S12(X) Debugger Manual

Revised: August 10, 2010

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 6501 William Cannon Drive West Austin, Texas 78735 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

Introduction

Book I - Debugger Engine

Book III - HC(S)12(X) Debugger Common Features

Book IV - Commands and Environment Variables

Book V - Debugger Legacy

Introduction

Manual Contents

The S12(X) Debugger Manual consists of the following books:

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

Book II: HC(S)12(X) Debug Connections – describes the connections available for debugging code written for HC12 CPUs.

Book III: HC(S)12(X) Debug Connections - Common Features – describes the common connections available for debugging code.

Book IV: Commands and Environment Variables – lists available debugger commands, and connection-specific commands, with a brief description of each. Lists environment variables for the debugger engine and connection-specific environment variables, with provides a brief description of each

Book V: Debugger Legacy

Book I - Debugger Engine

Book I Contents

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

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

This book is divided into the following chapters:

- Chapter 1 - Introduction describes the Debugger application and its features.
- Chapter 2 - Debugger Interface provides all details about the Debugger user interface environment i.e., menus, toolbars, status bars and drag and drop facilities.
- Chapter 3 - Debugger Components contains descriptions of each basic component and visualization utility.
- Chapter 4 - Control Points describes the control points and associated windows.
- Chapter 5 - Real-Time Kernel Awareness contains descriptions of the Real Time concept and related applications.
- Chapter 6 - How To... provides answers for common questions and describes how to use advanced features of the Debugger.
- Chapter 7 - CodeWarrior Integration explains how to configure the Debugger for use with CodeWarrior IDE.
- Chapter 8 - Debugger COM Capabilities provides information on the Component Object Model (COM) Interface which allows the user to control debugger using scripts or other application
- Chapter 9 - Synchronized Debugging through DA-C IDE explains the use of tools with the DA-C IDE from RistanCase

1

Introduction

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

Freescale Debugger

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

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

Debugger Application

A Debugger Application contains the Debugger Engine and a set of debugger components which perform specific tasks. The Debugger Engine monitors and coordinates the component tasks. Each Debugger Component has its own functionality (e.g., source level debugging, profiling, I/O stimulation).

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

You can also open several components of the same type.

Debugger Features

- True 32-bit application
- Powerful features for embedded debugging
- Special features for real-time embedded debugging
- Powerful true-time simulation features
- Powerful simulation and debugging capabilities
- Variety of target interfaces
- User Interface
- Graphical user interface (GUI) version including command line
- Configurable GUI with tool bar
- Visualization functions
- Versatile and intuitive drag and drop functions between components
- Folding and unfolding of objects like functions, structures, classes
- Graphical editing of user-defined objects
- Smart interactions with objects
- Extensibility function
- Show Me How Tool
- Context-sensitive help
- Smooth integration into third-party tools
- Supports both Freescale and ELF/DWARF Object File Format and S-Records.

Demonstration Version Limitations

When you start the Debugger in demonstration mode or with an invalid engine license, then all components protected with FLEXlm are in demonstration mode. The limitations of all components are described in their respective chapters.

2

Debugger Interface

This chapter describes the Debugger Graphic User Interface (GUI).

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

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

- Tiled component windows automatically resize when you resize main window
- Component windows overlap
- Minimized component windows appear as Debugger Main window icons

Application Programs

The CodeWarrior installer places executable programs in the `prog` subdirectory of the CodeWarrior installation directory. For example, if you install the CodeWarrior IDE software in `C:\Program Files\Freescale`, all program files are in the folder `C:\Program Files\Freescale\CWS12v5.1\Prog`.

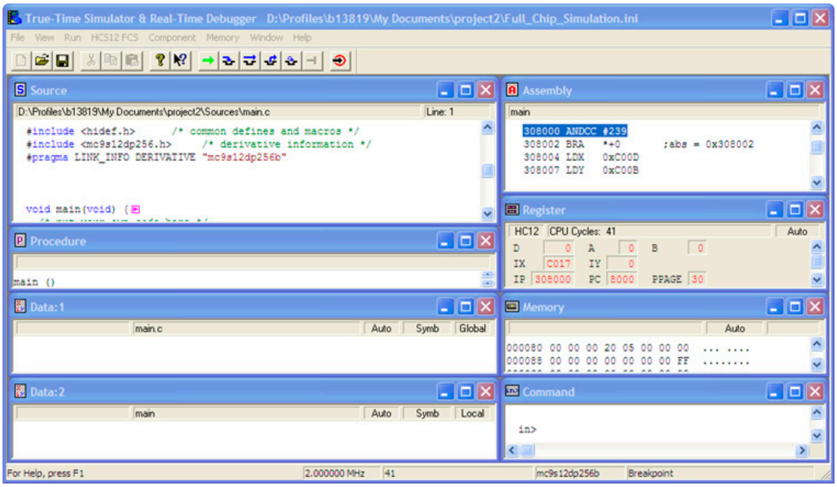
The CodeWarrior IDE uses the following files for C/C++ debugging:

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

Debugger Main Window

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

Figure 2.1 Debugger Main Window



Debugger Main Window Toolbar

The Debugger Main Window toolbar is the default toolbar. Most of the Main Window menu commands have a related shortcut icon on this toolbar. Figure 2.2 identifies each default icon.

Figure 2.2 Debugger Main Window Toolbar

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

Debugger Main Window Status Bar

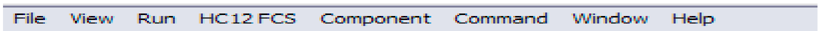
The status bar at the bottom of the Debugger Main Window, shown in Figure 2.3, contains a context sensitive help line for connection-specific information, including the number of CPU cycles for the **Simulator** connection and execution status.

Figure 2.3 Debugger Status Bar

Main Window Menu Bar

The Debugger Main Window Menu Bar, shown in Figure 2.4, is associated with the main function of the debugger application, connection, and selected windows.

Figure 2.4 Debugger Window Menu Bar



NOTE You can select menu commands from the keyboard by clicking the ALT key. A line appears under the initial letter in each item in the menu bar. Click the key corresponding to the menu of your choice, and click enter. Or use the directional arrows to move to the menu entry you want and click enter again.

Table 2.1 describes menu entries available in the menu bar.

Table 2.1 Description of the Main Menu Toolbar Entries

Menu Entry	Description
File	Use to manage debugger configuration files
View	Use to configure the toolbar
Run	Use to monitor a simulation or debug session
Connection	Use to select the debugger connection. Once you select a connection, the heading name changes.
Component	Use to select and configure extra component windows
Data	Use to select Data component functions
Window	Use to set the component windows
Help	Use to access a standard Windows Help menu

File Menu

The **File** menu shown in Figure 2.5 is dedicated to the debugger project.

Figure 2.5 File Menu

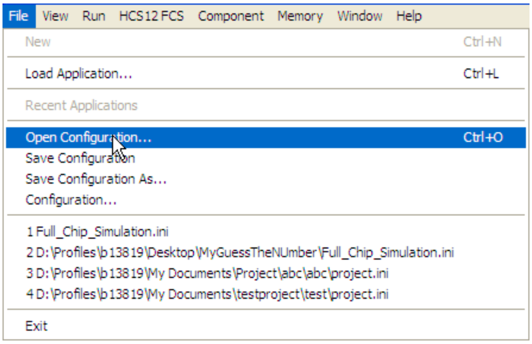


Table 2.2 describes File menu entries.

Table 2.2 File Menu Entry Description

Menu Entry	Description
New	Creates a new project
Load Application	Loads an executable file (or debugger connection if nothing is selected)
Recent Applications	Opens recently used applications
Open Configuration	Opens debugger project window. You can load a project file (`.PJT` or `.INI`) containing component names, associated window positions and parameters, window parameters, connection name, and `.ABS` application file to load. You can also load an existing `.HWC` file corresponding to a debugger configuration file.
Save Configuration	Saves the project file
Save Configuration As	Opens debugger project window to save the project file under a different path and name and/or format (such as `*.PJT`, `*.INI`)
Configuration	Opens Preferences window to set environment variables for current project

Table 2.2 File Menu Entry Description (*continued*)

Menu Entry	Description
1. Project.ini 2.Test.ini	Recent project file list
Exit	Quits the Debugger

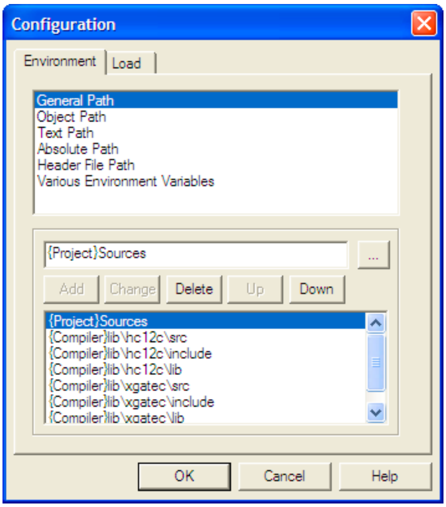
Use the toolbar icons as shortcuts for some of these functions (refer to the Debugger Main Window Toolbar section).

Configuration Window

Open the Configuration window by selecting **File > Configuration**. Use this window (shown in Figure 2.6) to set up environment variables for the current project. Click the **OK** button to save new variables in the current project file.

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

Figure 2.6 Configuration Window - Environment Tab



The **Environment** tab contains the following controls:

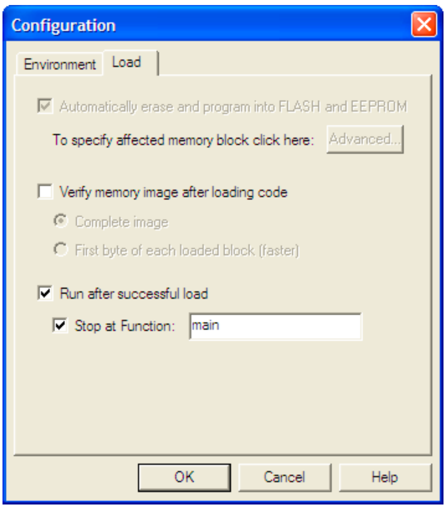
- A list box containing all available environment variables. Select a variable using the mouse or directional arrow keys.

- Command Line Arguments are displayed in the text box. You can add, delete, or modify options, and specify a directory with the browse button (...).
- A second list box contains the arguments for all of the environment variables defined in the corresponding Environment section. Select a variable using the mouse or directional arrow keys.
- **OK**: Confirms changes and saves in current project file.
- **Cancel**: Closes dialog box without saving changes.
- **Help**: Opens the help file.

The **Load** tab shown in Figure 2.7 contains the following controls:

- A checkbox that specifies automatic erase and program into Flash and EEPROM
- Advanced button specifies affected memory block
- Enable automatic memory image verification after loading code
- Enable automatic run after successful load
- Enable automatic stop at Function specified in text box.

Figure 2.7 Configuration Window - Load Tab



View Menu

In the Main Window View menu (Figure 2.8) you can choose to show or hide the toolbar, status bar, window component titles and headlines (see Component Windows Object Information Bar). You can select smaller window borders and customize the toolbar. Table 2.3 describes the View menu entries.

Figure 2.8 View Menu

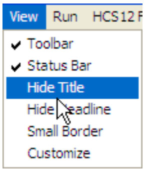


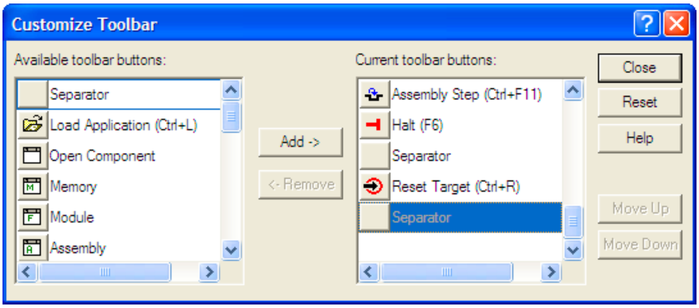
Table 2.3 View Menu Description

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

Customizing the Toolbar

When you select **View > Customize**, the Customize Toolbar dialog box appears. You can customize the toolbar of the Debugger, adding and removing component shortcuts and action shortcuts in this dialog box. You can also insert separators to separate icons. Almost all functions in **View**, **Run** and **Window** menus are available as shortcut buttons, as shown in Figure 2.9.

Figure 2.9 Customize Toolbar Dialog Box



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

Demo Version Limitations

The default toolbar cannot be configured.

Examples of View Menu Options

Figure 2.10 shows a typical component window display.

Figure 2.10 Typical Component Window Display

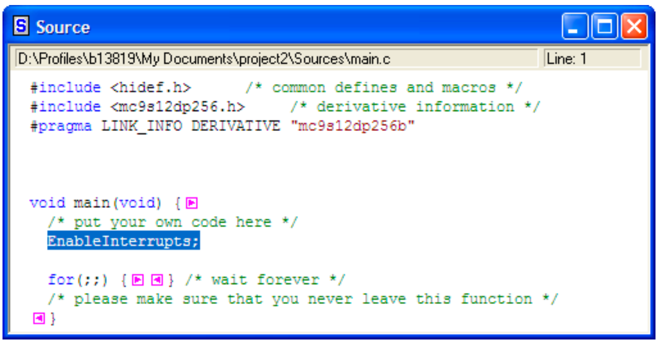


Figure 2.11 shows a component window without a title and headline.

Figure 2.11 Component Window without Title and Headline

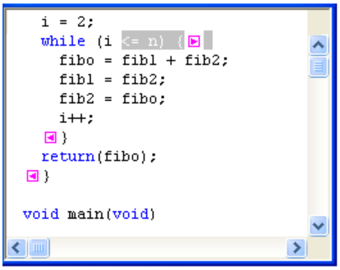


Figure 2.12 shows a component window without a title and headline, and with a small border.

Figure 2.12 Component Window without Title and Headline, and with Small Border

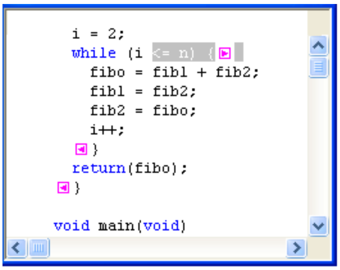
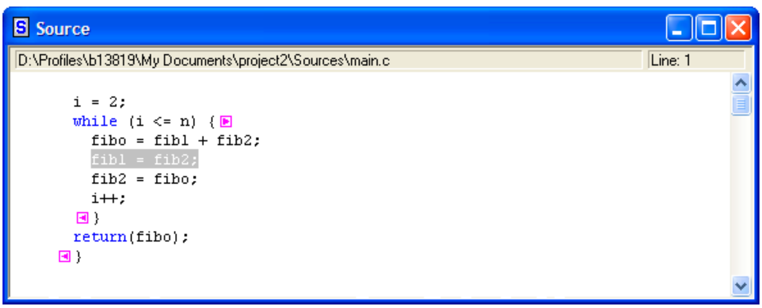


Figure 2.13 shows a component window without headline and small border.

Figure 2.13 Component Window without Headline and Small Border



Run Menu

The Main Window Run menu, shown in Figure 2.14 is associated with the debug session. You can monitor a simulation or debug session from this menu. Run menu entries are described in Table 2.4.

Figure 2.14 Run Menu

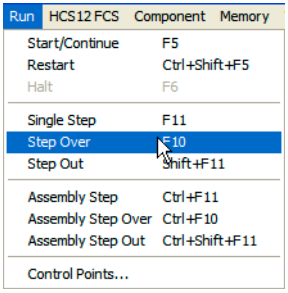


Table 2.4 Run Menu Description

Menu entry	Shortcut	Description
Start/ Continue	F5	Starts or continues execution of loaded application from current program counter (PC) until it reaches a breakpoint or watchpoint, detects a runtime error, or user stops application by selecting **Run > Halt**.
Restart	CTRL + Shift + F5	Starts execution of application from its entry point.
Halt	F6	Interrupts and halts a running application.Examine state of each variable in the application, set breakpoints, watchpoints, and inspect source code.
Single Step	F11	Performs a single step at source level in halted application. Execution continues until application reaches next source reference. If current statement is a procedure call, the debugger steps into procedure. Treats a function call as multiple statements, and steps into function.
Step Over	F10	Similar to **Single Step**, but does not step into called functions. Treats a function call as one statement.
Step Out	Shift + F11	If application halts inside a function, **Step Out** continues execution and stops at instruction following current function invocation. Has no effect if no function calls are present.

Table 2.4 Run Menu Description (*continued*)

Menu entry	Shortcut	Description
Assembly Step	CTRL + F11	Performs a single step at assembly level in halted application. Execution continues for one CPU instruction from the point at which it halted. Similar to **Single Step** command, but executes one machine instruction rather than a high-level language statement.
Assembly Step Over	CRTL + F10	Similar to **Step Over**, but steps over subroutine call instructions.
Assembly Step Out	CTRL + Shift + F11	If application halts inside a function, command continues execution and stops on CPU instruction following current function invocation. Similar to **Step Out**, but stops before assignment of result from function call.
Control Points	None	Opens Controlpoints Configuration Window to allow you to control breakpoints, watchpoints and markpoints (see Control Points).

You can provide toolbar shortcuts for some of these functions. Refer to Debugger Main Window Toolbar and Customizing the Toolbar for details. You can also set breakpoints and watchpoints from within the Source and Assembly component windows.

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

Connection Menu

This menu entry (Figure 2.15) appears between the **Run** and **Component** menus when no connection is specified in the `PROJECT.INI` file and no connection has been set. The **Connection** name is replaced by an actual connection name when the connection is set. If a connection has been set, the number of menu entries is expanded, depending on the connection. To set the connection, select **Component > Set Connection**. Refer to Component Menu for details.

Figure 2.15 Connection Menu

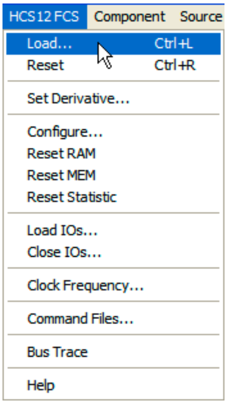


Table 2.5 describes the Connection menu entries.

Table 2.5 Connection Menu Common Options Description

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

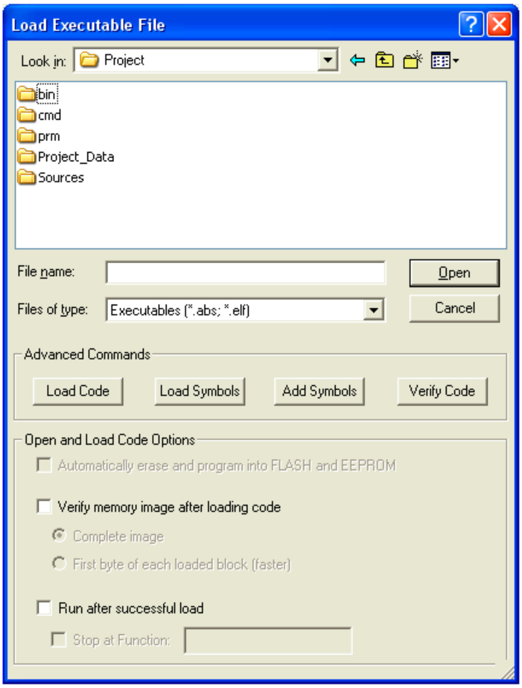
Loading a Connection

Choose ***Connection*** > **Load** in the Connection menu to load a debugger connection. This displays the Load Executable File window shown in Figure 2.16.

Load Executable File Window

From the Connection menu, choose **Load** to open the Load Executable File window, shown in Figure 2.16, then set the load options and choose a Simulation Execution Framework (an `.ABS` application file).

Figure 2.16 Load Executable File Window



Open Button

Clicking this button loads the application code and symbols.

Advanced Commands Buttons

These three buttons allow you to select which part of the executable file to load:

- **Load Code Button**: Loads only the application code into the target system. Use this button if no debugging is needed.
- **Load Symbols Button**: Loads symbols only. Only debugging information is loaded. Use this button if the code is already loaded into the target system or is programmed into a non-volatile memory device (ROM/Flash).
- **Add Symbols Button**: Loads additional symbolic information. Appends the loaded debugging information to the existing symbol table instead of replacing it. You can use this button if the executable file consists of several applications and code is already loaded into the target system or programmed into a non-volatile memory device.

