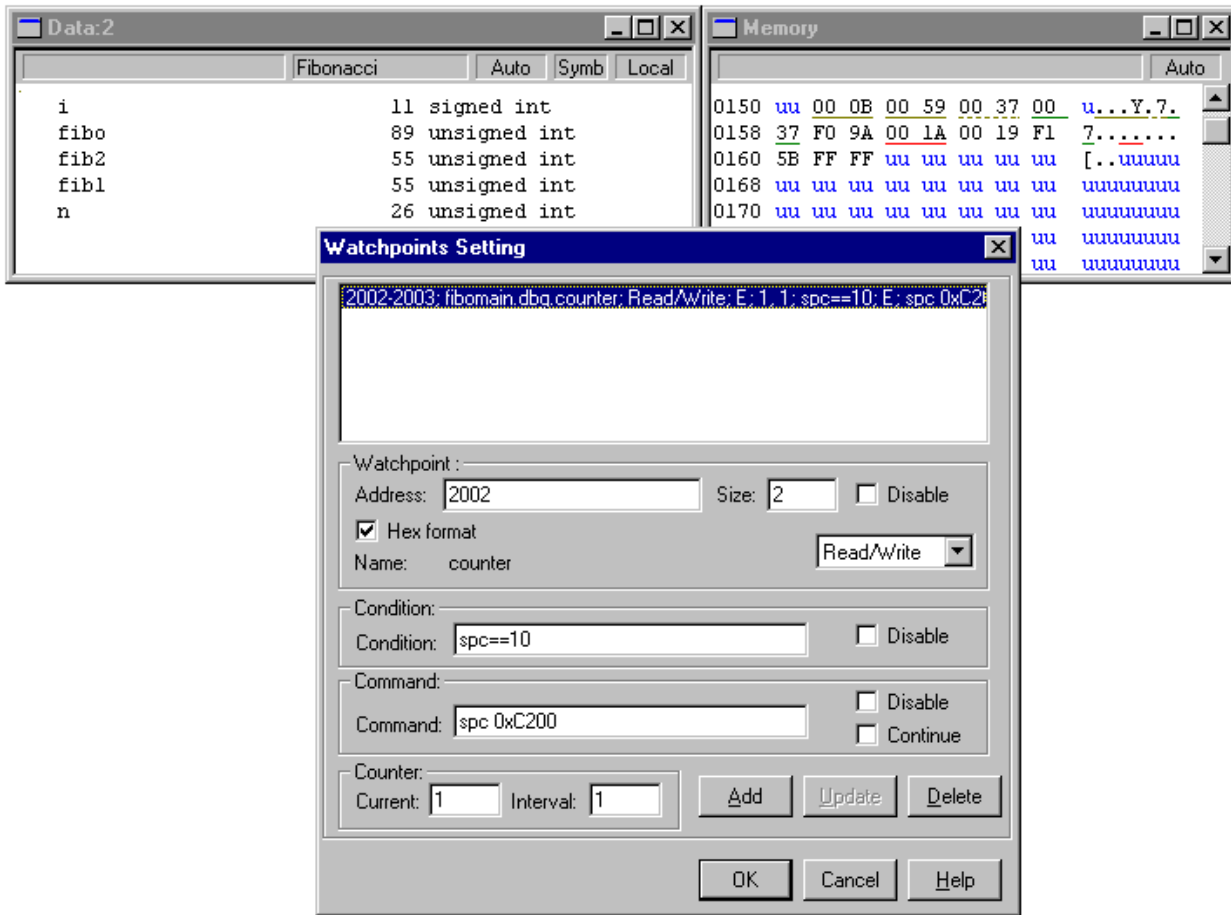
Demo Version Limitations

Only 2 breakpoints can be set.

Watchpoints setting dialog

Figure 6.2 shows the dialog used to set Watchpoints.

Figure 6.2 Watchpoints setting dialog



Description of the Dialog

The **Watchpoints Setting** dialog is based on:

- a list box that displays the list of currently defined watchpoints.
- a “**Watchpoint:**” group box that displays the address of the currently selected 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).



- a “**Condition:**” group box that displays the condition string to evaluate and the state of the condition (disabled or not).
- an **Update** button to Update all modifications in the dialog.
- a “**Command:**” group box that displays the command string to execute and state of the command (disabled or continue after command execution).
- **Delete:** Click delete button to remove currently selected watchpoint and select the watchpoint that is below the removed watchpoint.
- **OK:** Click OK to validate all modifications.
- **Add** button: adds 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.

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

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

Multiple selections in the dialog

For breakpoints, you can do multiple selections with  and  .

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 condition in the dialog

You can enter an expression in the condition 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.

Demo Version Limitations

Only 2 watchpoints can be set.

General Rules for Halting on a Control Point

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

Control Point with command: When the simulator halts on the control point, a specified command is executed.

Define 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 Simulator/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 watchpoint, 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 may be set in a Data or Memory component.

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

Defining a Read Watchpoint



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

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

- Use Popup Menu
1. **Point at a variable in the Data Component 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 [Watchpoints setting dialog](#) is displayed.**

5. Select the watchpoint you want to define as *read* access.
6. Select the Read type in the dropdown box.
7. A read access watchpoint is defined for the selected variable.

- Use  + 

1. Point at a variable in the Data Component Window and  + .
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.



Defining a Write Watchpoint

A red vertical bar is displayed in front of a variable associated with a write access watchpoint.

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

- Use Popup Menu
1. Point at a variable in the Data Component 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 [Watchpoints setting dialog](#) is displayed.
 5. Select the watchpoint you want to define as write access.
 6. Select the Write type in the dropdown box.
 7. A write access watchpoint is defined for the selected variable.

- Use  + 

1. **Point at a variable in the Data Component Window and  + .**
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 Simulator/Debugger provides two ways to define a read/write access watchpoint:

- Use Popup Menu
1. **Point at a variable in the Data Component Window and right-click. The Data Popup Menu is displayed.**
 2. **Choose Set Watchpoint from the Popup Menu.**
 3. **A read/write access watchpoint is defined for the selected variable.**

- Use  + .

1. **Point at a variable in the Data Component Window and  + .**
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 described previously (read, write, read/write).

The Simulator/Debugger provides two ways to define a counting watchpoint:

- Use Popup Menu
1. Point at a variable in the Data Component 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 [Watchpoints setting dialog](#) 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. Close the Watchpoints Setting dialog box by clicking OK.
 8. A counting watchpoint is defined for the selected variable.

Choose  + 

1. Point at a variable in the Data Component Window and  + . The [Watchpoints setting dialog](#) is displayed.
2. Select the watchpoint you want to define as a counting watchpoint.
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 [Watchpoints setting dialog](#) box by clicking OK.
5. 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 Simulator/Debugger provides two ways to define a conditional watchpoint:

- Use Popup Menu
1. **Point at a variable in the Data Component 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 [Watchpoints setting dialog](#) 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). Close the [Watchpoints setting dialog](#) box by clicking OK.**
 8. **A conditional watchpoint is defined for the selected variable.**

- Use  + 

1. **Point at a variable in the Data Component Window and  + .** The [Watchpoints setting dialog](#) 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.**
 - 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**)
4. **Close the [Watchpoints setting dialog](#) box by clicking OK.**
5. **A conditional watchpoint is defined for the selected variable.**

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



Deleting a Watchpoint

The Simulator/Debugger provides four ways to delete a watchpoint:

- Use **Delete Breakpoint** from Popup Menu

1. **In the Data Component, 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.**

- Use  + 

1. **In the Data Component, point at a variable where a watchpoint has been defined and  + .**
2. **The watchpoint is deleted and the vertical bar in front of the variable is removed.**

- Choose **Show Watchpoints** from Data Popup Menu

1. **Point in the Data Component Window and right-click. The Data Popup Menu is displayed.**
2. **Choose Show Watchpoints from the Popup Menu. The [Watchpoints setting dialog](#) 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 [Watchpoints setting dialog](#) box. The watchpoint is deleted and the vertical bar in front of the variable is removed.**

- Choose **Run>Watchpoints** menu command

1. **Choose Run>Watchpoints.... The [Watchpoints setting dialog](#) is displayed.**
2. **Select the watchpoint you want to delete.**
3. **Click Delete. The selected watchpoint is removed from the list of defined watchpoints.**

4. Click OK to close the [Watchpoints setting dialog](#) box. 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 setting dialog](#) box. There are two ways to open this dialog box:

- Choose **Show Watchpoints..** from Data Popup Menu
1. **Point in the Data Component Window and right-click.** The Data Popup Menu is displayed.
 2. **Select Show Watchpoints from the Popup Menu.** The [Watchpoints setting dialog](#) is displayed.
 - Choose **Run>Watchpoints...**
 1. **Choose Run>Watchpoints....** The [Watchpoints setting dialog](#) is displayed.
 2. **Once the [Watchpoints setting dialog](#) is open:**
 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](#), [GO](#) and [STOP](#) are not allowed. A command file can be associated with a breakpoint using the commands [CALL](#) or [CF](#) (Example CF breakCmd.cmd).**
 5. **Click OK to close the [Watchpoints setting dialog](#) box.**
 6. **When the watchpoint is detected, the command will be executed and the application will stop at this point. The Continue check button allows the application to continue after command execution.**

Debugger Commands

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.

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

- [Simulator/Debugger Commands](#)

Simulator/Debugger Commands

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, 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.

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.

Syntax of Simulator/Debugger command

To display the syntax of a command, type the command followed by a question mark.

Example:

```
in>printf?
PRINTF (<format>, <expression>, <expression>, ...)
```

List of Available Commands

Commands described on the following pages are sorted in 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 7.1](#) contains all available Kernel commands.

Table 7.1 List of Kernel Commands

Command, Syntax	Short Description
A	affects a value



Command, Syntax	Short Description
AT	sets a time delay for command execution
CALL fileName[;C][;NL]	executes a command file
DEFINE symbol [=] expression	defines a user symbol
ELSE	other operation associated with IF command
ELSEIF condition	other conditional operation associated with IF command
ENDFOCUS	resets the current focus (refer to FOCUS command)
ENDFOR	exits a FOR loop
ENDIF	exits an IF condition
ENDWHILE	exits a WHILE loop
FOCUS component	sets the focus on a specified component
FOR [variable =]range [“,” step]	FOR loop instruction
FPRINTF (fileName,format,parameters)	FPRINTF instruction
GOTO label	unconditional branch to a label in a command file
GOTOIF condition Label	conditional branch to a label in a command file
IF condition	conditional execution
PAUSETEST	displays a modal message box
PRINTF (“Text:,” value)	PRINT instruction
REPEAT	REPEAT loop instruction
RETURN	returns from a CALL command



Command, Syntax	Short Description
<u>TESTBOX</u>	displays a message box with a string
<u>UNDEF</u> symbol *	undefines a userdefined symbol
<u>UNTIL</u> condition	condition of a REPEAT loop
<u>WAIT</u> [time] [:s]	command file execution pause
<u>WHILE</u> 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 7.2](#) contains all available Base commands.

Table 7.2 Base Commands

Command, Syntax	Short Description
BC address *	deletes a breakpoint (breakpoint clear)
BS address function [P T[state]]	sets a breakpoint (breakpoint set)
CD [path]	changes the current working directory
CR [fileName][;A]	opens a record file (command records)
DASM [address range][;OBJ]	disassembles
DB [address range]	displays memory bytes
DL [address range]	displays memory bytes as longwords
DW [address range]	displays memory bytes as words
G [address]	starts execution of the application currently loaded
GO [address]	starts execution of the application currently loaded
LF [fileName][;A]	opens a log file
LOG type [=] state {[,] type [=] state }	enables or disables logging of a specified information type
MEM	displays the memory map



Command, Syntax	Short Description
<u>MS</u> range list	sets memory bytes
<u>NOCR</u>	closes the record file
<u>NOLF</u>	closes the log file
<u>P</u> [address]	single assembly steps into program
<u>RESTART</u>	restart the loaded application
<u>RD</u> [list *]	displays the content of registers
<u>RS</u> register[=]value{,register[=]value }	sets a register
<u>S</u>	stops execution of the loaded application
<u>STEPINTO</u>	stepping to the next source instruction of the loaded application
<u>STEPOUT</u>	executes program out of a function call
<u>STEPOVER</u>	stepping over the next source instruction of the loaded application
<u>STOP</u>	stops execution of the loaded application
<u>SAVEBP</u> on off	saves breakpoints
<u>T</u> [address][,count]	traces program instructions at the specified address
<u>WB</u> range list	writes bytes
<u>WL</u> range list	writes longwords
<u>WW</u> 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 7.3](#) contains all available Environment commands.

Table 7.3 Environment Commands

Command, Syntax	Short Description
ACTIVATE component	activates a component window
AUTOSIZE on off	autosize windows in the main window layout
BCKCOLOR color	set the background color
CLOSE component *	close a component
DDEPROTOCOL ON OFF SHOW HIDE STATUS	configure the Debugger/Simulator DDE protocol
FONT 'fontName' [size][color]	sets text font
LOAD applicationName	load a framework application (code and debug information)
LOADCODE applicationName	load the code of a framework application
LOADSYMBOLS applicationName	load debugging information of a framework application
OPEN component [[x y width height][:][i max]]	open a Windows component
OPENIO Iocomponentname	open an I/Os component
REGBASE <address> ;R	set the base address of the I/O register
REGFILE filename	load a registration entries file

Command, Syntax	Short Description
SET targetName	set a new target
SETCPU ProcessorName	set a new cpu simulator
SLAY fileName	save 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 of [Debugger Commands](#)). [Table 7.4](#) contains all available Component commands.

Table 7.4 List of Component Command

Command, Syntax	Short Description
CMDFILE	specify a command file state and full name
EXIT	terminates the application
HELP	displays a list of available commands
LOADMEM fileName	loads a memory configuration file
RESET	resets statistics
RESETCYCLES	resets Simulator CPU cycles counter
RESETMEM	resets all configured memory to ‘undefined’
RESETRAM	resets RAM to ‘undefined’
RESETSTAT	resets the statistical data
SHOWCYCLES	returns executed Simulator CPU cycles
SMEM range	shows a memory range



Command, Syntax	Short Description
SMOD module	shows module information in the destination component
SPC address	shows the specified address in a component window
SPROC level	shows information associated with the specified procedure
VER	displays version number of components and engine

Component Specific Commands

Component specific commands are associated with specific components. [Table 7.5](#) contains all available Component Specific commands.

Table 7.5 Component Specific Commands

Command, Syntax	Short Description
ADCPOR T (address ident) (address ident) (address ident)	sets the ports addresses used by the Adc_Dac component.
ADDCHANNEL ("Name")	creates a new channel "Name" for the Monitor component.
ADDXPR "expression"	adds a new expression in the data component
ATTRIBUTES list	sets up the display inside a component window
BASE code module	sets the Profiler base
BD	displays a list of all breakpoints
CF fileName [;C][;NL]	executes a command file
CLOCK frequency	sets the clock speed



Command, Syntax	Short Description
COPYMEM <Source addr range> dest-addr	copy memory
CPORT (address ident) (address ident) (address ident) (address ident) (address ident)	sets the 5 port addresses and control port address of the IO_Ports component
CYCLE on off	switches cycles and milliseconds
DELCHANNEL ("Name")	deletes the channel "Name" from the Monitor component
DETAILS assembly source	sets split view
DUMP	displays data component content
E expression [;O D X C B]	evaluates a given expression
EXECUTE fileName	executes a stimulation file
FILL range value	fills a memory range with a value
FILTER Options [<range>]	Select the output file filter options
FIND "string" [;B] [;MC] [;WW]	finds and highlights a pattern
FINDPROC ProcedureName	opens a procedure file
FOLD [*]	folds a source block
FRAMES number	sets the maximum number of frames
GRAPHICS on off	switches graphic bars on/off
INSPECTOROUTPUT [name {subname}]	prints content of Inspector to Command window
INSPECTORUPDATE	updates content of Inspector
ITPORT (address ident) (address ident)	sets the line and column port addresses of the IT_Keyboard component

Command, Syntax	Short Description
ITVECT (address ident)	sets the interrupt vector port address of the IT_Keyboard component.
KPORT (address ident) (address ident)	sets the line and column port addresses of the Keyboard component
LCDPORT (address ident) (address ident)	sets the data port and the control port address of the Lcd component
LINKADDR (address ident) (address ident) (address ident) (address ident) (address ident)	sets the components internal port addresses used with the IO_Ports as memory buffers
LS [symbol *][;C S]	displays the list of symbols
NB [base]	sets the base of arithmetic operations
OPENFILE fileName	opens a stimulation file
OUTPUT fileName	redirects the coverage component results
PBPORT (address ident)	sets the port address of the Push_Buttons component
PORT address	sets the Led components port address
PTRARRAY on off	switches on /off the pointer as array display
RECORD on off	switches on/off the frame recorder
SEGPORT (address ident) (address ident)	set the display selection port and the segment selection port addresses of the 7-Segments display component.
SLINE linenumber	shows the desired line number
SAVE range fileName [offset][;A]	saves a memory block in S-Record format
SETCOLORS ("Name") (Background) (Cursor) (Grid) (Line) (Text)	changes the colors attributes of the "Name" channel from the Monitor component

Command, Syntax	Short Description
SETCONTROL ("Name") (Ticks) (Pixels)	changes the number of ticks and pixels for the "Name" channel from the Monitor component
SREC fileName [offset]	loads a memory block in S-Record format
TUPDATE on off	switches on/off time update for statistics
UNFOLD [*]	unfolds a source block
UPDATERATE rate	sets the data and memory update mode
WPORT (address ident) (address ident	sets the ports addresses of the Wagon component
ZOOM address in out	zooms in/out a variable

Definitions of Terms Commonly Used in Command Syntaxes

address is 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.).

NOTE Please see also [Constant Standard Notation](#).

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. See also section [EBNF Notation](#) for [“Expression” Definition in EBNF](#).

Example: DEFINE IO_PORT = 0x210

WB IO_PORT 0xFF



range is 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 is 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 is 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

About 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., `fib.o`).

In **ELF** format, module name extensions are `.c`, `.cpp` or `.dbg` (`.dbg` for program sources in assembler) (e.g., `fib.o.c`), since all debugging information is contained in the `.abs` file and object files are not used.

Please consider or adapt the examples given in [Appendix](#) with your `.abs` application file format.

A

Description	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**.



Debugger Commands

Simulator/Debugger Commands

```
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

Description **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
```

This command activates the Memory Component and brings the window to the foreground.

ADDCHANNEL

Description 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 will be created in the Monitor component.

ADCPORT

Description 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`.

ADDXPR

Usage `ADDXPR "expression"`

Where the parameter expression is an expression to be added and evaluated in the data component.

Components Data component.

Description The **ADDXPR** command adds a new expression in the data component.

Example

```
in>ADDXPR "counter + 10"
```

The expression `"counter +10"` is added in the data component.

ATTRIBUTES

This command effective for various component is described in the next section.



In the Command Component

Description 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

Description 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

Description The **ATTRIBUTES** command allows you to set the display and state options for the Assembly component window. The **ADR** command

displays or hides the address of a disassembled instruction. **ON | OFF** is used to switch the address on or off. **SMEM** (show memory range) and **SPC** (show PC address) scroll the Assembly component to the corresponding address or range code location and select/highlight the corresponding assembler lines or range of code. The **CODE** command displays or hides the machine code of the disassembled instruction. **ON | OFF** is used to switch on or off the machine code. The **ABSADR** command shows or hides the absolute address of a disassembled instruction like 'branch to'. **ON | OFF** is used to switch on or off the absolute address. The **TOPPC** command scrolls the Assembly component in order to display the code location given as an argument on the first line of Assembly component window. The **SYMB** command displays or hides the symbolic names of objects. **ON | OFF** is used to switch the symbolic display on or off.

Usage **ATTRIBUTES list**

where **list=command{,command}**

command= **ADR (ON|OFF) | SMEM range | SPC address |
 CODE(ON|OFF) | ABSADR (ON|OFF) | TOPPC address | SYMB
 (ON|OFF)**

NOTE Also refer to [SMEM](#) and [SPC](#) command descriptions for more detail about these commands. The **SPC** command is similar to the **TOPPC** command but also highlights the code and does not scroll to the top of the component window.

Equivalent Operations

ATTRIBUTES ADR ~ Select menu **Assembly>Display Adr**

ATTRIBUTES SMEM ~ Select a range in Memory component window and drag it to the Assembly component window.

ATTRIBUTES SPC ~ Drag a register to the Assembly component window.

ATTRIBUTES CODE ~ Select menu **Assembly>Display Code**

ATTRIBUTES SYMB ~ Select menu **Assembly>Display Symbolic**

Example

```
Assembly < ATTRIBUTES ADR ON,SYMB ON, CODE ON, SMEM 0x800,16
```

Addresses, hexadecimal codes, and symbolic names are displayed in the Assembly component window, and assembly instructions at addresses 0x800,16 are highlighted.

In the Register Component

Description 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

Description 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 [Object Info Bar of the Simulator/Debugger Components](#). The **SPROC** (show procedure) command loads the corresponding module's source text, scrolls to the corresponding procedure and highlights the statement, that is in the procedure chain of this procedure. The **numberAssociatedToProcedure** is the level of the procedure in the procedure chain. The **MARKS** command (**ON** or **OFF**) displays or hides the marks.

NOTE Also refer to [SMEM](#), [SPC](#), [SPROC](#) and [SMOD](#) command descriptions for more detail about these commands.

Equivalent Operations

ATTRIBUTES SPC ~ Drag and drop from Register component to Source component.

ATTRIBUTES SMEM ~ Drag and drop from Memory component to Source component.

ATTRIBUTES SMOD ~ Drag and drop from Module component to Source component.

ATTRIBUTES SPROC ~ Drag and drop from Procedure component to Source component.

ATTRIBUTES MARKS ~ Select menu **Source>Marks** .

Usage

ATTRIBUTES list

where **list=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

Description

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|symp)**| **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.

In **Automatic** mode (default mode), variables are updated when target is stopped.

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

In **Locked** mode, values from variables displayed in the data component are updated when the target is stopped.

In **Frozen** mode, values from variables displayed in the data component are not updated when the target is stopped.

In **Periodical** mode, variables are updated at regular time intervals when the target is running. The default update rate is 1 second, but it can be modified by steps of up to 100 ms using the associated dialog box or the **UPDATERATE** command.

The **UPDATERATE** command sets the variables update rate (see also [UPDATERATE](#) command).

The **SPROC** (show procedure) and **SMOD** (show module) commands display local or global variables of the corresponding procedure or module.

The **SCOPE** command selects and displays global, local or user defined variables.

The **COMPLEMENT** command sets the display complement format of Data values: one sets the first complement (each bit is reversed), none unselects the first complement.

The **NAMEWIDTH** command sets the length of the variable name displayed in the window.

NOTE Refer to [SPROC](#), [UPDATERATE](#) and [SMOD](#) command descriptions for more detail about these commands.

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

Description 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) 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.



In **Automatic** mode (default mode), memory words are updated when target is stopped.

In **Frozen** mode, value from memory words displayed in the Memory component are not updated when the target is stopped.

In **Periodical** mode, memory words are updated at regular time intervals when the target is running. The default update rate is 1 second, but it can be modified by steps of up to 100 ms using the associated dialog box or **UPDATERATE** command.

The **UPDATERATE** command sets the variables update rate (see also [UPDATERATE](#) command).

NOTE Also refer to [SMEM](#), [SPC](#) and [SMOD](#) command descriptions for more detail about these commands.

Equivalent Operations

ATTRIBUTES FORMAT ~ Select menu **Memory>Format**

ATTRIBUTES WORD ~ Select menu **Memory>Word Size**

ATTRIBUTES ADR ~ Select menu **Memory>Display>Address**

ATTRIBUTES ASC ~ Select menu **Memory>Display>ASCII**

ATTRIBUTES ADDRESS ~ Select menu **Memory>Address...**

ATTRIBUTES COMPLEMENT ~ Select menu **Memory>Format**

ATTRIBUTES SMEM ~ Drag and drop from Data component (variable) to Memory component.

ATTRIBUTES SMOD ~ Drag and drop from Source component to Memory component.

ATTRIBUTES MODE ~ Select menu **Memory>Mode...**

ATTRIBUTES UPDATERATE ~ Select menu **Memory>Mode>Periodical**

Usage ATTRIBUTES list

where **list=command{,command}**)

**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

Description 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.

TIP 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

Usage **AT time**



where **time=expression** and **expression** is interpreted in milliseconds.

Components Debugger engine.

Description 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.

Example

AT 10 OPEN Command

This command (in command file) opens the **Command Line component** 10 ms after the command file is executed.

AUTOSIZE

Description **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

Description 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

Description **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 [Breakpoints setting dialog](#) and choose **Show Breakpoint**, select the breakpoint 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.

Example

in>BC 0x8000

This command deletes the breakpoint set at the address 0x8000. The breakpoint symbol is removed in the source and assembly window. The breakpoint is removed from the breakpoint list.



Example

```
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

Description 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

Description	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

Description	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, open the [Breakpoints setting dialog](#) 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., `fib.o.o`). In **ELF** format, module name extensions are `.c`, `.cpp` or `.dbg` (`.dbg` for program sources in assembler) (e.g., `fib.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 [Breakpoints setting dialog](#) 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 [Breakpoints setting dialog](#), 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 [Breakpoints setting dialog](#), for counting breakpoints.

interval is an **expression**. It matches the **Interval** field (**Counter**) in the [Breakpoints setting dialog](#), 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 Fib0.c:main{3}
```

Sets a breakpoint at the 3rd mark of the procedure **main**, where **main** is a function of the FIBO.C module.