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

Open and Load Code Options Area

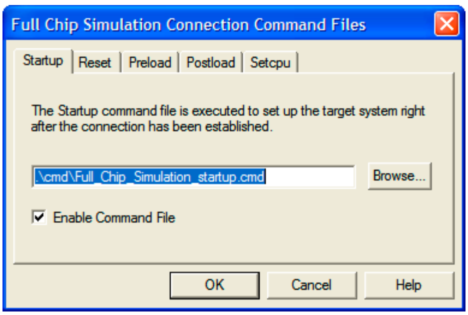
The checkboxes and buttons of this area of the Load Executable File window offer the following options:

- A checkbox specifying an automatic erase and program into Flash and EEPROM.
- A checkbox to automatically verify the memory image after loading code, with two radio buttons that let you define the memory image.
- Checkbox to automatically run after successful load.
- A checkbox to enable automatically stopping at the function specified in the textbox.

Connection Command Files Window

Choose **Connection > Command Files** to open the Connection Command Files window. Each tab of this window, shown in Figure 2.17, corresponds to an event on which a command file can be automatically run. See Startup Command File, Reset Command File, Preload Command File, and Postload Command File.

Figure 2.17 Connection Command Files Window



The command file in the edit box executes when the corresponding event occurs. Click the **Browse** button to set the path and name of the command file.

The **Enable Command File** check box allows you to enable/disable a command file on an event. By default, all command files are enabled:

- The default **Startup** command file is `STARTUP.CMD`,
- The default **Reset** command file is `RESET.CMD`,
- The default **Preload** command file is `PRELOAD.CMD`,

- The default **Postload** command file is `POSTLOAD.CMD`.

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

NOTE Setting a CPU stores the settings in this dialog 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 executes after the connection loads.

Specify the **Startup** command file full name and status (enable/disable) either with the **CMDFILE STARTUP** Command Line command or using the **Startup** property tab of the Connection Command Files Window.

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

Reset Command File

The **Reset** command file executes after the reset button, menu entry or Command Line command has been selected.

Specify the **Reset** command file full name and status (enable/disable) either with the **CMDFILE RESET** Command Line command or using the **Reset** property tab of the Connection Command Files Window.

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

Preload Command File

The **Preload** command file executes before an application loads to the target system through the connection.

Specify the **Preload** command file full name and status (enable/disable) either with the **CMDFILE PRELOAD** Command Line command or using the **Preload** property tab of the Connection Command Files Window.

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

Postload Command File

The **Postload** command file executes after an application loads to the target system through the connection.

Specify the **Postload** command file full name and status (enable/disable) either with the **CMDFILE POSTLOAD** Command Line command or using the **Postload** property tab of the Connection Command Files Window.

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

Component Menu

Figure 2.18 shows the Component menu.

Figure 2.18 Component Menu

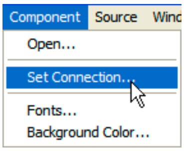


Table 2.6 describes the Component Menu entries.

Table 2.6 Component Menu Description

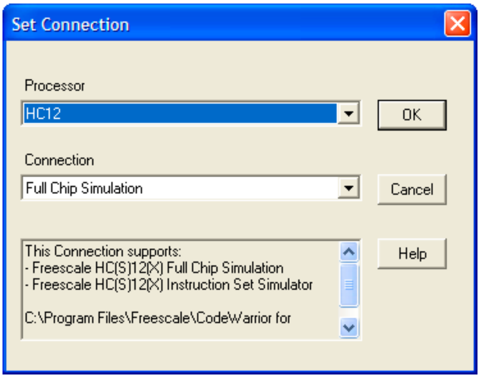
Menu entry	Description
Open	Loads an extra component window not loaded by Debugger at startup. Presents a set of components introduced in Typical Component Window Display.
Set Connection	Sets the Debugger connection.
Fonts	Opens standard Font Selection dialog to set font used by Debugger components.
Background Color	Opens standard Color Selection dialog to set background color used by Debugger component windows.

NOTE To enhance display readability, use a proportional font such as Courier or Terminal.

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

Open the Set Connection dialog box shown in Figure 2.19 by selecting **Component > Set Connection**.

Figure 2.19 Set Connection Dialog Box



1. Use the **Processor** context menu to select the desired processor.
2. Use the **Connection** context menu to select the desired connection.

   A text panel displays information about the selected connection.

NOTE When a connection cannot be loaded, the combo box displays the path where the missing `dll` must be installed.

3. Click **OK** to load connection in debugger.

NOTE For more information about which connection to load and how to set/reset a connection, refer to the How To... section of this manual.

Window Menu

In this menu, shown in Figure 2.20, you can set the general arrangement of the component windows. Figure 2.21 shows the Submenu **Window > Options** and Figure 2.22 shows the Submenu **Window > Layout**.

Figure 2.20 Window Menu

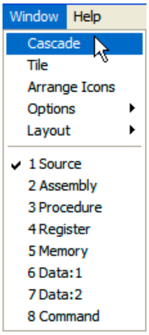


Figure 2.21 Window Menu Options Submenu

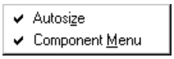


Figure 2.22 Window Menu Layout Submenu

Table 2.7 describes the Window menu entries.

Table 2.7 Window Menu Description

Menu entry	Description
Cascade	Use to arrange all open windows in cascade (overlapping).
Tile	Use 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 debugger window when you modify debugger window size.
Options - Component Menu	Select to display the component menu in the main menu when you select a component. For example, if you select the Source window, the Source menu displays in the main menu.
Layout - Load/ Store	Option to **Load / Store** your arrangements from a `.HWL` file.

NOTE Autosize and Component Menu are checked by default.

Help Menu

This is the Debugger Main window Help menu (Figure 2.23). Table 2.8 describes menu entries.

Figure 2.23 Help Menu

Table 2.8 Help Menu Description

Menu entry	Description
Help Topics	Choose to activate online help for specific information about a topic.
About	Displays information about debugger version, copyright, and license.

About Box

Select **Help > About** to display the About box. The about box lists directories for the current project, system information, program information, version number, copyright and registration information.

For more information on all components, click on the **Extended Information** button. Two hypertext links allow you to send an E-mail for a license request or information, and open the Freescale internet home page. Click **OK** to close this dialog box.

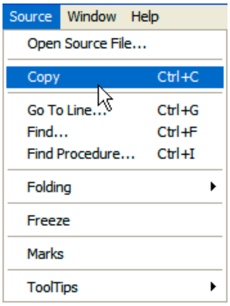
Component Associated Menus

Various Debugger Component windows are shown in Figure 2.1. Each component window has two menus. One menu is in the main menu and the other one is a context menu (also called **Associated Context Menu**) that you can open by right-clicking in an active window component.

Component Main Menu

This menu, shown in Figure 2.24, is always between the Component entry and the Window entry of the Debugger main window toolbar. It contains general entries of the current active component. Hide this menu by clearing **Window > Options > Component Menu**.

Figure 2.24 Example of Source Component Main Menu



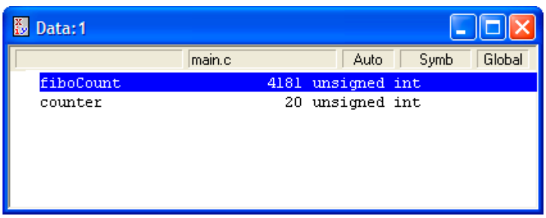
Component Files

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

Component Windows Object Information Bar

The object information bar of the debugger window, shown in Figure 2.25, provides information about the selected object.

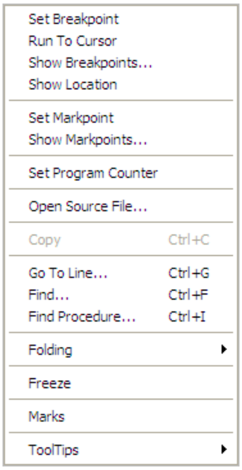
Figure 2.25 Object Information Bar of Debugger Component Windows



Component Context Menu

The context menu is a dynamic context-sensitive menu. It contains entries for additional facilities available in the current component. Context menus differ depending on the position of the mouse in the window. For example, if you click the mouse on a breakpoint, menu options allow you to delete, enable, or disable the breakpoint.

Figure 2.26 Example of a Component's Context Menu



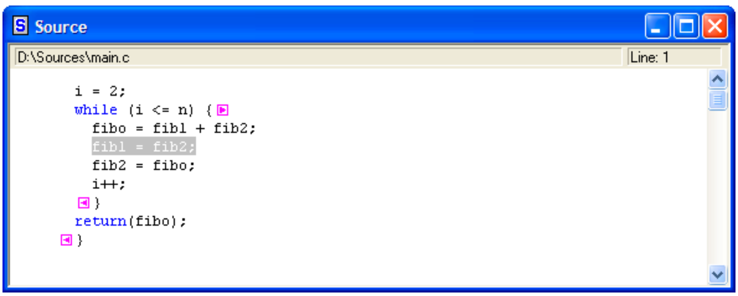
Features of the User Interface

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

Activating Services with Drag and Drop

You can activate services by dragging objects from one component window to another. This is known as drag and drop. Figure 2.27 shows an example.

Figure 2.27 Drag and Drop Example

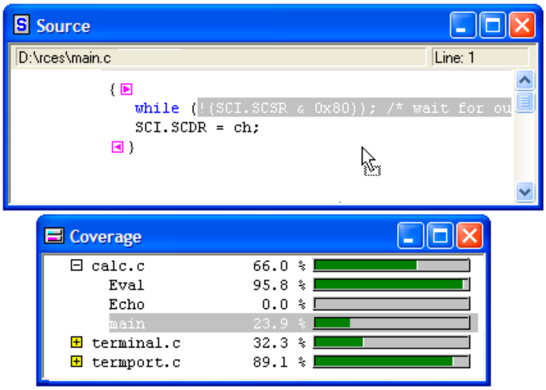


When an item cannot be dropped into a specific destination, the following cursor symbol appears: 

Example

Activate the display of coverage information on assembler and C statements by dragging the chosen procedure name from the Coverage component to the Source and Assembly components (Figure 2.28).

Figure 2.28 Dragging Procedure Name from Coverage to Source Component Window



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

Drag and Drop an Object

To drag an object from one component window to another:

1. Select the component containing the object you want to drag.
2. Make sure the destination component window to which you want to drag the object is visible.
3. Select the object you want.
4. Click and hold the left mouse button and drag the object into the destination component window.
5. Release the mouse button.

Drag and Drop Combinations

This section describes the possible combinations of drag and drop between components and associated actions. Dragging and dropping objects between different component windows is explained in each component description section.

Dragging from Assembly Component Window

Table 2.9 summarizes dragging from the Assembly Component.

Table 2.9 Dragging from the Assembly Component Window

Destination Component Window	Action
Command Line	Appends address of selected instruction to current command.
Memory	Dumps memory starting at selected instruction program counter (PC). Select PC location in Memory component.
Register	Loads destination register with PC of selected instruction.
Source	Source component scrolls to source statement and highlights it.

Dragging from Data Component Window

Table 2.10 summarizes dragging from the Data Component.

Table 2.10 Dragging from the Data Component Window

Destination Component Window	Action
Command Line	Appends address range of variable to current command in Command Line window. Dragging appends variable value to current command in Command Line window.
Memory	Dumps memory starting at the address where selected variable is located. Selects the memory area where the variable is located in memory component.
Register	Dragging the name loads destination register with address of selected variable. Dragging the value loads destination register with variable value.
Source	Dragging the name of a global variable in the source window displays the module in which the variable is defined. Source text is searched for the first occurrence of the variable and is highlighted.

NOTE It is not possible to drag an expression defined with the Expression Editor. The "forbidden" cursor appears.

Dragging from Source Component Window

Table 2.11 summarizes dragging from the Source Component.

Table 2.11 Dragging from the Source Component Window

Destination Component Window	Action
Assembly	Displays disassembled instructions starting at first high-level language instruction selected. Highlights assembler instructions corresponding to selected high-level language instructions in Assembly component.
Register	Loads destination register with PC of first instruction selected.
Memory	Displays memory area corresponding with selected high-level language source code. Memory area corresponding to selected instructions appears gray in memory component.
Data	A selection in the Source window is considered an expression in the Data window, as if entered through Data component Expression Editor (see Data Component and Expression Editor).

Dragging from the Memory Component Window

Table 2.12 summarizes dragging from the Memory Component.

Table 2.12 Dragging from the Memory Component Window

Destination Component Window	Action
Assembly	Displays disassembled instructions starting at first address selected. Highlights instructions corresponding to selected memory area in Assembly component.
Command Line	Appends selected memory range to Command Line window.

Table 2.12 Dragging from the Memory Component Window (*continued*)

Destination Component Window	Action
Register	Loads destination register with start address of selected memory block.
Source	Displays high-level language source code starting at first address selected. Instructions corresponding to selected memory area appear gray in the source component.

Dragging from Procedure Component Window

Table 2.13 summarizes dragging from the Procedure Component.

Table 2.13 Dragging from the Procedure Component Window

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

Dragging from Register Component Window

Table 2.14 summarizes dragging from the Register Component window.

Table 2.14 Dragging from the Register Component Window

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

Dragging from Module Component Window

Table 2.15 summarizes dragging from the Module Component.

Table 2.15 Dragging from the Module Component Window

Destination Component Window	Action
Data > Global	Displays global variables from selected module in data component.
Memory	Dumps memory starting at address of first global variable in module. Selects memory area where variable is located in the memory component.
Source	Displays source code from selected module.

Selection Dialog Box

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

Use this dialog box to do the following:

- Set Connection
- Open IO component
- Open Source File
- Open Module
- Open individual component windows

3

Debugger Components

This chapter explains how the different components of the Debugger work.

Debugger Kernel Components

The Debugger kernel includes various components. This section explains the types of components and their uses.

CPU Components

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

Window Components

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

Connection Components

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

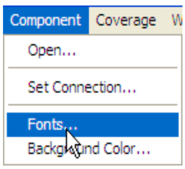
Different connections are available to connect the target system (hardware) to the debugger. For example, the connection may be connected using a Full Chip Simulator, an Emulator, a ROM monitor, a BDM pod cable, or any other supported device.

NOTE Connection components are covered in their respective manuals.

Loading Component Windows

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

Figure 3.1 Debugger Window Menu Bar

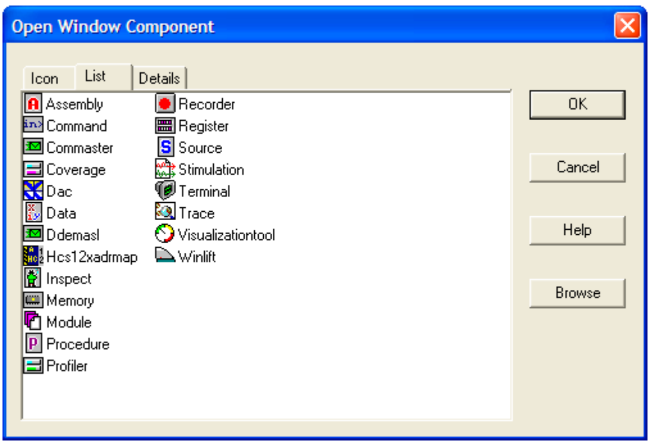


To open the window to choose one or more components:

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

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

Figure 3.2 Open Window Component Window



3. In the Open Window Component window, use the mouse to select a component.

4. Click the **OK** button to open the selected component.

There are three tabs in the Open Window Component window:

- The **Icon** tab shows components with large icons
- The **List** tab shows components with small icons
- The **Details** tab shows components with their descriptions

Multiple Component Windows

If you load a project that targets both HC12 and XGATE cores, the Debugger shows component windows as follows:

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

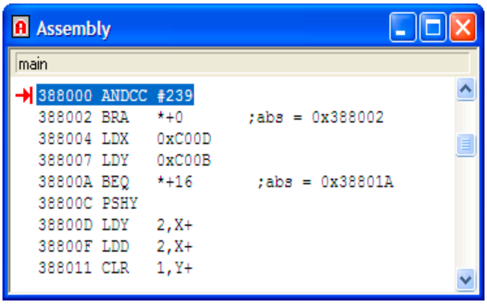
General Debugger Components

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

Assembly Component

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

Figure 3.3 Assembly Window



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

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

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

The **Object Information Bar** of the component window contains the procedure name, which contains the currently selected instruction. Double clicking a procedure in the Procedure component highlights the procedure's current assembly statement in the Assembly component.

Assembly Menu

The Assembly menu shown in Figure 3.4 contains all functions associated with the assembly component. Table 3.1 describes these entries.

Figure 3.4 Assembly Menu

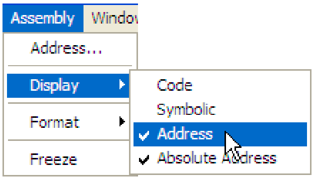


Table 3.1 Assembly Menu Description

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

Setting Breakpoints

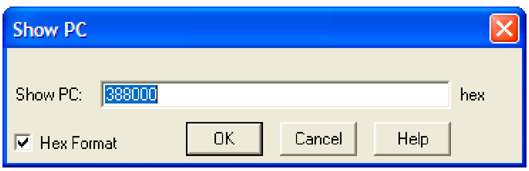
Use the context menu to set, edit and delete breakpoints. Right-click on any statement in the Source component window, then choose Set Breakpoint, Delete Breakpoint, etc.

NOTE For information on using breakpoints, see Control Points.

Show PC Dialog Box

If a hexadecimal address is entered in the **Show PC** dialog box shown in Figure 3.5, memory contents are interpreted and displayed as assembler instructions starting at the specified address.

Figure 3.5 Show PC Dialog Box



Associated Context Menu

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

Figure 3.6 Assembly Context Menu

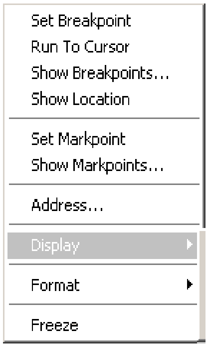


Table 3.2 Assembly Context Menu Description

Menu Entry	Description
Set Breakpoint	Appears in context menu if no breakpoint is set or disabled on specified instruction. Select to set a permanent breakpoint on instruction. When program execution reaches instruction, program halts and current program state displays in all window components.
Delete Breakpoint	Appears in context menu if a breakpoint is set or disabled on the specified instruction. Select to delete breakpoint.
Enable Breakpoint	Appears in context menu only if a breakpoint is disabled on an instruction. Select to enable breakpoint.
Disable Breakpoint	Appears in context menu if a breakpoint is set on an instruction. Select to disable breakpoint.
Run To Cursor	Select to set a temporary breakpoint on specified instruction and continue program execution. Disabling a permanent breakpoint at this position disables the temporary breakpoint as well and the program will not halt. Temporary breakpoints are automatically removed once reached.
Show Breakpoints	Opens Controlpoints Configuration Window Breakpoints Tab and displays list of breakpoints defined in application (refer to Control Points).
Show Location	Select to highlight source statement that generated the specified assembler instruction and the assembler instruction. Also highlights the memory range corresponding to this assembler instruction in memory component.

Table 3.2 Assembly Context Menu Description (*continued*)

Menu Entry	Description
Set Markpoint	Select to set a markpoint at this location.
Delete Markpoint	Appears in context menu only if a markpoint is set at the nearest code position (visible with marks). When selected, disables markpoint.
Show Markpoints	Opens Controlpoints Configuration Window Markpoints Tab and displays list of markpoints defined in application (refer to Control Points).
Address	Table 3.1 describes remaining context menu entries.

Retrieving Source Statement

Retrieve a source statement using one of these methods:

- Point to an instruction in the Assembly component window, drag and drop it into the Source component window.

  The Source component window scrolls to the source statement generating this assembly instruction and highlights it.

- Left click the mouse and click the L key.

  This highlights a code range in the Assembly component window corresponding to the first line of code selected in the Source component window in which the operation is performed. This line or code range is also highlighted.

Drag Out

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

Table 3.3 Assembly Component Drag Actions

Destination Component Window	Action
Command Line	The Command Line component appends the address of the specified instruction to the current command.
Memory	Dumps memory starting at the selected instruction PC. Selects the PC location in the memory component.

Table 3.3 Assembly Component Drag Actions (*continued*)

Destination Component Window	Action
Register	Loads the destination register with the PC of the selected instruction.
Source	Source component scrolls to the source statements and highlights it.

Drop Into

Table 3.4 shows the drop actions possible in the Assembly component.

Table 3.4 Drop Into Assembly Component

Source Component Window	Action
Source	Displays disassembled instructions starting at first high-level language instruction selected. Highlights assembler instructions corresponding to selected high-level language instructions in Assembly component.
Memory	Displays disassembled instructions starting at first address selected. In Assembly component, highlights instructions corresponding to selected memory area.
Register	Displays disassembled instructions starting at address stored in source register. Highlights instruction starting at address stored in register.
Procedure	In Assembly component, highlights current assembly statement inside procedure.

Demo Version Limitations

No limitations.

Associated Commands

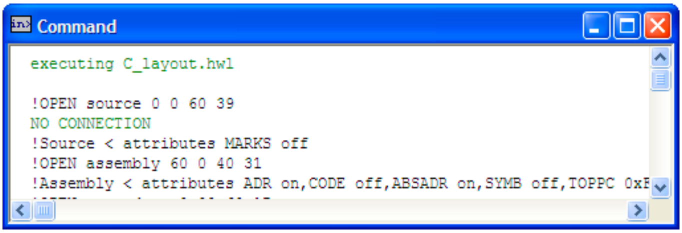
Following commands are associated with the Assembly component:

ATTRIBUTES, SMEM, SPC.

Command Line Component

The Command Line window shown in Figure 3.7 interprets and executes all Debugger commands and functions. The command entry always occurs in the last line of the Command component. Characters can be typed in or pasted on to the edit line.

Figure 3.7 Command Line Window



Keying In Commands

You can type Debugger commands after the **in>** terminal prompt in the Command Line Component window.

Recalling a Line from the Command Line History

To recall a command in the DOS window use the up or down arrow, or the **F3** function key, to retype the previous command.

Scrolling the Command Component Window Content

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

Clearing the Line or a Character of the Command Line

Selected text can be deleted by clicking the left arrow. To clear the current line, click the ESC key.

Command Interpretation

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

When a command executes and runs from the Command Line component, the component cannot be closed. In this case, closing the Command Line component with the window close button (X) or with the **Close** entry of the system menu displays the following message:

`Command Component is busy. Closing will be delayed`

The Command Line component closes as soon as command execution completes. Applying the CLOSE command to this Command Line component (for example, from another Command Line component), closes the component as soon as command execution finishes.

Variable Checking in the Command Line

When you specify a single name as an expression in a command line, the system checks for the expression in the following manner:

- First checked as a local variable in the current procedure.
- Next, as a global variable in the current module.
- Next, as a global variable in the application.
- Next, as a function in the current module.
- Then, as a function in the application,
- Finally if the expression is 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 you close the Command Line component with the close button or with the **Close** entry of the system menu, the following message displays:

`Command Component is busy. Closing will be delayed`

The Command component closes as soon as command execution completes. If you apply the **Close** command to this Command component, the Command component closes as soon as command execution completes.

Command Menu

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

Figure 3.8 Command Menu

Clicking **Execute File** opens a dialog in which you can select a file containing Debugger commands to be executed. These files generally have the `.cmd` default extension.

Copy selected text in the Command Line window to the clipboard by:

- Selecting the menu entry **Command > Copy**.
- Pressing **CTRL + C**
- 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.

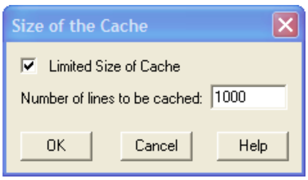
Paste the first line of text contained in the clipboard where the caret is blinking (end of current line) by:

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

Cache Size

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

Figure 3.9 Cache Size Dialog Box



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

Drag Out

Nothing can be dragged out.

Drop Into

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

Table 3.5 Drop Into Command Component

Source Component Window	Action
Assembly	Command Line component appends address of specified instruction to current command.
Data	Dragging the name appends the variable address range 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 selected memory range to Command Line window.
Register	Appends address stored in selected register to current command.

Demo Version Limitations

Only 20 commands can be entered and the command component closes. It is no longer possible to open a new command component in the same Debugger session.

NOTE Command files with more than 20 commands cannot be executed.

Associated Commands

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

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

ComMaster Component

The ComMaster component allows you to easily control one more debugger instance from the master debugger like you do it through the COM interface from within another application.

NOTE The ComMaster component is accessible through the debugger commands only. Its window is always minimized and has no associated menus.

Associated Commands

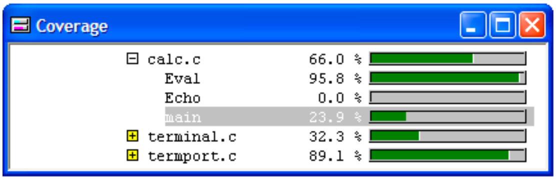
COM_START, COM_EXE, COM_EXIT

Coverage Component

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

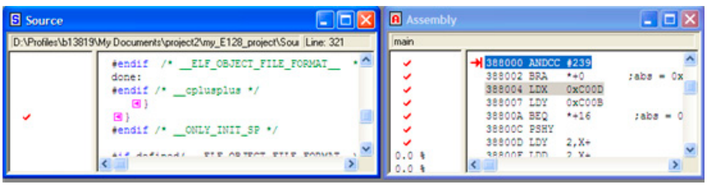
NOTE In cases of advanced code optimizations (like linker overlapping ROM/code areas) the coverage output/data is affected. In such a case, it is recommended to switch off such linker optimizations.

Figure 3.10 Coverage Window



The Coverage window contains percentage numbers and graphic bars. From this component, you can split views in the Source window and Assembly window, as shown in Figure 3.11. A red check mark is displayed in front of each source or assembler instruction that has been executed. Split views are removed when the Coverage window is closed or by selecting **Delete** in the split view context menu.

Figure 3.11 Split Views



Coverage Operations

Click the fold/unfold icons (⊞ ⊟) to unfold/fold the source module and display/hide the functions defined.

Coverage Menu

The **Coverage** menu and submenus are shown in Figure 3.12.

Figure 3.12 Coverage Menu

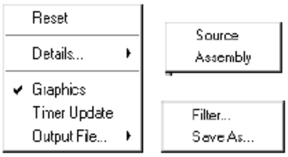


Table 3.6 Coverage Menu Description

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

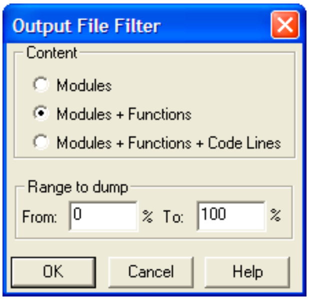
Output File

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

Output File Filter

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

Figure 3.13 Output File Filter Dialog Box



Output File Save

The **Save As** entry opens a **Save As** dialog in which you can specify the output file name and location. Listing 3.1 shows an example.

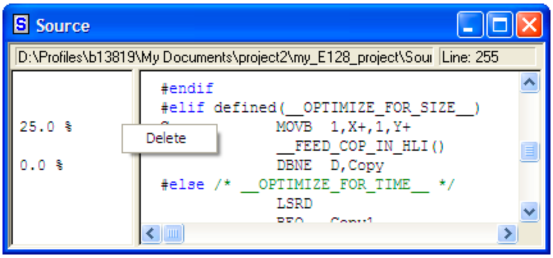
Listing 3.1 Example Output File with Modules and Functions

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

Split-View Associated Context Menu

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

Figure 3.14 Coverage Split-View Associated Context Menu



Drag Out

All displayed items can be dragged into a Source or Assembly component. The 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

Displays only modules and disables the Save function.

Associated Commands

DETAILS, FILTER, GRAPHICS, OUTPUT, RESET, TUPDATE

DA-C Link Component

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

Figure 3.15 DA-C Link Window

DA-C Link Operation

When you load the DA-C Link component, you establish communication 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 Synchronized Debugging through DA-C IDE.

DA-C Link Menu

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

Figure 3.16 DA-C Link Menu

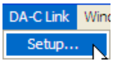


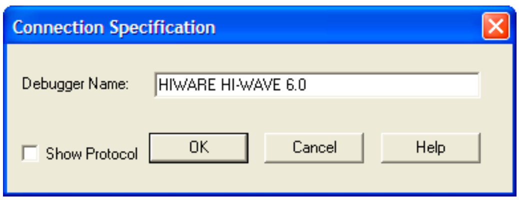
Table 3.7 DA-C Link Menu Description

Menu Entry	Description
Setup	Opens the Connection Specification dialog box.

Connection Specification Dialog Box

Set the DA-C debugger name in the **Connection Specification** dialog box.

Figure 3.17 Connection Specification Dialog Box



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

NOTE If problems exist, refer to Troubleshooting in the DA-C documentation.

Drag Out

Nothing can be dragged out.

Drop Into

Nothing can be dropped into the DA-C Component window.

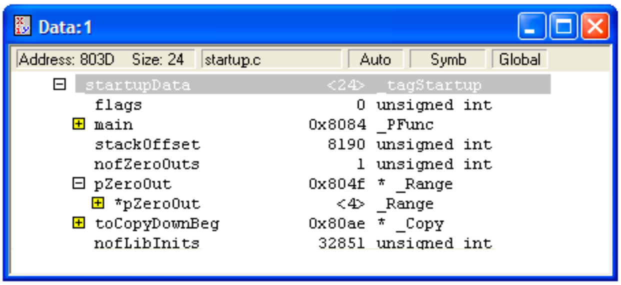
Demo Version Limitations

None.

Data Component

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

Figure 3.18 Data Window



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

The Component Windows Object Information Bar contains the address and size of the selected variable. It also contains the module name or procedure name in which 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 to which they point.

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

Data Operations

- Double click a variable line to edit the value.
- Click the fold/unfold icons to unfold/fold the structured variable.

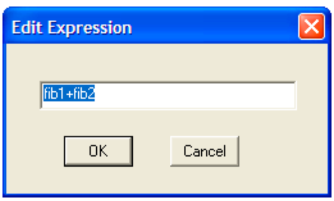
- Double click a blank line: Opens the Expression editor so you can insert an expression in the Data Component window.
- Select a variable in the Data component, and click the left mouse button + R key to set a *Read* watchpoint on the selected variable. A green vertical bar appears on the left side of the variables on which a read watchpoint is defined. If a read access on the variable is detected during execution, the program halts and the current program state displays in all window components.
- Select a variable in the Data component, and click the left mouse button + W key to set a *Write* watchpoint on the selected variable. A red vertical bar appears on the left side of the variables on which a write watchpoint is defined. If write access is detected on the variable during execution, the program halts and the current program state displays in all window components.
- Select a variable in the Data component, and click the left mouse button + B key to set a *Read/Write* watchpoint on the selected variable. A yellow vertical bar appears for the variables on which a read/write watchpoint is defined. If the variable is accessed during execution, the program halts and the current program state displays in all window components.
- Select a variable on which a watchpoint was previously defined in the Data component, and click the left mouse button + D key to delete the watchpoint on the selected variable. The vertical bar previously displayed for the variables is removed.
- Select a variable in the Data component, and click the left mouse button + S key to set a watchpoint on the selected variable. The Watchpoints Setting dialog box opens. A grey vertical bar appears for the variables on which a watchpoint is defined.

Expression Editor

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

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

Figure 3.19 Edit Expression Dialog Box



Example

With two variables **variable_1**, **variable_2**;

Entering the expression **(variable_1<<variable_2)+ 0xFF) <= 0x1000** results in a boolean type.

Entering the expression **(variable_1>>~variable_2)* 0x1000** results in an integer type.

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

Expression Command File

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

Example When loading `fibo.abs`, the debugger executes `Fibo.xpr`

Data Menu

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

Figure 3.20 Data Menu

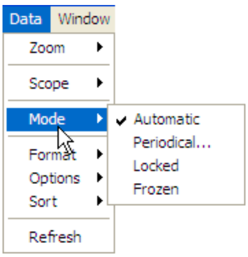


Table 3.8 Data Menu Entry Description

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

Scope Submenu

Activate the Scope submenu by highlighting the Scope entry on the Data menu.

Figure 3.21 Scope Submenu

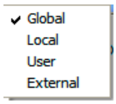


Table 3.9 describes the Scope submenu entries.

Table 3.9 Scope Submenu Entries

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

NOTE If the data component mode is not automatic, entries are gray (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

Activate the Format submenu by highlighting the format entry on the Data menu.

Figure 3.22 Format Submenu

Table 3.10 describes the Format submenu entries.

Table 3.10 Format Submenu Entries

Menu Entry	Description
Selected	Applies the changes to the selection only
All	Applies the changes to all items

Format Selected and Format All Submenu

Activate the Format Selected and Format All submenu by highlighting this entry on the Data Component menu.

Figure 3.23 Format Selected and All Submenus

Table 3.11 describes the Format Selected Mode and Format All Mode submenu entries.

Table 3.11 Format Selected and All Submenu

Menu entry	Description
Symbolic	Selects **Symbolic** display format (display format depends on variable type). Default display.
Hex	Selects hexadecimal data display format.
Bin	Selects binary data display format.
Oct	Selects octal data display format.
Dec	Selects signed decimal data display format.
UDec	Selects unsigned decimal data display format.
Bit Reverse	Selects bit reverse data display format (reverse each bit).

Mode Submenu

Activate the Mode submenu by highlighting the **Mode** entry on the Data menu.

Figure 3.24 Mode Submenu

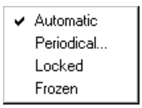


Table 3.12 describes the Mode submenu entries.

Table 3.12 Mode Submenu

Menu Entry	Description
Automatic	Switches to **Automatic** mode (default): updates variables when connection stops. Displays variables from currently executed module or procedure in data component.
Periodical	Switches to **Periodical** mode: updates variables at regular time intervals when connection is running. The default update rate is 1 second, but can be modified by steps of up to 100 ms using the associated dialog box (see below).

Table 3.12 Mode Submenu (*continued*)

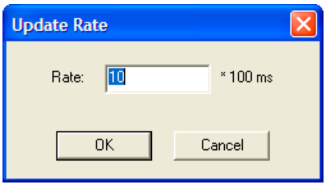
Menu Entry	Description
Locked	Switches to **Locked** mode: updates values from variables displayed in data component when connection stops.
Frozen	Switches to **Frozen** mode: Does not update values from variables displayed in data component when the connection stops.

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

Update Rate Dialog Box

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

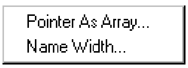
Figure 3.25 Update Rate Dialog Box



Options Submenu

Activates the Options submenu by highlighting the Options entry on the Data menu.

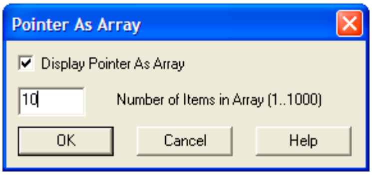
Figure 3.26 Options Submenu



Pointer as Array Option

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

Figure 3.27 Pointer as Array Dialog Box

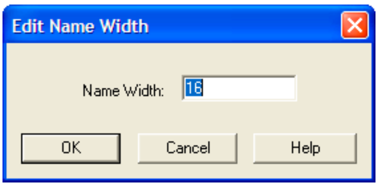


Within this dialog box, you can display pointers as arrays, assuming that the pointer points to the first item (**pointer[0]**). Note that this setup is valid for all pointers displayed in the Data window. Check the **Display Pointer as Array** checkbox and set the number of items that you want to be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

Name Width Option

In the Data Menu's Options submenu, choose **Options > Name Width** to open the dialog box shown in Figure 3.28.

Figure 3.28 Edit Name Width Dialog Box



This dialog box allows you to adjust the width of the variable name displayed in the Data window. Maximum name width is 16 characters. By increasing the value you can adapt the window to longer names.

Zoom and Sort Submenus

Figure 3.29 Zoom and Sort Submenus

Associated Context Menu

Figure 3.30 Data Context Menu

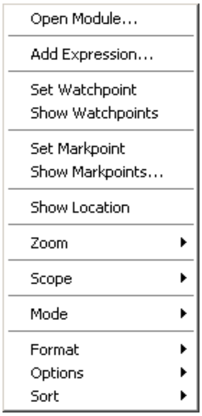


Table 3.13 describes the Data context menu entries.

Table 3.13 Data Context Menu

Menu Entry	Description
Open Module	Opens the Open Module dialog box.
Set Watchpoint	Appears only in context menu if no watchpoint is set or disabled on specified variable. When selected, sets a read/write watchpoint on this variable. Displays a yellow vertical bar for the variables on which a read/write watchpoint is defined. If variable is accessed during execution, the program halts and current program state displays in all window components.
Delete Watchpoint	Only appears in context menu if a watchpoint is set or disabled on the specified variable. When selected, deletes this watchpoint.
Enable Watchpoint	Only appears in context menu if a watchpoint is disabled on the specified variable. When selected, enables this watchpoint.
Disable Breakpoint	Only appears in context menu if a breakpoint is set on the specified instruction. When selected, disables this watchpoint.

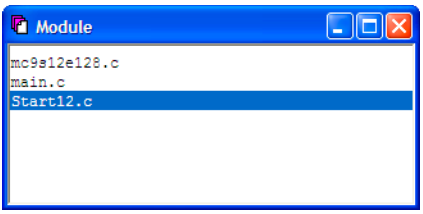
Table 3.13 Data Context Menu (*continued*)

Menu Entry	Description
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 specified variable (e.g., the Memory component selects memory range where variable is located).

Open Module Submenu

The dialog shown in Figure 3.31 lists all source files bound to the application. Displays global variables from the selected module in the data component. This is only supported when the component is in **Global** scope mode.

Figure 3.31 Open Modules Dialog Box



Drag Out

Table 3.14 describes the drag actions possible from the Data component.

Table 3.14 Dragging Data Possibilities

Destination Component Window	Action
Command Line	Dragging the name appends the address of the variable to the current command in the Command Line Window. Dragging the value appends the variable value to the current command in the Command Line Window.
Memory	Dumps memory starting at the address at which selected variable is located. Selects memory area at which the variable is located in memory component.

Table 3.14 Dragging Data Possibilities (*continued*)

Destination Component Window	Action
Source	Dragging the name of a global variable in source Window displays the module at which the variable is defined and highlights first occurrence of the variable.
Register	Dragging the name loads the destination register with the address of the selected variable. Dragging the value loads the destination register with the value of the variable.

NOTE It is important to distinguish between dragging a variable name and dragging a variable value. Both operations are possible. Dragging the name drags the address of the variable. Dragging the variable value drags the value.

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

Drop Into

Table 3.15 describes the drop actions possible in the Data component.

Table 3.15 Data Drop Possibilities

Source Component Window	Action
Source	A selection in the Source window is considered an expression in the Data window, as if entered through the Data component Expression Editor. Refer to Data Component, Expression Editor.
Module	Displays global variables from the selected module in Data component.

Demo Version Limitations

Only two variables can be displayed.

Only two members of a structure are visible when unfolded.

Only one expression can be defined.

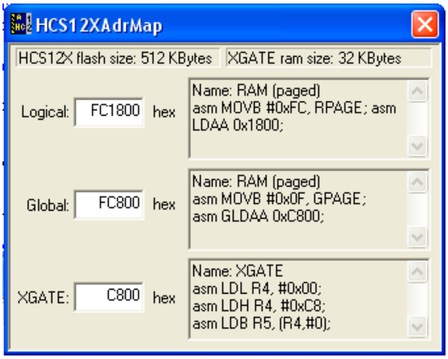
Associated Commands

ADDXPR, ATTRIBUTES, DUMP, PTRARRAY, SMOD, SPROC, UPDATERATE, ZOOM.

HCS12XAdrMap Component

The HCS12XAdrMap window, shown in the Figure 3.32 displays the address on Logical, Global and XGATE memory maps for HCS12X derivatives.

Figure 3.32 HCS12XAdrMap Window



The object information bar of the component window contains the derivative's memory settings.

HCS12XAdrMap Operations

Input the address into appropriate edit box in hex format. The rest two edit boxes will display representation of this address in the corresponding memory maps. If any of the edit boxes is empty that means that the address cannot be mapped to the corresponding memory map.

Text boxes in the right part of the component window display the following information for each memory map.

1. Name of the memory where the displayed address is located (Flash, Ram, etc.)
2. An example of assembly code that illustrates how to obtain data from the displayed address.

HCS12XAdrMap Menu

Figure 3.33 shows the HCS12XAdrMap menu.

Figure 3.33 HCS12XAdrMap Menu

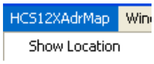


Table 3.16 describes HCS12XAdrMap menu entries.

Table 3.16 HCS12XAdrMap Menu Description

Menu Entry	Description
Show Location	Forces the Memory component to select data at the address displayed in the HCS12XAdrMap component window

Drag Out

NONE

Drop Into

Table 3.17 describes the drop actions possible in the HCS12XAdrMap component.

Table 3.17 HCS12XAdrMap Drop Possibilities

Source Component Window	Action
Assembly	Maps memory address at selected PC instruction.
Data	Maps memory address where selected variable is located.
Register	Maps memory address stored in selected register.