Debugger Commands

Simulator/Debugger Commands

```
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

Description Executes 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

Description 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:\Metrowerks\demo
in>cd
C:\Metrowerks\demo
in>cd /Metrowerks/prog
C:\Metrowerks\prog
```

The new project directory is C:\Metrowerks\prog

CF

Description The **CF** command reads the commands in the specified command file, which are then executed by the command interpreter. The command file contains ASCII text commands. Command files can be nested. By default, after executing the commands from a nested command file, the command interpreter resumes execution of remaining commands in the calling file. Any error halts execution of **CF** file commands. When the command is entered with no parameter, the **Open File** dialog is displayed. The **CALL** command is equivalent to the **CF** command.

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

Usage CF **fileName** [;C][;NL]

Where **fileName** is a file (and path) containing Simulator/Debugger commands.



;C specifies chaining the command file. This option is meaningful in a nested command file only.

When the **;C** option is given in the calling file, the command interpreter quits the calling file and executes the called file. (i.e. in the calling file, commands following the **CF ... ;C** command are never executed).

When the option is omitted, execution of the remaining commands in the calling file is resumed after the commands in the called file have been executed.

;NL: when set, the commands that are in the called file are not logged in the Command Line window (and not to log file, when a file has been opened with an **LF** command), even if the **CMDFILE** type is set to **ON** (see **LOG** command).

Components Debugger engine.

Examples:

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

The **COMMANDS . TXT** file is executed. It should contain debugger commands like those described in the [Debugger Commands](#) chapter.

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
```

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

Description 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

Description 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

Description 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 overlaid.

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

- Description** 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.

CPORT

- Description** 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.

CR

Description The **CR** command initiates writing records of commands to an external file.

Writing records continues until a close record file ([NOCR](#)) command is executed.

NOTE Drag & drop actions are also translated into commands in the record file.

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

Usage CR [fileName][;A]

If fileName is not specified, a standard **Open File** dialog is opened.

;A specifies to open a file **fileName** in append mode. Records are appended at the end of an existing record file.

If the **;A** option is omitted and **fileName** is an existing file, the file is cleared before records are written to it.

Components Debugger engine.

Example

```
in>cr /Metrowerks/demo/myrecord.txt ;A
```

The myrecord.txt file is opened in “Append” mode for a recording session.

CYCLE

Description In the **SoftTrace component**, the **CYCLE** command displays or hides cycles. When cycle is off, milliseconds (ms) are displayed.

Usage CYCLE on|off

Components Softrtrace component.

Example

in>CYCLE on

DASM

Description 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  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: This is a constant expression representing the **address** where disassembly begins.

range: This is 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 for PPC

```
in>dasm 0x3fc2f0
3FC2F0 bla      0x003fc27c
3FC2F4 lis      r4, 64
3FC2F8 stw      r3, -26624(r4)
3FC2FC addi     r31, r31, 1
```

Debugger Commands

Simulator/Debugger Commands

```

3FC300 cmpwi    crf5, r31, 25
3FC304 ble     crf5, 0x003fc2d4
3FC308 b       0x003fc2c4
3FC30C lwz    r31, 12(r1)
3FC310 addi   r1, r1, 16
3FC314 lwz    r0, 4(r1)
3FC318 mtlr   r0
3FC31C blr

```

NOTE Depending on the target, the above code may vary.

Disassembled instructions are displayed in the Command Line component window.

NOTE It is necessary to open the Command Line component before executing this command to see the dumped code.

DB

Description 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  key.

Usage DB [address|range]

When **address** and **range** are omitted, the first longword displayed is taken from the address following the last longword displayed by the most recent **DB**, **DW**, or **DL** command, or from address **0x0000** (for the first **DB**, [DW](#), [DL](#) command of a session).

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  
p_Eý_C'.5íììlipf
```

Memory bytes are displayed in the Command Line component window, with matching ASCII characters.

NOTE 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

Description 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.



Debugger Commands

Simulator/Debugger Commands

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 [Debugger DDE capabilities](#).

DEFINE

Usage DEFINE symbol [=] expression

Components Debugger engine.

Description 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.

TIP 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.

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](#).

DELCHANNEL

Description 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.

DETAILS

Description 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

Description 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  key.

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

Usage DL [address|range]

When **range** is omitted, the first longword displayed is taken from the address following the last longword displayed by the most recent **DB**, **DW**, or **DL** command, or from address **0x0000** (for the first [DB](#), [DW](#), [DL](#) command of a session).

Components Debugger engine.

Example

```
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

Description 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

Description 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  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](#), **DW**, [DL](#) 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

Description 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 a 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.

NOTE Refer to [“Expression” Definition in EBNF](#) in [Appendix](#) for more detail about expression.

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](#) and [LOG](#) command descriptions).

ELSE

Description The **ELSE** keyword is associated with the [LF](#) command.

Usage ELSE

Components Debugger engine.

Example

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

ELSEIF

Description The **ELSEIF** keyword is associated with the [IF](#) command.

Usage ELSEIF condition

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



Debugger Commands

Simulator/Debugger Commands

Components Debugger engine.

Example

```
if CUR_TARGET == 1000          /* Simulator */
    set sim
elseif CUR_TARGET == 1001 /* BDI */
    set bdi
```

ENDFOCUS

Description The **ENDFOCUS** command resets the current focus. It is associated with the **FOCUS** command. Following commands are broadcast to all currently open components. This command is only valid in a command file.

Usage ENDFOCUS

Components Debugger engine.

Example

```
FOCUS Assembly
ATTRIBUTES code on
ENDFOCUS
FOCUS Source
ATTRIBUTES marks on
ENDFOCUS
```

The **ATTRIBUTES** command is first redirected to the Assembly component by the **FOCUS** Assembly command. The code is displayed next to assembly instructions. Then the Assembly component is released by the **ENDFOCUS** command and the second **ATTRIBUTES** command is redirected to the Source component by the **FOCUS** Source command. Marks are displayed in the Source window.

ENDFOR

Description The **ENDFOR** keyword is associated with the [FOR](#) command.

Usage ENDFOR

Components Debugger engine.

Example

```
for i = 1..5
  define multi5 = 5 * i
endfor
```

After the **ENDFOR** instruction, i is equal to 5.

ENDIF

Description The **ENDIF** keyword is associated with the [IF](#) command.

Usage ENDIF

Components Debugger engine.

Example

```
if (CUR_CPU == 12)
  DW &counter
else
  DB &counter
endif
```

ENDWHILE

Description The **ENDWHILE** keyword is associated with the [WHILE](#) command.

Usage ENDWHILE

Components Debugger engine.

Example

```
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 from the currently highlighted statement or from the beginning of the file (if nothing is highlighted). If the item is found, the Source window is scrolled to the position of the item and the item is highlighted in grey.

Equivalent Operation

You can select **Source>Find...** or open the Source popup menu and select **Find...** to open the **Find** dialog.

Usage FIND “string” [;B] [;MC] [;WW]

Where **string** is the “**pattern**” to match. It has to be enclosed in double quotes. See the example below.

;B the search is backwards, default is forwards.

;MC match case sensitive is set.

;WW match whole word is set.

Components Source component.

Example

```
in>FIND "this" ;B ;WW
```

The “**this**” string (considered as a whole word) is searched in the Source component window. The search is performed backward.

FINDPROC

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

Equivalent Operation

You can select **Source>Find Procedure...** or open the Source popup menu and select **Find Procedure...** to open the **Find Procedure** dialog.

Usage FINDPROC procedureName

Components Source component.

Example

```
in>findproc Fibonacci
```

The “**Fibonacci**” procedure is displayed and the title is highlighted.

FOCUS

Description The **FOCUS** command sets the given component (**component**) as the destination for all subsequent commands up to the next [ENDFOCUS](#) command. Hence, the focus command releases the user from repeatedly specifying the same command redirection, especially in the case where command files are edited manually. This command is only valid in a command file.

NOTE It is not possible to visually notice that a component is “FOCUSed”. However, you can use the [ACTIVATE](#) command to activate a component window.

Usage FOCUS component

Components Debugger engine.

Example

```
FOCUS Assembly
ATTRIBUTES code on
ENDFOCUS
FOCUS Source
ATTRIBUTES marks on
ENDFOCUS
```

The ATTRIBUTES command is first redirected to the Assembly component by the **FOCUS** Assembly command. The code is displayed next to assembly instructions. Then the Assembly component is released by the ENDFOCUS command and the second ATTRIBUTES command is redirected to the Source component by the **FOCUS** Source command. Marks are displayed in the Source window.

FOLD

Description In the Source component, the **FOLD** command hides the source text at the program block level. Folded program text is displayed as if the program block was empty. When the folded block is unfolded, the hidden program text reappears. All text is folded once or (*) completely, until there are no more folded parts.

Usage FOLD [*]

Where * means fold completely, otherwise fold only one level.

Components Source component.

Example

```
in>FOLD *
```

FONT

Description FONT sets the font type, size and color.

Equivalent Operation

The **Font** dialog is available by selecting the **Component>Fonts...** menu entry.

Usage FONT 'FontName' [size][color]

Components Debugger engine.

Example

```
FONT 'Arial' 8 BLUE
```

The font type is “Arial” 8 points and blue.

FOR

Description The **FOR** loop allows you to execute all commands up to the trailing [ENDFOR](#) a predefined number of times. The bounds of the range and the optional steps are evaluated at the beginning. A **variable** (either a symbol or a program variable) may be optionally specified, which is assigned to all values of the range that are met during execution of the for loop. If a variable is used, it must be defined before executing the **FOR** command, with a [DEFINE](#) command.

Assignment happens immediately before comparing the iteration value with the upper bound. The variable is only a copy of the internal iteration value, therefore modifications on the variable don't have an impact on the number of iterations.

This command can be stopped by pressing the  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
```



Debugger Commands
Simulator/Debugger Commands

```
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

Description 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

Description The conditional commands ([IF](#), [ELSEIF](#), [ELSE](#) and [ENDIF](#)) allow you to execute different sections depending on the result of the corresponding condition. The conditional command may be nested. Conditions of the **IF** and **ELSEIF** commands, respectively, guard all commands up to the next **ELSEIF**, **ELSE** or **ENDIF** command on the same nesting level. The **ELSE** command guards all commands up to the next **ENDIF** command on the same nesting level. Any occurrence of a subcommand not in sequence of “**IF**, zero or more **ELSEIF**, zero or one **ELSE**, **ENDIF**” is an error.

Usage IF condition
 Where **condition** is same as defined in “C” language.

Components Debugger engine.

Example

```
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

Description 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

Description 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
```

ITPORT

Description 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

Description 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

Description 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

Description 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.

LF

Description The LF command initiates logging of commands and responses to an external file or device. While logging remains in effect, any line that is appended to the command window is also written to the log file.

Logging continues until a close log file ([NOLF](#)) command is executed. When the LF command is entered with no filename, the Open File Dialog is displayed to specify a filename.

Use the logging option ([LOG](#)) command to specify information to be logged.

If a path is specified in the file name, this path must be a valid path. When a relative path is specified, ensure that the path is relative to the project directory.

Usage LF [fileName][:A]

fileName is a DOS filename that identifies the file or device where the log is written. The command interpreter does not assume a filename extension.



;A opens the file in append mode. Logged lines are appended at the end of an existing log file.

If the **;**A option is omitted and **fileName** is an existing file, the file is cleared before logging begins.

Components Debugger engine.

Example

```
in>lf /mcuez/demo/logfile.txt ;A
```

The logfile.txt file is opened as a Log File in “append” mode.

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

LOAD

Description 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

Description 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.

LOADMEM

Description This command loads a memory configuration file.

Usage LOADMEM fileName

Components Simulator component.

Equivalent Operation

You can select the **Open** button in the **Memory Configuration** dialog box to load a memory configuration file.

Example

```
in>LOAD DEFAULT.MEM
```

The memory configuration file DEFAULT .MEM is loaded.

LOADSYMBOLS

Description 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.

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

LOG

Description The **LOG** command enables or disables logging of information in the Command Line component window (and to logfile, when it as been opened with an **LF** command). If **LOG** is not used, all types are **ON** by default i.e. all information is logged in the Command Line component and log file.

NOTE - **about RESPONSES**: Responses are results of commands. For example, for the DB command, the displayed memory dump is the response of the command. Protocol messages are not responses. - **about ERRORS**: Errors are displayed in red in Command Line component. Protocol messages are not errors. - **about NOTICES**: Notices are displayed in green in the Command Line.



Usage LOG type [=] state {[, type [=] state }

where **type** is one of the following types:

CMDLINE: Commands entered on the command line.

CMDFILE: Commands read from a file.

RESPONSES: Command output response.

ERRORS: Error messages.

NOTICES: Asynchronous event notices, such as breakpoints.

where **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**.

Example 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
```



Debugger Commands

Simulator/Debugger Commands

When commands executed from a command file are logged, all executed commands that are in the **WHILE** loop are logged the number of times they have been executed. That is, a command file executed with the **CF** or **CALL** command without the **NL** option and with the **CMDFILE** flag of the **LOG** command set to **TRUE**.

Example 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**.

Example 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

Description In the Command Line window, the **LS** command lists the values of symbols defined in the symbol table and by the user. There is no limit to the number of symbols that can be listed. The size of memory determines the symbol table size. Use the [DEFINE](#) command to define symbols, and the [UNDEF](#) command to delete symbols.

The symbols that are listed with the LS command are split in two parts: Applications Symbols and User Symbols.

Usage LS [symbol | *][;C|S]

Where **symbol** is a restricted regular expression that specifies the symbol whose values are to be listed.

* specifies to list all symbols.

;C specifies to list symbols in canonical format, which consists of a DEFINE command for each symbol.

;S specifies to list symbol table statistics following the list of symbols.

Components Debugger engine.



Debugger Commands

Simulator/Debugger Commands

Example

```
in>ls
```

User Symbols:

```
j      0x2 (2)
```

Application Symbols:

```
counter  0x80 (128)
```

```
fibonacci 0x81 (129)
```

```
j      0x83 (131)
```

```
n      0x84 (132)
```

```
fib1    0x85 (133)
```

```
fib2    0x87 (135)
```

```
fibonacci 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

Usage	MEM
Components	Debugger engine.
Description	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.

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

Description The **MS** command sets a specified block of memory to a specified list of byte values. When the **range** is wider than the **list** of byte values, the **list** of byte values is repeated as many times as necessary to fill the memory block.

When the **range** is not an integer multiple of the length of the **list**, the last copy of the **list** is truncated appropriately. This command is identical to the write bytes ([WB](#)) command.

Usage MS range list

range: is an address range constant that defines the block of memory to be set to the values of the bytes in the list.

list: is a list of byte values to be stored in the block of memory.

Components Debugger engine.



Example

```
in>MS 0x1000..0x100F 0xFF
```

The memory range between addresses 0x1000 and 0x100F is filled with the 0xFF value.

NB

Usage NB [base]

where **base** is the new number base (2, 8, 10 or 16).

Components Debugger engine.

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 7.6](#).

Table 7.6 ANSI C constant notation

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

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 7.7](#).

Table 7.7 Assembler notation for constant

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

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 7.8](#). Otherwise, the command line interpreter cannot detect if you are specifying a number or a symbol.

Table 7.8 Base is 16: constants starting with a letter A, B, C, D, E or F

Notation.	Meaning
5AFD	Hexadecimal constant \$5AFD
AFD	Hexadecimal constant \$AFD

Example

in>NB 16



The number base is hexadecimal.

NOCR

Description The **NOCR** command closes the current record file. The record file is opened with the [CR](#) command.

Usage NOCR

Components Debugger engine.

Example

```
in>NOCR
```

The current record file is closed.

NOLF

Description The **NOLF** command closes the current Log File. The log file is opened with the [LF](#) command.

Usage NOLF

Components Debugger engine.

Example

```
in>NOLF
```

The current Log File is closed.

OPEN

Description 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.

OPENFILE

Description In the **Stimulation component**, the **OPENFILE** command opens a specified file to run a Stimulation.

Usage OPENFILE **fileName**

Where **fileName** is name of stimulation file.

Components Stimulation component.

Example

```
in>OPENFILE d:\demo\io_demo.txt
```

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

OPENIO

Description The **OPENIO** command is used to open a I/O component (components whose DLL file name has a “.io” extension).

Usage OPENIO "IOcomponentName"



Debugger Commands

Simulator/Debugger Commands

Where IOcomponentName is the name (with an optional path), without extension, of the I/O component to open.

Components Debugger engine.

Example

```
in>OPENIO "demo"
```

The demo I/O component is opened.

```
in>OPENIO "c:\Metrowerks\prog\myio\Myio"
```

The Myio I/O component is opened.

OUTPUT

Description 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:\Metrowerks\myfile.txt
```

The Coverage output results are redirected to the file `myfile.txt` from the directory `C:\Metrowerks`.

P

Description The **P** command executes a CPU instruction, either at a specified address or at the current instruction, (pointed to by the program counter). This command traces through subroutine calls, software interrupts, and operations involving the following instructions (two are target specific):

- Branch to SubRoutine (**BSR**)
- Long Branch to Subroutine (**LBSR**)
- Jump to Subroutine (**JSR**)



- Software Interrupt (**SWI**)
- Repeat Multiply and Accumulate (**RMAC**)

For example: if the current instruction is a **BSR** instruction, the subroutine is executed, and execution stops at the first instruction after the **BSR** instruction. For instructions that are not in the above list, the **P** and **T** commands are equivalent.

When the instruction specified in the **P** command has been executed, the software displays the content of the CPU registers, the instruction bytes at the new value of the program counter and a mnemonic disassembly of that instruction.

Usage P [address]

address: is an address constant expression, the address at which execution begins.

If **address** is omitted, execution begins with the instruction pointed to by the current value of the program counter.

Components Debugger engine.

Example

Example for PPC

```
in>p
```

```
TRACED
STARTED
RUNNING
STOPPING
HALTED
R0=0x3FC1E8 R1=0x3F9AE8 R2=0x0 R3=0x18 R4=0xD R5=0xE9 R6=0x90
R7=0x90 R8=0x0 R9=0x0 R10=0x0 R11=0x0 R12=0x0 R13=0x0
R14=0x0 R15=0x0 R16=0x0 R17=0x0 R18=0x0 R19=0x0 R20=0x0
R21=0x0 R22=0x0 R23=0x0 R24=0x0 R25=0x0 R26=0x0 R27=0x0
R28=0x0 R29=0x0 R30=0x0 R31=0x17 FR00=???? FR01=???? FR02=????
FR03=???? FR04=???? FR05=???? FR06=???? FR07=???? FR08=????
FR09=????
```

Debugger Commands

Simulator/Debugger Commands

```
FR10=???? FR11=???? FR12=???? FR13=???? FR14=???? FR15=????
FR16=????
FR17=???? FR18=???? FR19=???? FR20=???? FR21=???? FR22=????
FR23=????
FR24=???? FR25=???? FR26=???? FR27=???? FR28=???? FR29=????
FR30=????
FR31=???? CR=0x40000800
```

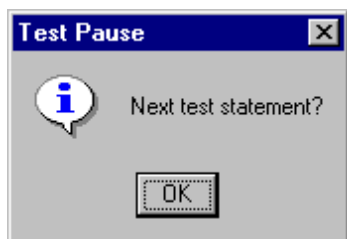
```
3FC298 7CA62B79 mr.          r6, r5
```

Contents of registers are displayed and the current instruction is disassembled.

PAUSETEST

Description Displays a modal message box shown in [Figure 7.1](#) for testing purpose.

Figure 7.1 PAUSETEST message box



Usage PAUSETEST

Components Debugger engine.

Example

```
in> pausetest
```

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
```

PRINTF

Description 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

- Description** The **PTRARRAY** command allows to specify if a pointer should be displayed as an array.
- Usage** PTRARRAY on|off [nb]
- Where
- **on** displays pointers as arrays.
 - **off** displays pointers as usual (*pointer).
 - **nb** is the number of elements to display in the array when unfolding a pointer displayed as array.
- Components** Data component.

Example

```
in>Ptrarray on 5
```

Display content of pointers as array of 5 items.

RD

- Description** 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. You can load a register file with the command [REGFILE](#). The address and size of each register is displayed. 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.

Example for PPC

```
in>rd cpu
```

```
R0=0x3FC1E8 R1=0x3F9AE8 R2=0x0 R3=0xC R4=0x8 R5=0x15 R6=0x15
R7=0xD R8=0x0 R9=0x0 R10=0x0 R11=0x0 R12=0x0 R13=0x0 R14=0x0
R15=0x0 R16=0x0 R17=0x0 R18=0x0 R19=0x0 R20=0x0 R21=0x0 R22=0x0
R23=0x0 R24=0x0 R25=0x0 R26=0x0 R27=0x0 R28=0x0 R29=0x0 R30=0x0
R31=0xB FR00=0xAAAAAAAAFF FR01=00 FR02=00 FR03=00 FR04=00?
FR05=00 FR06=00 FR07=00 FR08=00 FR09=00 FR10=00 FR11=00 FR12=00
FR13=00 FR14=00 FR15=00 FR16=00 FR17=00 FR18=00 FR19=00 FR20=00
FR21=00 FR22=00 FR23=00 FR24=00 FR25=00 FR26=00 FR27=00 FR28=00
FR29=00 FR30=00 FR31=00 CR=0x40000800
```

RECORD

Description 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
```

REGBASE

Description 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.

REGFILE

Description This command allows you to load a file containing I/O register descriptions from a register file.

Usage Regfile <filename>

Where Regfile is a register filename (with a .REG extension).


Components Debugger engine.

Example

in>REGFILE MDEF.REG

REPEAT

Description 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  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

Description 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
```

RESETCYCLES

Description 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>



Debugger Commands

Simulator/Debugger Commands

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

Description 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

Description 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

Description This command resets the statistics (read and write counters to zero)



Debugger Commands

Simulator/Debugger Commands

Usage RESETSTAT

Components Simulator component.

Example

in>RESETSTAT

After the **RESETSTAT** command, all counters are initialized to zero.

RESTART

Description 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

Usage RETURN

Components Debugger engine.

Description The **RETURN** command terminates the current command processing level (returns from a [CALL](#) command). If executed within a command file, control is returned to the caller of the command file (i.e. the first instance that did not chain execution).

Example in file d:\demo\cmd1.txt:

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

in 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](#) Trace instruction is executed.

RS

Description 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 for PPC

```
in>rs R4=0x8000
in>rd R4
```

```
R4=0x8000
```

S

Description The **S** command stops execution of the emulation processor. Use the Go [G](#) command to start the emulator.



Debugger Commands

Simulator/Debugger Commands

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

Description The **SAVE** command saves a specified block of memory to a specified file in Motorola S-record format. The memory block can be reloaded later using the load S-record ([SREC](#)) command.

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

Usage SAVE **range** fileName [offset][;A]

offset: an optional offset to add or subtract from addresses when writing S-records. The default offset is **0**.

;A: appends the saved S-records to the end of an existing file. If this option is omitted, and the file specified by fileName exists, the file is cleared before saving the S-records.

Components Debugger engine.

Example

```
in>SAVE 0x1000..0x2000 DUMP.SX ;A
```

The memory range 0x1000..0x2000 is appended to the DUMP .SX file.

SAVEBP

Description 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 [Breakpoints setting dialog](#). It is used to store currently defined breakpoints (**SAVEBP on**) when the user quits the Simulator/Debugger or loads another .ABS file.

NOTE For more information about this syntax, refer to [BS](#) command, [Control Points](#) chapter, and [Saving Breakpoints](#) section.

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
```

SEGPOR

Description The SEGPOR command is used to set the display selection port and segment selection port addresses of the 7-Segments display component.

Usage SEGPOR 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>SEGPOR 0x100 0x200
```

The ports of the 7-Segments display are now defined at addresses 0x100 and 0x200.

SET

Description 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

Description 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 : 0x00bbgrr).

Cursor is the new color for the channel cursor (the format is: 0x00bbgrr).

Grid is the new color for the channel grid (the format is: 0x00bbgrr).

Line is the new color for the channel lines (the format is: 0x00bbgrr).

Text is the new color for the channel text (the format is: 0x00bbgrr).

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.

SETCONTROL

Description 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

Description 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 PPC
```

The simulator PPC.sim is loaded.

SHOWCYCLES

Description The **SHOWCYCLES** command returns the number of CPU cycles already done since the beginning of the simulation in the Command Line component (**RESETCYCLES** is performed internally), or since the last [RESETCYCLES](#) command. The number of cycles executed is also the number displayed in the status bar (CPU cycles counter).

Usage SHOWCYCLES

Components Debugger engine.

Example

```
in>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.

SLAY

Description The **SLAY** command is used to save the layout of all window components in the main application window to a specified file.

TIP 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

Description 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.



Debugger Commands

Simulator/Debugger Commands

Usage SLINE line number

Components Source component

Example

```
in>sline 15
```

SMEM

Description 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

Description 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., `fib.o`). In **ELF** format, module name extensions are `.c`, `.cpp` or `.dbg` (`.dbg` or program sources in assembler) (e.g., `fib.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 `fib.o.c` module are displayed in the `Data:1` component window.

SPC

Description 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

Description 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.

TIP 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

Description The **SREC** command initiates the loading of Motorola 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

Description 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

Description 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

Description 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

Description The **STOP** command stops execution of the emulation processor. Use the Go [G](#) command to start the emulator.

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

Description 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  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 for PPC

```
in>t
```

```
TRACED
R0=0x3FC1E8 R1=0x3F9AE8 R2=0x0 R3=0xC R4=0x8001 R5=0x15 R6=0x15
R7=0xD R8=0x0 R9=0x0 R10=0x0 R11=0x0 R12=0x0 R13=0x0
R14=0x0 R15=0x0 R16=0x0 R17=0x0 R18=0x0 R19=0x0 R20=0x0
R21=0x0 R22=0x0 R23=0x0 R24=0x0 R25=0x0 R26=0x0 R27=0x0
R28=0x0 R29=0x0 R30=0x0 R31=0xB FR00=???? FR01=???? FR02=????
```

```
FR03=???? FR04=???? FR05=???? FR06=???? FR07=???? FR08=????
FR09=????
FR10=???? FR11=???? FR12=???? FR13=???? FR14=???? FR15=????
FR16=????
FR17=???? FR18=???? FR19=???? FR20=???? FR21=???? FR22=????
FR23=????
FR24=???? FR25=???? FR26=???? FR27=???? FR28=???? FR29=????
FR30=????
FR31=???? CR=0x40000800
```

```
3FC2A0 7E841840 cmlw      crf5, r4, r3
```

Contents of registers are displayed and current instruction is disassembled.

TESTBOX

Description Displays a modal message box shown in [Figure 7.2](#) with a given string.

Figure 7.2 TESTBOX message box



Usage TESTBOX "<String>"

Components Debugger engine.

Example

```
in>TESTBOX "Step 1: init all vars"
```

TUPDATE

Description 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

Description The **UNDEF** command removes a symbol definition from the symbol table. This command does not undefine the symbols defined in the loaded application.

Program variables whose names were redefined using the [UNDEF](#) command are visible again. Undefining an undefined symbol is not considered an error.

Usage UNDEF symbol | *

If * is specified, all symbols defined previously using the command DEFINE are undefined.

Components Debugger engine.

Example

```
DEFINE test = 1
...
UNDEF test
```

When the test variable is no longer needed in a command program, it can be undefined and removed from the list of symbols. After UNDEF test, the test variable can no longer be used without (re)defining it.

NOTE See also examples of the [DEFINE](#) command.

Examples The value of an existing symbol can be changed by applying the DEFINE command again. In this case, the previous value is replaced and lost. It is not put on a stack. Then when UNDEF is applied to the symbol, it no longer exists, even if the value of the symbol has been replaced several times:



```
in>DEFINE apple 0
in>LS
```

```
apple          0x0 (0)    // apple is equal to 0
```

```
in>DEFINE apple = apple + 1
in>LS
```

```
apple          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
fibonacci        0x800 (2048)
counter          0x802 (2050)
_startupData     0x84D (2125)
Fibonacci        0x867 (2151)
```



Debugger Commands

Simulator/Debugger Commands

```
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
fibonacci     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
fibonacci     0x800 (2048)
counter       0x802 (2050)
_startupData  0x84D (2125)
Fibonacci     0x867 (2151)
main          0x896 (2198)
Init          0x810 (2064)
_Startup      0x83D (2109)
```

UNFOLD

Description	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

Description The **UNTIL** keyword is associated with the [REPEAT](#) command.

Usage UNTIL condition

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

Components Debugger engine.

Example

```
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.



Debugger Commands

Simulator/Debugger Commands

Example

```
in>Memory < updatarate 30
```

This commands sets the Memory component updatarate to 3 seconds.

VER

Description 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

Description 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

Description The **WB** command sets a specified block of memory to a specified list of byte values. When the range is wider than the list of byte values, the list of byte values is repeated as many times as necessary to fill the memory block. When the range is not an integer, a multiple of the length of the list and the last copy of the list is truncated accordingly. This command is identical to the memory set ([MS](#)) command.

Usage WB range list

range: is an address range constant that defines the block of memory to be set to the values of the bytes in the list.

list: is a list of byte values to be stored in the block of memory.

Alias MS

Components Debugger engine.

Example

```
in>WB 0x0205..0x0220 0xFF
```

This command fills up the memory range 0x0205..0x0220 with the 0xFF byte value.

WHILE

Description 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  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

Description 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 0x0FFFFFF0F
```

This command fills up memory starting at address 0x2000 with the 0x0FFFFFF0F longword value. The addresses 0x2000 to 0x2003 will be modified.

```
in>WL 0x2000, 2 0x0FFFFFF0F
```

This command fills up the memory area 0x2000 to 0x2007 with the longword value 0x0FFFFFF0F.

WPORT

Description 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.

WW

Description 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 times 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

Description In the Data component, the **ZOOM** command is used to display the member fields of structures by ‘diving’ into the structure. In contrast to the [UNFOLD](#) command, where member fields are not expanded in place. The display of the member fields replaces the previous view. The **ZOOM out** command is used to return to the nesting level indicated by the given identifier.



TIP 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).

True Time I/O Stimulation

The Simulator/Debugger I/O Stimulation component is a facility to trigger I/O events. With the Stimulation component loaded, interrupts and register modifications invoked by the hardware can be simulated. In this tutorial, examples of stimulation files are introduced and explained.

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

- [Stimulation Program examples](#)
- [Stimulation Input File Syntax](#)

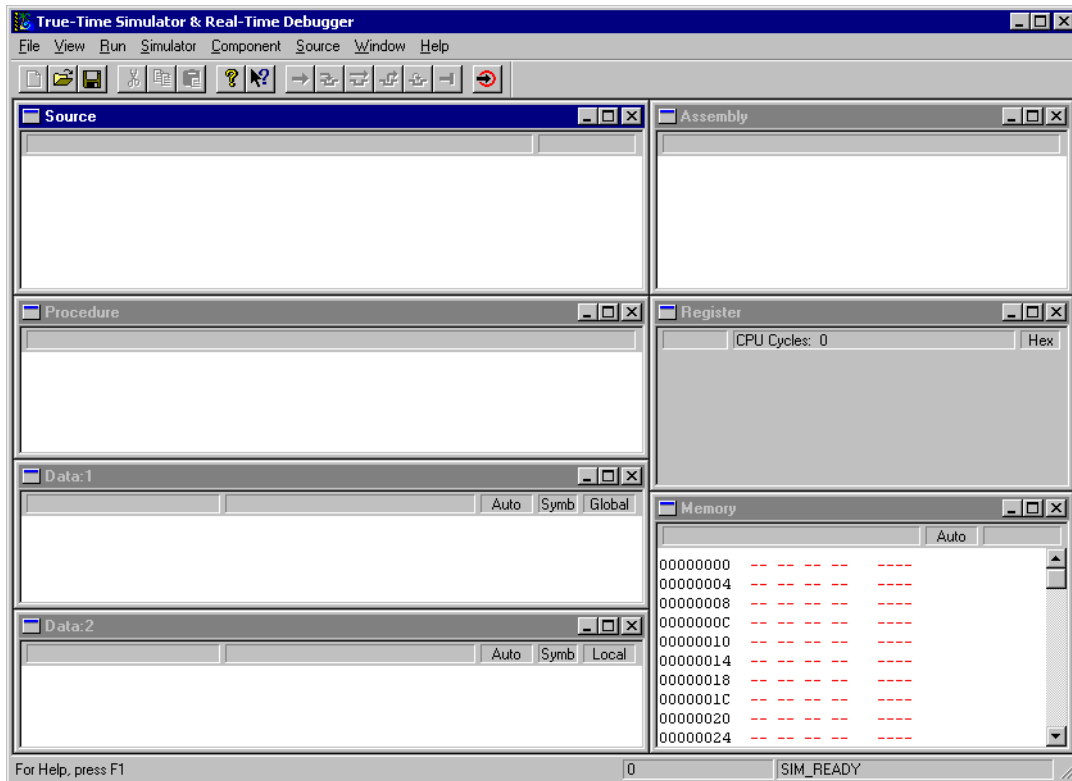
Stimulation Program examples

Running an Example Program Without Stimulation

1. **Run the Simulator/Debugger.**

The Main Window is shown in [Figure 8.1](#).

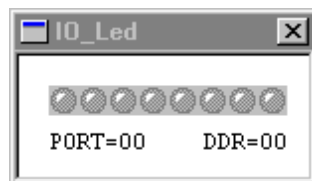
Figure 8.1 Simulator/Debugger I/O-Simulation main window



2. Choose **S**imulator > **S**et > **S**im.
3. Choose **C**omponent > **O**pen > **I**o_led.

The IO_Led component is shown in [Figure 8.2](#).

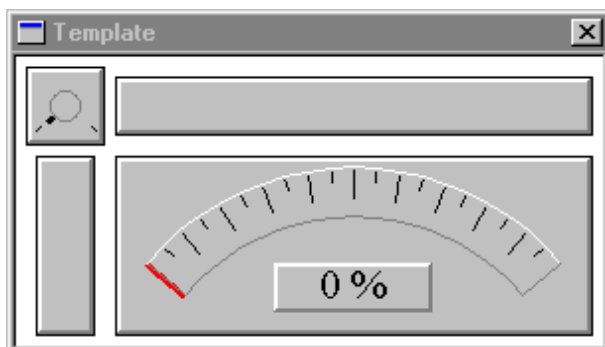
Figure 8.2 IO_Led Component window



4. Choose **C**omponent > **O**pen > **T**emplate.

The Template component is shown in [Figure 8.3](#).

Figure 8.3 Template component window



5. Choose **S**imulator>**L**oad io_demo.abs.
6. Choose **R**un>**S**tart/Continue or click the 'green arrow' icon.
7. If the program halts in startup, click the **S**tart/Continue command again.
8. Choose **R**un > **H**alt 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 [Listing 8.1](#), 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 8.1 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));
    }
}
```

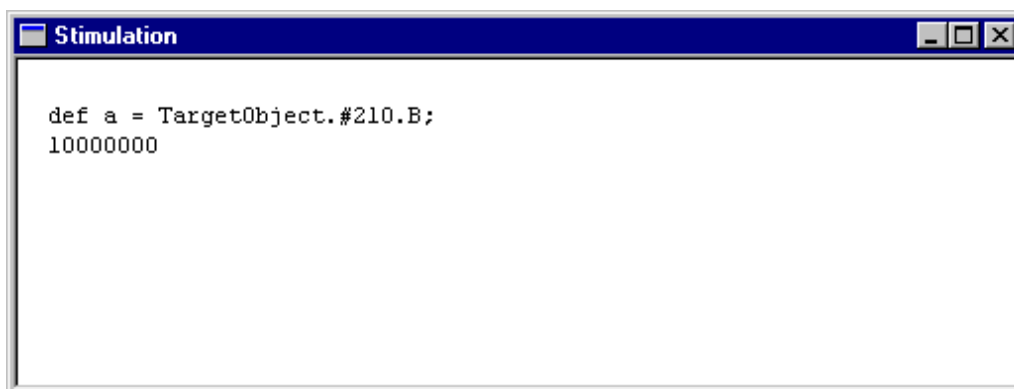
```
dir = -1;
do {
    Delay();
    PORT_DATA--;
} while ((dir == -1) && (PORT_DATA != 0));
}
```

Example Program with Periodical Stimulation of a Variable

1. Choose **S**imulator > **R**eset.
2. Choose **S**imulator | **L**oad Io_demo.abs.
3. Choose **C**omponent | **O**pen | **S**timulation

The Stimulation component is shown in [Figure 8.4](#).

Figure 8.4 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 **R**un > **S**tart/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 simulator demo directory. This file looks like [Listing 8.2](#).

Listing 8.2 `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.

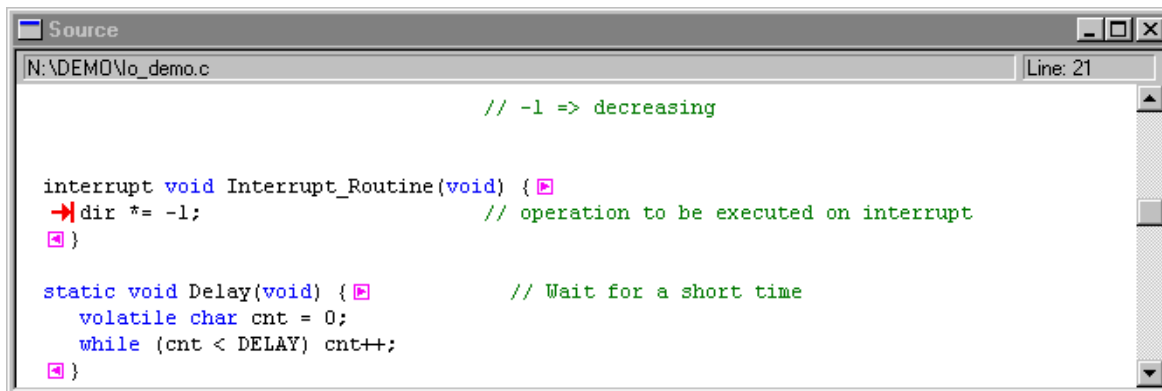
Example Program with Stimulated Interrupt

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 8.5](#).

Figure 8.5 Source component window



8. Activate Stimulation Window by clicking on it.

9. Choose Stimulation>Execute.

10. Choose Run>Start/Continue.

After about 300,000 cycles the simulator stops on the breakpoint in the interrupt routine and the corresponding source line is highlighted. The interrupt has been called. Start the simulator. It stops approximately each 100,000 cycles on the same breakpoint. Restart and repeat these actions until 1,200,000 cycles. Start again, the simulator runs until 10,000,000 cycles and stops on the breakpoint. Start the simulator. 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 [Listing 8.3](#).

Listing 8.3 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, 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 simulator will stop when interruption is generated. The exception mnemonic (matching the interrupt vector number) is displayed in the status bar of the Simulator/Debugger.

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

[Listing 8.4](#) 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 8.4 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 Simulator, 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



Freescale Semiconductor, Inc.

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 Simulator 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.

Real Time Kernel Awareness

The Simulator/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:

- [Real Time Kernel Awareness Introduction](#)
- [Task description language](#)
- [Example of application](#)
- [Inspecting data structures of the Kernel](#)
- [Register assignments for the RTK awareness](#)
- [OSEK Kernel Awareness](#)

Real Time Kernel Awareness Introduction

Often operating systems (Real Time Kernels) are used to coordinate the different tasks in more complex systems. This chapter describes how applications built of several tasks can be handled with the Simulator/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 Simulator/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 will be described in this chapter.

Inspecting the state of a task


Each multitasking operating system will store 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 will be more information in the task descriptor, depending on the specific implementation of the kernel.

The Simulator/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 an 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'. This file describes the algorithm on how to get all 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 will be 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  key while clicking 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](#)

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 | ErrorMessage | If.
Assignment       =  Ident ':=' Expression.
ErrorMessage     =  'MSG' ':=' String.
IfStatement      =  '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 dynamic link (data base) to be set (if not available, same as SP).

STATUS error number to be set (refer to manual).

Rnn processor registers to be set (mapping to CPU registers see manual).

MSG is error message (has to be specified if $N \geq 1000$).

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.



Example of application

[Listing 9.1](#) shows an example of "OSPARAM.PRM" file for SOOM System/REM.

Listing 9.1 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 = 0THEN MSG := "ReadyInCQSc"
ELSIF i = 1THEN MSG := "BlockedByAccept"
ELSIF i = 2THEN MSG := "WaitForDReply"
ELSIF i = 3THEN MSG := "WaitForMail"
ELSIF i = 4THEN MSG := "DelayQueue"
ELSIF i = 5THEN MSG := "BlockedByReceive"
ELSIF i = 6THEN MSG := "WaitForSemaphore"
ELSIF i = 7THEN MSG := "Dummy"
ELSIF i = 8THEN MSG := "SysBlocked"
ELSE MSG := "invalid"
END;
```

Inspecting data structures of the Kernel

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 9.2](#).

Listing 9.2 kernel data structures 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 9.3](#)).

Listing 9.3 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 in a way 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 9.4](#):



Real Time Kernel Awareness

Register assignments for the RTK awareness

Listing 9.4 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 Simulator/Debugger. There is no restriction in the complexity of the structures to describe the global data of the kernel.

NOTE We do not recommend opening the terminal window during testing. Errors detected during reading of a PRM file are written to this window.

Register assignments for the RTK awareness

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 (OSEK Run Time Interface) compliant real-time operating systems and offers dedicated

kernel awareness, by using the information stored in your application's ORTI file.

With the CodeWarrior OSEK kernel awareness, you can monitor kernel task information, semaphores, messages, queues, resources allocations, synchronization, communicating between tasks, etc.

ORTI is intended for the description of applications in any OSEK implementation. It describes a set of attributes for system objects and a method for interpreting the data obtained.

OSEK ORTI

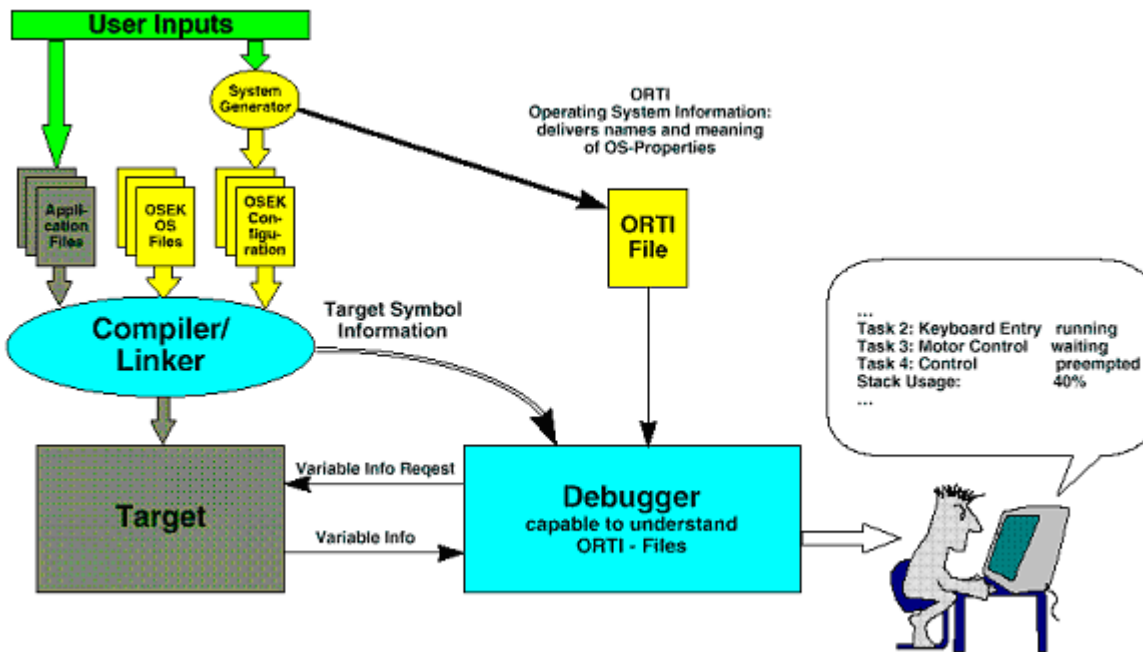
The OSEK Run Time Interface (ORTI) is intended as an interface for development tools to the OSEK Operating System. It is a part of the OSEK standard (refer to www.osek-vdx.org).

OSEK ORTI Definition

The OSEK ORTI intends to enable 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.

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 9.1 ORTI Aware debugging system



Two types of data shall be made available to the CodeWarrior debug tool. One type shall describe static configuration data that will remain unchanged during program execution. The second type of data shall be dynamic and this data will be re-evaluated each time by Code Warrior. 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.

An [OSEK ORTI File Sample](#) is described in Appendix.

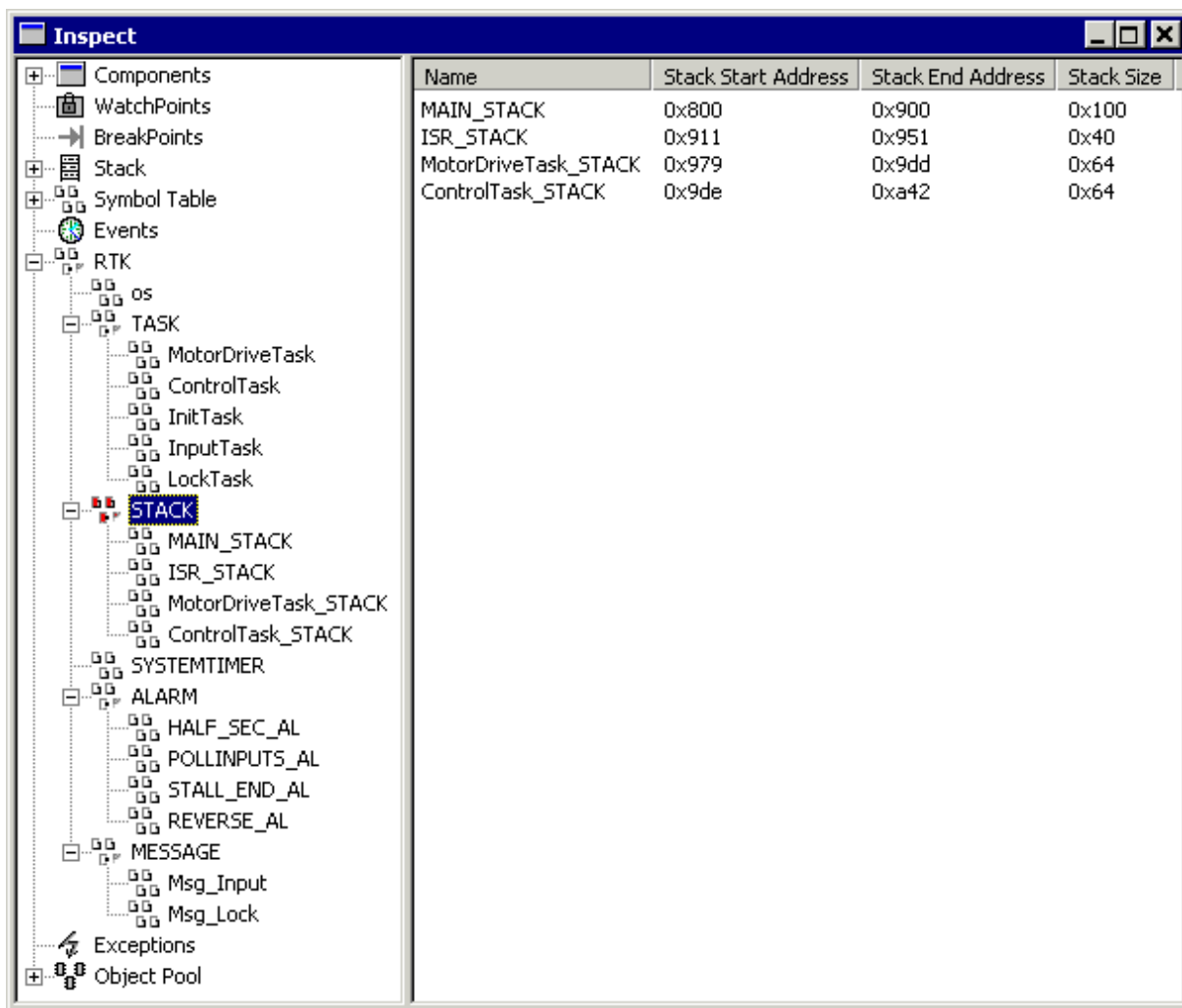
OSEK RTK Inspector component

OSEK awareness is described through the Code Warrior RTK Inspector component as show in [Figure 9.2](#).

Inspector window is displayed by clicking on **Component>Open...** menu entry and then by clicking on **Inspect** icon in the “Open Window Component” window.

When the RTK components icon is selected in the hierarchical content of the items, the right side displays various information about OSEK Awareness.

Figure 9.2 Code Warrior RTK Inspector



The OSEK RTK Inspector provides all these information. As defined in the ORTI file, objects of the same type are grouped and can be viewed together.

- Task
- Stack
- SystemTimer
- Alarm
- Message.

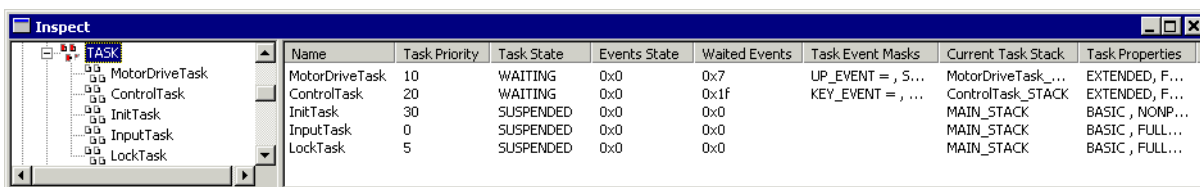
Below you can find a description of typical objects along with their attributes and how they are presented:

NOTE Be aware that objects and their attributes depend on the OSEK implementation and OSEK configuration, and therefore may differ from this description.

Task

The Task shown in [Figure 9.3](#) displays the current state of OSEK task trace.

Figure 9.3 Inspector Task



Name	Task Priority	Task State	Events State	Waited Events	Task Event Masks	Current Task Stack	Task Properties
MotorDriveTask	10	WAITING	0x0	0x7	UP_EVENT = , S...	MotorDriveTask_...	EXTENDED, F...
ControlTask	20	WAITING	0x0	0x1f	KEY_EVENT = , ...	ControlTask_STACK	EXTENDED, F...
InitTask	30	SUSPENDED	0x0	0x0		MAIN_STACK	BASIC , NONP...
InputTask	0	SUSPENDED	0x0	0x0		MAIN_STACK	BASIC , FULL...
LockTask	5	SUSPENDED	0x0	0x0		MAIN_STACK	BASIC , FULL...

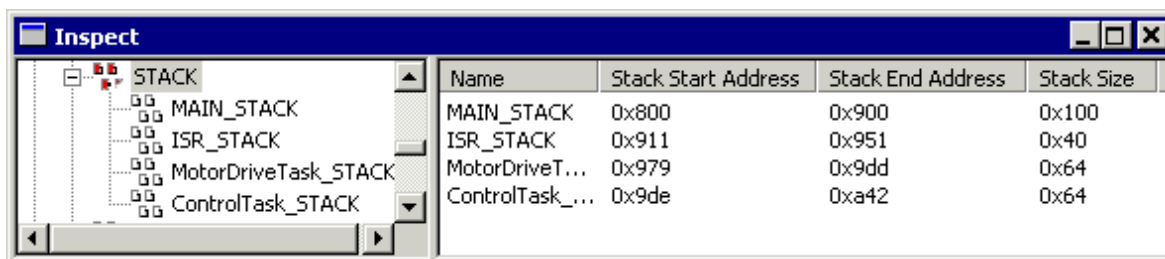
When selecting Task in the hierarchical tree on the left side, additional information concerning tasks is displayed on the right side:

- **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 States:** the event is represented by its mask. The event mask is the number which range is from 1 to 0xFFFFFFFF. When the event mask value is set to 1, the event is activated. When it is set to 0, the event is disabled.
- **Waited Events:** when the bit is set to 0, the event is not expected. When the bit is set to 1, the event is expected.
- **Task Event Mask:** describes the current task event mask.
- **Current Task Stack:** displays the name of the current stack used by the task.
- **Task Priorities:** describes task priorities. Possible value are BASIC/EXTENDED, NONPREMPT/FULLPREMPT, Priority value, AUTO. The ORTI file defines the possible values.