Demo Version Limitations

NONE

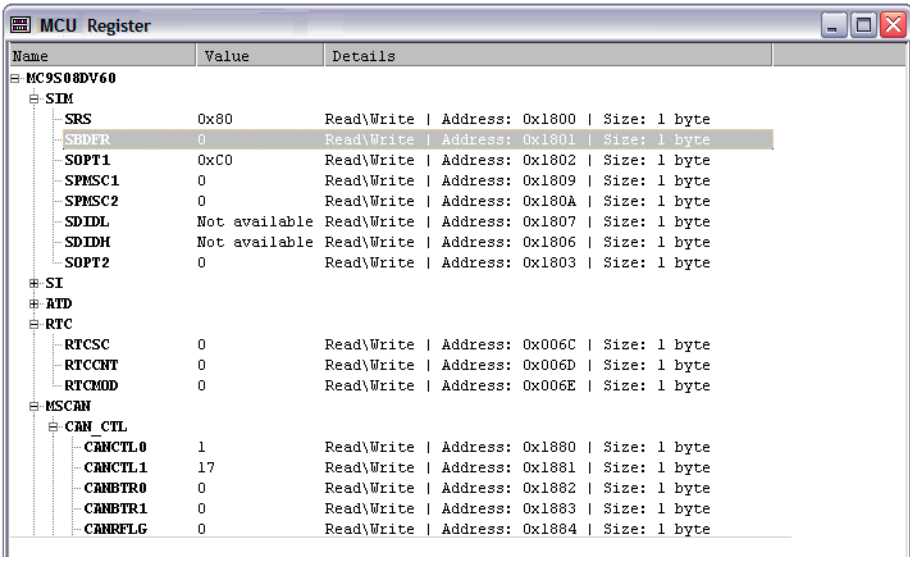
Associated Commands

NONE

MCURegisters Component

The MCURegisters window, shown in Figure 3.34 displays the names, values and details (access, size, address (id in case of CPU registers)) of CPU and device registers. The registers are arranged on the basis of groups and modules in tree view structure. The root item of the tree view contains the board name. The content of child node can be hidden or displayed by folding or unfolding corresponding parent node.

Figure 3.34 MCURegisters Window



The purpose of the MCURegisters component is to provide the user with convenient representation of the CPU and device registers. The changed register values are displayed in red. Register values can be displayed in binary, hexadecimal, octal, decimal or unsigned decimal format. When binary or hexadecimal format is set the values are formatted to the size of the register. These values can be edited.

Editing Registers

- To modify the value, double-click on a register to open an edit box.
- Click the **ESC key** to ignore changes and retain previous content of the register.
- Click the **Enter key** to confirm the chnages. If the new value is valid the register content is changed.

MCURegisters Menu

Figure 3.35 shows the MCURegisters component menu.

Figure 3.35 MCURegisters Menu

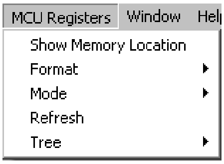


Table 3.18 describes MCURegisters Menu entries.

Table 3.18 MCURegisters Menu Entry Description

Menu Entry	Description
Show Memory Location	Forces the Memory component to select the memory range where the pointed register is located (applicable only for memory mapped registers).
Format	Displays Bin (binary), Hex (hexadecimal), Oct (octal), Dec (signed decimal), UDec (unsigned decimal) format.
Mode	Switches between Automatic and Periodical update mode.
Refresh	Refreshes the display.
Tree	Expands and collapses the whole register tree.

Format Submenu

Figure 3.36 shows the Format submenu. The Format submenu is activated by highlighting the Format entry on the MCURegisters menu.

Figure 3.36 Format Submenu

Table 3.19 describes the Format submenu entries.

Table 3.19 Format Submenu Entries

Menu Entry	Description
Selected	Apply changes to the selection only.
All	Apply changes to all items.

NOTE Format can be applied to a register only. For board, module and group items Selected Format is not ticked, however after activating Selected Format is applied for all the child register items.

Format Selected and All Submenu

Figure 3.37 shows Format Selected and All submenu. The Format Selected and All submenu is activated by highlighting this entry on the MCURegisters component menu.

Figure 3.37 Format Selected and All Submenus

Table 3.20 describes the Format Selected and All Submenu entries.

Table 3.20 Format Selected and All Submenus

Menu Entry	Description
Bin	Select the binary MCURegisters display format.
Hex	Select the hexadecimal MCURegisters display format.
Oct	Select the octal MCURegisters display format.

Table 3.20 Format Selected and All Submenus

Menu Entry	Description
Dec	Select the signed decimal MCURegisters display format.
Udec	Select the unsigned decimal MCURegisters display format.

Mode Submenu

Figure 3.38 shows the Mode submenu. The Mode submenu is activated by highlighting the Mode entry on the MCURegisters menu.

Figure 3.38 Mode Submenu

Table 3.21 describes the Mode submenu entries.

Table 3.21 Mode Submenu Entries

Menu Entry	Description
Selected	Apply changes to the selection only.
All	Apply changes to all items.

NOTE The Selected Mode can be applied to a board, modules and register group items only. For register items selected mode is not ticked.

Mode Selected and All Submenu

Figure 3.39 shows Mode Selected and All submenu. The Mode Selected and All submenu is activated by highlighting this entry on the MCURegisters component menu.

Figure 3.39 Mode Selected and All Submenus

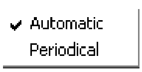


Table 3.22 describes the Mode Selected and All submenu entries.

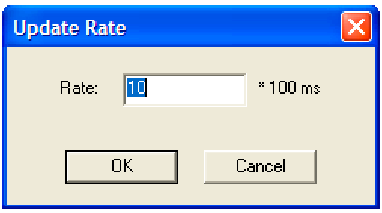
Table 3.22 Mode Selected and All Submenu

Menu Entry	Description
Automatic	Switches to Automatic mode (default); registers are updated when the connection is stopped.
Periodical	Switches to Periodical mode; registers are updated at regular time intervals when the connection is running. The default update rate is 1 second, but it can be modified by steps of up to 100 ms using the associated dialog box as shown in Figure 3.40.

Update Rate Dialog Box

Figure 3.40 shows Updat Rate dialog box. The Update Rate dialog box allows you to modify the default update rate using steps of 100 ms.

Figure 3.40 Update Rate Dialog Box



Tree Submenu

Figure 3.41 shows the Tree submenu. The Tree submenu is activated by highlighting the Tree entry on the MCURegisters menu.

Figure 3.41 Tree Submenu

Table 3.23 describes Tree submenu entries.

Table 3.23 Tree Submenu

Menu Entries	Description
Expand	Unfolds the whole register tree
Collapse	Folds the whole register tree

Drag Out

Table 3.24 describes the drag actions possible from the MCURegisters component.

Table 3.24 Dragging MCURegisters Possibilities

Destination Component Window	Action
Command Line	Dragging the register appends the register value to the current command in the Command Line Window.
Memory	Dumps memory starting at the address of the selected register value.
Register	Dragging the register loads the destination register with the value of the register.

Drop Into

NONE

Demo Version Limitations

NONE

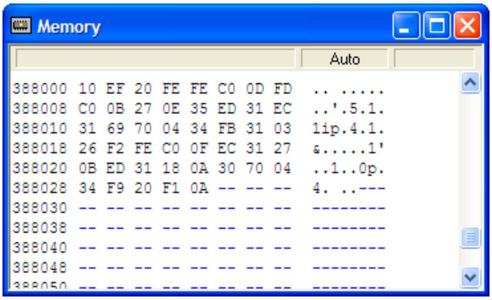
Associated Commands

DUMP, ATTRIBUTES, UPDATERATE, EXPAND, COLLAPSE

Memory Component

The Memory window shown in Figure 3.42 displays unstructured memory content, or memory dump, that is, continuous memory words without distinction between variables.

Figure 3.42 Memory Window



You can define watchpoints and specify various data formats (byte, word, double) and data displays (hexadecimal, binary, octal, decimal, unsigned decimal) for the display and editing of memory content.

NOTE Refer to Control Points for more information about watchpoints.

Use the Fill Memory Dialog Box box to initialize memory areas with a fill pattern.

Checking/unchecking **ASCII** in the **Display** menu entry adds or removes an ASCII dump on the right side of the numerical dump.

Checking/unchecking **Address** in the **Display** menu entry adds or removes the location address on the left side of the numerical dump.

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

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

"uu" memory value means: not initialized (for Simulation only).

"pp" memory value means: protected from being read, or protected from being read and written.

"rr" memory value means: not accessible because the hardware is running.

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

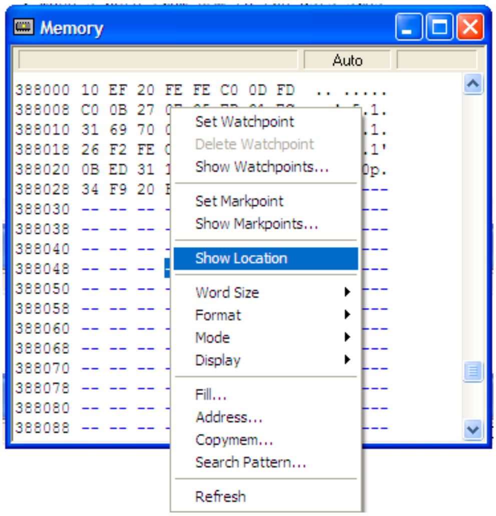
NOTE Memory values that have changed since the last refresh status are displayed in red. However, if a memory item is edited or rewritten with the same value, the display for this memory item remains black.

Memory Address Spaces

Some devices might have one or more additional address spaces. Select the Address Space menu entry to display the different address spaces in the Memory window.

TIP HCS12X devices have three address spaces. The **Logical** address space covers physical/local and logical displays (see Banked/Window Paged Memory: Physical/Local vs. Logical display for further details). The **Global** address space covers the Global Memory range (covering the memory as one single linear range), as accessed by Global core instruction set. The **XGATE** address space covers the memory as seen by the XGATE on-chip core.

Figure 3.43 Example: HCS12X Device Address Space Selection



Banked/Window Paged Memory: Physical/Local vs. Logical display

This section applies only to devices having on-chip program pages or data pages. For Legacy reasons, the debugger provides two ways to display the banked/window paged memory, such as the PPAGE window $8000-$BFFF range with HCS12 devices with on-chip banked memory, or EEPROM windows EPAGE selectable:

- The Debugging Memory Map (DMM) interface calls the default display the **physical** memory. Device specifications sometimes call the default display **local** memory, and it matches exactly what the CPU sees for silicon memory. This means that what displays in the Memory window at a specific suspended time (debugger halted) matches the current setup of page registers, like PPAGE or EPAGE for EEPROM. Changing the page registers, then refreshing the Memory window immediately shows changes in the window range.
- The **logical** display gives a constant Memory view at a specific address. For example, if we define, in a window address range, the concatenation of `PPAGE<<16` added with the physical/local address, we obtain a 24-bit address that does not represent anything for the CPU, but that is directly readable by the user in the Memory window.

By default, for 8/16-bit devices, the debugger displays memory addresses greater than address 0xFFFF as logical. These addresses no longer represent real addresses, but are required by the debugger to synchronize the program flow display and data accesses within all windows.

The debugger defines page range accessibility in the DMM interface. For 8/16-bit devices, window ranges in the physical/local memory $0000-$FFFF can be defined as logical in the DMM interface, to make them constant at display. For example, changing the $8000-$BFFF program window from physical to paged (or EEPROM paged for paged EEPROM) in the DMM graphical user interface makes the debugger display the PPAGE $00 instead of what the CPU sees, when looking at addresses in the $008000-$00BFFF range.

The default debugger display is mixed. You can change the display when you edit the module setup in the DMM interface. Refer to the Debugging Memory Map section for further details.

Memory Operations

- Double click a memory position to edit it. If the memory is not initialized, this operation is not possible.
- Drag the mouse in the memory dump to select a memory range.
- Hold down the left mouse button + A key to jump to a memory address. The specified value is interpreted as an address and the memory component dumps memory starting at this address.
- Select a memory range, and hold down the left mouse button + R key to set a **Read** watchpoint for the selected memory area. Memory ranges at which a read watchpoint is defined are underlined in green. If read access on the memory area is detected during execution, the program halts and the current program state displays in all window components.
- Select a memory range, and hold down the left mouse button + W key to set a **Write** watchpoint on the selected memory area. Memory ranges at which a write watchpoint is defined are underlined in red. If write access on the memory area is detected during execution, the program halts and the current program state displays in all window components.
- Select a memory range, and hold down the left mouse button + B key to set a **Read/ Write** watchpoint on the selected memory area. Memory ranges at which a read/ write watchpoint is defined are underlined in black. If the memory range is exceeded during execution, the program halts and the current program state displays in all window components.
- Select a memory range on which a watchpoint was previously defined, and hold down the left mouse button + D key to delete the watchpoint on the selected memory area. The underline disappears.
- Select a memory range, and hold down the left mouse button + S key to set a watchpoint on the selected memory area. The Watchpoints Setting dialog box opens. Memory ranges at which a watchpoint is defined are underlined in black.

Memory Menu

The Memory menu shown in Figure 3.44 provides access to memory commands. Table 3.25 describes the menu entries.

Figure 3.44 Memory Menu

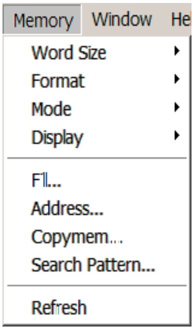


Table 3.25 Memory Menu Description

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

Word Size Submenu

With the Word Size submenu shown in Figure 3.45, you can set the memory display unit. Table 3.26 describes the menu entries.

Figure 3.45 Word Size Submenu

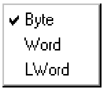


Table 3.26 Word Size Submenu Description

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

Format Submenu

With the Format submenu shown in Figure 3.46, you can set the memory display format. Table 3.27 describes the menu entries.

Figure 3.46 Format Submenu

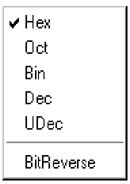


Table 3.27 Format Submenu Description

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

Table 3.27 Format Submenu Description (*continued*)

Menu Entry	Description
UDec	Selects unsigned decimal memory display format
Bit Reverse	Selects bit reverse memory display format (reverses each bit)

Mode Submenu

With the Mode submenu shown in Figure 3.47, you can set the memory mode format. Table 3.28 describes the menu entries.

Figure 3.47 Mode Submenu

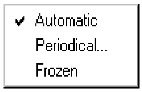


Table 3.28 Mode Submenu Description

Menu Entry	Description
Automatic	Selects **Automatic** mode (default). Updates memory dump when connection stops.
Periodical	Selects **Periodical** mode. Updates memory dump at regular time intervals while connection runs. Default update rate is 1 second, but can be modified by steps of up to 100 ms using associated dialog box.
Frozen	Selects **Frozen** mode. Does not update memory dump displayed in the memory component when connection stops.

Display Submenu

With the Display submenu shown in Figure 3.48, you can set the memory display (Address/ASCII). Table 3.29 describes the menu entries.

Figure 3.48 Display Submenu

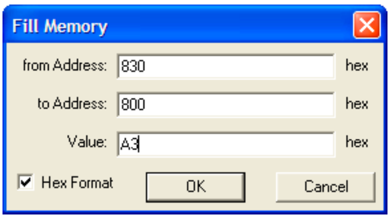
Table 3.29 Display Submenu Description

Menu Entry	Description
Address	Toggle the display of address dump.
ASCII	Toggle the display of ASCII dump.

Fill Memory Dialog Box

The Fill Memory dialog box shown in Figure 3.49 allows you to fill a memory range (**from Address** edit box and **to Address** edit box) with a bit pattern (**value** edit box).

Figure 3.49 Fill Memory Dialog Box



NOTE If *Hex Format* is checked, numbers and letters are interpreted as hexadecimal numbers. Otherwise, type expressions and prefix Hex numbers with **0x** or **$**.

Display Address Dialog Box

With the Display Address dialog box, shown in Figure 3.50, the memory component dumps memory starting at the specified address.

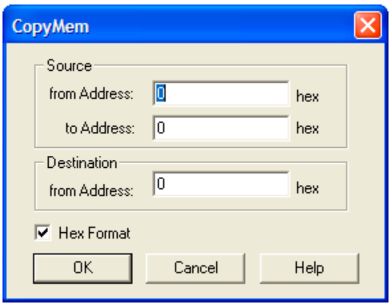
Figure 3.50 Display Address Dialog Box

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

CopyMem Dialog Box

The CopyMem dialog box shown in Figure 3.51 allows you to copy a memory range to a specific address.

Figure 3.51 CopyMem Dialog Box



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

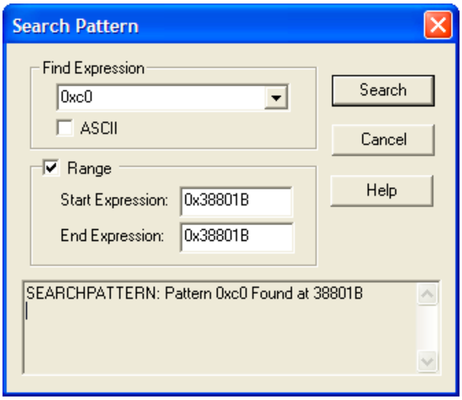
If you check **Hex Format**, all given values are in Hexadecimal Format. It is not necessary to add `0x`. For instance, type 1000 instead of 0x1000.

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

Search Pattern

The Search Pattern dialog box shown in Figure 3.52 allows you to search memory or a memory range for a specific expression.

Figure 3.52 Search Pattern Dialog Box



Using ANSI-C syntax, enter a list of hexadecimal bytes separated by white spaces (e.g., `0x0F 0x2F 0x20`) in the Find Expression text box. The hexadecimal string entered must be at least one byte.

When you check the ASCII checkbox, you can enter a text string in the text box (e.g., `my &%\ string`).

Check the Range checkbox and enter a Start Expression and an End Expression in the text fields. The string must be a hexadecimal value using ANSI-C syntax (e.g., `0xF000`).

NOTE Checking Range and using a Start Expression and an End Expression is recommended. Without these values, the debugger searches through the entire device memory mapped in the Memory window.

The lower part of the dialog box displays the search results at the end of the search, in the format: `SEARCHPATTERN: Pattern “my &%”\ string” Found at 20C0’L`.

Click Search button to start the search, or click Cancel to close the dialog box.

Refresh

Select the Refresh menu entry to refresh the Memory window current data cache. The debugger refreshes the data cache as if the debugger was halted or stepped.

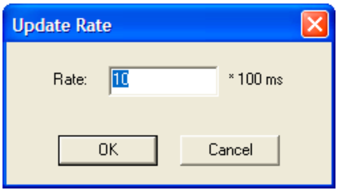
Only memory ranges defined with the **Refresh memory when halting** option in the Debugging Memory Map (DMM) interface will be refreshed. The Refresh menu entry addresses, by DMM factory setup, the volatile memory, i.e. the RAM and on-chip I/O Registers.

TIP To refresh other memory ranges, either set the Refresh memory when halting option for those ranges in the DMM dialog, or enter the `DMM RELEASECACHES` command in the Command window. You can disable caching for the debug session when entering the `DMM CACHINGOFF` command in the Command window.

Update Rate

This dialog box shown in Figure 3.53 allows you to modify the update rate in steps of 100ms.

Figure 3.53 Update Rate Dialog Box



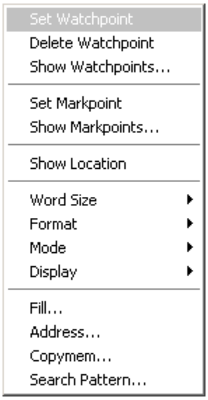
NOTE Periodical mode is not available for all hardware connections and some hardware connections require additional configuration to work.

When you set the **Refresh memory periodically when halted** checkbox, the debugger continues refreshing caches even if it is not running. This allows you to see I/O Register changes even if the CPU is not running.

Associated Context Menu

The memory context menu, shown in Figure 3.54, gives the user access to memory commands.

Figure 3.54 Memory Context Menu



The Memory context menu entries shown in Table 3.30 allow you to execute memory associated commands.

Table 3.30 Memory Context Menu Description

Menu Entry	Description
Set Watchpoint	Appears in context menu only if no watchpoint is set or disabled on selected memory range. When selected, sets a Read/Write watchpoint at this memory area. Memory ranges at which a read/write watchpoint is defined are underlined in yellow. If memory area is accessed during application execution, program halts and current program state displays in all window components.
Delete Watchpoint	Appears in context menu only if a watchpoint is set or disabled on selected memory range. When selected, deletes this watchpoint.
Show Watchpoints	When selected, brings up the Controlpoints Configuration Window - Watchpoints Tab. This is the interface through which watchpoints are controlled (see Control Points).
Set Markpoint	Appears in Context Menu only if no watchpoint is set or disabled on selected memory range. When selected, sets a Read/Write watchpoint at this memory area.
Show Markpoints	When selected, brings up Controlpoints Configuration Window - Markpoints Tab. This is the interface through which markpoints are controlled (see Control Points).

Table 3.30 Memory Context Menu Description (*continued*)

Menu Entry	Description
Show Location	Forces all opened windows to display information about selected memory area.
Word Size	Table 3.25 describes remaining menu entries.

Drag Out

Table 3.31 describes the drag actions possible from the Memory component.

Table 3.31 Memory Component Drag Possibilities

Destination Component Window	Action
Assembly	Displays disassembled instructions starting at first address selected. Highlights instructions corresponding to selected memory area in Assembly component.
Command Line	Appends selected memory range to Command Line window.
Register	Loads destination register with start address of selected memory block.
Source	Displays high-level language source code starting at first address selected. Instructions corresponding to selected memory area are gray in source component.

Drop Into

Table 3.32 shows the drop actions possible in the Memory component.

Table 3.32 Memory Component Drop Possibilities

Source Component Window	Action
Assembly	Dumps memory starting at selected PC instruction. Selects PC location in memory component.
Data	Dumps memory starting at address where selected variable is located. Selects memory area where variable is located in memory component.

Table 3.32 Memory Component Drop Possibilities (*continued*)

Source Component Window	Action
Register	Dumps memory starting at address stored in selected register. Selects corresponding address in memory component.
Module	Dumps memory starting at address of first global variable in module. Selects memory area where this variable is located in memory component.

Demo Version Limitations

No limitations.

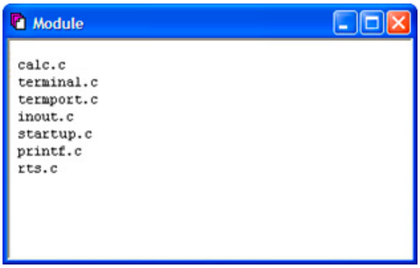
Associated Commands

ATTRIBUTES, FILL, SMEM, SMOD, SPC, UPDATERATE.

Module Component

The Module window shown in Figure 3.55 gives an overview of source modules building the application.

Figure 3.55 Module Window



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

Module Operations

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

Module Menu

The Module component window has no menu.

Drag Out

Table 3.33 shows the drag actions possible from the Module component.

Table 3.33 Module Component Drag Possibilities

Destination Component Window	Action
Data > Global	Displays global variables from selected module in data component.
Memory	Dumps memory starting at address of first global variable in module. Select memory area at which this variable is located in memory component.
Source	Displays source code from selected module.

Drop Into

Nothing can be dropped into the Module component window.

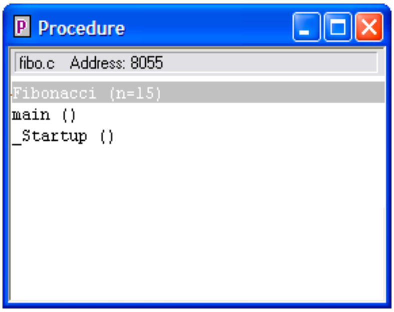
Demo Version Limitations

Displays only two modules.

Procedure Component

The Procedure window shown in Figure 3.56 displays the list of procedure or function calls that have been made up to the moment the program halts. This list is known as the *procedure chain* or the *call chain*.

Figure 3.56 Procedure Window



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

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

Procedure Operations

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

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

Procedure Menu

Figure 3.57 shows the Procedure menu and Table 3.34 describes its entries.

Figure 3.57 Procedure Menu

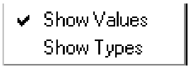


Table 3.34 Procedure Menu Description

Menu Entry	Description
Show Values	Displays function parameter values in procedure component.
Show Types	Displays function parameter types in procedure component.

Drag Out

Table 3.35 shows the drag actions possible from the Procedure component.

Table 3.35 Procedure Component Drag Possibilities

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

Drop Into

Nothing can be dropped into the Procedure component.

Demo Version Limitations

Displays only the last two procedures.

Associated Commands

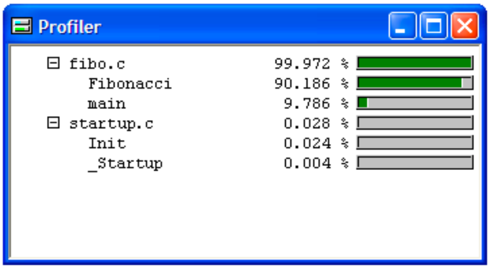
ATTRIBUTES, FINDPROC

Profiler Component

The Profiler window shown in Figure 3.58 provides information on application profile.

NOTE Advanced code optimizations (like linker overlapping ROM/code areas) affects the profiler output/data. In such cases, switching off such linker optimizations is recommended.

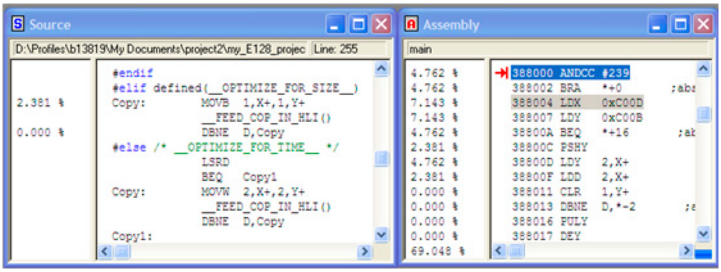
Figure 3.58 Profiler Window



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

The Profiler window can set a split view in the Source and Assembly windows, as shown in Figure 3.59. To obtain a split view in either the Source or Assembly windows, select: **Details > Source** or **Details > Assembly** or both from the Profiler menu and submenu. The split windows effect ends when you close the Profiler window.

Figure 3.59 Split View in the Source and Assembly Windows



Percentage values representing the time spent in each source or assembler instruction are displayed on the left side of the instruction. The split view can also display graphic bars. Split views close when you close the Coverage component or if you open the split view list menu and select **Delete**.

The value displayed may reflect percentages either from total code or from module code.

Profiler Operations

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

Profiler Menu

Figure 3.60 shows the Profiler menu entries, with the Details submenu and the Base submenu. Figure 3.61 shows the **Profiler Output File** submenu. Table 3.36 describes menu entries.

Figure 3.60 Profiler Menu and Submenus

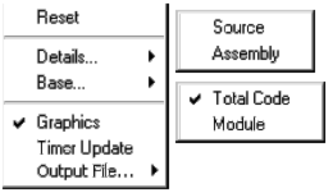


Figure 3.61 Profiler Output File Submenu

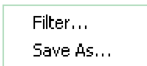


Table 3.36 Profiler Menu Entries Description

Menu Entry	Description
Reset	Resets all statistics.
Details	Sets a split view in chosen component (Source or Assembly)
Base	Sets base of percentage (total code or module code).
Graphics	Toggles display from graphics bar.
Timer Update	Switches periodic update of the Coverage component on or off. If activated, statistics update once per second.
Output File	Sets up Profiler Output File Functions.

Split View Associated Context Menu

Figure 3.62 shows the Profiler context menu, Table 3.37 describes the **Delete** and **Graphics** menu entries.

Figure 3.62 Profiler Split View Associated Context Menu

Table 3.37 Profiler Split View Associated Context Menu Description

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

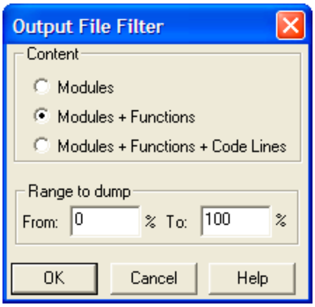
Profiler Output File Functions

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

Output File Filter

By choosing **Output Filter**, the dialog box shown in Figure 3.63 lets you select what you want to display, i.e. modules only, modules and functions, or modules, functions and code lines. You can also specify a range of coverage to be logged in your file.

Figure 3.63 Output File Filter Dialog Box



Output File Save

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

Associated Context 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

Displays only modules, and the Save function is disabled.

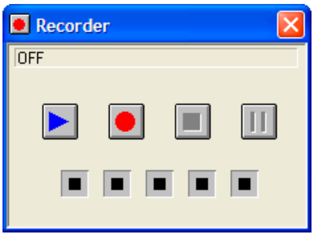
Associated Commands

GRAPHICS, TUPDATE, DETAILS, RESET, BASE.

Recorder Component

The Recorder window shown in Figure 3.64 provides record and replay facilities for debug sessions.

Figure 3.64 Recorder Window



The Recorder window enables the user to record and replay command files. The recorded file may also contain the command execution time.

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

Play **Record** **Stop** **Pause**

An animation occurs during recording, replaying, and pausing.

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

Recorder Operations

When the window is open but no record or play session is in progress, 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 recording session starts and the **Stop** button is enabled. Click the **Stop** button to end the recording 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 enables only the stop and pause buttons. Click the **Pause** button to stop file execution and enable the play and stop buttons. Click the **Play** button to resume file execution from the point at which it stopped. Click the **Stop** button to stop the replay session.

Terminal and TestTerm Record

Data typed in the Terminal component and TestTerm component is recorded during a recording session.

NOTE You must record the time as well to be able to replay the recording (**Record Time** menu entry of the recorder must be checked before recording).

Recorder Menu

The Recorder menu shown in Figure 3.65 changes according to the current session. Table 3.38 describes the menu items.

Figure 3.65 Recorder Menu

Table 3.38 Recorder Menu Description

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

The code in Listing 3.2 loads an `.abs` file, sets a breakpoint, and configures the assembly component to display the code and addresses. The **Data1** component switches the display to local variables, starts the application, and stops at the breakpoint.

Listing 3.2 Record File Example

```
at 4537 load C:\Freescale\DEMO\fibo.abs
at 9424 bs 0x1040 P
```

```
at 11917 Assembly < attributes code on
at 14481 Assembly < attributes adr on
at 20540 Data:1 < attributes scope local
at 24425 g
wait ;s
```

Drag Out

Nothing can be dragged out.

Drop Into

Nothing can be dropped in.

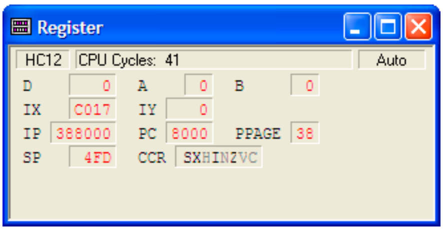
Demo Version Limitations

Records and replays only 20 commands.

Register Component

The Register window, shown in Figure 3.66, displays the content of registers and status register bits of the target processor.

Figure 3.66 Register Window



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

Status Register Bits

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

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

Editing Registers

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

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

Pressing the **Enter** key outside the edited register validates the new value and changes the register content.

Pressing the **Tab** key validates the new value, changes the register content, and selects the next register value for modification if desired.

Double clicking a status register bit toggles it.

Holding down the left mouse button and clicking the **A** key changes the contents of Source, Assembly and Memory component windows. The Source window shows the source code located at the address stored in the register. The Assembly window shows the disassembled code starting at the address stored in the register. The Memory window dumps memory starting at the address stored in the register.

Register Menu (Format Submenu)

The Register menu contains the items shown in Figure 3.67. Table 3.39 describes the menu entries.

Figure 3.67 Register Menu

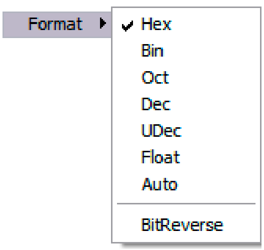


Table 3.39 Register Menu Description

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

Table 3.39 Register Menu Description (*continued*)

Menu Entry	Description
UDec	Selects unsigned decimal register display format
Float	Selects float register display format (displays all 32/64 bit registers as floats, all others as hex)
Auto	Selects auto register display format (displays all floating point 32/64 bit registers as floats, all others as hex)
Bit Reverse	Selects bit reverse data display format (reverses each bit)

Drag Out

Table 3.40 contains the drag actions possible from the Register window.

Table 3.40 Register Component Drag Possibilities

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

Drop Into

Table 3.41 shows the drop actions possible into the Register component.

Table 3.41 Register Component Drop Possibilities

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

Demo Version Limitations

No limitations.

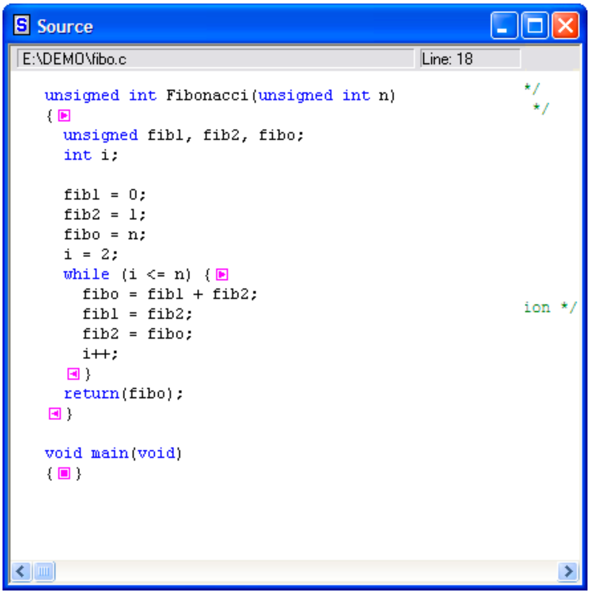
Associated Commands

ATTRIBUTES.

Source Component

The Source window shown in Figure 3.68 displays the source code of your program, i.e. your application file.

Figure 3.68 Source Window



The Source window allows you to view, change, monitor and control the current execution location in the program. The Source component window emphasizes language keywords, comments and strings with blue, green, and red, respectively. Select a word by double clicking it. Select a section of code by holding down the left mouse button and dragging the mouse.

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

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

When the source statement matching the current PC is selected in this window, (e.g., in a C source: `fib1 = fib2;`), the matching assembler instruction in the Assembler component window is also selected. The CPU executes this instruction next.

If breakpoints have been set in the program, a special symbol marks the breakpoint in the program source. The type of symbols depends on the types of breakpoint. For information on breakpoints, refer to Control Points. If execution stops, the current position is marked in the source component by highlighting the corresponding statement.

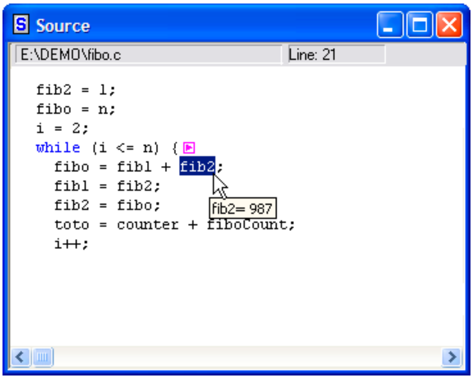
The complete path of the displayed source file is written in the Object Information bar of this window.

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

ToolTips Features

The Debugger source component provides tool tips to display variable values. The tool tip is a small rectangular pop-up window that displays the value of the selected variable (shown in Figure 3.69) or the parameter value and address of the selected procedure. Select a parameter or procedure by double clicking it.

Figure 3.69 ToolTips Features



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

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

Select **ToolTips > Format** from the source menu entry to select the tool tip display format (decimal, hexadecimal, octal, binary or ASCII).

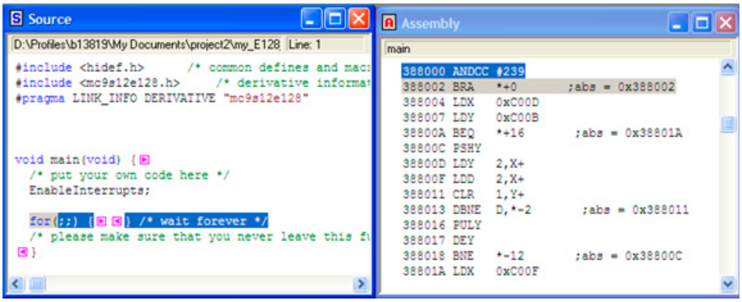
On-Line Disassembling

For information about performing on-line disassembly, refer to Consulting Assembler Instructions Generated by a Source Statement.

- Select a range of instructions in the source component and drag it into the assembly component. The corresponding range of code is highlighted in the Assembly component window, as shown in Figure 3.70.
- Holding down the left mouse button and clicking the T key highlights a code range in the Assembly component window corresponding to the first line of code selected in

the Source component window in which the operation is performed. This line or code range is also highlighted.

Figure 3.70 On Line Disassembling

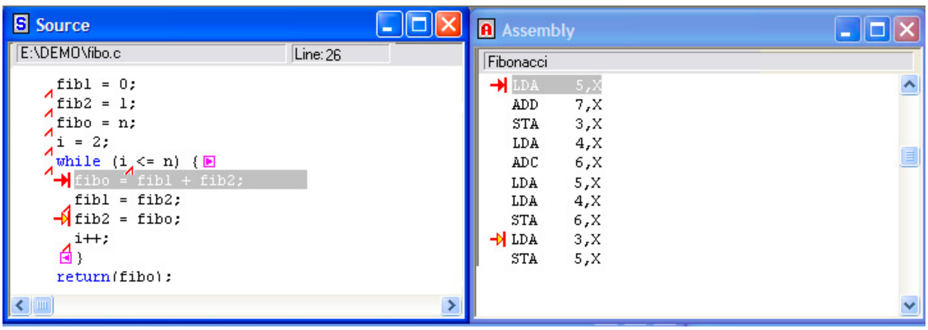


Setting Temporary Breakpoints

For information on how to set breakpoints refer to Control Points.

- Point to an instruction in the Source component window and click the right mouse button to display the Source window context menu. Select **Run To Cursor** from the context menu. The application continues execution and stops at this location.
- Holding down the left mouse button and pressing the T key sets a temporary breakpoint at the nearest code position (visible with marks). Thereafter the program runs and breaks at this location, as shown in Figure 3.71.

Figure 3.71 Setting Breakpoints



Setting Permanent Breakpoints

- Point to an instruction in the Source component Window and click the right mouse button to display the Source component context menu. Select **Set Breakpoint** from the context menu. This displays the permanent breakpoint icon in front of the selected source statement.

- Holding down the left mouse button and pressing the P key sets a permanent breakpoint at the nearest code position (visible with marks). The permanent breakpoint icon displays in front of the selected 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 unfolds at loading.

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

Sections of code that can be unfolded are hidden under .

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

Double click an unfolding mark ( ) to unfold the text that is hidden behind the mark.

Source Menus

Figure 3.72 shows the Source menu and Figure 3.73 shows the functions associated with the Source context menu. Table 3.42 describes these functions.

Figure 3.72 Source Menu

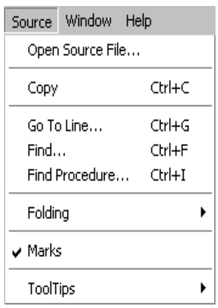


Figure 3.73 Source Associated Context Menu

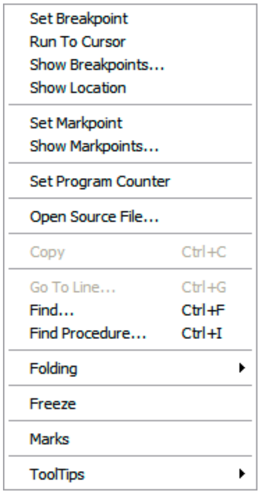


Table 3.42 Associated Context Menu Description

Menu Entry	Description
Set Breakpoint	Appears in context menu only if no breakpoint is set or disabled at nearest code position (visible with marks). When selected, sets a permanent breakpoint at this position. If program execution reaches this statement, program halts and current program state displays in all window components.
Delete Breakpoint	Appears in context menu only if a breakpoint is set or disabled at nearest code position (visible with marks). When selected, deletes this breakpoint.
Enable Breakpoint	Appears in context menu only if a breakpoint is disabled at nearest code position (visible with marks). When selected, enables this breakpoint.
Disable Breakpoint	Appears in context menu only if a breakpoint is set at nearest code position (visible with marks). When selected, disables this breakpoint.
Run To Cursor	When selected, sets a temporary breakpoint at nearest code position and continues program execution immediately. Disabling a breakpoint at this position disables the temporary breakpoint also and the program will not halt. Temporary breakpoints are automatically removed when reached.