Stack

The Stack shown in [Figure 9.4](#) displays the current state of OSEK stack trace.

Figure 9.4 Inspector Stack



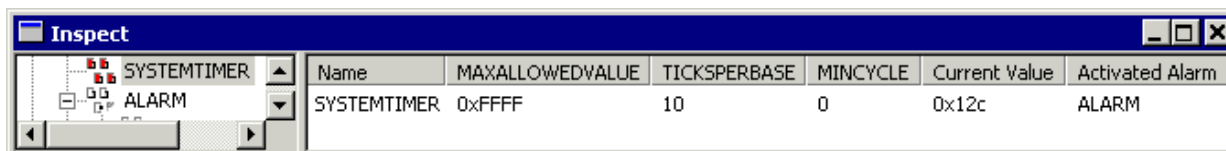
When selecting Stack in the hierarchical tree on the left side, additional information concerning task are displayed on the right side:

- **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.

SystemTimer

The SystemTimer shown in [Figure 9.5](#) displays the current state of OSEK SystemTimer trace.

Figure 9.5 Inspector SystemTimer



When selecting SystemTimer in the hierarchical tree on the left side, additional information concerning task are displayed on the right side:

- **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.

Alarm

The Alarm shown in [Figure 9.6](#) displays the current state of OSEK alarm trace.

Figure 9.6 Inspector Alarm

Name	Alarm State	Assigned Counter	Notified Task	Event to set	Time to expire	Cycle period
HALF_SEC_AL	ALARMSTOP	SYSTEMTIMER	ControlTask	HALF_SEC...	0xfed4	0x0
POLLINPUTS_AL	ALARMRUN	SYSTEMTIMER	InputTask		0x3	0x3
STALL_END_AL	ALARMSTOP	SYSTEMTIMER	ControlTask	STALL_EN...	0xfed4	0x0
REVERSE_AL	ALARMSTOP	SYSTEMTIMER	ControlTask	REVERSE_...	0xfed4	0x0

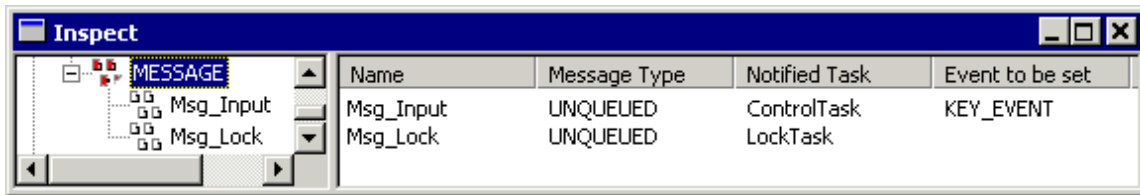
When selecting Alarm in the hierarchical tree on the left side, additional information concerning task are displayed on the right side:

- **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:** the 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. Notified Task defines the task to be notified (by activation or event setting) when the alarm expires.
- **Event to Set:** the 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.

Message

The Message shown in [Figure 9.7](#) displays the current state of OSEK message trace.

Figure 9.7 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.

Environment

This chapter describes the environment variables used by the Simulator/Debugger. Some of these environment variables are also used by other tools (for example, Linker), so also consult their respective manual.

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

- [Debugger environment](#)
- [Local Configuration File \(usually project.ini\)](#)
- [ABSPATH](#)
- [DEFAULTDIR](#)
- [ENVIRONMENT](#)
- [GENPATH](#)
- [LIBRARYPATH](#)
- [OBJPATH](#)
- [TMP](#)
- [USELIBPATH](#)
- [Searching order for sources files](#)
- [Files of the Simulator/Debugger](#)

Debugger environment

Various parameters of the Simulator/Debugger may be set in an environment using environment variables. The syntax is always the same:

Parameter = KeyName "=" ParamDef.

NOTE Normally no blanks are allowed in the definition of an environment variable.

Example



```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/lib;/home/me/  
my_project
```

These 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](#).

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 Simulator/Debugger tools, the current directory is the directory containing the local project file. Changing the current project file also changes the current directory, if the other project file is in a different directory. Note that browsing for a C file does not change the current directory.

To overwrite this behavior, the environment variable [DEFAULTDIR: Default Current Directory](#) may be used.

Global Initialization File (MCUTOOLS.INI) (PC only)

All tools may store global data in MCUTOOLS . INI. The tool first searches for this file in the directory of the tool itself (path of executable). If there is no MCUTOOLS.INI file in this directory, the tool looks for the file in the MS Windows installation directory (for example, C : \WINDOWS).

Example

```
C : \WINDOWS\MCTYPETOOLS . INI
D : \INSTALL\PROG\MCTYPETOOLS . 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\MCTYPETOOLS . INI).

If the tool is started outside the D : \INSTALL\PROG directory, the project file in the Windows directory is used (C : \WINDOWS\MCTYPETOOLS . INI).

Environment*Local Configuration File (usually project.ini)*

NOTE For more information about MCUTOOLS . INI entries, see the compiler manual.

Local Configuration File (usually project.ini)

The Simulator/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 Simulator/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 Simulator/Debugger. Apart from this, the Simulator/Debugger can use any file name for the project file. The configuration file has the same format as windows .ini files. The Simulator/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 `COMOPTIONS` 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 `COMOPTIONS`.

When a project is loaded using the first `default.env`, its `COMOPTIONS` 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 Simulator/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.

Configuration of the Default Layout for the Simulator/Debugger: the PROJECT.INI File

The default layout activated when starting the Simulator/Debugger is defined in the PROJECT.INI file located in the project directory, as shown in [Listing 10.1](#). All default layout related parameters are stored in section [DEFAULTS].

Listing 10.1 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 Simulator/Debugger (loads the file <target> with a .tgt extension), for example, Target=Sim for Simulator, or Target=Motosil, Target=Bdi.

Window<n>: Specifies coordinates of the windows that must be open when the Simulator/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**.



Environment

Local Configuration File (usually project.ini)

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.

Please see section [Window Menu](#) for more information about Layouts.

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:



Freescale Semiconductor, Inc.

Project=<ProjectName>

where **ProjectName** is the name of the file describing the project to be loaded,

for example, **Project=Proj1.hwc**

NOTE The project path can be specified if the project is not in the project directory. This option can be used for compatibility with the old .hwp format (Project=oldProject.hwp) and will be opened as a new project file.

See [File Menu](#) section for more details about Projects.

NOTE If **Layout** and **Project** are defined in PROJECT.INI, the **Project** parameter overwrites the **Layout** parameter, also containing layout information.

**MainFrame=<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>,
<nbr.>,<nbr.>,<nbr.>,<nbr.>**

This variable is used to save and load the Simulator/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:

Toolbar = (0 | 1)

If 1 is specified, the toolbar is shown, otherwise the toolbar is hidden.

The status bar can be shown or hidden with the following syntax:

Statusbar = (0 | 1)

If 1 is specified, the status bar is shown, otherwise the toolbar is hidden.

Title bars can be shown or hidden with the following syntax:



Environment

Local Configuration File (usually project.ini)

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):

1. The old Project file is closed.
2. Target Component is unloaded
3. 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]

4. The environment variables are loaded from the default.env file.
5. If an entry 'Layout=lll' exists, the layout file lll.hwl is loaded and executed.
6. The target is set (if entry 'Target=ttt' exists load target 'ttt').
7. If an entry 'Project=ppp' exists, the command file 'ppp' is executed.
8. The configuration file (*.hwc) is loaded (entry configuration=*.hwc).

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
```



Environment

Local Configuration File (usually project.ini)

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 Variable Details

The remainder of this section is devoted to describing each of the environment variables available for the Simulator/Debugger. The options are listed in alphabetical order and each is divided into several sections described in the [Environment Variable Details](#).

Table 10.1 Environment Variable Details

Topic	Description
Tools	Lists of other tools that are using this variable
Synonym	For some environment variables a synonym also exists. The synonyms may be used for older releases of the Simulator/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

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 ABS_PATH 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

See also

None

DEFAULTDIR

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](#) and [Global Initialization File \(MCUTOOLS.INI\) \(PC only\)](#)

ENVIRONMENT

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](#) 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=\Metrowerks\prog\global.env
```

See also

None:

GENPATH

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 Simulator/Debugger searches in the current directory, then in the directories given by GENPATH and finally in the directories given by [LIBRARYPATH](#).

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

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](#) and finally in directories given by [LIBRARYPATH](#).

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](#)

Environment variable [USELIBPATH](#)

OBJPATH

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](#) and finally in HIPATH.

Example

```
OBJPATH=\sources\obj
```



See also

None.

TMP

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 alsoSection '[The Current Directory](#)'

USELIBPATH

USELIBPATH: Using LIBPATH Environment Variable

Tools

Compiler, Linker, Debugger.

Synonym

None.

Syntax

```
"USELIBPATH=" ( "OFF" | "ON" | "NO" | "YES" )
```

Arguments

"ON", "YES": The environment variable [LIBRARYPATH](#) is used to look for system header files <*.h>.

"NO", "OFF": The environment variable [LIBRARYPATH](#) is not used.

Default

ON

Description

This environment variable allows a flexible usage of the [LIBRARYPATH](#) environment variable, because [LIBRARYPATH](#) may be used by other software (for example, version management PVCS).

Example

```
USELIBPATH=ON
```

See alsoEnvironment variable [LIBRARYPATH](#)

Searching order for sources files

This section describes the searching order (from first to last) used by the debugger.

Searching Order in the Simulator/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

Searching Order in the Simulator/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

Searching Order in the Simulator/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)



Files of the Simulator/Debugger

The Simulator/Debugger comes with several program, application, configuration files and examples. These files are listed in the following table.

Table 10.2 Simulator/Debugger and Metrowerks files extension.

Filename.	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	Simulator/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	Simulator/Debugger Default Environment file.



Filename.	Description.
* .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 Simulator/Debugger is provided for supported library and extended functions.
* .H	Header file
HIWAVE . EXE	The Simulator/Debugger for Windows executable program.
* .HWL	Simulator/Debugger Layout file, for example, default.hwl
* .HWC	Simulator/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
* .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



Filename.	Description.
* .MCP	Metrowerks 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	Simulator/Debugger Project Initialization File
* .REC	Recorder File
* .REG	Register Entries files, for example, mcu081e.reg
* .SIM	CPU simulator file, for example, st7.sim
* .SX	Motorola S-Record file, for example, fibo.sx
* .TXT	General Text Information file.
* :TGT	Target File for the Simulator/Debugger, for example, xtend-g3.tgt
* .WND	Simulator/Debugger Window Component File, for example,, recorder.wnd
* .XPR	Simulator/Debugger User Expression file, for example, Fibo.xpr

How To ...

This chapter provides answers to frequently asked questions.

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

- [How To Configure the Simulator/Debugger](#)
- [How To Start the Simulator/Debugger](#)
- [Automating startup of the Simulator/Debugger](#)
- [How To Load an Application](#)
- [How To Start an Application](#)
- [How To Stop an Application](#)
- [How To Step in the Application](#)
- [How To Work on Variables](#)
- [How To Work on Register](#)
- [How to Modify the content of a Memory Address](#)
- [How to Modify the content of a Memory Address](#)
- [How to Consult Assembler Instructions Generated by a Source Statement](#)
- [How To view Code](#)
- [How to Communicate with the Application](#)
- [About startup.cmd, reset.cmd, preload.cmd, postload.cmd](#)

How To Configure the Simulator/Debugger

If you have installed the Simulator/Debugger under Windows 95, 98, NT 4.0 and Windows2000 or higher, the Simulator/Debugger can be started from the desktop, from the Start menu, or external editor (WinEdit, CodeWright, etc.). In order to work efficiently (find all requested configuration and component files), the Simulator/Debugger must be associated with a working directory.

How To Configure the Simulator/Debugger for Use from Desktop on Win 95, Win 98, Win NT4.0 or Win2000

When starting the Simulator/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]**.

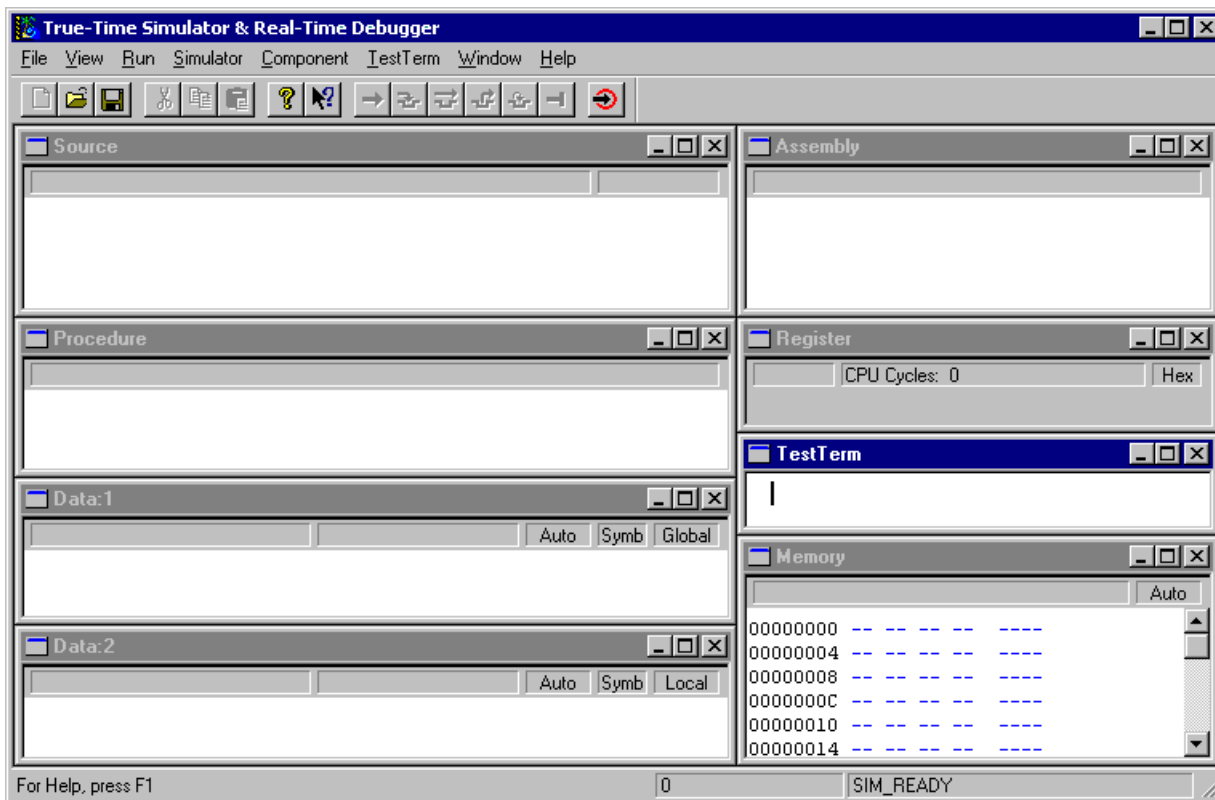
How To Start the Simulator/Debugger

This section describes various ways to start the Debugger.

How To Start the Simulator/Debugger from WinEdit

The Simulator/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 11.1](#).

Figure 11.1 Simulator/Debugger after startup



READY displayed in the status bar indicates that the simulator is ready.

Automating startup of the Simulator/Debugger

Often the same tasks have to be performed after starting the Simulator/Debugger. These tasks can be automated by writing a command file that contains all commands to be executed after startup of the Simulator/Debugger, as shown in [Listing 11.1](#).

Listing 11.1 Example of a command file to automate tasks

```
load fibo.abs
bs &main t
g
```




Freescale Semiconductor, Inc.

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 Simulator/Debugger (for example, Editor, Explorer, Make utility, ...).

Example:

```
\Metrowerks\PROG\HIWAVE.EXE -c init.cmd
```

When the Simulator/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 11.2](#)). The project file where the layout and target 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 [Configuration of the Default Layout for the Simulator/Debugger: the PROJECT.INI File](#)), commands in this file are executed.

Listing 11.2 Calling a command file from the project file:

```
set Sim
CLOSE *
call \Metrowerks\DEMO\test.hwl
call init.cmd
```

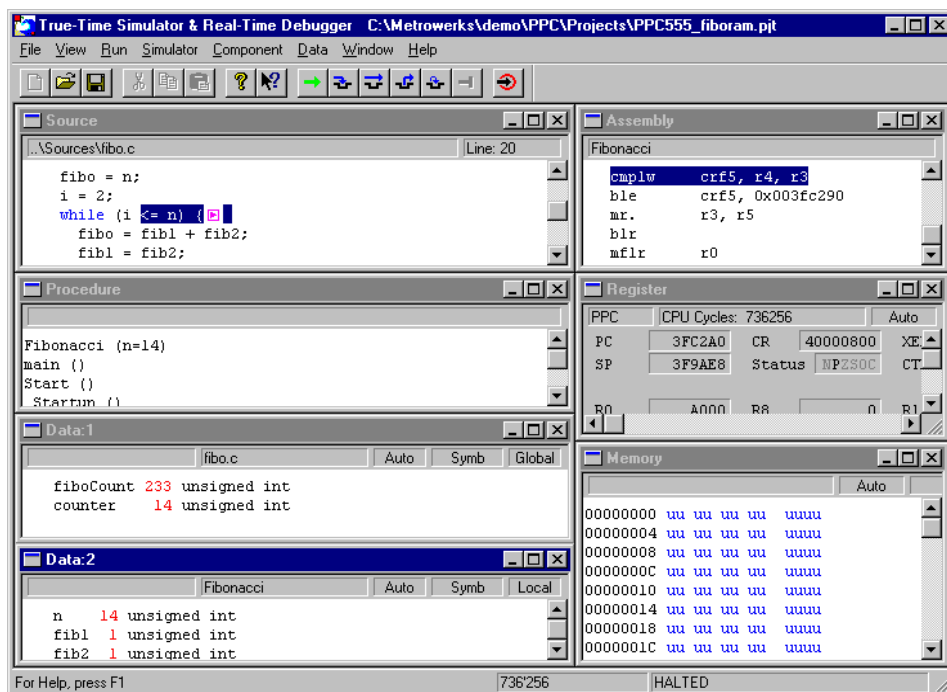
- Calling the command file when the Target Component is loaded. Most target components will execute the command file `STARTUP.CMD` once the target component is loaded and initialized. By adding the call command file in this file (for example, **CALL INIT.CMD**), it will automatically execute when the target component is loaded.

NOTE Refer to section [Starting the Debugger from a Command Line](#).

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 Simulator/Debugger ([Listing 11.2](#)).

Figure 11.2 Loading of 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](#)).
- 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.

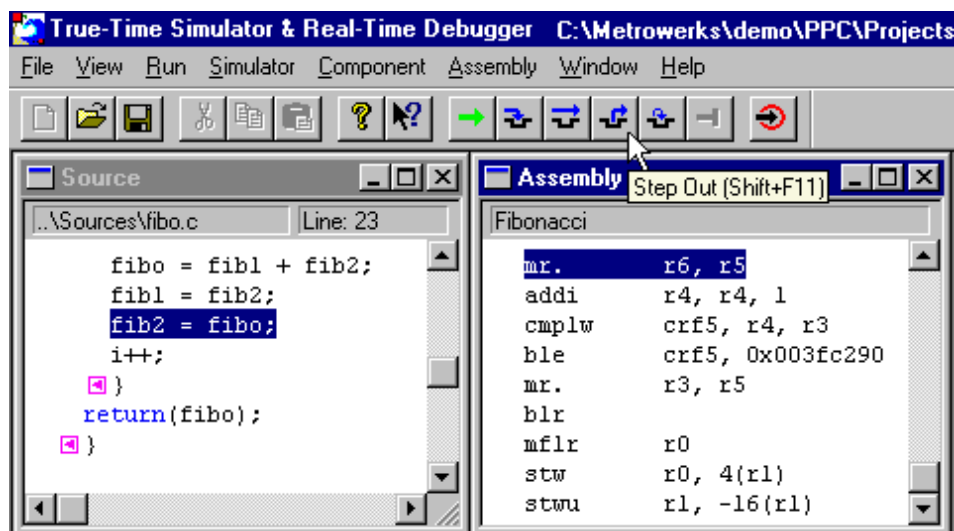
Data window with attribute **Local** displays the name and values of the local variables defined in the current procedure. The name of the procedure is specified in the Data info bar.

How To Step in the Application

The Simulator/Debugger provides stepping functions at the application source level and assembler level ([Listing 11.3](#)).


How to step on Source Level

Listing 11.3 Stepping on source level.



How to Step on the next source instruction

The Simulator/Debugger provides two ways of stepping to the next source instruction:

1. Choose **Run>Single Step**
2. Click the **Single Step** icon from the Simulator/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.

How to Step Over a Function Call (Flat Step)

The Simulator/Debugger provides two ways of stepping over a function call:

1. **Choose Run >Step Over**
2. **Click the Step Over icon from the Simulator/Debugger tool bar** 

STEPPED OVER (or STOPPED) 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.

How to Step Out from a Function Call

The Simulator/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 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.

How to Step on Assembly Level

The Simulator/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.

How to Display Local Variable from a Function

The Simulator/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.

How to Display Global Variable from a Module

The Simulator/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.

How to Change the Format for the Display of Variable Value

The Simulator/Debugger allows you to see the value of variables in different formats. This is set by entries in **Format** menu ([Table 11.1](#)).

Table 11.1 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.

- Values for pointer variables are displayed in hexadecimal format.**
- Values for function pointer variables are displayed as function name.**
- Values for character variables are displayed in ASCII character and decimal format.**
- 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:

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

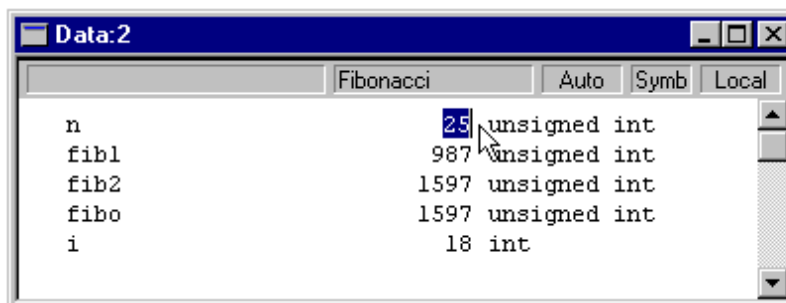
The format selected is valid for the whole Data component. Values from all variables in the data component are displayed according to the selected format.

How to Modify a Variable Value

The Simulator/Debugger allows you to change the value of a variable, as shown in [Figure 11.3](#).


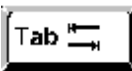
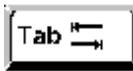

Modify a Variable Value

Figure 11.3 Modifying a Variable Value



The Simulator/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  or .**
3. **If an input value has been validated by  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  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.

How to Get the Address Where a Variable is Allocated

The Simulator/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.

How to Inspect Memory starting at a Variable Location Address



The Simulator/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.**

- Using  + 

1. **Point to a variable name in a Data Component.**

2. **Press the left mouse button and  + .**

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.

How to Load an Address Register with the Address of a variable

The Simulator/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 Register

This section describes how to work with the Register.

How to Change the Format of the Register display

The Simulator/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.

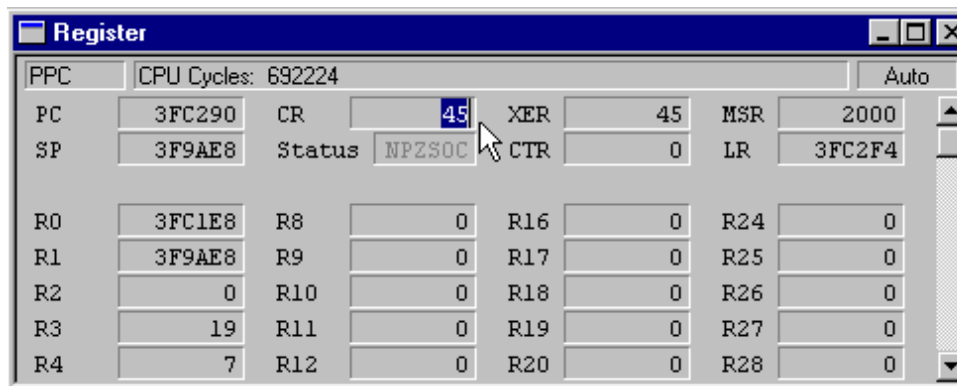
How to Modify a Register Content


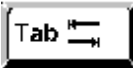
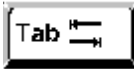

The Simulator/Debugger allows you to change the content of indexes, accumulators or bit registers.

How to Modify an Index or Accumulator Register Content

Double-click a register. The current register content is highlighted and may be edited.

Figure 11.4 Modifying an 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  or , or select another register.
3. If an input value has been validated by , 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 .

How to Modify a 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.

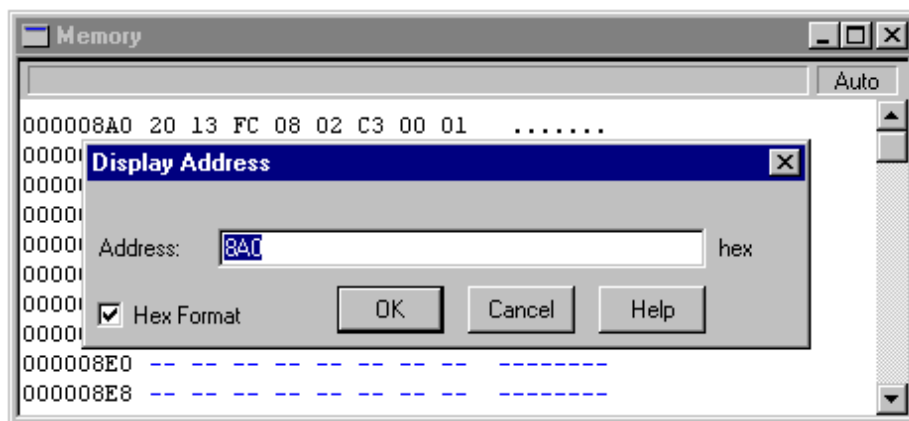
Single bits inside the bit register can be toggled by double-clicking the corresponding mnemonic character.

How to Get a Memory Dump starting at the Address where a Register is pointing

The Simulator/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 11.5 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 11.5](#) 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.


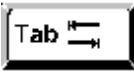
NOTE If “Hex Format” is checked, numbers and letters are considered to be hexadecimal numbers. Otherwise, expressions can be typed and Hex numbers should be prefixed with “0x” or “\$”. Refer to [Constant Standard Notation](#) section.

How to Modify the content of a Memory Address

The Simulator/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**

 or , or select another variable.

4. **To stop editing and restore the previous memory value, press .**

How to Consult Assembler Instructions Generated by a Source Statement

The Simulator/Debugger provides an on-line disassembly facility, which allows you to disassemble the hexadecimal code directly from the Simulator/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.**

• Using  + 

1. **In the Source component, point to the instruction you want to disassemble.**

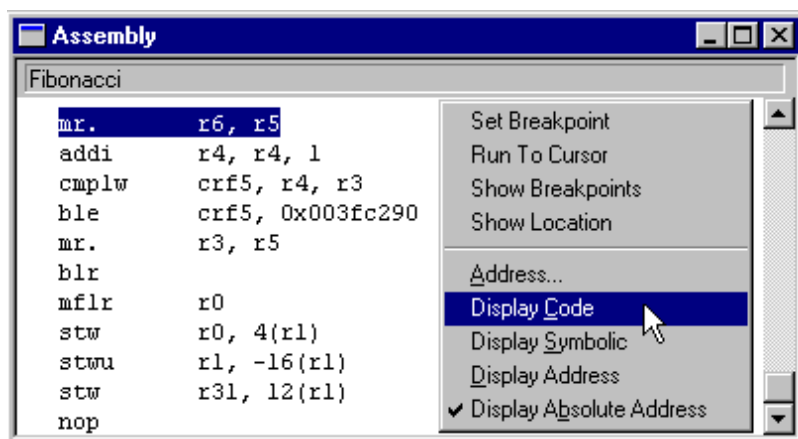


The disassembled code associated with the selected source instruction is greyed in the Assembly component.

How To view Code

The Simulator/Debugger allows you to view the code associated with each assembler instruction.

Figure 11.6 Viewing code associated with an 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 11.6).**
 - 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 Simulator/Debugger has a pseudo-terminal facility. Use the 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 Simulator/Debugger and choose Open... from the Component menu.**
2. **Open the Component.**
3. **Choose Load... from the Simulator menu.**
4. **Load the program `CALC.ABS`.**

Data entered in the 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.

Refer to sections [TestTerm Component](#) and [Terminal Component](#) for more information.

About `startup.cmd`, `reset.cmd`, `preload.cmd`, `postload.cmd`

The command files `startup.cmd`, `reset.cmd`, `preload.cmd`, and `postload.cmd` are Simulator/Debugger system command files. All these command files do not exist automatically. They could be installed when installing a new target.

However, the Simulator/Debugger is able to recognize these command files and execute them.

- `startup.cmd` is executed when a target is loaded (the target defined in the `project.ini` file or loaded when you select **Component>Set Target**).
- `reset.cmd` is executed when you select **“Target Name”>Reset** in the menu (**Target Name** is the real name of the target, such as `MMDS0508`, `SDI`, etc.).
- `preload.cmd` is executed before loading a `.ABS` application file or `Srecords` file (when you select **“Target Name”>Load...** in the menu).



Freescale Semiconductor, Inc.

- `postload.cmd` is executed after loading a `.ABS` application file or `Srecords` file (when you select “**Target Name**”>**Load...** in the menu).

Depending on the target used, other command files can be recognized by the Simulator/Debugger. Refer to the appropriate target manual for information and properties of these command files.

CodeWarrior Integration

This chapter provides information on how to use and configure the Simulator/Debugger within CodeWarrior.

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

- [Requirements](#)
- [Debugger Configuration](#)

Requirements

CodeWarrior IDE - version 4.1 or later

Debugger V6.1 or later

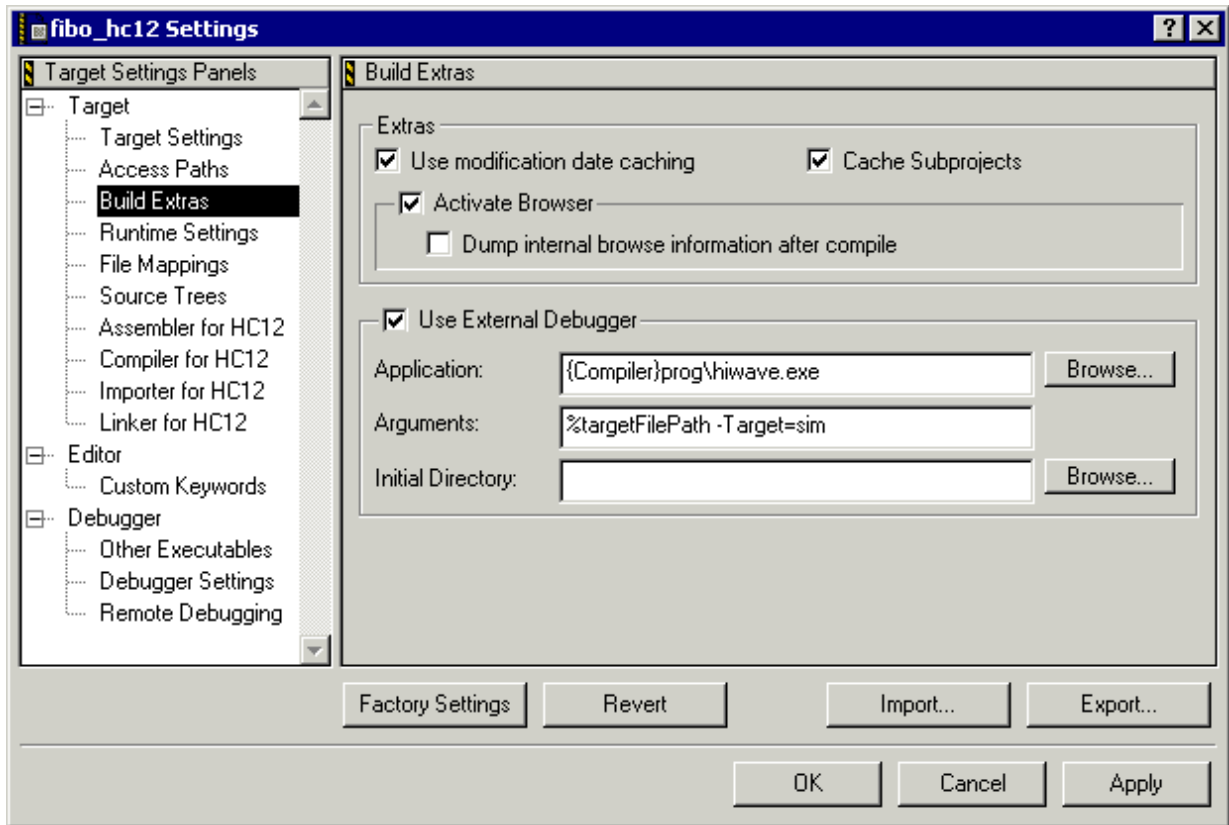
NOTE This chapter provides information on how to use and configure the Simulator/Debugger within the CodeWarrior IDE, for more information, refer to the CodeWarrior documentation.

Debugger Configuration

To configure the Real Time Debugger and True Time Simulator, in the CodeWarrior IDE open the **Target Settings Panel** and select **Build Extras** ([Figure 12.1](#)).

In the **Build Extras** pane check the **Use External Debugger** checkbox. In the Application field, type the Debugger path, for example, **{Compiler}proghiwave.exe** and arguments, for example, **%targetFilePath -Target=sim** in the Argument field. Click on **Apply** to validate these changes.

Figure 12.1 IDE Build Extras Panel



Debugger DDE capabilities

This chapter provides information on debugger capabilities and how to use and configure the Simulator/Debugger within CodeWarrior.

NOTE **The DDE capabilities of the Debugger are deprecated.** Future versions of the Debugger will have no DDE capabilities. Its recommended to use the Component Object Model (COM) Interface. See the chapter [Scripting](#) for more information about this.

Click the following link to jump to the corresponding section of this chapter:

- [Debugger DDE Server](#)

Debugger DDE Server

DDE 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.

Debugger DDE implementation

The Simulator/Debugger integrates a DDE server and DDE client implementation in the KERNEL.

The DDE application name of the IDF server is "HI-WAVE".

The Simulator/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 the Simulator/Debugger through DDE

The DDE implementation in the Debugger allows you to drive it easily by using the DDE command.

For 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 Simulator/Debugger DDE Server is **"HI-WAVE"** and the Topic name for the Simulator/Debugger DDE Server is **"Command"**.

The following example is done with DDECLient.exe from Microsoft.

1. **Run the Simulator/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 Simulator/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 Simulator/Debugger.**

NOTE You can disconnect the DDE in the Simulator/Debugger. The Simulator/Debugger can be started without DDE (this is saved in the project file). To view the current state, open a command line component and type the following command: "DDEPROTOCOL STATUS". The state must be: "DDEPROTOCOL ON" to ensure the DDE works properly.

Synchronized debugging through DA-C IDE

This chapter provides information on how to use and configure Metrowerks tools within DA-C IDE.

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

- [Requirements](#)
- [Configuring DA-C IDE for Metrowerks Tool Kit](#)
- [Debugger Interface](#)
- [Synchronized debugging](#)
- [Troubleshooting](#)

Requirements

DA-C - version 3.5 build 555 or later - (Development Assistant for C - RistanCASE).

Simulator/Debugger V6.0 or later.

NOTE This chapter provides information on how to use and configure Metrowerks tools within DA-C IDE. For more information on DA-C, refer to the "Development Assistant for C" documentation v 3.5.

Configuring DA-C IDE for Metrowerks Tool Kit

Install the DA-C software. The Metrowerks 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 Metrowerks Tools within DA-C IDE.

- Create a new project
- Configure the working directories
- Configure the file types
- Configuration of the Metrowerks library path
- Adding files to project
- Building the Database
- Configure the tools

In the following sections, we assume that the Metrowerks tool kit is installed in "C:\Metrowerks" 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 Metrowerks.

Creating a 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

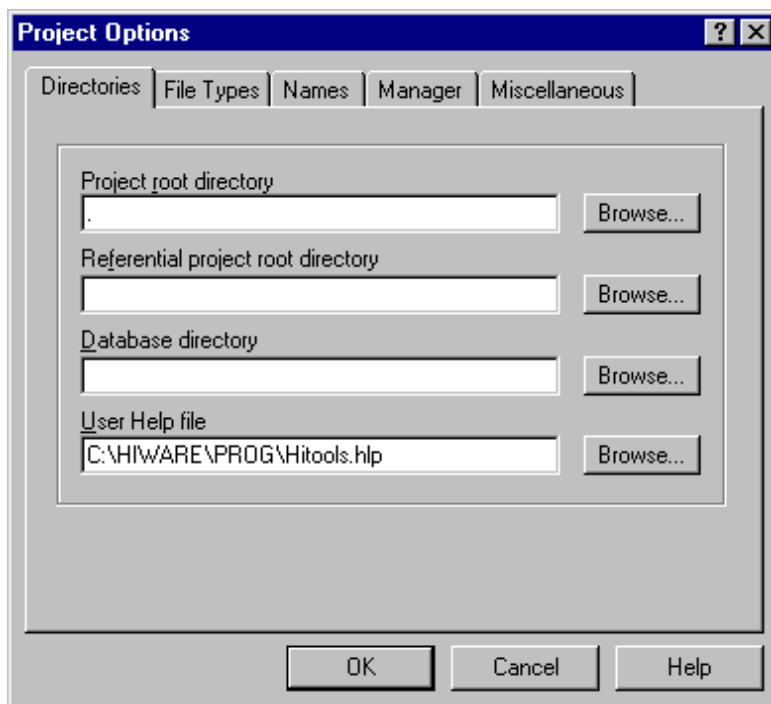
```
"C:\Metrowerks\work\<<processor>c\myproject "
```

and change the <processor> field to your CPU). A specific project file is created with ".dcp" extension (for example "myproject.dcp").

Configure the working directories

Choose **Options>Project** from the main menu of DA-C. The dialog box shown in [Figure 14.1](#) contains options, which establish directories for the project.

Figure 14.1 DA-C Project Options dialog



- Project root directory

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, specifies 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

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.

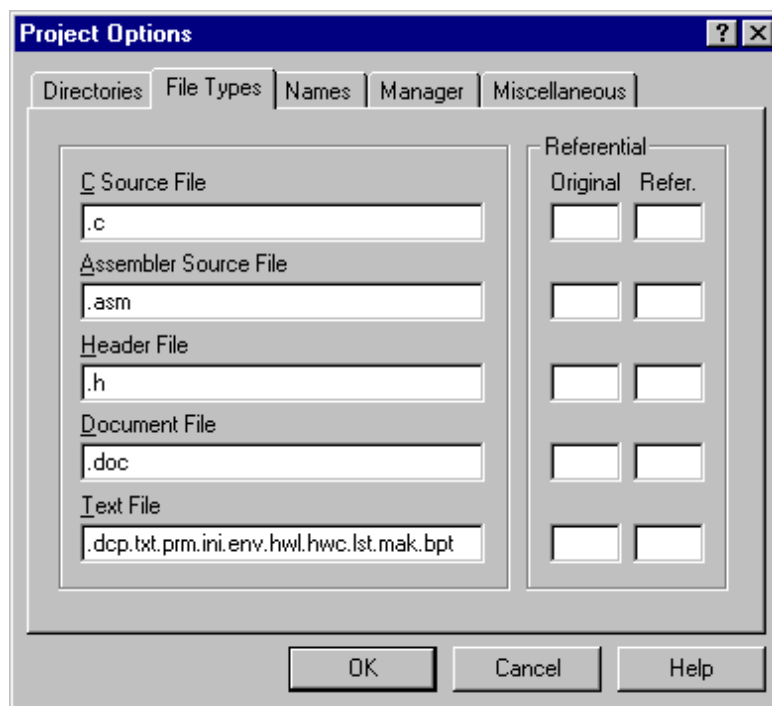
- User help file

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 Metrowerks installation and select the help file matching your CPU.

- Configure the file types

In the previous menu choose "File Types" to configure the basic file types. This dialog box contains options, which determine file types of the project. For an efficient use of Metrowerks tools, [Figure 14.2](#) shows file extension types that can be defined.

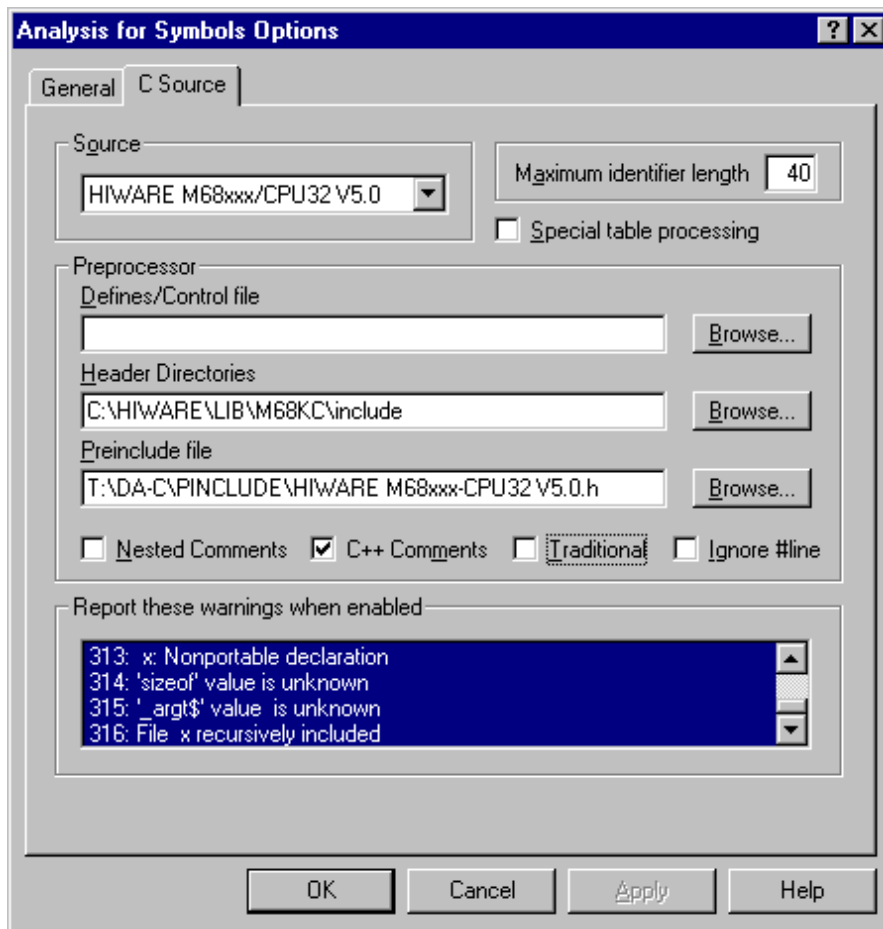
Figure 14.2 Definition of file types extension



Configuration of the 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 dialog box shown in [Figure 14.3](#) contains options that determine parameters of the C source code analysis.

Figure 14.3 Analysis for Symbols Options dialog



- Source

The supported C dialects of the C language used in the current project can be selected in this field. In our example we chose the Metrowerks M68k language (adapt it to your needs).

- Preprocessor | Header Directories

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



Freescale Semiconductor, Inc.

root) with prefix @. Directories are separated with a semi-colon or new line.

Define the library path matching your CPU (assuming Metrowerks tools are installed on "C:\Metrowerks"):

```
C:\Metrowerks\lib\

---


```

- Preprocessor | Preinclude file

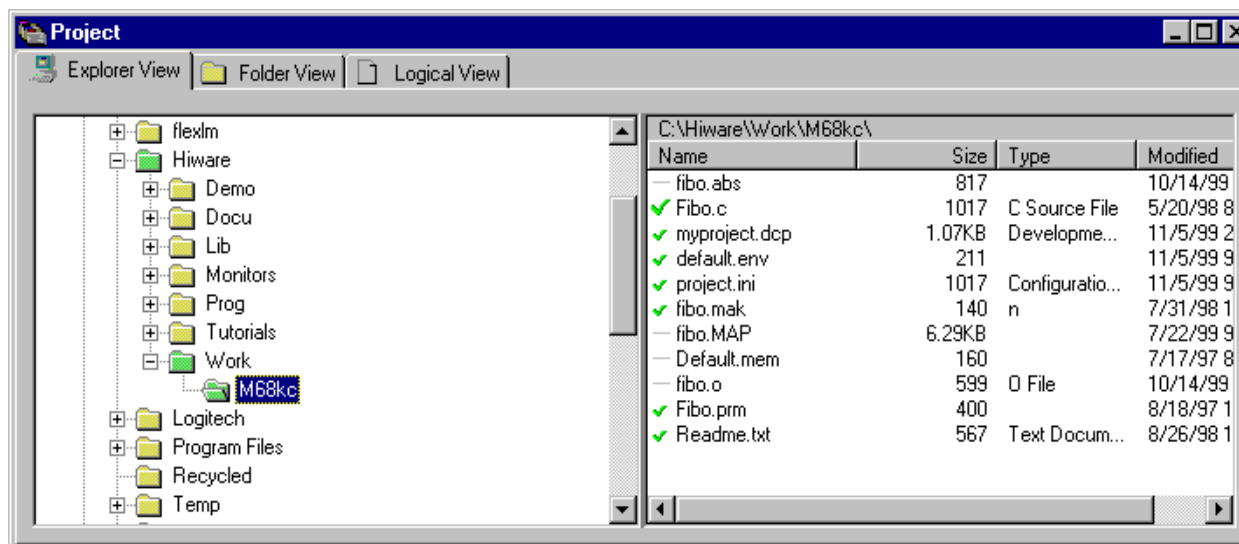
Determines the name of the file that will be included automatically at the beginning of every source module during analysis, in the same way as if #include "string" were present in the first line. The preinclude file can be used to specify predefined macros and variable and function declarations for 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 Manager's window the Explorer View 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 "fibonacci" example.

In the Explorer View, browse to the ">Metrowerks>WORK><processor>c" directory of your Metrowerks installation and select "fibonacci.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 14.4](#)).

Figure 14.4 Adding files to project with the Explorer View



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 in Project Manager.
- 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 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

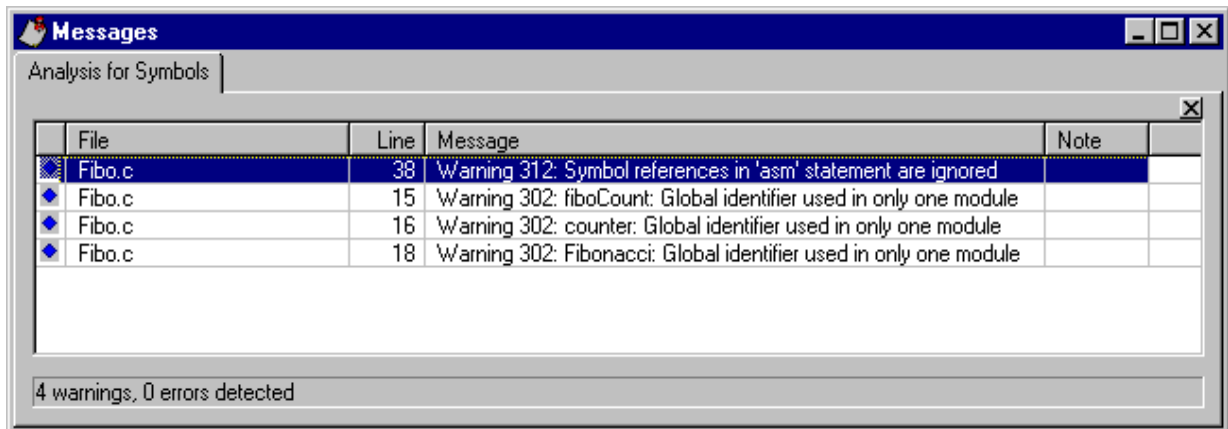
Development Assistant for C provides the static code analysis of C source files, as well as generating various data based on the results.

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 14.5](#) (for `Fibo.c` sample file):

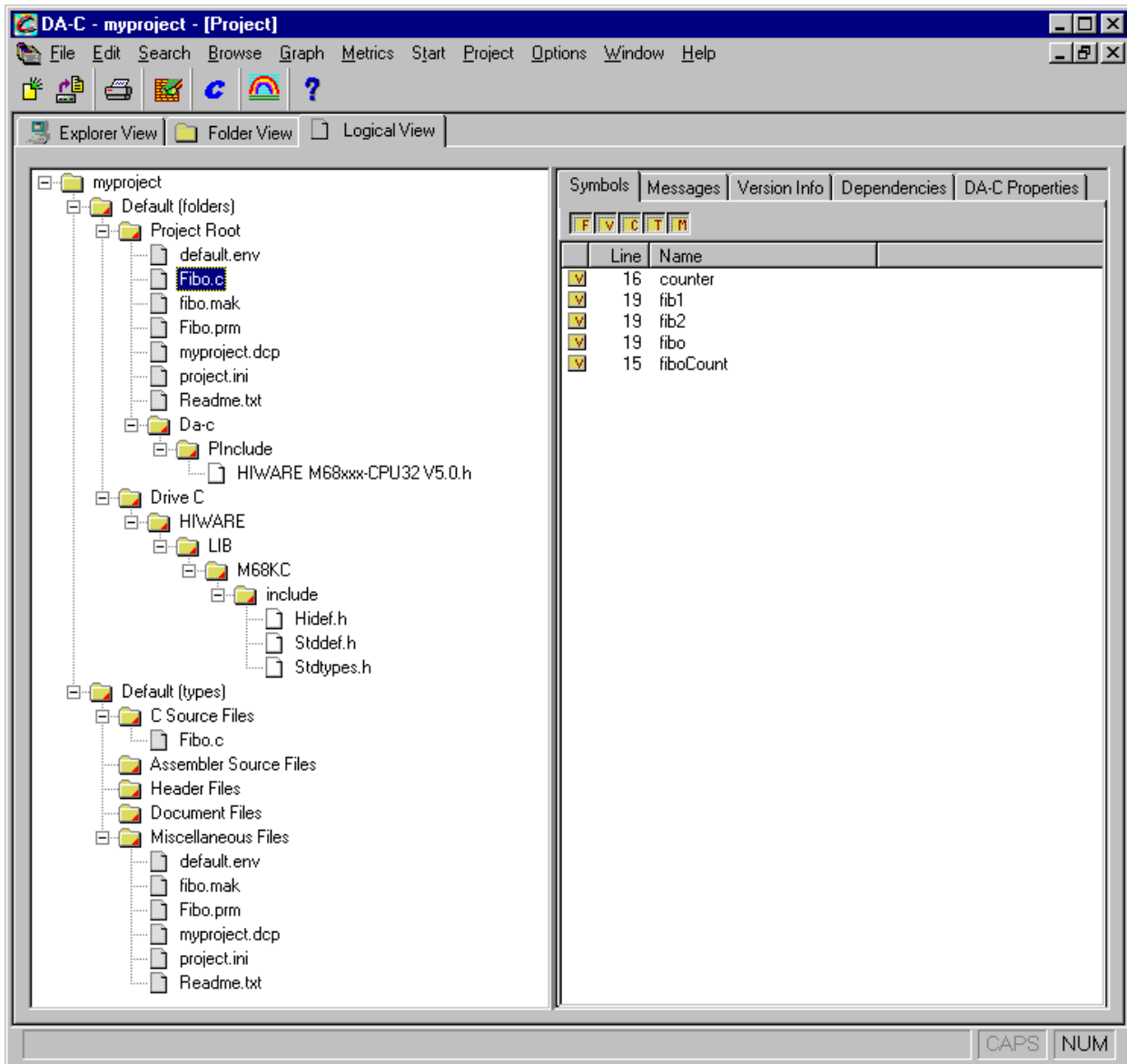
Figure 14.5 DA-C Message Window



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** property page shown in [Figure 14.6](#) and unfold all fields, you will now have the overview of your project.

Figure 14.6 Logical View



Double-click on "Fib0.c" file to open it.

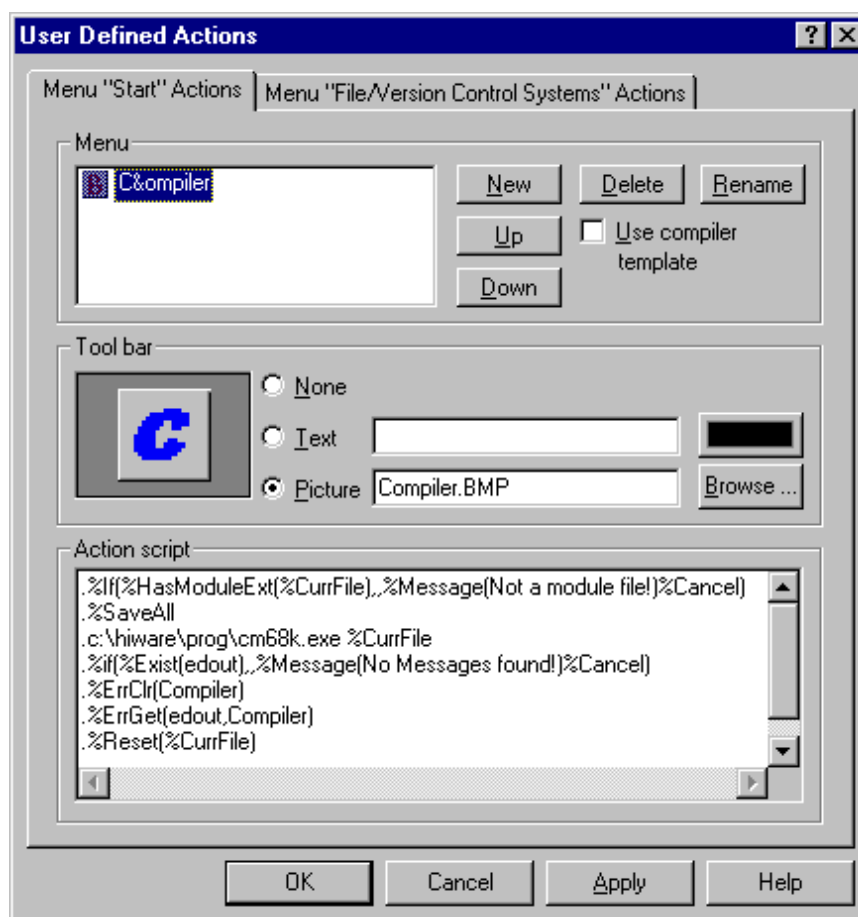
Configuring the tools

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

In **Menu "Start" Actions**, click on **new** and fill in the **New Action** box with "C&ompile", then press ENTER ([Figure 14.7](#)). In the **Toolbar** field, you can associate a bitmap with each tool, for example click on the **Picture** radio button and browse to the "\Bitmap" directory of your current DA-C installation and choose Compiler.bmp. This is a default bitmap delivered with DA-C IDE (here you are able to add your own bitmap).

Figure 14.7 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 14.1](#) in the Action Script field and change the directory to where the compiler is located.



Listing 14.1 Script for compiler action association

```
.%If(%HasModuleExt(%CurrFile),,%Message(Not a module  
file!)%Cancel)
```

```
.%SaveAll  
.c:\Metrowerks\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.

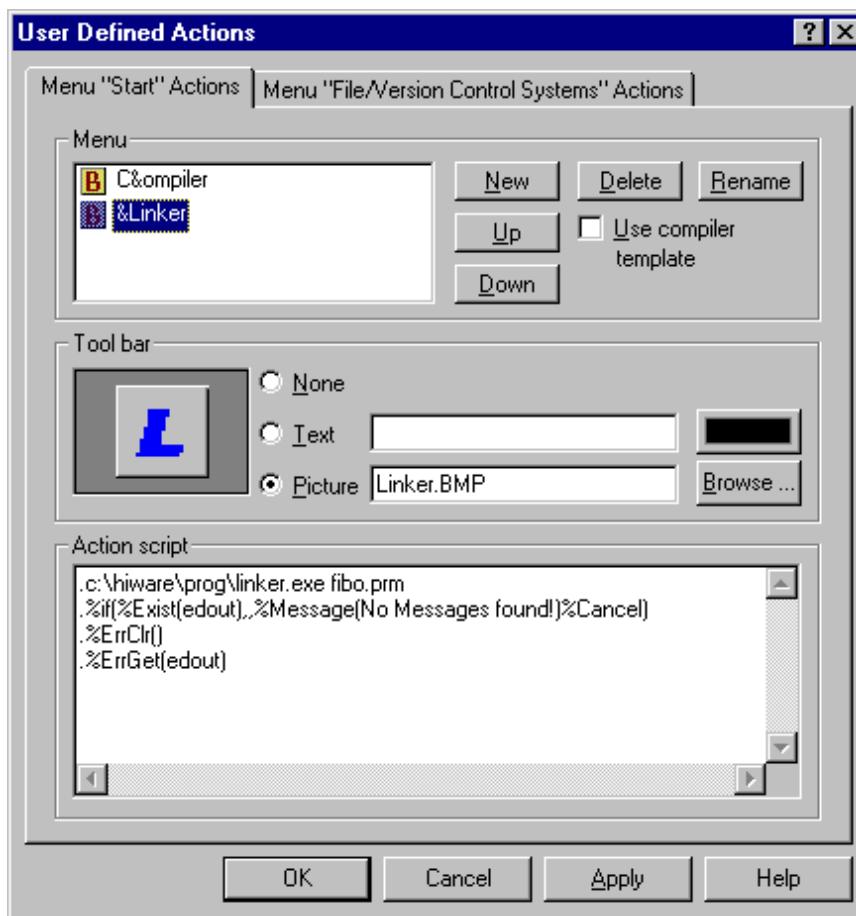
Linker

In the same way, you can now configure the linker as illustrated in [Figure 14.8](#). In the **Menu "Start" Actions**, click on new and fill in the created **New Action** box with "&Link", then validate with ENTER. After setting the corresponding bitmap, copy the following lines shown in [Listing 14.2](#) in the **Action Script** field and change the directory to where the linker is located.

Listing 14.2 Script for Linker action association

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

Figure 14.8 DA-C Linker Settings



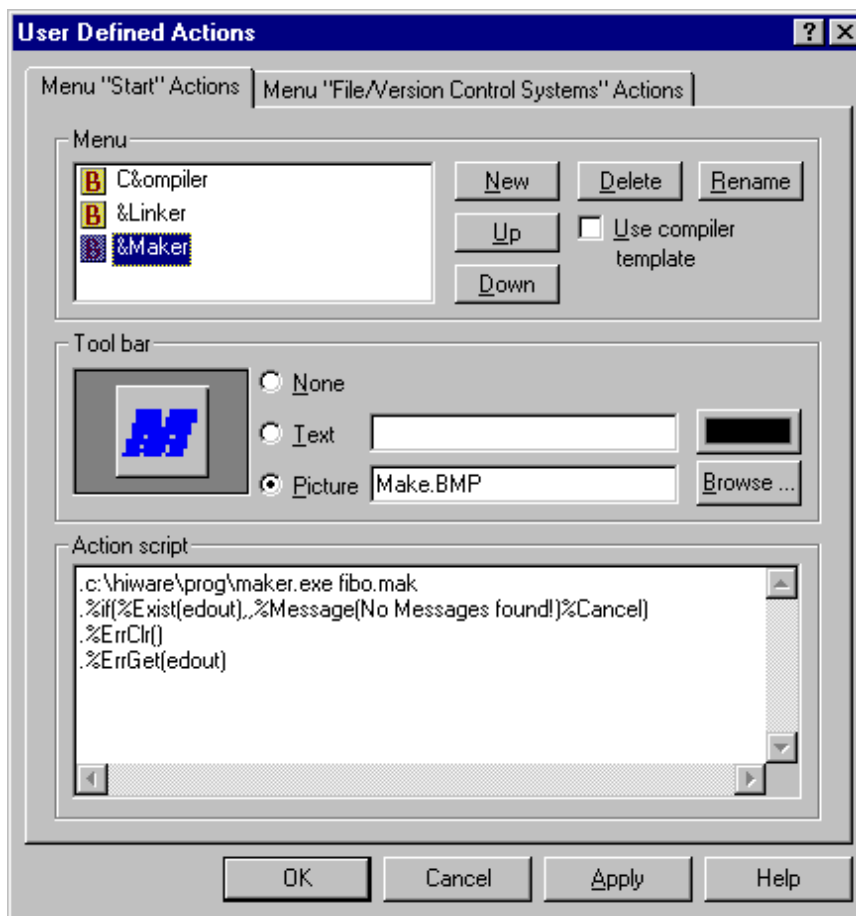
Maker

In the same way, you can now configure the maker as illustrated in [Figure 14.9](#). In the **Menu "Start" Actions**, click on new and fill in the created **New Action** box with "**&Make**", then press ENTER. After setting the corresponding bitmap, copy the lines from [Listing 14.3](#) in the **Action Script** field and change the directory to where the maker is located.

Listing 14.3 Script for Maker action association

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

Figure 14.9 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 Simulator/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 Simulator/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 Simulator/Debugger is increased. Some configurations are required in order to make an efficient use of this Debugger Interface:

- Installation of communication DLL



Freescale Semiconductor, Inc.

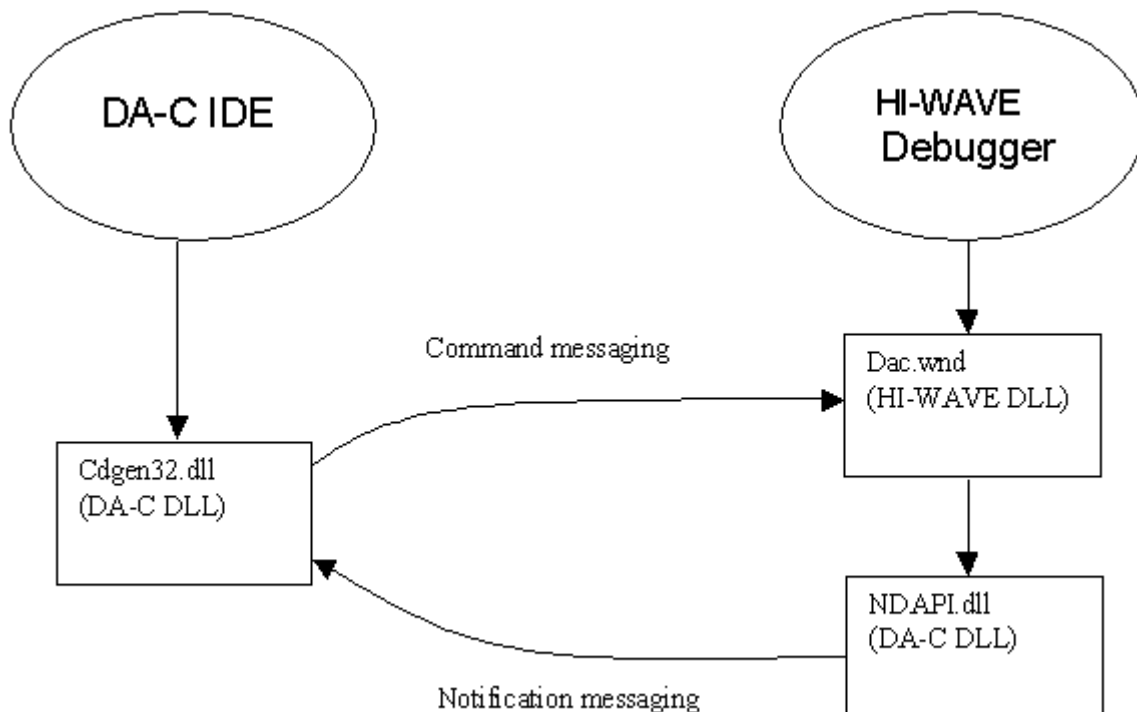
- Configuration of Debugger properties
- Configuration of the Simulator/Debugger project file

Principle of Communication between DA-C IDE and Simulator/Debugger

DA-C and the Simulator/Debugger are both Microsoft Windows applications and communication is based on the DDE protocol ([Figure 14.10](#)). The whole system contains:

- DA-C
- Simulator/Debugger
- cDAPI interface implementation DLL - which is used by DA-C (Cdgen32.dll)
- nDAPI communication DLL (provided by DA-C), which is used by Simulator/Debugger
- Simulator/Debugger specific DLL for bridging its interface to debugging environment and DA-C's nDAPI (DAC.wnd)

Figure 14.10 Principle of Communication between DA-C IDE and Simulator/Debugger



Installation of communication DLL

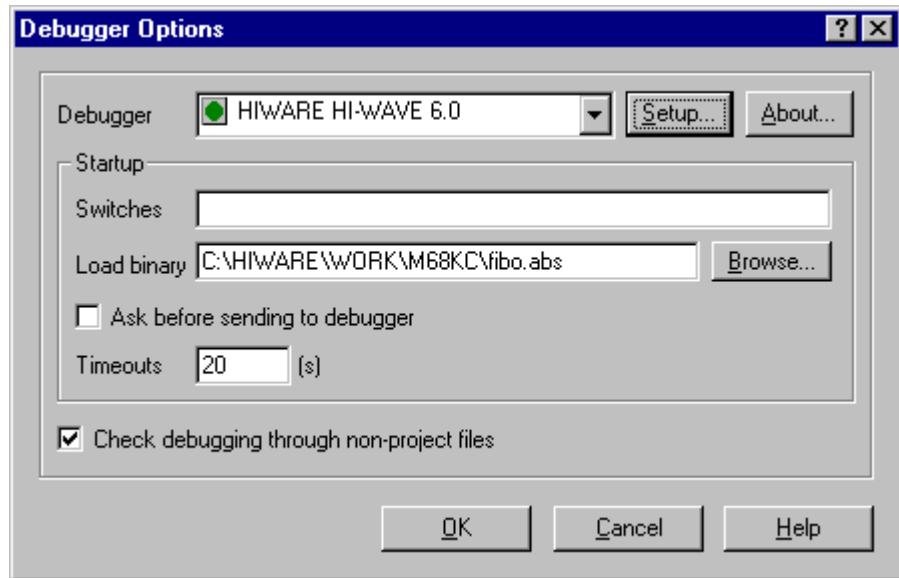
As described previously, the Simulator/Debugger needs the nDAPI communication DLL (provided by DA-C IDE). This dll (called Ndapi.dll) is automatically installed during the Metrowerks Tool Kit installation. However, if you install a new release of DA-C you have to follow this procedure:

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 "Metrowerks\PROG" directory (where Simulator/Debugger is located). Then rename it to "Ndapi.dll".

Configuration of Debugger properties

In the DA-C main menu, choose **Options>Debugger**, the dialog shown in [Figure 14.11](#) is opened.

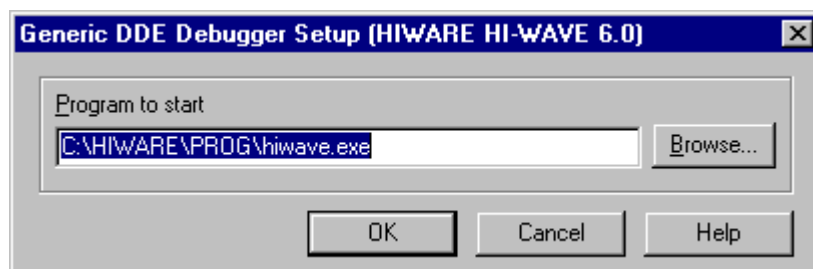
Figure 14.11 DA-C Debugger Options



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 "fibonacci.abs" file.

Then click on the **Setup...** button. The dialog shown in [Figure 14.12](#) is opened.

Figure 14.12 DDE Debugger Setup



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

Configuration of the Simulator/Debugger project file

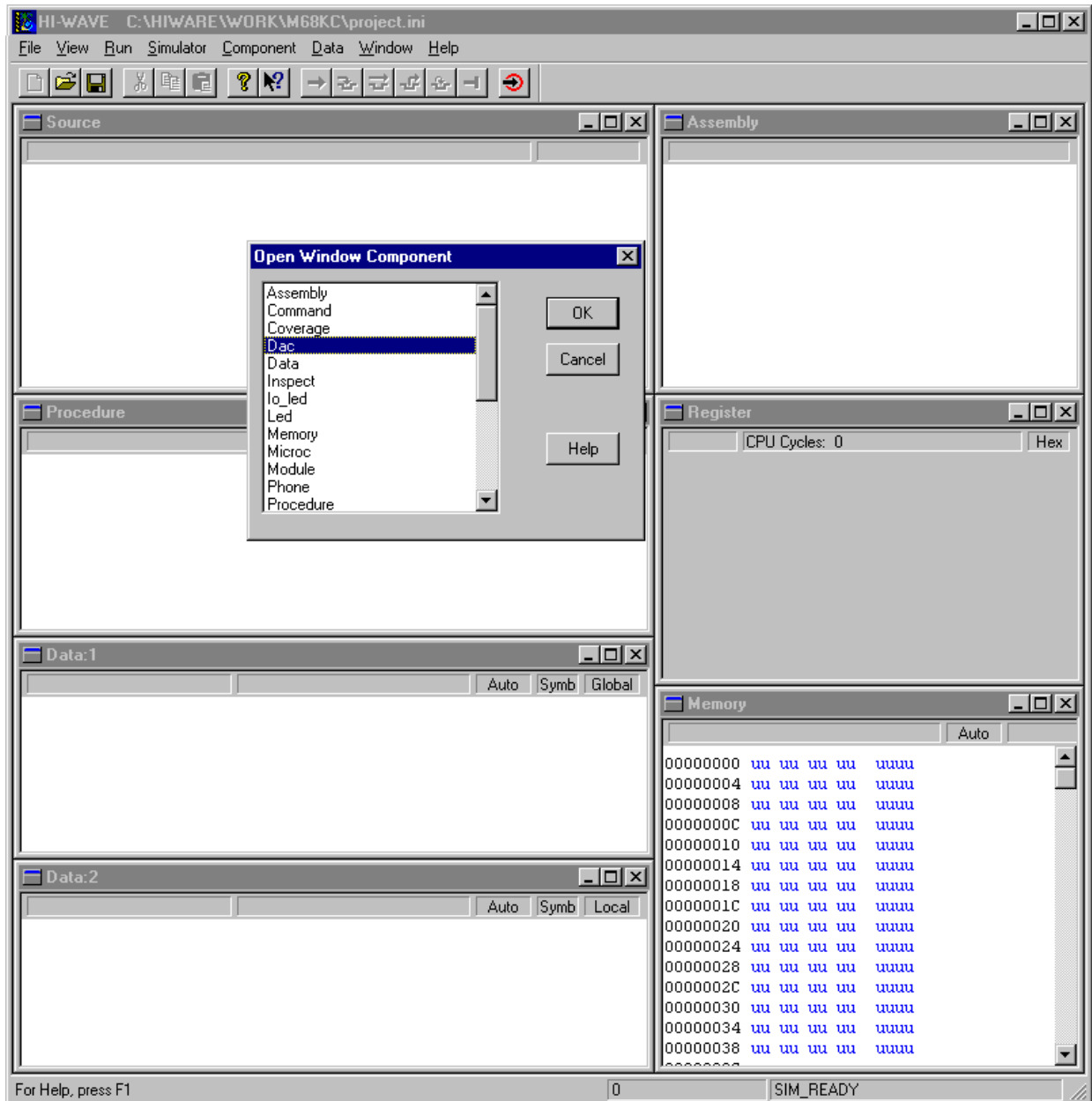
Before configuring the project file, close DA-C. Open Simulator/Debugger (for example, from a shell) and select **File>Open Project...** from the main



Freescale Semiconductor, Inc.

menu bar. Select the "Project.ini" file from the currently defined working directory (in our case "C:\Metrowerks\WORK\Component >Open from the main menu bar and choose "**Dac**", as shown in [Figure 14.13](#).

Figure 14.13 DA-C component opening



The Simulator/Debugger DAC window, which is needed for communication with DA-C IDE is now opened ([Figure 14.14](#)).

Figure 14.14 DA-C Window



You have to save this configuration by selecting **File>Save Project** from the main menu of the Simulator/Debugger. This component will be automatically loaded the next time this project is called. Close the Simulator/Debugger.

Synchronized debugging

We can now test the synchronization between DA-C IDE and Simulator/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 Simulator/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 14.15](#) or from the Simulator/Debugger.

Figure 14.15 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 Simulator/Debugger and DA-C source positions.

Troubleshooting

This section describes possible trouble when trying to connect the Simulator/Debugger with the DA-C IDE.

1. **When loading DAC component into the Simulator/Debugger, if the message box shown in [Figure 14.16](#) is displayed:**

Figure 14.16 DA-C component loading error.



check if the `Ndapi.dll` is located in the "`\prog`" directory of your current Metrowerks installation. If not, copy the specified DLL into this directory.

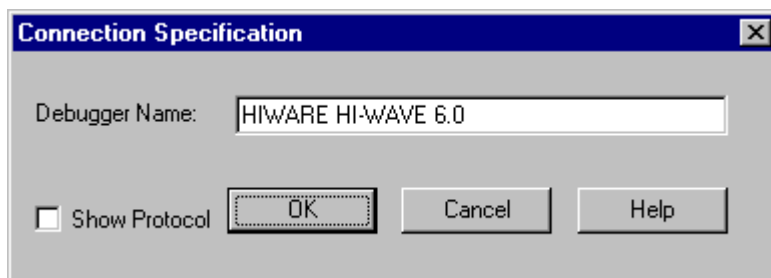
2. If the message box shown in [Figure 14.17](#) is displayed in DA-C IDE:

Figure 14.17 DA-C debugger support.



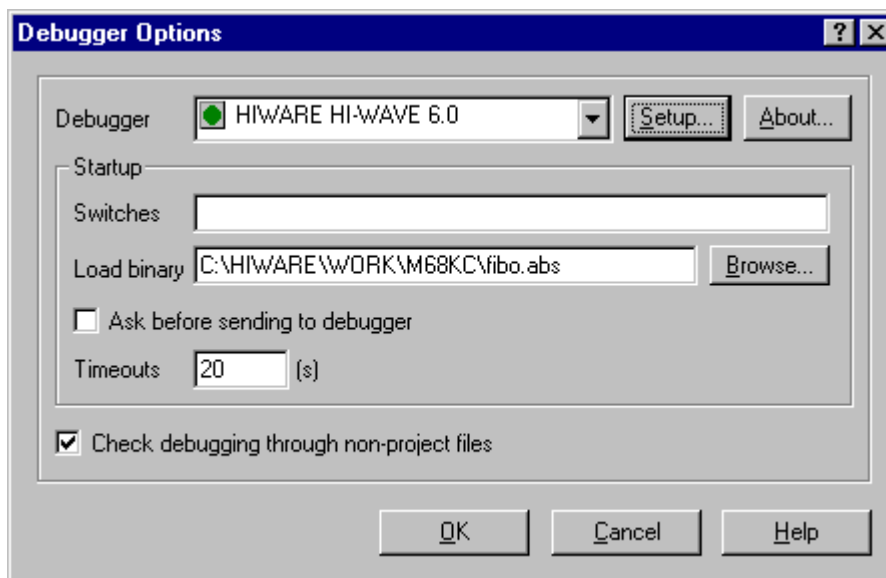
This means that the current name specified in the **Options>Debugger** main menu of DA-C doesn't match the debugger name specified in the Simulator/Debugger. Open the setup dialog in the Simulator/Debugger by clicking on the DA-C Link component and choose **DA-C Link>Setup...** from the main menu. The "Connection Specification" dialog is opened ([Figure 14.18](#)).

Figure 14.18 DA-C connection specification



Compare the "**Debugger Name**" from this dialog with the selected Debugger in DA-C IDE (**Options>Debugger**), shown in [Figure 14.19](#).

Figure 14.19 DA-C Debugger Options



Both must be the same. If it's not the case, change it in the Simulator/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 14.4](#).



Listing 14.4 DA-C section in project file.

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



Freescale Semiconductor, Inc.

synchronized debugging through DA-C IDE
Troubleshooting

















Freescale Semiconductor, Inc.

For More Information: www.freescale.com































Freescale Semiconductor, Inc.

For More Information: www.freescale.com











Scripting

This chapter explains how to use the Debugger's **Component Object Model (COM)** Interface. The Debugger's Interface inherits from IDispatch. This enables the feature to control the Debugger from external scripts. The script language can be any language that supports the Component Object Model: e.g. Visual Basic Script, Perl, Java Script, etc.