Table 3.42 Associated Context Menu Description (*continued*)

Menu Entry	Description
Show Breakpoints	Opens Controlpoints Configuration window's Breakpoints Tab, which allows you to view the list of breakpoints defined in application and modify their properties (see Control Points).
Show Location	Highlights a code range in Assembly component window matching the line or selected source code. Highlights the line or the source code range as well.
Set Markpoint	Appears in context menu only if a markpoint is disabled at nearest code position (visible with marks). When selected, enables this markpoint.
Delete Markpoint	Appears in context menu only if a markpoint is set at nearest code position (visible with marks). When selected, disables this markpoint.
Show Markpoints	Opens Controlpoints Configuration Window's Markpoints Tab which allows you to view the list of markpoints defined in application and modify their properties (see Control Points).
Set Program Counter	Sets Program Counter to the address of selected source code.
Open Source File	Opens Source File dialog if a CPU is loaded (see next section).
Copy (CTRL+C)	Copies selected area of source component into the clipboard. Select a word by double clicking it. You can select a text area with the mouse by moving the pointer to the left of the lines until it changes to a right-pointing arrow, and then drag up or down; automatic scrolling is activated when the text is not visible in the windows.
Go to Line (CTRL+G)	Opens dialog box to scroll window to a number line.
Find (CTRL+F)	Opens a dialog box prompting for a string and then searches file displayed in source component. To start searching, click **Find Next**. Search begins at current selection or at first line visible in source component.
Find Procedure (CTRL+I)	Opens a dialog box for searching a procedure.
Foldings	Opens folding window (see chapter below)

Table 3.42 Associated Context Menu Description (*continued*)

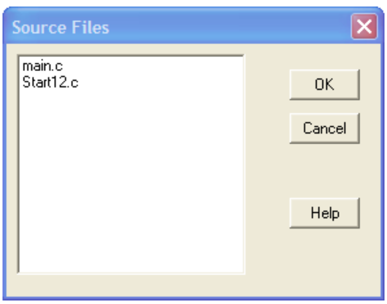
Menu Entry	Description
Marks	Toggles display of source positions where breakpoints may be set. If on, these positions are marked by small triangles.
ToolTips	Allows you to enable or disable the source tool tips feature, to set up the tool tip mode, and to set up tool tip format.

NOTE If some statements do not show marks although the mark display is switched on, the following 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 was never used.

Open Source File

The Open Source File dialog box shown in Figure 3.74 allows you to open the Source File (if a CPU is loaded). A source file is a file that has been used to build the currently loaded absolute file. An assembly file (`*.dbg`) is searched for in the directory given by the OBJPATH and GENPATH variables. C and C++ files (`*.c`,`*.cpp`,`*.h`, etc.) are searched for in the directories given by the GENPATH variable.

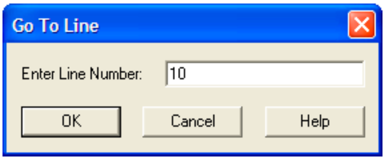
Figure 3.74 Open Source File Dialog Box



Go To Line

This menu entry is only enabled if a source file is loaded. It opens the dialog box shown in Figure 3.75. In this dialog box, enter the line number you want to go to in the source component: the selected line displays at the top of the source window. If the line number is incorrect, a message displays.

Figure 3.75 Go To Line Dialog Box



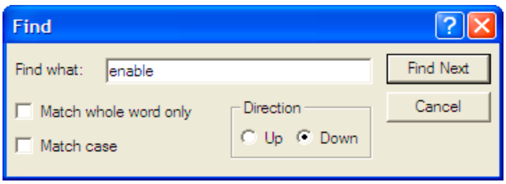
When this dialog box is open, the line number of the first visible line in the source is displayed and selected in the **Enter Line Number** edit box.

Find Operations

Use the Find dialog box, shown in Figure 3.76, 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 at 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 displays.

Figure 3.76 Find Dialog Box



This dialog box allows you to specify the following options:

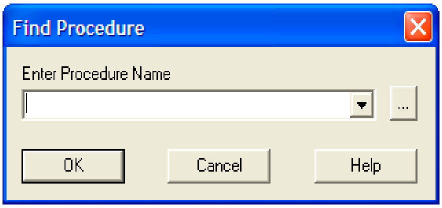
- **Match whole word only**: If this box is checked, only strings separated by special characters are recognized.
- **Match case**: If this box is checked, the search is case sensitive.

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 is copied into the **Find what** edit box.

Find Procedure

Use the Find Procedure dialog box, shown in Figure 3.77, to find the procedure name in the currently loaded application. Enter the procedure name you want to search for in the **Find Procedure** edit box. To start searching, click **OK**, the search starts at the current selection or at the first line visible in the source component when nothing is selected.

Figure 3.77 Find Procedure Dialog Box



If a valid procedure name is given as a parameter, the source file where the procedure is defined opens in the Source Component. The procedure's definition displays and highlights the procedure's title.

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

Folding Menu

The Folding menu shown in Figure 3.78 allows you to select the Fold functions described in Table 3.43.

Figure 3.78 Folding Menu

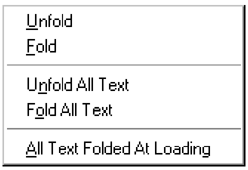


Table 3.43 Folding Menu Description

Menu Entry	Description
Unfold	Unfolds the displayed source code
Fold	Folds the displayed source code
Unfold All Text	Unfolds all displayed source code

Table 3.43 Folding Menu Description (*continued*)

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

Drag Out

Table 3.44 shows the drag actions possible from the Source component.

Table 3.44 Source Drag Possibilities

Destination Component Window	Action
Assembly	Displays disassembled instructions starting at first high-level language instruction selected. Highlights assembler instructions corresponding to selected high-level language instructions in Assembly component.
Register	Loads destination register with PC of first instruction selected.
Data	A selection in the Source window is considered an expression in the Data window, as if entered through the Expression Editor of the Data component (see Data Component or Expression Editor).

Drop Into

Table 3.45 shows the drop actions possible into the Source component.

Table 3.45 Source Drop Possibilities

Source Component Window	Action
Assembly	Source component scrolls to source statement corresponding with selected assembly instruction and highlights it.
Memory	Displays high-level language source code starting at first address selected. Instructions corresponding to selected memory area are gray in source component.
Module	Displays source code from selected module.

Demo Version Limitations

Displays only one source file of the currently loaded application.

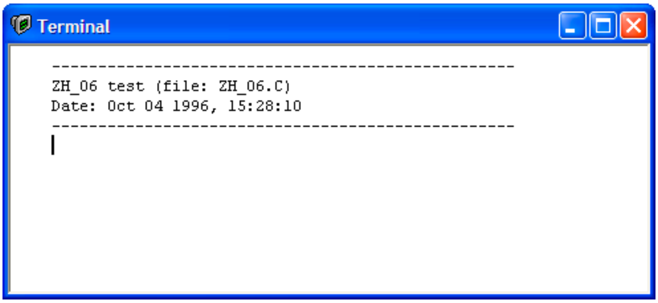
Associated Commands

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

Terminal Component

The Terminal Component window shown in Figure 3.79 can be used to simulate input and output. It can receive characters from several input devices and send them to other devices.

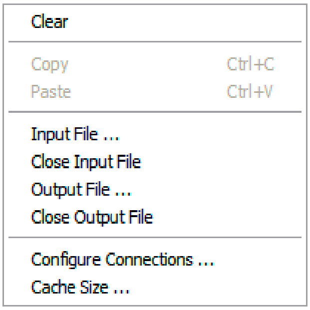
Figure 3.79 Terminal Window



You can use a virtual Serial Communication Interface (SCI) port provided by the framework for communication with the target, but it is also possible to use the keyboard, the display, some files or even the serial port of your computer as I/O devices.

To control and configure a terminal component use the Terminal menu of the terminal window, shown in Figure 3.80.

Figure 3.80 Terminal Context Menu

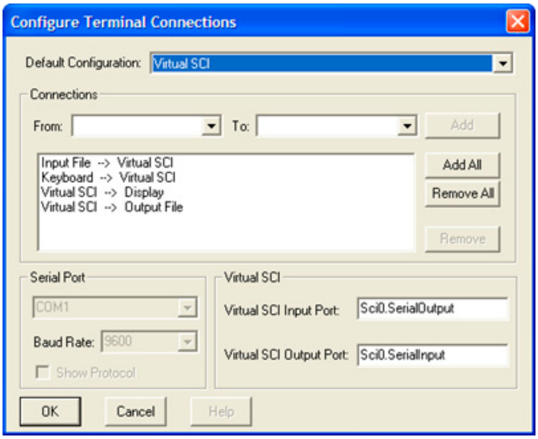


To open the context menu, right click in the terminal window.

Configure Terminal Connections

The terminal window is very flexible and can redirect characters received from any available input device to any available output device. You can specify these connections by choosing **Configure Connections** in the context menu of the terminal component. This opens the dialog box shown in Figure 3.81.

Figure 3.81 Configure Terminal Connections Dialog Box



You can simply choose one of the default configurations in the **Default Configuration** combo box. In the **Connections** section all active connections are listed in a list box. There you can customize which input devices will be redirected to which output devices by adding and removing connections.

To add a connection, specify the source and target devices using the **From** and **To** list boxes and then click the **Add** button. The new connection then appears in the list below, which shows all active connections.

To remove connections, select them in the list of active connections and click the **Remove** button.

In the **Serial Port** section you can specify which serial port to use and its properties. This is only possible if there is at least one connection from or to the serial port.

If a connection from or to the virtual SCI port has been chosen it is also possible to specify in the **Virtual SCI** section which ports will be taken as virtual SCI ports. This enables you to make a connection to any port in the Full Chip Simulation framework.

Input and Output File

It is also possible to take a file as an input stream for the terminal component or redirect the output to a file.

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 choosing **Input File** from the context menu. As soon as you click the **OK** button in the **File Open** dialog, input from the file starts. The file closes as soon as the end of file is reached or you choose **Close Input File** from the context menu.

When the input file reaches its end a CTRL-Z character (ASCII code 26 decimal) is sent to all output devices receiving characters from the input file, to notify them that the file transfer is finished.

You can redirect some input devices to an output file in a similar fashion. Make sure that you have chosen your connections from input devices to the output file. Then open or create your output file by choosing **Output File** from the context menu. If the file does not exist it is created. Otherwise you can choose to overwrite or append the existing file. To stop writing to the output file 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 3.46 illustrates the different possible commands and associated Escape sequences where *filename* is a sequence of characters terminated by a control character (e.g. CR) and is a valid filename, and ESC is the ESC Character (ASCII code 27 decimal).

Table 3.46 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.

You can give these commands 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. Listing 3.3 shows the source code in `terminal.c`.

Listing 3.3 TERM_Direct Source Code

```
void TERM_Direct(TERM_DirectKind what, const char* fileName) {
  /* sets direction of the terminal */
  if (what < TERM_TO_WINDOW || what > TERM_APPEND_FILE) return;
  TERM_Write(ESC); TERM_Write('h');
  TERM_Write((char)(what + '0'));
  if (what != TERM_TO_WINDOW && what != TERM_FROM_KEYS) {
    TERM_WriteString(fileName); TERM_Write(CR);
  }
}
```

In the example, the parameter `what` is one of the following constants:

- **TERM_TO_WINDOW**: send output to terminal window
- **TERM_TO_BOTH**: send output to file and window
- **TERM_TO_FILE**: send output to file `fileName`
- **TERM_FROM_KEYS**: read from keyboard (close input file)
- **TERM_FROM_FILE**: read input from file `fileName`
- **TERM_APPEND_BOTH**: append output to file and window
- **TERM_APPEND_FILE**: append output to file `fileName`

See also `terminal.h` for further details.

Using Virtual SCI

In its default **Virtual SCI** configuration, the terminal component accesses the target through the Object Pool interface.

To make the terminal component work in this default configuration, the target must provide an object with the name **Sci0**. If no **Sci0** object is available, no input or output happens. It is possible to check, through the Inspector component, if the environment currently provides an **Sci0** object.

NOTE Not all Full Chip Simulation components currently have an **Sci0** object. For all other Full Chip Simulation components the default virtual SCI port does not work unless a user-defined **Sci0** object with the specified register name is loaded.

Write access to the target application is done with the Object Pool function **OP_SetValue** at the address **Sci0.SerialInput**.

Input from the target application is handled with a subscription to an Object Pool register with the name **Sci0.SerialOutput**. When this register changes (sends a notification), a new value is received.

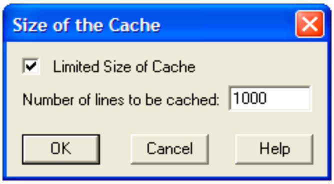
For implementations of this register with help of the **IOBase** class, use the **IOB_NotifyAnyChanges** flag. Otherwise only the first of two identical characters are received.

It is also possible to configure the terminal to use another object in the Object Pool instead of **Sci0** with which to communicate. Refer to Configure Terminal Connections for information about how to do this.

Cache Size

The item **Cache Size** in the context menu allows you to set the number of lines in the terminal window with the dialog shown in Figure 3.82.

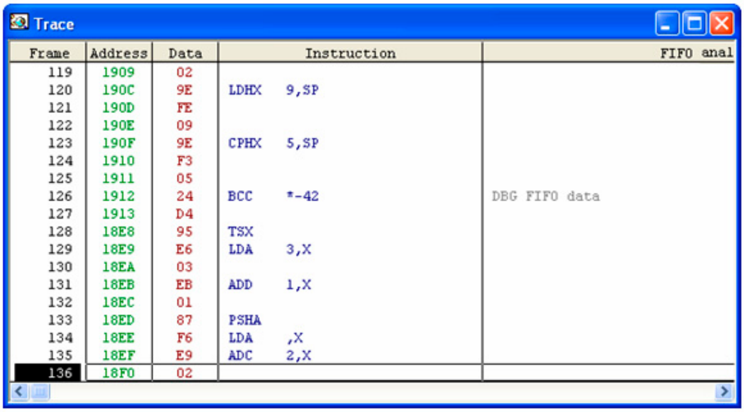
Figure 3.82 Size of the Cache Dialog Box



Trace Component

The Trace window shown in Figure 3.83 records and displays instruction frames and time or cycles if the information is available from the hardware. The capability of the trace component depends on the selected derivative and connection.

Figure 3.83 Trace Window



Trace Operations

Pointing at a frame and dragging the mouse forces all open windows to show the corresponding code or location. All other frames evaluate time and cycles relative to this base.

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

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

Trace Menu

The Trace menu shown in Figure 3.84 contains the functions described in Table 3.47. The exact content of the Trace menu can vary depending on the connection and derivative.

Figure 3.84 Trace Menu

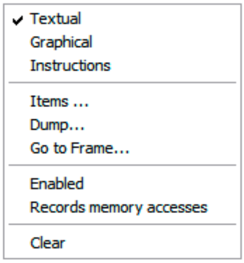


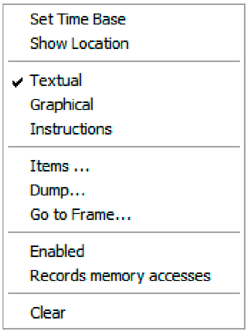
Table 3.47 Trace Menu Description

Menu Entry	Description
Textual	Displays window contents in text.
Graphical	Displays window content in graphical format.
Instructions	Displays instructions in window.
Items	Specifies the window display items.
Dump	Selects a file to dump or a range of frames to dump.
Go to Frame	Searches for specific frame.
Enabled	Disable or enable tracing function.
Records memory accesses	Enables recording of memory accesses.
Clear	Clears the trace comp

Associated Context Menu

The Trace context menu shown in Figure 3.85 contains functions described in Table 3.47 associated with the selected frame.

Figure 3.85 Trace Assocaited Context Menu



Selecting Show Location in the Trace window context-sensitive menu displays the frame matching source and assembly code in the Source and Assembly windows.

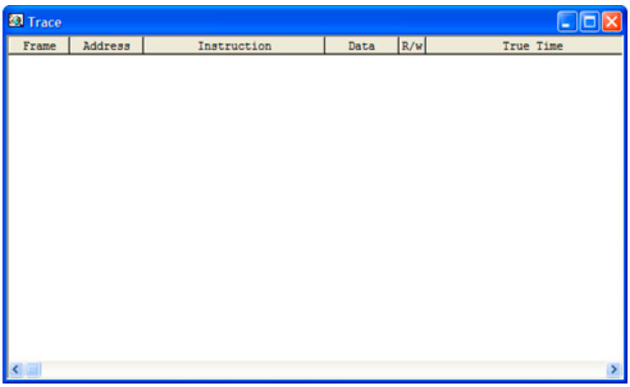
Display Modes

The information collected by the Trace component can be presented in different modes - Instructions, Textual and Graphical display.

Instructions Display

The Instruction display is the default display mode or you can switch to this mode by selecting Instructions in the Trace window menu. This mode provides the display of disassemble information.

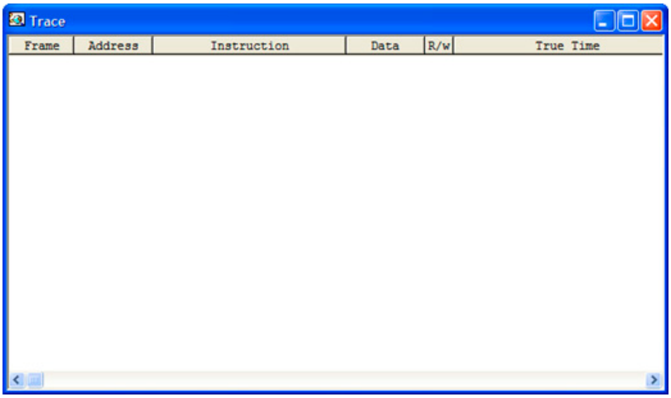
Figure 3.86 Trace Window - Instruction Display



Textual Display

Activate this display mode by selecting Textual in the Trace window menu. Textual display mode simply expands instruction assembly code in the Trace window.

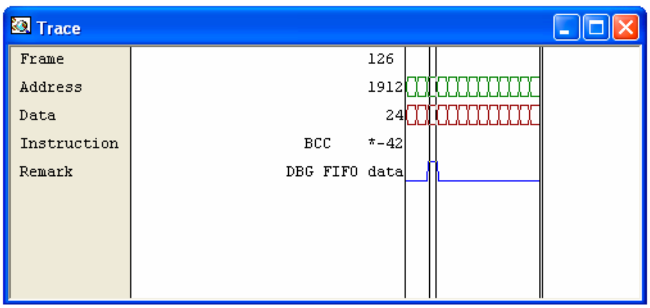
Figure 3.87 Trace Window - Textual Display



Graphical Display

Graphical Display mode provides a graphical representation of the same information. Activate this display mode by selecting Graphical in the Trace window menu.

Figure 3.88 Trace Window - Graphical Display



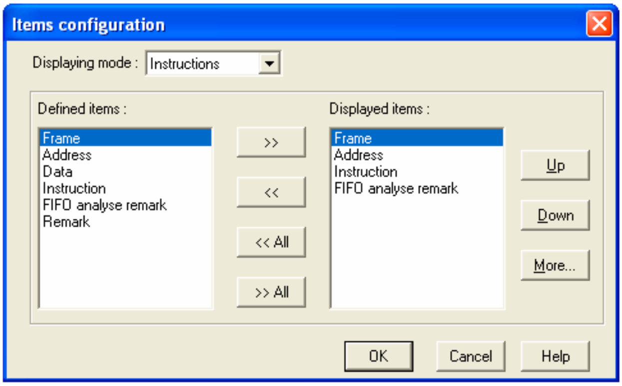
Column Display and Moving

The information columns shown by the Trace component can be configured with the configuration dialog shown on Figure 3.89. The Items menu item shall be used to open this dialog. This dialog allow to specify columns to view in each display mode. The

Displaying mode list allows selection for Textual, Instructions or Graphical mode. Use the right arrow to move items to the Displayed Items list, and the left arrow to hide the item. Moving the item Up in the list moves it to the left in the Trace component window.

Select More for more options. Select OK to save your changes.

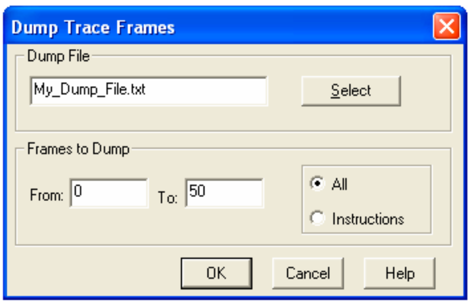
Figure 3.89 Items Configuration Dialog Box



Dumping Frames to File

Selecting Dump in the Trace window contest-sensitive menu opens a dialog that allows you to specify the number of Trace component frames to save, and the name of the text file to which to save the frames.

Figure 3.90 Dump Trace Frames Dialog Box



Go to Frame

Selecting Go to Frame in the Trace window context-sensitive menu opens a Search Frame dialog to allow you to look for a specific frame in the Trace window.

Drag Out

Nothing can be dragged out.

Drop Into

Nothing can be dropped in.

Demo Version Limitations

Limits the number of frames to 50.

Associated Commands

CLOCK, CYCLE, FRAMES, RECORD, RESET.