This chapter contains the following sections:

- [The Component Object Model Interface](#)
- [Manual Registration](#)
- [Scripting Example](#)
- [Remote Scripting another HI-WAVE](#)

The Component Object Model Interface

The Interface Name is "**Metrowerks.Hiwave**" and consists of two methods. Both make the same except that the later one returns the result message that the given command will produce.

Listing 15.1 Interface Methods

```
HRESULT ExecuteCmd([in] BSTR command);  
HRESULT ExecuteCmdRes([in] BSTR command,  
                      [out, retval] BSTR *result);
```

Parameters:

command

For this command you can use the debugger commands that are specified in the chapter [Debugger Commands](#).

result

Returns the result message that the command produced. This is the same message you would see in the command window when executing the command.

Return Values:

If the method succeeds, the return value is S_OK.
If the method fails, the return value is an error.

Manual Registration

NOTE The Component Object Model Interface will be automatically registered during the installation.

When executing the batch file **bin/regservers.bat** the Component Object Model Interface from the Code Warrior Debugger will be explicitly registered. Or use **prog/hiwave.exe /RegServer**.

Scripting Example

The following Visual Basic Script demonstrates the use of the Component Object Model Interface from the CodeWarrior Debugger. This small example will start the CodeWarrior Debugger (HI-WAVE), open a command window, set the target interface to simulator and loads an application named "filbo.abs".

Listing 15.2 example.vbs

```
' Code Warrior Debugger COM Scripting Example

Dim h
Set h = CreateObject("Metrowerks.Hiwave")

h.ExecuteCmd("open command")

Dim result
result = h.ExecuteCmdRes("set sim")
```

```
If result <> "" Then
    msgbox result
End If

h.ExecuteCmd("load fibo.abs")
```

Remote Scripting another HI-WAVE

Its also possible to remote control another HI-WAVE from within a running HI-WAVE. To do so open the component ComMaster. This will add additional commands. You can see them by entering help in the command window.

NOTE Make sure that the HI-WAVE you want to remote control is registered.

COM_START

Description The **COM_START** command starts another HI-WAVE. Its only possible to start one HI-WAVE at once. If you want to have several remote HI-WAVE applications simply open several ComMaster components.

Usage COM_START

Components ComMaster component.

Example

```
in>COM_START
```

The remote Debugger application is started.

COM_EXIT

Description This command will quit the started reomte HI-WAVE.

Usage COM_EXIT

Components ComMaster component.



Scripting

Remote Scripting another HI-WAVE

Example

in>COM_EXIT

The remote Debugger application is closed.

COM_EXE

Description With this command you can send commands to the remote HI-WAVE

Usage COM_EXE "<MyCommand>"

Components ComMaster component.

Example

in>COM_EXE "load fibo.abs"

Loads an application named "fibo.abs" in the remote Debugger.

Appendix

This chapter contains the followings sections:

- [Messages in Status Bar](#)
- [EBNF Notation](#)
- [Constant Standard Notation](#)
- [Register Description File](#)
- [OSEK ORTI File Sample](#)
- [Bug Reports](#)
- [Technical Support](#)

Messages in Status Bar

This section describes debugger status messages.

Status Messages

This section describes the different status messages.

READY

The Simulator/Debugger is ready and waits until a new target or application is loaded. This message is generated once the Simulator/Debugger has been started.

HALT

Program execution has been stopped by a request of the application. The predefined macro HALT (defined in HIDEF.H) has been reached in the application code during execution of the application.

RUNNING

The application is currently executing in the Simulator/Debugger.

HALTED

Execution 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 Simulator/Debugger has been reset on user request. The menu entry **Simulator>Reset** or the Reset icon in the tool bar has been selected, or the reset command has been used.

HARDWARE RESET

This message is generated when the Simulator/Debugger has been Reset on user request and when a target is specified. The menu entry **Simulator>Reset** or the Reset icon in the tool bar has been selected, or the Reset command has been used.

Stepping, Breakpoint and Watchpoints Messages

This section describes the different Stepping, Breakpoint and Watchpoints messages.

STEPPED

Program execution has been stopped after a single step at source level. The menu entry **Run>Single Step** or the **Single Step** icon in the tool bar has been selected.

STEPPED OVER

Execution has been stopped after stepping over a function call. The menu entry **Run>Step Over** or the **Step Over** icon in the tool bar has been selected.

STOPPED

Execution has been stopped after stepping out of a function call. The menu entry **Run>Step Out** or the **Step Out** icon in the tool bar has been selected.

TRACED

Execution has been stopped after a single step at assembler level. The menu entry **Run>Assembly Step** or the **Assembly Step** icon in the tool bar has been selected.

BREAKPOINT

Program execution has been stopped because a breakpoint has been reached.

WATCHPOINT

Execution has been stopped because a watchpoint has been reached. The format from this message is:

Watchpoint at **address: size**

Where:

- **address** is the start address in memory where the watchpoint has been defined.
- **size** is the size of the memory area where the watchpoint has been defined.

The name of the variable is displayed (if available).

CPU Specific Messages

Some error messages depend on the CPU used. These are messages related to exceptions. The Simulator/Debugger make a distinction between predefined exceptions (which have a specific meaning for all derivatives in the CPU family) and user defined exceptions (which can be freely configured by the user or does not have the same meaning for all derivatives in the CPU family).

Format for exception message is:

Exception **string** | **number**

Where:

- **string** describes the reason for the exception. This string is only specified when a predefined exception is detected.

- **number** is the entry in the vector table that generates the exception. This number is only specified when a user defined exception is detected.

Two exceptions are treated differently; the address error and the bus error exception.

ADDRESS ERROR

An address error exception for the target processor has been generated. Check your hardware manual for the reason of the Address Error Exception.

BUS ERROR

A bus error exception for the target processor has been generated. Check your hardware manual for the reason of the Bus Error Exception.

OTHER EXCEPTION

An exception has been generated for a vector that is not associated with an interrupt function.

Possible reasons:

- You have forgotten to disable an interrupt source. Insert code to disable the interrupt source in your application.
- You have forgotten to initialize the corresponding entry in the vector table with the address of the function associated with the interrupt. Initialize the vector table.

Target Specific Messages

Some messages are closely related to the debugging interface used (Simulator, Emulator,...).

These messages are listed in the corresponding Target Manual.

Examples: Simulator/Debugger Simulator Messages

This section describes the different Simulator/Debugger Simulator messages.

SIM_READY

The Simulator/Debugger simulator is ready and waits for user commands. This message is generated when an application has been loaded into the Simulator/Debugger Simulator.

More Simulator Peculiar Messages: Memory Access Messages

This section describes the different Simulator Peculiar Messages: Memory Access Messages

READ_UNDEFINED

The Simulator/Debugger detects a read access on a RAM area, where there was no previous write access. This allows you to track read access on uninitialized local variables.

NO MEMORY

The Simulator/Debugger has detected an attempt to access a memory area that is not defined (no memory).

Possible reasons:

- Your code is not correct and tries to access an address where there is no memory available. Correct your code.
- Your memory configuration is not correct. Check the current configuration in the **Memory Configuration** dialog box.

PROTECTED

The Simulator/Debugger has detected a write access on a ROM area.

Possible reason:

- Your code is not correct and tries to write in a ROM area. Correct your code.
- Your memory configuration is not correct. Check the current configuration in the **Memory Configuration** dialog box.

EBNF Notation

This chapter gives a short overview of the EBNF notation, which is frequently used in this manual to describe file formats and syntax rules.

Introduction to EBNF

Extended Backus–Naur Form (EBNF) is frequently used in this reference manual to describe file formats and syntax rules. Therefore, a short introduction to EBNF is given in [Listing 16.1](#).

Listing 16.1 EBNF Example

```
ProcDecl=PROCEDURE "(" ArgList ")".
ArgList=Expression {"," Expression}.
Expression=Term ("*" | "/" ) Term.
Term=Factor AddOp Factor.
AddOp="+" | "-".
Factor=(["-"] Number) | "(" Expression ")".
```

The EBNF language is a formalism that can be used to express the syntax of context-free languages. An EBNF grammar is a set of rules called **productions** of the form:

```
LeftHandSide=RightHandSide.
```

The left hand side is a so-called nonterminal symbol, the right hand side describes how it is composed.

EBNF consists of the following symbols:

- Terminal symbols (terminals for short) are the basic symbols, which form the language described. In above example, the word **PROCEDURE** is a terminal. Punctuation symbols of the language described (not of EBNF itself) are quoted (they are terminals, too), while other terminal symbols are printed in **boldface**.
- Nonterminal symbols (nonterminals) are syntactic variables and have to be defined in a production. They have to appear on the left hand side of a production somewhere. In above example, there are many nonterminals, for example, ArgList or AddOp.
- The vertical bar "|" denotes an alternative; either the left or the right side of the bar can appear in the language described, but one of them has to

appear. For example, the 3rd production above means “an expression is a term followed by either a "*" or a "/" followed by another term”.

- Parts of an EBNF production enclosed by "[" and "]" are optional. They may appear exactly once in the language, or they may be skipped. The minus sign in the last production above is optional, both -7 and 7 are allowed.
- The repetition is another useful construct. Any part of a production enclosed by "{" and "}" may appear any number of times in the language described (including zero, that is, it may also be skipped). ArgList above is an example: an argument list is a single expression or a list of any number of expressions separated by commas. (Note that the syntax in the example does not allow empty argument lists...)
- For better readability, normal parentheses may be used for grouping EBNF expressions, as is done in the last production of the example. Note the difference between the first and the second left bracket: the first one is part of EBNF itself, the second one is a terminal symbol (it is quoted) and therefore may appear in the language described.
- A production is always terminated by a period.

EBNF-Syntax

We can now give the definition in EBNF:

```

Production=NonTerminal "=" Expression ".".
Expression=Term {"|" Term}.
Term=Factor {Factor}.
Factor=NonTerminal
| Terminal
| "(" Expression ")"
| "[" Expression "]"
| "{" Expression }".
Terminal=Identifier | "\"" <any char> "\".
NonTerminal=Identifier.
    
```

The identifier for a nonterminal can be any name you like; terminal symbols are either identifiers appearing in the language described or any character sequence that is quoted.

Extensions

In addition to this standard definition of EBNF, we use the following notational conventions:

- The counting repetition: Anything enclosed by "{" and "}" and followed by a ^{superscripted} expression x must appear exactly x times. x may

also be a nonterminal. In the following example, exactly four stars are allowed:

```
Stars = {"*"}4.
```

- The size in bytes. Any identifier immediately followed by a number *n* in square brackets ("[" and "]") may be assumed to be a binary number with the most significant byte stored first, having exactly *n* bytes.
Example:

```
Struct = RefNo FilePos[4].
```

- In some examples, we enclose text by "<" and ">". This text is a meta-literal. Whatever the text says may be inserted in place of the text. (cf. <any char> in the above example, where any character can be inserted).

“Expression” Definition in EBNF

```
expression= lorExpr.
lorExpr= landExpr {"|" landExpr} // logical OR
landExpr = orExpr {"&&" orExpr} // logical AND
orExpr = xorExpr {"|" xorExpr} // bitwise OR
xorExpr= andExpr {"^" andExpr} // bitwise XOR
andExpr = eqExpr {"&" eqExpr} // bitwise AND
eqExpr = relExpr {"==" | "!=") relExpr}
relExpr = shiftExpr {"<" | ">" | "<=" | ">=")
    shiftExpr}
shiftExpr = addExpr {"<<" | ">>") addExpr}
addExpr = mulExpr {"+" | "-" ) mulExpr}
MulExpr = castExpr {"*" | "/" | "%") castExpr}
castExpr= ["~" | "!" | "+" | "-" ] parenExpr
parenExpr= "(" expression ")"
    | cObject
    | symbol
    | register
    | variable
    | string
    | number
cObject= ["(" cType ")"] expression
    | "&" itemName
    | "*" itemName
    | itemName {"( "." | "->")identifier} |
    ("[" expression "]" )
```

```

cType= [qualifier] [specifier] type
      | [qualifier] specifier
      | specifier
      | "void *"
qualifier= "const" | "volatile"
specifier= "signed" | "unsigned"
type= "char" | "short" | "long" | "int" |
      "float" | "double"
symbol defined with the DEFINE command
register= IOReg
variable= ObjectReg
ObjectReg= ["OBJPOOL::"] ObjectSpec
ObjectSpec= ObjectName ["." FieldName].
ObjectName= ident [":" Index].
FieldName= IdentNum ( [".." IdentNum] | ["." Size] ).
IdentNum= ident | "#" HexNumber.
Size= "B" | "W" | "L".
ident is an identifier as defined in ANSI-C

```

```

IOReg= ["IOREG::"] group | regName
group refer to the silicon vendor I/O register file definition
regName refer to the silicon vendor Register Name definition

```

```

itemName = module | [[module] ":"] procedure |
            [[module] ":" [procedure] ":"] variable
variable = ident { "." ident | number }
module = ident ["." extension]
procedure = ident
extension is an identifier as defined in ANSI-C
number is a number as defined in ANSI-C
ident is an identifier as defined in ANSI-C

```

Module names can have an extension. If no extension is specified, the parser will look for the first module that has the same name (without extension).

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 (.cpp for program

sources in assembler) (e.g., `fibonacci.c`), since all debug information is contained in the `.abs` file and object files are not used. Please adapt the following examples with your `.abs` application file format.

Semantic

A scope represents either a module or a procedure. A scope is recognized by the presence of the double colon which terminates the scope. If the scope identification contains at least one colon, it is assumed to represent a procedure, otherwise a module.

Empty module or procedure names represent the current module or procedure, respectively. The current procedure is the procedure that the `pc` of the simulator points into. The current module is the module that contains the current procedure.

Items are identified either absolutely or relatively, corresponding to the presence or absence of a scope.

An item is identified absolutely by specifying its scope, that is, the module and/or procedure where the item is located.

An item is identified relatively, if a scope is omitted. In this case, the item is assumed to be located in the current procedure.

Examples

fibonacci:fib1 matches the local variable **fib1** of the procedure **Fibonacci** in the module **fibonacci**.

:main matches the procedure **main** in the current module.

start12:_Startup matches the procedure **_Startup** in the module **start12**.

::counter matches global variable **counter** of the current module.

:Fibonacci:fib1 matches the local variable **fib1** of the procedure **Fibonacci** of the current module.

fibonacci::counter matches the global variable **counter** of the module **fibonacci**.

fib1 matches the local or global variable or module of the current procedure and/or the current module.

startupData.flags matches the field flags of the local or global variable **startupData** (which is a structure) of the current module or procedure.

Constant Standard Notation

Inside an expression, the ANSI C standard notation for constant is supported. That means that independently from the current number base you can specify hexadecimal or octal constants using the standard ANSI C notation.

Example

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

In the same way, the Assembler notation for constant is supported. That means that independently from the current number base you can specify hexadecimal, octal or binary constants using the assembler prefixes.

Example

Notation	Meaning
\$----	Hexadecimal constant
@	Octal constant
%	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 **\$**. Otherwise, the command line detects a symbol and not a number.

Example

Notation	Meaning
5AFD	Hexadecimal constant \$5AFD .
AFD	Symbol, whose name is AFD .

Register Description File

When loading a Simulator/Debugger target, the definition of the I/O registers is loaded from a file. This allows you to use the names of these registers as parameters of the commands or as operands in an expression. The syntax of the file is given below.

There may be several different files depending on the MCU used. The name of the correct file is derived from the MCU identification number (MCUID) in the following way:

MCUIxxxx.REG

where nnn is the MCUID in hexadecimal representation. This file is expected to be found in the directory where the program files are located (e.g., . . \PROG). If this file is not found, a file with the name 'DEFAULT.REG' is searched for and loaded, if found. If no file is found, an error message is displayed.

File format

The register description file contains the following information (for details refer to the EBNF definition in [Appendix](#)). First, a header contains the name, identification number and location of the register block of the MCU. The header is followed by a list of module descriptors. Each of those contain register definitions and optionally a memory map specification. The register definitions may be grouped under a group name. Each register definition defines the name, address and size of an I/O register. The memory map specification is used by the MEM command to display the configured memory of that module.

Description using EBNF.

The format of the register file is described in [Listing 16.2](#) in EBNF.

Listing 16.2 Register file description EBNF.

```
MCUDescription=Header {Module}.
Header="MCU" McuName McuId RegBase RegSize.
Module="MODULE" ModuleName {RegDef} {GroupDef | MapDef}.
GroupDef="GROUP" GroupName {RegDef}.
RegDef=RegName RegOffset Size.
```



```
MapDef="MEMMAP" BlkName BaseMapDef {MapSecifier}.
BaseMapDef="BASE" Exp "SIZE" Exp "ENABLED" Exp.
MapSpecifier="SPECIFIER" [Label] Exp.
Exp=CExpression | SwitchExpr.
SwitchExpr=CExpression ":" {CaseSpec}.
CaseSpec=[" ConstValue ":" (CExpression | StringDef) "].
```

```
McuName=StringDef.//name of the MCU
McuId=ConstValue.//identification number of the MCU
RegBase=ConstValue.//base address of the registers after reset
ModuleName=Name.//name of the module
GroupName=Name.//name of a group of registers
RegName=Name.//name of the register
```

```
RegOffset=ConstValue.//offset from the register base address
Size=ConstValue.//size of the register in bits.
BlkName=Name.//name of the memory block.
Label=StringDef.//name to be used to label the specifier
CExpression=// expression as defined in ANSI-C which
    contains integer values only.
ConstValue=// constant value as defined in ANSI-C
Name=// identifier as defined in ANSI-C
StringDef=// any number of printable character in double quotes
(")
```

[1] evaluation of expressions is done using signed 32 bit arithmetic.

[2] all non-printable characters are interpreted as white spaces.

Example

[Listing 16.3](#) describes a hypothetical MCU. It contains the modules ABC, SQIM and FLASH. The SQIM has two groups of registers, the PORTS and CHIPSELECTS.

Listing 16.3 MCU examples

```
MCU "MY_MCU" 0x07A5 0xFFF000 0x1000
  MODULE ABC
    ABCMCR 0x700 16
```



Appendix

Register Description File

```
PORTABC      0x706      16
MODULE SQIM
SQIMCR       0xA00      16
SYNCR        0xA04      16
GROUP PORTS
  PORTA       0xA10      8
  PORTB       0xA11      8
GROUP CHIPSELECTS
  CSPAR0      0xA44      16
  CSBARA      0xA60      16
  CSORA       0xA62      16
MEMMAP CSA
  BASE (CSBARA & 0xFFF8) << 8
  SIZE CSBARA & 7 :
    [0:0x800] [1:0x2000] [2:0x4000]
    [3:0x10000] [4:0x20000] [5:0x40000]
    [6:0x80000] [7:0x80000]
  ENABLED (CSPAR0 & 3) >= 2
  SPECIFIER "ACCESS" (CSORA >> 11) & 3 :
    [0:"None"] [1:"Read"]
    [2:"Write"] [3:"Both"]
  SPECIFIER "BYTE" (CSORA >> 13) & 3 :
    [0:"None"] [1:"Lower"]
    [2:"Upper"] [3:"Both"]
  SPECIFIER (CSORA >> 4) & 3 :
    [0:"None"] [1:"Lower"]
    [2:"Upper"] [3:"Both"]
MODULE FLASH
  FEEMCR      0x820      16
  FEEBAH      0x824      16
  FEEBAL      0x826      16
MEMMAP FLASH
  BASE (FEEBAH << 16)
  SIZE 0x8000
  ENABLED (FEEMCR & 0x8000) == 0
```

<eof>

OSEK ORTI File Sample

When building an OSEK project in CodeWarrior, the OSEK ORTI file is automatically generated by the the OSEK System Generator. The generated file has the same name and the same location as executable file but its extension is .ort.

Listing 16.4 OSEK ORTI File Sample

```
IMPLEMENTATION Motorola_ORTI_OSEKturbo_OS12_2_1_1_17 {

OS {
    ENUM UINT8 [ "NO_TASK" = 0,
                "MotorDriveTask" = 1,
                "ControlTask" = 2,
                "InitTask" = 3,
                "InputTask" = 4,
                "LockTask" = 5
    ] RUNNINGTASK, "Running Task Identification";
    ENUM UINT8 [ "NO_SERVICE" = 0, "StartOS" = 0x01, "ShutdownOS"
= 0x02,
                "GetActiveApplicationMode" = 0x03,
                    /* task management services*/
                "ActivateTask" = 0x10, "TerminateTask" = 0x11,
"ChainTask" = 0x12,
                "Schedule" = 0x13, "GetTaskId" = 0x14, "GetTaskState" =
0x15,
                    /* interrupt handling services*/
                "EnterISR" = 0x20, "LeaveISR" = 0x21,
                "EnableInterrupt" = 0x22, "DisableInterrupt" = 0x23,
"GetInterruptDescriptor" = 0x24,
                "ResumeOSInterrupts" = 0x25, "SuspendOSInterrupts" =
0x26,
                "EnableAllInterrupts" = 0x27, "DisableAllInterrupts" =
0x28,
                    /* resource management services*/
                "GetResource" = 0x30, "ReleaseResource" = 0x31,
                    /* event control services*/
                "SetEvent" = 0x40, "ClearEvent" = 0x41, "GetEvent" =
0x42, "WaitEvent" = 0x43,
```



Appendix

OSEK ORTI File Sample

```
        /* messages services*/
"SendMessage" = 0x50, "ReceiveMessage" = 0x51,
        /* counters and alarms services*/
"GetAlarmBase" = 0x60, "GetAlarm" = 0x61, "SetRelAlarm" =
0x62,
"SetAbsAlarm" = 0x63, "CancelAlarm" = 0x64,
        /* OSEK OS v1.0 specs*/
"InitCounter" = 0x65, "CounterTrigger" = 0x66,
"GetCounterValue" = 0x67, "GetCounterInfo" = 0x68,
        /* hook routines*/
"ErrorHook" = 0x70, "PreTaskHook" = 0x71, "PostTaskHook"
= 0x72,
"StartupHook" = 0x73, "ShutdownHook" = 0x74,
        /* extra services*/
"IdleLoopHook" = 0x75] CURRENTSERVICE, "OS Services
Watch";
ENUM UINT8 [ "TASK_LEVEL" = 0
            , "SYSTEM_TIMER" = 1
            , "StallInt" = 3
        ] RUNNINGISR, "Executed ISR Identification";
};
TASK {
    ENUM UINT8 [ "0" = 1, "5" = 2, "10" = 3, "20" = 4, "30" = 5]
    PRIORITY, "Task Priority";
    ENUM UINT8 [ "RUNNING" = 0, "WAITING" = 1, "READY" = 2,
    "SUSPENDED" = 3 ] STATE, "Task State";

    UINT8EVENTS, "Events State";
    UINT8WAITEVENTS, "Waited Events";
    STRING MASKS, "Task Event Masks";

    ENUM ADDRESS [ "MAIN_STACK" = "&_OsOrtiStackStart",
"MotorDriveTask_STACK" = "OsMotorDriveTaskStack+1",
    "ControlTask_STACK" = "OsControlTaskStack+1",
    "NO_STACK" = 0 ] STACK, "Current Task Stack";
    STRING PROPERTY, "Task Properties";

};
STACK {
    ADDRESSSTARTADDRESS , "Stack Start Address";
    ADDRESSENDADDRESS , "Stack End Address";
    UINT16SIZE , "Stack Size";
```



```
};
COUNTER{
    STRINGMAXALLOWEDVALUE, "MAXALLOWEDVALUE";
    STRING TICKSPERBASE, "TICKSPERBASE";
    STRING MINCYCLE, "MINCYCLE";
    UINT16 VALUE, "Current Value";
    ENUM UINT8[ "NO_ALARM" = 0, "ALARM" = 1 ] STATE, "Activated
Alarm";
};

ALARM{
    ENUM UINT8 [ "ALARMSTOP" = 0, "ALARMRUN" = 1 ] STATE, "Alarm
State";
    STRING COUNTER, "Assigned Counter";
    STRING TASK, "Notified Task";
    STRING EVENT, "Event to set";
    UINT16 TIME, "Time to expire";
    UINT16 CYCLE, "Cycle period";
};

MESSAGE{
    STRING TYPE, "Message Type";
    STRING TASK, "Notified Task";
    STRING EVENT, "Event to be set";
};
};

/* Application Description Part */

OS os {
    RUNNINGTASK = "OsRunning";
    CURRENTSERVICE = "OsOrtiRunningServiceId";
    RUNNINGISR = "OsOrtiRunningISRId";
};

TASK MotorDriveTask {
    PRIORITY = "3";
    STATE = "( OsRunning != 1 ) * ( ( ( OsTaskStatus[1] & 0x02 )
!= 0 ) + ( ( OsTaskStatus[1] & 0x02 ) == 0 ) * ( ( OsTaskStatus[1]
& 0x04 ) != 0 ) * 2 + ( OsTaskStatus[1] == 0 ) * 3 )";
};
```



Appendix

OSEK ORTI File Sample

```
STACK = "OsMotorDriveTaskStack+1";
EVENTS = "OsTaskSetEvent[0]" ;
WAITEVENTS = "OsTaskWaitEvent[0]" ;
MASKS = "UP_EVENT = , STOP_EVENT = , DOWN_EVENT = ";
        PROPERTY = "EXTENDED, FULLPREEMPT, Priority: 10 ";

};
TASK ControlTask {
PRIORITY = "4";
STATE = "( OsRunning != 2 ) * ( ( ( OsTaskStatus[2] & 0x02 )
!= 0 ) + ( ( OsTaskStatus[2] & 0x02 ) == 0 ) * ( ( OsTaskStatus[2]
& 0x04 ) != 0 ) * 2 + ( OsTaskStatus[2] == 0 ) * 3 )";
STACK = "OsControlTaskStack+1";
EVENTS = "OsTaskSetEvent[1]" ;
WAITEVENTS = "OsTaskWaitEvent[1]" ;
MASKS = "KEY_EVENT = , HALF_SEC_EVENT = , STALL_EVENT = ,
STALL_END_EVENT = , REVERSE_EVENT = ";
        PROPERTY = "EXTENDED, FULLPREEMPT, Priority: 20 ";

};
TASK InitTask {
PRIORITY = "5";
STATE = "( OsRunning != 3 ) * ( ( ( OsTaskStatus[3] & 0x04 )
!= 0 ) * 2 + ( OsTaskStatus[3] == 0 ) * 3 )";
STACK = "&_OsOrtiStackStart";
EVENTS = "0" ;
WAITEVENTS = "0" ;
MASKS = "";
        PROPERTY = "BASIC , NONPREEMPT, Priority: 30 , AUTOSTART";

};
TASK InputTask {
PRIORITY = "1";
STATE = "( OsRunning != 4 ) * ( ( ( OsTaskStatus[4] & 0x04 )
!= 0 ) * 2 + ( OsTaskStatus[4] == 0 ) * 3 )";
STACK = "&_OsOrtiStackStart";
EVENTS = "0" ;
WAITEVENTS = "0" ;
MASKS = "";
        PROPERTY = "BASIC , FULLPREEMPT, Priority: 0 ";

};
```



```
TASK LockTask {
  PRIORITY = "2";
  STATE = "( OsRunning != 5 ) * ( ( ( OsTaskStatus[5] & 0x04 )
!= 0 ) * 2 + ( OsTaskStatus[5] == 0 ) * 3 )";
  STACK = "&_OsOrtiStackStart";
  EVENTS = "0" ;
  WAITEVENTS = "0" ;
  MASKS = "";
  PROPERTY = "BASIC , FULLPREEMPT, Priority: 5 ";

};

STACK MAIN_STACK {
  STARTADDRESS = "&_OsOrtiStackStart";
  ENDADDRESS = "&_OsOrtiStart";
  SIZE = "&_OsOrtiStart - &_OsOrtiStackStart";
};

STACK ISR_STACK {
  STARTADDRESS = "OsIsrStack";
  ENDADDRESS = "OsIsrStack + 64";
  SIZE = "64";
};

STACK MotorDriveTask_STACK {
  STARTADDRESS = "OsMotorDriveTaskStack+1";
  ENDADDRESS = "OsMotorDriveTaskStack + 101";
  SIZE = "100";
};

STACK ControlTask_STACK {
  STARTADDRESS = "OsControlTaskStack+1";
  ENDADDRESS = "OsControlTaskStack + 101";
  SIZE = "100";
};

COUNTER SYSTEMTIMER{
  MAXALLOWEDVALUE = "0xFFFF";
  TICKSPERBASE = "10";
  MINCYCLE = "0";
```



Appendix

OSEK ORTI File Sample

```
VALUE = "OsCtrValue[0]";
STATE = "(OsCtrLink[0] != 0xFF)";
};

ALARM HALF_SEC_AL{
    STATE = "(OsAlmLink[0] != 0)";
    COUNTER = "SYSTEMTIMER";
    TASK = "ControlTask";
    EVENT = "HALF_SEC_EVENT () ";
    TIME = "OsAlmValue[0] - OsCtrValue[OsAlmCtr[0]] +
((OsAlmValue[0] - OsCtrValue[OsAlmCtr[0]]) < 0)*(0xFFFF+1)";
    CYCLE = "OsAlmCycle[0]";
};

ALARM POLLINPUTS_AL{
    STATE = "(OsAlmLink[1] != 1)";
    COUNTER = "SYSTEMTIMER";
    TASK = "InputTask";
    EVENT = " ";
    TIME = "OsAlmValue[1] - OsCtrValue[OsAlmCtr[1]] +
((OsAlmValue[1] - OsCtrValue[OsAlmCtr[1]]) < 0)*(0xFFFF+1)";
    CYCLE = "OsAlmCycle[1]";
};

ALARM STALL_END_AL{
    STATE = "(OsAlmLink[2] != 2)";
    COUNTER = "SYSTEMTIMER";
    TASK = "ControlTask";
    EVENT = "STALL_END_EVENT () ";
    TIME = "OsAlmValue[2] - OsCtrValue[OsAlmCtr[2]] +
((OsAlmValue[2] - OsCtrValue[OsAlmCtr[2]]) < 0)*(0xFFFF+1)";
    CYCLE = "OsAlmCycle[2]";
};

ALARM REVERSE_AL{
    STATE = "(OsAlmLink[3] != 3)";
    COUNTER = "SYSTEMTIMER";
    TASK = "ControlTask";
    EVENT = "REVERSE_EVENT () ";
    TIME = "OsAlmValue[3] - OsCtrValue[OsAlmCtr[3]] +
((OsAlmValue[3] - OsCtrValue[OsAlmCtr[3]]) < 0)*(0xFFFF+1)";
    CYCLE = "OsAlmCycle[3]";
};
```




```
MESSAGE Msg_Input {  
    TYPE = "UNQUEUED";  
    TASK = "ControlTask ";  
    EVENT = "KEY_EVENT ";  
};  
MESSAGE Msg_Lock {  
    TYPE = "UNQUEUED";  
    TASK = "LockTask ";  
    EVENT = "";  
};
```



Bug Reports

If you cannot solve your problem, you may need to contact our Technical Support Department. Isolate the problem – if it's a Debugger problem, write a short program reproducing the problem. Then send us a bug report.

Send or fax your bug report to your local distributor, it will be forwarded to the Technical Support Department.

The report type gives us a clue how urgent a bug report is. The classification is:

Information

Things you'd like to see improved in a future major release, that would be handy, but you can live without.

Bug

An error for which you have a work around or would be satisfied for the time being if we could supply a work around. If you already have a work around, we'd like to know it, too. Bugs will be fixed in the next release.

Critical Bug

A grave error that makes it impossible for you to continue with your work.

Electronic Mail (email) or Fax Report Form

If you send the report by fax or email, the following template can be used:

Metrowerks REPORT FORM

Fill this form and send it to Metrowerks:

EMail: support_europe@metrowerks.com

Fax : +(41) 61 690 75 01

CUSTOMER INFORMATION



Customer Name:
Company :
Customer Number:
Phone Number:
Fax Number:
Email Address:

PRODUCT INFORMATION

Product (HI-CROSS+, Simulator/Debugger, Smile Line,...):
Host Computer (PC, ...):
OS/Window Manager (WinNT, Win95, Win98, Win2000, Win XP ...):
Target Processor:
Language (C, C++, ...):

TOOL INFORMATION

Tool (Compiler, Linker, ...):
Version Number (Vx.x.xx):
Options Used:
For the Simulator/Debugger only: Target Interface Used:



REPORT INFORMATION

Report Type (Bug, Wish, Information):

Severity Level (0: Higher, ... 5: Lower):

(0 : No workaround, development stopped.

**1 : Workaround found, can continue development, problem seems to
be a common one.**

2 : Workaround found, problem with very special code.

3 : Has to be improved.

4 : Wish

5 : Information

)

Description:

Technical Support

The following methods are available to receive technical support for the CodeWarrior Interactive Development Environment (IDE). Whichever method you choose, we at Metrowerks listen and act.

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

- [“E-mail”](#)
- [“FAX”](#)
- [“Support by MAIL”](#)
- [“Internet”](#)

E-mail

The best way to get technical support is through e-mail. You can attach examples to the email using a compression utility or simply uuencode.

The email addresses are:

EUROPE: support_europe@metrowerks.com

USA: support@metrowerks.com

ASIA/PACIFIC: j-emb-sup@metrowerks.com

FAX

You can fax your problem to the following numbers:

EUROPE: Fax: +41 61 690 7501

USA: Fax: +512 997 4901

ASIA/PACIFIC: +3-3780-6092

Support by MAIL

To reach technical support by normal mail, use the addresses below:



EUROPE: **Metrowerks Europe** - Riehenring 175 - CH-4058 Basel
(Switzerland)

USA: **Metrowerks** - 9801 Metric Blvd - Austin, TX 78758

ASIA/PACIFIC: **Metrowerks Japan** - Metrowerks Co., Ltd., Shibuya
Mitsuba Building 5F, Udagawa-cho 20-11, Shibuya-ku, Tokyo 150-0042
Japan

Internet

For the latest updates and product-enhancement information, go to:

<http://www.metrowerks.com>

Index

Symbols

.abs file 71
 .cmd 88
 .hidefaults 416, 427, 428, 432
 .hwl 420
 .HWP 37
 .hwp 421
 .INI 37
 .PJT 37
 .rec 163
 .sim 49
 .tgt 48
 .WND 72
 .wnd 61
 .xpr file 101

A

A 285
 About Box 59
 About True Time Simulator and Real Time
 Debugger 59
 ABSPATH 426
 ACTIVATE 286
 ADCPORT 287
 Add New Instrument 231, 232, 233
 ADDCHANNEL 286
 Address 114, 116
 ADDRESS ERROR 520
 Address... 81
 ADDXPR 287
 Align 232, 233
 All Text Folded At Loading 189
 Analog 235
 AND Mask 238, 239, 242
 Appendix 517
 Application
 Assembly Step 446
 Embedded 23
 Loading 442
 Starting 443
 Step In 444

 Step Out 445
 Step Over 445
 Stopping 443
 Target 23
 ArbPrio 398
 Arrange Icons 58
 ASCII 116
 Assembly Step 44
 Assembly Step Out 45
 Assembly Step Over 44
 Assignment 397
 AssignmentList 397
 Associated Commands 125
 AT 299
 ATTRIBUTES 287
 Auto 168
 Automatic 103, 116
 AUTOSIZE 300

B

Background Color 56
 Backgroundcolor 234, 236
 Bar 235
 Barcolor 237
 Bardirection 237
 BASE 300
 BC 301
 BCKCOLOR 302
 BD 303
 Bin 104, 115, 168, 448
 Binary 448, 451
 Bit Reverse 115, 168
 Bitnumber to Display 240
 BitRange 397
 Bottom 234
 Bounding Box 235
 BREAKPOINT 519
 Breakpoint 80, 179
 BREAKPOINT 519
 Checking condition 251
 Command 260
 Conditional 258, 264



- Counting 256, 264
- Definition 247
- Deleting 259
- Message 518
- Multiple selection 251
- Permanent 247, 256
- Position 254
- Temporary 247, 255
- breakpoint 393
- Breakpoint with Register Condition 259
- Breakpoints... 45
- BS 303
- BUS ERROR 520
- Byte 114
- C**
- C 32
- CALL 306
- Call Chain 151
- Cascade 58
- CD 306
- CF 307
- CLOCK 310
- Clone Attributes 232, 233
- CLOSE 310
- Cmd 32
- CMDFILE 311
- CodeWarrior Integration 458
- Color if 241
- Color if Bit 240
- COM 513
- Command 245
 - Syntax 211, 272
- Command File Dialog 52
- Command File menu entry 52
- Command File Playing 88
- Command Line 31
- COMPLEMENT
 - DATA Component 294
 - Memory Component 295
 - Register Component 290
- Component
 - Analog Meter 211
 - Assembly 80, 442, 443
 - Associated Menus 61
 - Command Line 86
 - Coverage 91
 - CPU 71
 - DAC 96
 - Data 98, 442, 443, 446
 - Framework 27, 28, 71
 - Inspector 213
 - IO_Led 222
 - LED 224
 - Led 224
 - Main Menu 61
 - Memory 111, 454
 - MicroC 144
 - Module 149
 - Phone 226
 - Pop Up Menu 61
 - Procedure 151
 - Profiler 154
 - Recorder 162
 - Register 166, 442, 451
 - SoftTrace 175
 - Source 178, 442, 443
 - Stimulation 192
 - Target 72
 - Terminal 456
 - VisualizationTool 229
 - Window 71
- Component Object Model 513
- Components File 61
- COMPOPTIONS 418
- Configuration 37
- Control Point
 - Definition 247
 - Dialogs 247
- Control Points 247
- Copy 232, 233
- COPYMEM 310
- CopyMem 114
- Copyright 59
- Copyrights 17
- CPORT 311
- CPU
 - Cycle 34
 - cycle 166
- CPU Message 519
 - ADDRESS ERROR 520
 - BUS ERROR 520
- CR 312
- Cross-debugging 23



Ctrl+E 231
Ctrl+L 231
Ctrl+S 231
CTRL-P 234
Current Directory 416, 427
Customize 39
Cut 232, 233
CYCLE 312
Cycle 176
Cycles 396

D

DAC

- communication DLL 476
- Configure the file types 465
- Configuring 462
- Configuring the tools 470
- database 468
- Database directory 464
- Debugger Interface 474
- Debugger name 481
- IDE 462
- library path 465
- Ndapi.dll 481
- new project 463
- Preprocessor | Header Directories 466
- Preprocessor | Preinclude file 467
- Project root directory 464
- Referential project root directory 464
- Requirements 462
- True Time Simulator and Real Time Debugger
 - project file 477
- Source 466
- Synchronized debugging 480
- Troubleshooting 480
- User help file 464
- working directories 463

DASM 313

DB 314

DDE

- HI-WAVE server 460

DDEPROTOCOL 315

Debugger DDE Server 460

Debugger Start Option -C 32

Debugger Start Option -Cmd 32

Debugger Start Option -ENVpath 32

Debugger Start Option -

- Instance=%currentTargetName 31

Debugger Start Option -Nodefaults 32

Debugger Start Option -Prod 32

Debugger Start Option -T 31

Debugger Start Option -Target 31

Debugger Start Option -W 31

Debugging 23

Dec 104, 115, 168, 448

Decimal 448

Decimalmode 241

DEFAULT.ENV 416, 427, 428, 432

DEFAULT.REG 528

DEFAULTDIR 427

DefaultDir 439

DEFINE 316

DELCHANNEL 317

Delete Breakpoint 83, 184

Demo Version Limitations 125

DETAILS 318

Disable Breakpoint 83, 184

Display 113

Display Absolute Address 82

Display Address 82

Display Address Dialog 117

Display Code 82

Display Headline 234

Display Scrollbars 234

Display Symbolic 82

Display Version 241

Displayfont 243

DL 318

Drag Out 125

Dragging 62, 63

Driving True Time Simulator and Real Time Debugger

- trough DDE 461

Drop Into 125

DUMP 319

DW 319

E

E 320

EBNF 522

Editing

- Memory 454

- Register 451



Variable 449
Editmode 231, 234
Editor 99
ELSE 321
ELSEIF 321
Enable Breakpoint 83, 184
ENDFOCUS 322
ENDFOR 322
ENDIF 323
ENDWHILE 323
Environment
 ABSPATH 426
 DEFAULTDIR 427
 ENVIRONMENT 416
 File 416
 GENPATH 429, 431
 HIENVIRONMENT 428
 HIPATH 429, 431
 LIBPATH 430, 433
 LIBRARYPATH 431
 OBJPATH 431
 TMP 432
 USELIBPATH 433
 Variable 425
-ENVpath 32
EQUAL Mask 238, 242
Events 388
Exception 397
EXECUTE 324
EXIT 324
Exit 37
Explorer 417
Expression 397
Expression Command File 101
Expression definition (EBNF) 524
Expression Editor 99
Extended Backus-Naur Form, see EBNF

F

Field Description 244, 245
File
 Environment 416
File Manager 417
Filename 238
FILL 324
Fill Memory Dialog 117

FILTER 325
FIND 325
Find 185, 187
Find Procedure 185, 188
FINDPROC 326
FLEXIm 28
Float 168
FOCUS 326
FOLD 327
Fold 189
Fold All Text 189
Folding 182
 Mark 182
Folding Menu 188
Foldings 185
FONT 328
Fonts 56
FOR 328, 342
Format 113, 448, 451
Format mode 244
Format... 103
FPRINTF 329
FRAMES 329
Frames 175
Frozen 103, 105, 116

G

G 330
GENPATH 429, 431
Global 103
Global Variable
 Displaying 447
GO 330
Go To Line 188
Go to Line 185, 186, 187
GOTO 331
GOTOIF 331
Graphic bar 91, 154
GRAPHICS 332
Grid Color 235
Grid Mode 235
Grid Size 235

H

HALT 517



Index

Halt 43
HALTED 518
Hardware 23
Height 235
HELP 332
Help Topics 59
Hex 104, 115, 168, 448, 452
Hexadecimal 448, 451, 455
Hide Headline 39
Hide Tile 39
HIENVIRONMENT 428
High Display Value 237, 240, 244
HIPATH 429
Horiz. Text Alignment 243
Horizontal Size 234
How To ... 438

I

IdDeclaration 397
IDF 460
IDispatch 513
IF 333, 342
I-LOGIX 144
Important 17
Indicatorcolor 237, 240
Indicatorlength 237
init.cmd 441
INSPECTORUPDATE 334
-Instance=%currentTargetName 31
Instruction Syntax 273
Interrupt
 Example 392
 Stimulated 392
Interrupt_Function 392
interruption 122
Introduction 23
io_demod 390
Io_demod.abs 391
io_ex.txt 394
io_int.txt 392, 393, 394
IO_Led 389
IO_Show 390
io_var.txt 391, 392
iodemo.c 392
IO-Simulation

Main window 389
IPATH 431
ITPORT 335
ITVECT 335

J

j-emb-sup@metrowerks.com 541

K

keyword DAC
 True Time Simulator and Real Time Debugger
 project file 477
Kind of Port 236
KPORT 336

L

Layout 29, 420
Layout - Load/Store 58
LCDPORT 336
Led 222
Leds 394
Left 234
LF 337
LIBPATH 433
LIBRARYPATH 430, 431
Line Continuation 424
LINKADDR 337
LOAD 338
Load Application 36
Load Layout 231, 232, 233
Load Target 46, 48
LOADCODE 340
Loading an Application 442
LOADMEM 340
LOADSYMBOLS 341
Local 103
Local Variable
 Displaying 447
Locked 103, 105
LOG 341
Low Display Value 237, 240, 244
LS 345
Lword 114



M

Main Menu Bar 35
MainFrame 421
Marks 185
MCUID 528
MCUIOnnn.REG 528
MCUTOOLS.INI 417, 439
MEM 346
Memory
 Dump 111
 Word 111
Memory Access Message 521
 NO MEMORY 521
 PROTECTED 521
 READ UNDEFINED 521
Menu
 Help 58
 Run 42
 Target 45, 56
 View 39
 Window 57
MicroC 144
Mode 113
Module 149
MS 347
ms 176

N

Name 397
NB 348
NbTimes 398
New 36
NO MEMORY 521
NOCR 350
-Nodefaults 32
NOLF 350
NoOfBits 397

O

Object 27
Object Info Bar 34
ObjectField 397
ObjectId 397
ObjectSpec 397, 398
OBJPATH 431

Oct 104, 115, 168, 448
Octal 448
OPEN 350
Open Component 56
Open Configuration 37
Open Source File 185
OPENFILE 351
OPENIO 351
Options 439
 Pointer As Array. 103
Options - Autosize 58
Options - Component Menu 58
OSEK Kernel Awareness 406
OSEK ORTI 407
OSEK RTK Inspector 409
OSPARAM.PRM 401
Outlinecolor 241
OUTPUT 352

P

P 352
Paste 232, 233
PATH 423
Pause 163
PAUSETEST 354
PBPORT 354
Percentage 91, 154
PERIODICAL 193, 397
Periodical 103, 116, 391
PeriodicEvent 397
PerTimedEvent 397, 399
Play 162
Pointer as Array 103, 106
PORT 355
Port to Display 236
PORT_DATA 390, 391, 392
Port_Register 395
Postload command file 54
postload.cmd 457
Preference panel 38
Preferences dialog 37
Preload command file 54
preload.cmd 456
PRINTF 355
Priority 398

prn file 394
 Procedure Chain 151
 -Prod 32
 Project 421
 PROJECT.INI 45, 419
 project.ini 419
 Properties 232, 233
 PROTECTED 521
 PTRARRAY 356
 PVCS 433

R

RAISE 393
 RD 356
 READ UNDEFINED 521
 READY 517
 real time 23
 Real Time Kernel Awareness 400
 Real Time Kernels 400
 RECORD 357
 Record 162
 REGBASE 358
 REGFILE 358
 Register 166
 Register values 259, 269
 Registers 528
 Description file 528
 Registration 59
 Relative Mode 244
 Release Notes 20
 Remove 232, 233
 REPEAT 342, 358
 Replay 164
 RESET 359
 Reset command file 53
 Reset Target 46, 49
 reset.cmd 456
 RESETCYCLES 359
 RESETMEM 360
 RESETRAM 361
 RESETSTAT 361
 RESTART 362
 Restart 43
 RETURN 362
 RHAPSODY 144

Right 234
 RS 363
 Run To Cursor 83, 184
 RUNNING 518

S

S 363
 SAVE 364
 Save Configuration 37
 Save Configuration As 37
 Save Layout 231, 232, 233
 SAVEBP 365
 Scope... 103
 SDI 72
 search order 434
 Searching Order
 Assembly source files 434
 C source files 434
 Object files source files 434
 SEGPORT 366
 Send to Back 232, 233
 Send to Front 232, 233
 SET 366
 Set Breakpoint 83, 184
 Set Target 56
 Set Zero Base 177, 192
 SETCOLORS 366
 SETCONTROL 367
 SETCPU 368
 Setcpu command file 54
 Setup 231
 Show Breakpoints 83, 184
 Show Location 84, 185
 SHOWCYCLES 368
 SIM_READY 521
 Simulation 23
 Simulator 72
 Simulators File 49
 Single Step 44
 Size 234
 Size of Port 236
 SLAY 369
 SLINE 369
 Sloping 241
 Small Borders. 39



SMEM 370
SMOD 370
Source 393
SPC 371
Splitting View 91
SPROC 372
SREC 373
ST1619-HDS
 Postload command file 54
 Preload command file 54
 Reset command file 53
 Startup command file 53
Start 164, 398
Start/Continue 43
StartBit 397
Starting an Application 443
startup 418
Startup command file 53
startup.cmd 456
Statistics 156
Status Bar 34, 39
 Message 517
Status Message 517
 HALT 517
 HALTED 518
 Hardware Reset 518
 READY 517
 Reset 518
 RUNNING 518
Status register bits 166
Step In 444
 Assembly Instruction 446
 Source Instruction 444
Step Out 44, 444
 Function Call 445
Step Over 44, 444, 445
STEPINTO 373
STEPOUT 374
STEPOVER 374
STEPPED 518
STEPPED OVER 518
Stepping Message 518
 STEPPED 518
 STOPPED 518
 TRACED 519
Stimulation 391
 Example 392

 File 394
StimulationFile 397
STOP 375
STOPPED 518
Stopping an Application 443
Support
 FAX 541
 MAIL 541
support@metrowerks.com 541
support_europe@metrowerks.com 541
Symbolic 104, 448

T

-T 31
T 376
-Target 31
Target files 48
Target Message 520
 SIM_READY 521
TargetObject 390, 391, 393, 394
task 400
Template 390
TESTBOX 377
Text 240
Text Mode 243
Textcolor 243
Tile 58
Time 398
TimedEvent 397
Timer Update 93
TMP 432
Toolbar 33, 39
 Customizing 40
ToolTips 185
ToolTips Activation 180
ToolTips format 180
ToolTips mode 180
Top 232, 233
TRACED 519
Trademarks 17
True Time IO Stimulation 388
True Time Simulator and Real Time Debugger
 Concept 26
 Configuration 438
 Default Layout Configuration 419
 Demo Version Limitations 28



Index

- Drag and Drop 64
- Engine 24
- Execution framework 25, 26
- Framework component 27
- Layout 420
- Objects and Services 27
- Project 421
- project.ini 419
- Running from a command line 31
- Smart User Interface 62
- Tool tip 34
- Toolbar 33
- User Interface 29, 62
- Using on Windows 95 or Windows NT 4.0/
WIN2000 439
- TUPDATE 377

- U**
- UDec 104, 115, 168, 448
- UNDEF 378
- UNFOLD 380
- Unfold 189
- Unfold All Text 189
- Unfolding 182
 - Mark 182
- Unsigned Decimal 448
- UNTIL 381
- UPDATERATE 381
- USELIBPATH 433
- User 104

- V**
- VA 386
- Variable 391
 - Address 450
 - Displaying Global Variables 447
 - Displaying Local Variables 447
 - Editing Value 449
 - Format 98
 - Local and Global 98
 - Mode 103
 - Scope 98
 - Showing Location 450
 - Type 98
 - Value 448
- Vector 398
- VER 382

- Version number 59
- Vert. Text Alignment 243
- Vertical Size 234
- VisualizationTool
 - 7 Segment Display 240
 - Analog 236
 - Bar 237
 - Bitmap 238, 239
 - Demo 246
 - Demo limitation 246
 - Demo Version Limitations 246
 - DILSwitch 239
 - Instrument 235
 - Knob 239
 - LED 240
 - Setup 234
 - Switch 241
 - Text 243
- Vppoff command file 55
- Vppon command file 55

- W**
- W 31
- WAIT 382
- Warranty 18
- WATCHPOINT 519
- Watchpoint
 - Checking condition 264
 - Command 271
 - Conditional 264, 268
 - Counting 264, 267
 - Definition 247
 - Deleting 270
 - Message 518
 - Read 265
 - Read, Write 248
 - Read/Write 267
 - WATCHPOINT 519
 - Write 266
- Watchpoints... 45
- WB 383
- WHILE 342, 384
- Width 235
- Windows 416
- WinEdit 416, 417
- WL 385
- Word 114



Word size 113
WorkDir 439
WorkingDirectory 439
WPORT 385
WW 386

X

X-Position 235

Y

Y-Position 235

Z

ZOOM 386
Zoom in 103
Zoom out 103



Freescale Semiconductor, Inc.

CodeWarrior

True-Time Simulator & Real-Time Debugger

For More Information: www.freescale.com



Freescale Semiconductor, Inc.

For More Information: www.freescale.com