Visualization Utilities

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

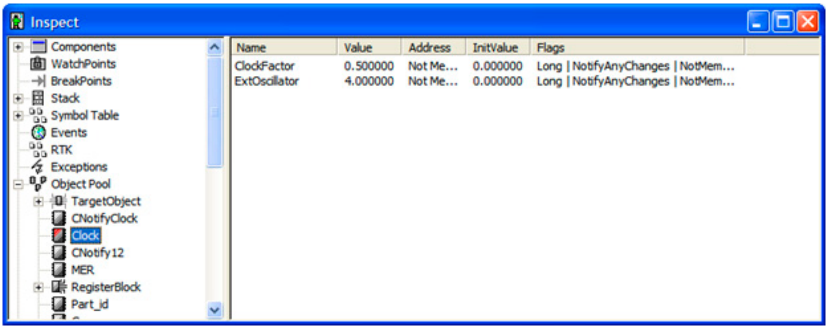
Among these components, there are visualization utilities that graphically display values, registers, and memory cells, or provide an advanced graphical user interface to simulated I/O devices, program variables, and so forth.

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

Inspect Component

The Inspect window shown in Figure 3.91 displays information about loaded components, the visible stack, pending events, pending exceptions and loaded I/O devices.

Figure 3.91 Inspect Component Window



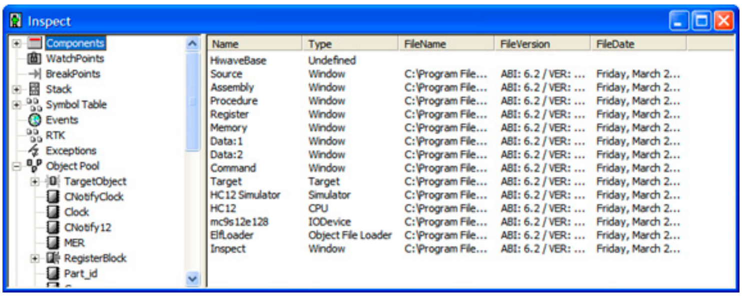
The hierarchical content of the items displays in a tree structure. Select any item on the left side and additional information displays on the right side.

In the figure above, for example, the Object Pool is expanded. The Object Pool contains the TargetObject, which contains LEDs and Swap peripheral devices. Select the Swap peripheral device to display Swap device registers.

Components Icon

Select the components icon in the Inspect window, as shown in Figure 3.92, and the right side displays various information about all loaded components. A Component is the unit of dynamic loading, therefore all windows, the CPU, the connection and perhaps the connection simulator are listed.

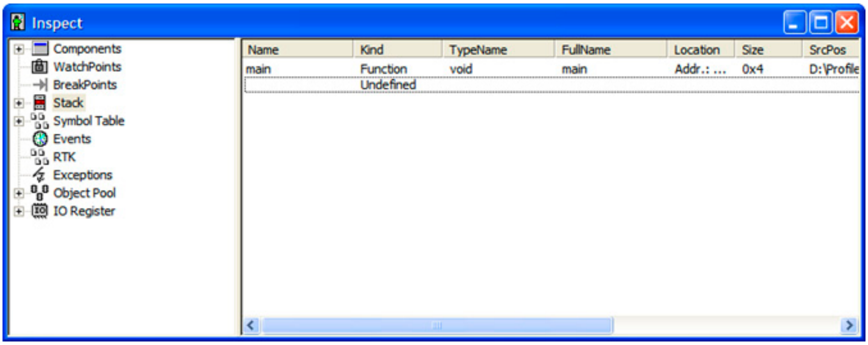
Figure 3.92 Inspect Window Components Icon



Stack Icon

The Stack icon shown in Figure 3.93 displays the current stack trace. Every function on the stack has a separate icon on the trace. In the stack-trace, the content of a local variable is accessible.

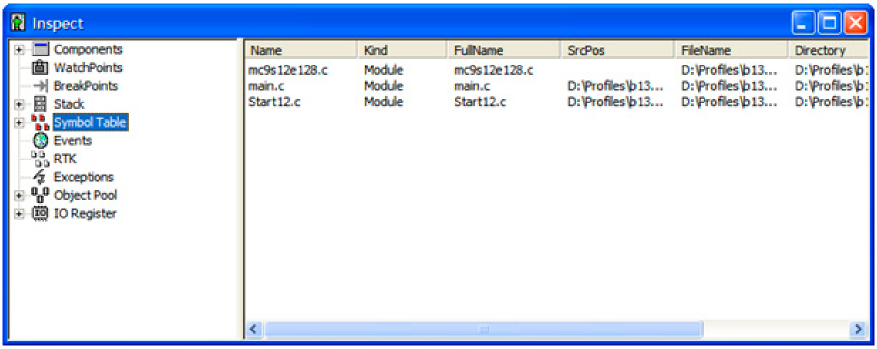
Figure 3.93 Inspector Window Stack Icon



Symbol Table

The symbol table shown in Figure 3.94 displays all loaded symbol table information in raw format. There are no stack frames associated with functions, therefore the content of local variables is not displayed. Global variables and their types are displayed.

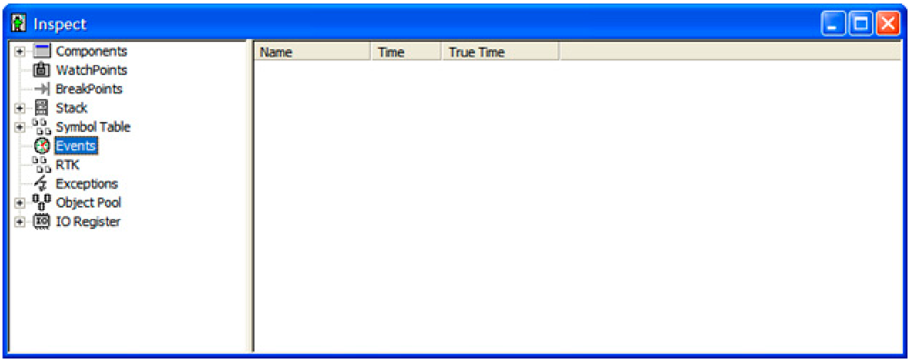
Figure 3.94 Inspector Window Symbol Table



Events Icon

The Inspect window Events icon shown in Figure 3.95 shows all currently installed events. Events are handled by peripheral devices, and notified at a given time. The Event display shows the name of the event and remaining time until the event occurs.

Figure 3.95 Inspector Window Events Icon



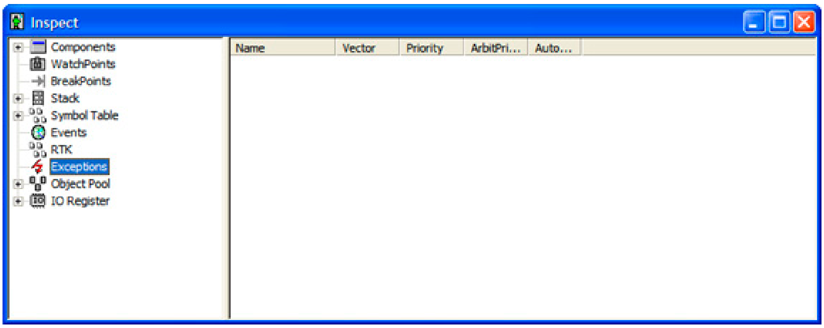
Events are only used in the HC(S)12(X) Full Chip Simulator. Use this information for simulation I/O device development.

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

Exceptions Icon

The Inspect window Exceptions icon shown in Figure 3.96 shows all currently raised exceptions. Exceptions are pending interrupts.

Figure 3.96 Inspector Window Exceptions Icon



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

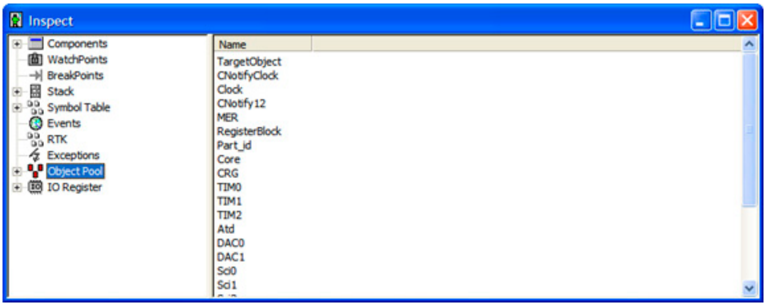
Since interrupts are usually simulated immediately when they are raised, the Exceptions are usually empty. Only when interrupts are disabled or an interrupt is handled is something visible in this item.

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

Object Pool

The Object Pool shown in Figure 3.97 is a pool of objects. It can contain any number of Objects, which can communicate together and also with other parts of the Debugger.

Figure 3.97 Inspector Window Object Pool



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

For the HC(S)12(X) Full Chip Simulator, the Object Pool usually contains the TargetObject, which represents the address space. All loaded Objects display in the Object Pool. The TargetObject also shows the objects that are mapped to the address space.

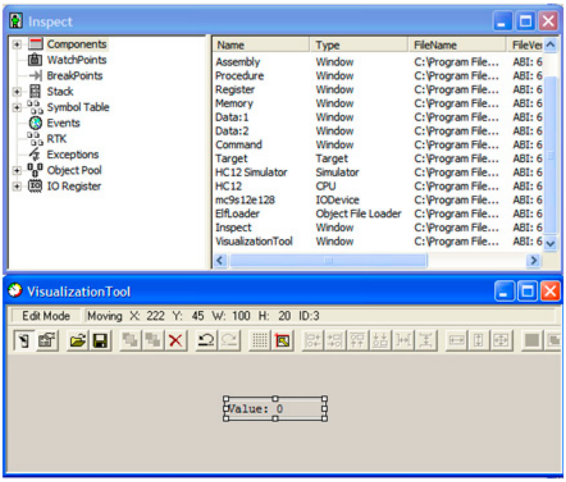
Inspector Operations

- Click the folded/unfolded icons to unfold/fold the tree and display/hide additional information.
- Click on any icon or name to see the corresponding information displayed on the right side.

On the right side, some value fields can be edited by double clicking on them. Only accessible values 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. Enter values in hexadecimal (with preceding **0x**), decimal, octal (with preceding **0**), or binary (with preceding **&**).

To see the component in the Inspector, for example VisualizationTool, as shown in Figure 3.98, open the **VisualizationTool** 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 "**VisualizationTool**" component.

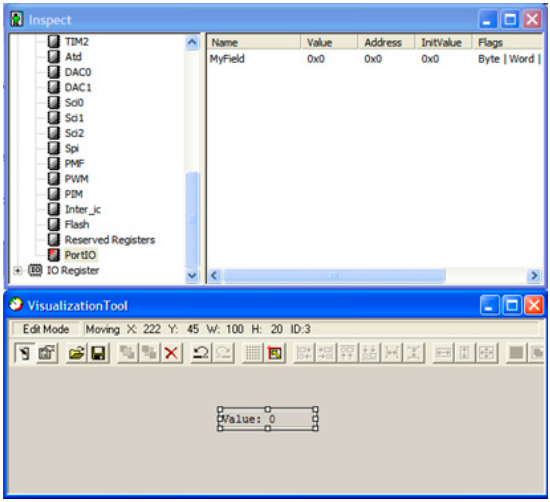
Figure 3.98 Seeing the VisualizationTool in the Inspect Window



Now, you can create the instrument in the **VisualizationTool** and view it in **Inspector Window**. Use **Add New Instrument** menu in the **VisualizationTool** and select required instrument type, for example "**Value as Text**", please see "Visualization Tool Component" on page 143 for more details. Specify kind of port as "**Substitute**" and port to display as "**TargetObject.MyField**".

Expand Object Pool to see the added **PortIO** icon. Click on the **PortIO** icon. On the right side, the **MyField** is displayed with current value. Double click on the values to change them. Figure 3.99 displays PortIO icon and MyField value.

Figure 3.99 Changing MyField Value in the Inspector Window



Inspect Menu

The Inspect menu contains entries described in Table 3.48.

Table 3.48 Inspect Menu Entries

Menu Entry	Description
Update	Updates all displayed information. Removes items that no longer exist and adds new items.
Periodical Update	Switches to Periodical mode. Updates displayed information at regular time intervals when connection is running. The update rate is 1 second.

Associated Context Menu

Commands in the Inspect context menu depend on the selected item. Table 3.49 describes possible context menu entries.

Table 3.49 Inspector Context Menu Entries Description

Menu Entry	Context	Description
Update	All items	Updates all displayed information. Removes items that no longer exist and adds new items.
Periodical Update	All items	Switches to Periodical mode. Updates displayed information at regular time intervals when connection is running. The update rate is 1 second.
Max. Elements	All items	Configure the maximum number to display large arrays element by element. It is also possible to display a dialog that prompts the user.
Format	All items	Use to display numerical values in different formats.
Close	Single selected Component only	Closes the corresponding component

Drag Out

Items that can be dragged depend on which icon is selected. Table 3.50 gives a brief description.

Table 3.50 Inspector Component Drag Possibilities

Dragging Item	Description
Components	The components cannot be dragged
Stack	The Stack Icon itself cannot be dragged. Subitems can be dragged depending on their type: • Modules: Modules can be dragged to the source and global data window to specify a specific module. • Functions: Functions can be dragged to display the function or code range. • Variables: Variables can be dragged to display their content in memory. • Indirections: Indirections can be dragged to display their content in memory.

Table 3.50 Inspector Component Drag Possibilities (*continued*)

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

Drop Into

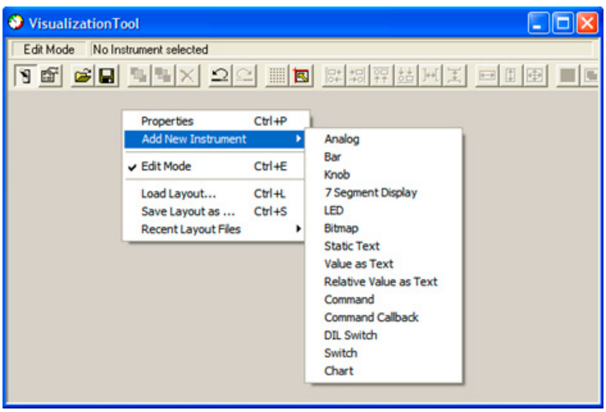
Nothing can be dropped in.

Visualization Tool Component

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

The VisualizationTool window, shown in Figure 3.100, consists of a plain workspace that can be equipped with many different instruments.

Figure 3.100 VisualizationTool Window



Edit Mode and Display Mode

The VisualizationTool operates in two modes:

- Edit mode

  Use Edit mode to design the workspace to suit your needs.

- Display mode

  In Display mode you can use what you have done in the Edit mode to view values, interact with your application and instruments, and click buttons.

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

Add New Instrument

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

Instrument Selection

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

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

Move Instruments

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

Resize Instruments

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

VisualizationTool Menu

Once you launch the Visualization Tool component, its menu appears in the debugger menu bar. Figure 3.101 displays the Visualization Tool menu.

Figure 3.101 Visualization Tool Menu

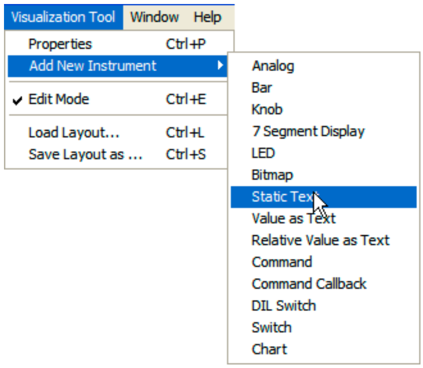


Table 3.51 describes the visualization tool menu entries.

Table 3.51 VisualizationTool Menu Description

Menu Entry	Shortcut	Description
Properties	**<Ctrl+P>**	Displays the properties of the currently selected instrument.
Add New Instrument	None	Enables you to choose an instrument from the list and add it to the view.
Edit Mode	**<Ctrl+E>**	Switches between Display mode and Edit mode. Checked in Edit mode.
Load Layout...	**<Ctrl+L>**	Loads a VisualizationTool Layout (`*.vtl`). The actual instruments are not removed.
Save Layout as...	**<Ctrl+S>**	Saves the current layout to a file (`*.vtl`).

Associated Context Menu

The context menu of the VisualizationTool varies depending on the current selection.

Table 3.52 describes possible menu entries.

Table 3.52 VisualizationTool Context Menu

Menu entry	Appears in Menu	Shortcut	Description
Edit mode	Always	None	Switches between Display mode and Edit mode. In Edit mode, this entry is checked.
Setup	Always	None	Shows Setup dialog of the VisualizationTool.
Load Layout	Edit mode	None	Loads a VisualizationTool Layout (`*.vtl`).
Save Layout	Always	None	Saves current layout to a file (`*.vtl`).
Add New Instrument	Edit mode	None	Shows new context menu with all available instruments.
Properties	Only one instrument selected	**Ctrl + P**	Shows property dialog box for currently selected instrument.
Remove	At least one selection	**Delete**	Removes all currently selected instruments.
Copy	At least one selection	**Ctrl + C**	Copies data of currently selected instruments into clipboard.
Cut	At least one selection	**Ctrl + X**	Cuts currently selected instruments into clipboard.
Paste	Edit mode	**Ctrl + V**	Adds instruments, which are temporarily stored in clipboard, to workspace.
Send to Back	At least one selection	None	Sends current instrument to back of Z order.
Send to Front	At least one selection	None	Brings current instrument to front of Z order.
Clone Attributes	More than one selection	**<Ctrl + Enter>**	Clones common attributes to all selected instruments according to the last selected.
Align	At least two selections	None	Gives access to new menu for alignment.
Top	Align	None	Aligns instruments to top line of last selected instrument.

Table 3.52 VisualizationTool Context Menu (*continued*)

Menu entry	Appears in Menu	Shortcut	Description
Bottom	Align	None	Aligns instruments to bottom line of last selected instrument.
Left	Align	None	Aligns instruments to left line of last selected instrument.
Right	Align	None	Aligns instruments to right line of last selected instrument.
Size	Align	None	Makes size of all selected instruments the same as last selected.
Vertical Size	Align	None	Makes vertical size of all selected instruments same as last selected.
Horizontal Size	Align	None	Makes horizontal size of all selected instruments same as last selected.

VisualizationTool Properties

Like other instruments, the VisualizationTool itself has Properties. There are several configuration possibilities for the VisualizationTool, shown in Table 3.53. To view the property dialog box of the VisualizationTool, use the shortcut <CTRL+P> or double click on the background.

Table 3.53 VisualizationTool Properties

Menu Entry	Description
Edit Mode	Switches from Edit mode to Display mode.
Display Scrollbars	Switches scrollbars on, off, or sets to automatic mode.
Display Headline	Switches headline on or off.
Backgroundcolor	Specifies background color of VisualizationTool.
Grid Mode	Specifies 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 distance between two grid points (vertical, horizontal).

Table 3.53 VisualizationTool Properties (*continued*)

Menu Entry	Description
Grid Color	Specifies color of the grid points.
Refresh Mode	Specifies window refresh mode. 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 the instrument. Table 3.54 shows attributes common to all instruments.

Table 3.54 Instruments Properties Attributes

Attribute	Description
X-Position	Specifies X-coordinate of the upper left corner.
Y-Position	Specifies Y-coordinate of the upper left corner.
Height	Specifies instrument height.
Width	Specifies instrument width.
Bounding Box	Specifies look of the bounding box. Available displays are: No Box, Flat (outline only), Raised, Sunken, Etched, and Shadowed.
Backgroundcolor	Defines color of instrument's background. Allows setting a color or letting instrument be transparent.
Kind of Port	Specifies kind of port to be used to get the value to display. The location must be specified in **Port to Display** field.

Table 3.54 Instruments Properties Attributes (*continued*)

Attribute	Description
Port to Display	Defines location of value to be used for instrument's visualization. Examples: Substitute: *TargetObject.#210* Subscribe: *TargetObject.#210* Subscribe: *PORTB.PORTB (check exact spelling using Inspector)* Variable: *counter* Register: *SP* Memory: *0x210*
Size of Port	If you use the Memory Port, you can also specify width of memory to display (up to 4 bytes).

Kind of Port

Table 3.55 lists the kind of port.

Table 3.55 Kind of Port

Attribute	Description

Substitute	Used to substitute a field of an object from the Object Pool.
	
	Objects can substitute other objects' fields, that is they can replace a certain field range of the substituted object, and hence become the actor for these fields. In this case, the substituted object forwards all requests concerning such fields to the appropriate substituting object.
	
	Requests for subscription to substituted fields are forwarded to the appropriate substituting object, i.e. subscribers always deal with the corresponding object which manages the fields (that is: the actor).
	
	Objects' fields are identified by their names. Names have the following syntax:
	
	`objSpec = objname ["." fieldname].`
	
	`objname = ident [":" index].`
	
	`fieldname = identnum [".." identnum]["." size].`
	
	`identnum = ident | "#" hexnumber.`
	
	`size = "B" | "W" | "L".`
	
	Example:
	
	Use it to create IO Port with needed name or address
	
	Substitute: `TargetObject.MyField`
	
	New temporary PortIO object is created in the Object Pool. It contains field with name MyField. The field's address equals to zero. Instrument displays and controls the field's value. If the instrument is removed the temporary PortIO object will be removed too.
	
	Example:
	
	Substitute: `TargetObject.#0x210`
	
	New temporary PortIO object is created in the Object Pool. It contains field that has no name. The address of this field equals to 210. Instrument displays and controls the field's value. If the instrument is removed the temporary PortIO object will be removed too.

Attribute	Description

	Used to subscribe to a field of an object from the Object Pool.
	
	Objects can subscribe to other objects' fields. The subscribing object is called subscriber, and the subscribed object is called actor. If a subscribed field of an actor changes, it notifies all subscribers of that fact.
	
	Objects' fields are identified by their names. Names have the following syntax:
	
	`objSpec = objname ["." fieldname].`
	
	`objname = ident [":" index].`
	
	`fieldname = identnum [".." identnum]["." size].`
	
	`identnum = ident | "#" hexnumber.`
	
	`size = "B" | "W" | "L".`
	
	Example:
	
	• Use it to subscribe the instrument to existing IO Port (field of an object from the Object Pool).
	
	Subscribe: `PORTB. fieldname` (check exact spelling using Inspector)
	
	New temporary PortIO object is created in the Object Pool. It contains no fields.
	
	If the instrument is removed the temporary PortIO object will be removed too.
	
	The Instrument exchanges data with fieldname field of PORTB object.
	
	Subscribe: `TargetObject.#210`
	
	New temporary PortIO object is created in the Object Pool. It contains no fields.
	
	If the instrument is removed the temporary PortIO object will be removed too.
	
	The Instrument exchanges data with the field located at `0x210`.

Attribute	Description
	Example: • PORTA is defined in the MEBI block. The name of the field for Pin0 from PORTA is PORTAPin0 In order to be able to stimulate this Pin by means of visualization tool: - Add a LED component - Set Kind of port to "subscribe" - Set Port to display to Mebi.PORTAPin0 - Set Visualization tool to Display Mode Starting from here you can toggle the Pin by clicking on the LED in the component.
Variable	Used to display value of a variable with given name Variable: *counter*
Register	Used to display value of a register with given name Register: *SP*
Memory	Used to display value of memory with given address (you can also specify width of memory to display) Memory: *0x210*

Analog Instrument

The Analog instrument (Figure 3.102) represents the classical pointer instrument.

Figure 3.102 Analog Instrument

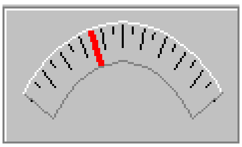


Table 3.56 shows the analog instrument attributes.

Table 3.56 Analog Instrument Attributes

Attribute	Description
Low Display Value	Defines zero point of the indicator. Values below this definition are not displayed.
High Display Value	Defines highest position of the indicator: the value at which indicator reads 100%.
Indicatorlength	Defines length of the small indicator. Minimal value is set to 20.
Indicator	Defines color of the indicator. Default color is red.
Marks	Defines color of the marks. Default color is black.

Bar Instrument

Using the Bar instrument, values are displayed by a bar strip. This instrument (see Figure 3.103) may be used as a position state of a water tank.

Figure 3.103 Bar Instrument

Table 3.57 shows bar instrument attributes.

Table 3.57 Bar Instrument Attributes

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

Bitmap Instrument

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

Figure 3.104 Bitmap Instrument

Additionally, it can also be used as a bitmap animation. Table 3.58 shows the Bitmap instrument attributes.

Table 3.58 Bitmap Instrument Attributes

Attribute	Description
Filename	Specifies location of the bitmap. With the button behind, you can browse for files.
AND Mask	Performs bitwise-AND operation with this value. AND the value of selected port. Default value is 0.
EQUAL Mask	Compares this value to the result of the AND operation. Bitmap displays only if both values are the same. Default value is 0.

In general, to show the bitmap, the following condition must be true:

```
(port_memory & ANDmask) == EQUALmask
```

Example using the AND and EQUAL masks

You want to show a taillight of a car in the visualization. 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 whether a lamp is on (e.g., bit 0 says that the flasher is on, where bit 1 says that the brake light is on). For your simple application you need the 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

Chart Instrument

The Chart instrument helps you to graphically represent any change in data.

Figure 3.105 Chart Instrument

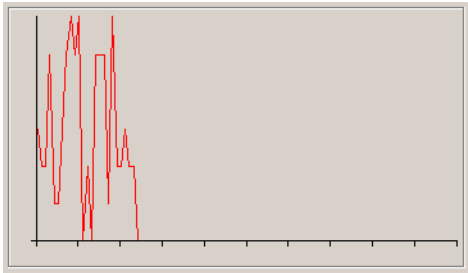


Table 3.59 Chart Instrument Attributes

Attribute	Description
High Display Value	Defines the maximum value for the Y axis.
Low Display Value	Defines the minimum value for the Y axis, the crossing with X axis.
Display Mode	Specify what type of chart will be used - Point, Line or Surface can be used to display the chart.
Type of Unit	The capturing of the value for the chart can be driven either the host or target time. Select Target Periodical to capture values relative to target time.
Unit Size	Defines the period of update.
Number of Units	Defines the number of items displayed on time axis (X).
Line / Fill Color	Defines color of points, line or surface.
Draw Frame	Specifies whether the axis are displayed or not.
Horiz. Index Step	Defines the interval between markers on horizontal axis (in frames).
Vert. Index Step	Defines the interval between markers on vertical axis.

DILSwitch Instrument

Use the Dual-in-Line Switch (DILSwitch) instrument (Figure 3.106) for configuration purposes. Use it for viewing or setting bits of one to four bytes.

Figure 3.106 DILSwitch Instrument

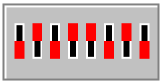


Table 3.60 lists DILSwitch instrument attributes.

Table 3.60 DILSwitch Instrument Attributes

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

Knob Instrument

The Knob instrument is normally known as an adjustment instrument. For example, it can simulate the volume control of a radio (Figure 3.107).

Figure 3.107 Knob Instrument

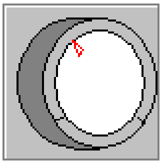


Table 3.61 shows the Knob instrument attributes.

Table 3.61 Knob Instrument Attributes

Attribute	Description
Low Display Value	Defines zero point of the indicator. Values below this definition are not displayed.
High Display Value	Defines highest position of the indicator: the value at which the indicator reads 100%.
Indicator Color	Defines color and width of pen used to draw the indicator.
Knob Color	Defines color of the knob side.

LED Instrument

Use the LED instrument for observing one definite bit of one byte (Figure 3.108). There are only two states: On and Off.

Figure 3.108 LED Instrument

Table 3.62 shows LED instrument attributes.

Table 3.62 LED Instrument Attributes

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

7-Segment Display Instrument

This is a 7-Segment Display instrument for numbers and characters. It has seven segments and one point. These eight units represent eight bits of one byte (Figure 3.109).

Figure 3.109 7-Segment Display Instrument

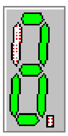


Table 3.63 shows 7-Segment Display instrument attributes.

Table 3.63 7-Segment Display Instrument Attributes

Attribute	Description
Decimalmode	Displays first or second four bits of one byte in hexadecimal mode. When switched off, each segment represents one bit of one byte.
Sloping	Switches sloping on or off.
Display Version	Selects appearance of instrument. Two versions available.
Color if Bit = = 1	Defines color of activated segment. Color may be set to transparent.

Table 3.63 7-Segment Display Instrument Attributes (*continued*)

Attribute	Description
Color if Bit = = 0	Defines color of deactivated segment. May be set to transparent.
Outlinecolor	Defines color of segment outlines. Color may be set to transparent.

Switch Instrument

Use the Switch instrument to set or view a definite bit (Figure 3.110). The Switch instrument also provides an interesting debugging feature: you can let it simulate bounces, and thus check whether your algorithm is robust enough. Four different looks of the switch are available: slide switch, toggle switch, jumper or push button.

Figure 3.110 Switch Instrument

Table 3.64 shows Switch instrument attributes.

Table 3.64 Switch Instrument Attributes

Attribute	Description
Bitnumber to Display	Specifies number of bit to display.
Display 0/1	Allows display of the bit value in its upper left corner.
Top Position is	Specifies whether the 'up' position is zero or one. Especially useful to easily transform push button into a reset button.
Kind of Switch	Changes look of the instrument. The 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 color of switch.
Bounces	If enabled, gives access to following other attributes to configure the way the switch bounces.
Nb Bounces	Specifies the number of bounces before stabilization.
Bounces on Edge	Specifies whether switch bounces on falling, rising or both edges.

Table 3.64 Switch Instrument Attributes (*continued*)

Attribute	Description
Type of Unit	Synchronizes frequency of bouncing either on timer of host machine, or on CPU cycles.
Pulse Width (100ms)	Defines duration of one bounce. Fill in if you chose **Host Periodical** in the **Type of Unit** attribute.
CPU Count	Represents number of CPU cycles to reach before switch changes state. Fill in if you chose **CPU Cycles** in **Type of Unit** attribute.

Text Instrument

The Text instrument has several functions: Static Text, Value, Relative Value, and Command (Figure 3.111).

Figure 3.111 Text Instrument

Use **Text Mode** to switch between the five available modes. Table 3.65 shows Text instrument common attributes.

Table 3.65 Text Instrument Attributes

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

Use **Static Text** for adding descriptions on the workspace. Table 3.66 shows Static Text attributes.

Table 3.66 Static Text Attributes

Attribute	Description
Field Description	Contains the text to display

Use **Value** for displaying a value in different ways (decimal, hexadecimal, octal, or binary). Table 3.67 shows Value attributes.

Table 3.67 Value Attributes

Attribute	Description
Field Description	Contains additional description to display in front of value. Add a colon and/or space as you wish. The default setting is **Value:**
Format mode	Defines format. Choose from: Decimal, Hexadecimal, Octal, and Binary formats.

Use **Relative Value** for showing a value in a range of 0 up to 100% or 1000‰. Table 3.68 shows Relative Value attributes.

Table 3.68 Relative Value Attributes

Attribute	Description
Field Description	Add additional description text to display in front of value. Add a colon and/or space if desired. The default setting is **Value:**
Low Display Value	Fixes minimal value that will represent 0%. Values below this definition appear as an error: #ERROR.
High Display Value	Fixes maximal value that will represent 100%. Values above this definition appear as an error: #ERROR.
Relative Mode	Switches between percent and permill.

Use **Command** instrument mode to specify a command that will be executed by clicking on this field. For more information about commands, read the chapters on Debugger Commands. Table 3.69 shows Command mode attributes.

Table 3.69 Command Attributes

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

CMD Callback is the same as Command, but shows the returned value as text instead of **Field Description**. Table 3.70 shows CMD Callback attributes.

Table 3.70 CMD Callback Attributes

Attribute	Description
Field Description	Warning: text written in this field is overwritten the first time you execute specified command.
Command	Contains command line command to be executed after clicking button.

Drop Into

In Edit mode, the drag and drop functionality supplies an 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 now contains the address of the variable. 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 Inspect component: pick an object from the object pool.

Demo Version Limitations

Only one VisualizationTool window can be loaded. The number of instruments is limited to three.

4

Control Points

This chapter provides an overview of the three kinds of debugger control points:

- Breakpoints (also called data breakpoints)

  Breakpoints are located at an address. They can be temporary or permanent.

- Watchpoints

  Watchpoints are located at a memory range. They start from an address, have a range, and a read and/or write state.

- Markpoints

  Markpoints are marked points of observation that give the programmer easily accessible program markers. They can be located in data, source or memory.

Control Point Configuration

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

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

- Select a single control point from a list box and click **Delete**.
- Select multiple control points from a list box and click **Delete**.
- Enable/disable a selected control point by checking or unchecking the related checkbox.
- Enable/disable multiple control points by checking or unchecking the related checkbox.
- Enter or modify the condition of a selected control point.
- Enable/disable the condition of a selected control point by checking/unchecking the related checkbox.
- Enter or modify the command of a selected control point.
- Enable/disable the command of a selected control point by checking/unchecking the related checkbox.

- Enable/disable multiple control point commands by selecting control points and checking/unchecking the related checkbox.
- Modify the counter and/or limit of a single control point.

With breakpoints, you can also perform the following operations:

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

With watchpoints, you can also perform the following operations:

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

Breakpoints

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

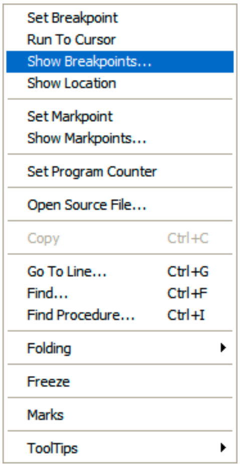
- Temporary breakpoints, which activate the next time the instruction executes.
- Permanent breakpoints, which activate each time the instruction executes.
- Counting breakpoints, which activate after the instruction executes a certain number of times.
- Conditional breakpoints, which activate when a given condition is TRUE.

Breakpoints are controlled through the Breakpoints tab of the Controlpoints Configuration window. Open this window using the Source window context menu, by following these steps:

1. Point at a C statement in the Source window, and click the right mouse button.

   The Source Window context menu appears (see Figure 4.1).

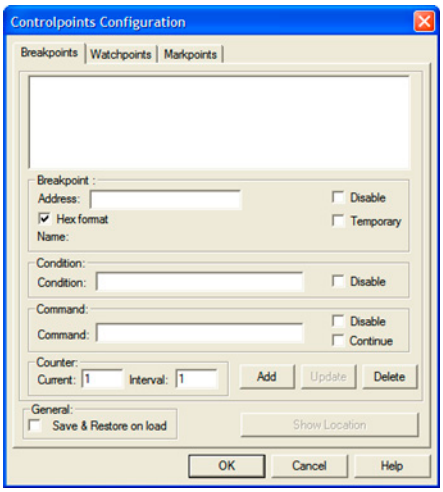
Figure 4.1 Source Window Context Menu



2. Select **Show Breakpoints** from this menu

The Breakpoints tab of the Controlpoints configuration window opens (Figure 4.2).

Figure 4.2 Controlpoints Configuration Window (Breakpoints Tab)



Breakpoints Tab

The Breakpoints tab contains:

- List box that displays the list of currently defined breakpoints
- **Breakpoint** group box that displays the address of the currently selected breakpoint, name of procedure in which the breakpoint has been set, state of the breakpoint (disabled/enabled), and type of breakpoint (temporary or permanent).
- **Condition** group box that displays the condition string to evaluate, and the state of the condition (disabled/enabled).
- **Command** group box that displays the command string to execute and the state of the command (disable or continue after command execution).
- **Counter** group box that displays the current counter value and interval counter value.

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. Changing the Interval value automatically sets the Counter value to the Interval value.

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

Multiple Selections in List Box

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

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

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

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

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

Checking Expressions

You can enter an expression in the **Condition** group edit box. The debugger checks the expression syntax 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 the debugger detects a syntax error, a message box appears:

Incorrect Condition. Do you want to correct it?

If you click **OK**, correct the error in the **Condition** edit box.

If you click **Cancel**, the **Condition** edit box is cleared.

Saving Breakpoints

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

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

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

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

Listing 4.1 Breakpoint (.BPT) File Syntax

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

In the code above:

– `address` is the address where the breakpoint is to be set. This address is specified in ANSI C format. `address` can also be replaced by an **expression** as shown in the example below.

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

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

Breakpoint File (.BPT) Example

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

Listing 4.2 Breakpoint File with Save & Restore on load Checked

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

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

```
savebp on
```

This saves only the flag and removes the breakpoints.

NOTE If only one or a few functions change after a recompilation, not all **BP**s are lost. To achieve that, **BP**s are disabled only if the size of a function changes. 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 an `.ABS` file is loaded and the matching `.BPT` file exists, for each **BS** command, the Debugger checks if the code size (in bytes) and the source size (in characters) are different in the matching function (given by the symbol table). If there is a difference, the breakpoint is set and disabled. If there is no difference, the breakpoint is set and enabled.

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

Setting Breakpoints

The Debugger supports different types of breakpoints:

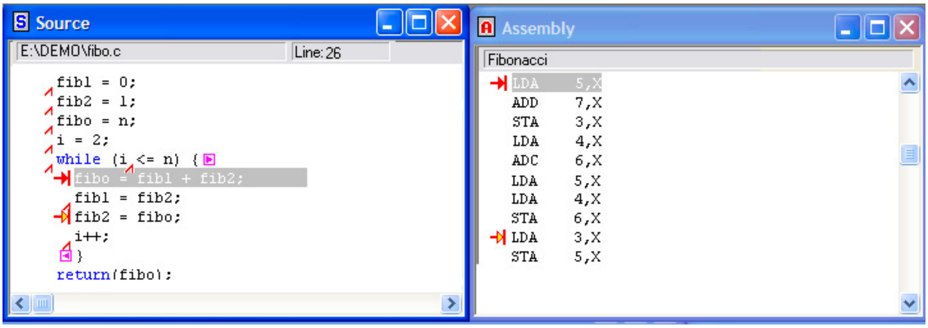
- Temporary breakpoints activate the next time the instruction executes.
- Permanent breakpoints activate each time the instruction executes.
- Counting breakpoints activate after the instruction executes a certain number of times.
- Conditional breakpoints activate when a given condition is TRUE.

Set breakpoints in a Source or Assembly component window.

Possible Breakpoint Positions

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

Figure 4.3 Source and Assembly Windows



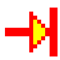
The Debugger helps you detect all positions where you can set a breakpoint.

1. Right click in the Source component to display the Source context menu on the screen.
2. Choose **Marks** from the context menu. All statements where a breakpoint can be set are identified by a red inverted check mark:

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

Temporary Breakpoints

Temporary breakpoints activate the next time the instruction executed. The following icon identifies a temporary breakpoint:



Setting Temporary Breakpoints

Using the Source Window Context Menu

1. Point at a C statement in the Source window and right click to display the Source context menu.
2. Choose **Run To Cursor** from the context menu. The application continues execution and stops before executing the statement. You have executed a temporary breakpoint.

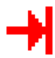
Holding down the left mouse button, pressing the T key

1. Point at a C statement in the Source window, hold down the left mouse button, and click the T key. This defines a temporary breakpoint.
2. Choose **Run To Cursor** from the context menu. The application continues execution and stops before executing the statement.

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

Permanent Breakpoints

Permanent breakpoints activate each time the instruction executes. The following icon identifies a permanent breakpoint:



Setting Permanent Breakpoints

Using the Source Window Context Menu

1. Point at a C statement in the Source window and right click. The Source context menu appears.
2. Select **Set BreakPoint** from the context menu. A permanent breakpoint mark appears in front of the selected statement.

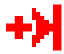
Holding down the left mouse button, pressing the P key

1. Point at a C statement in the Source window.
2. Hold down the left mouse button and click the P key. A permanent breakpoint mark appears in front of the selected statement.

Once you define a permanent breakpoint, you can continue program execution. The application stops before executing the statement. Permanent breakpoints remain active until they are disabled or deleted.

Counting Breakpoints

Counting breakpoints activate after the instruction executes a certain number of times. The following icon identifies a Counting breakpoint:



Setting Counting Breakpoints

Counting breakpoints can only be set using the Breakpoints tab. There are two ways to set a counting breakpoint:

Hold down the left mouse button, click the S key

1. Point at a C statement in the Source window.
2. Hold down the left mouse button and click the S key.

   The Controlpoints Configuration window with the Breakpoints tab opens and inserts a new breakpoint in the list of breakpoints defined in the application.
3. Select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints at the top of the tab.
4. In the **Counter** group of this tab specify the interval for the breakpoint detection in the **Interval** field.
5. Close the window by clicking the **OK** button.

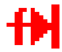
Use the Source Context Menu

1. Point at a C statement in the Source window and right click to display the Source context menu.
2. Choose **Set BreakPoint** from the context menu. This defines a breakpoint on the selected instruction.
3. Point in the Source window and right click again.
4. Choose **Show Breakpoints** from the context menu to display the Controlpoints Configuration Window (Breakpoints Tab).
5. Select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints at the top of the tab.
6. In the **Counter** group of this tab specify the interval for the breakpoint detection in the **Interval** field.
7. Close the window by clicking the **OK** button.

If you continue program execution, the content of the **Current** field decrements each time the program reaches the instruction containing the breakpoint. When **Current** equals zero, the application stops. If the **Temporary** checkbox is unchecked (not a temporary breakpoint), **Current** is reloaded with the value stored in **Interval** to enable the counting breakpoint again.

Conditional Breakpoints

Conditional breakpoints activate when a given condition is TRUE. The following icon identifies a conditional breakpoint:



Setting Conditional Breakpoints

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

Hold down the left mouse button, click the S key

1. Point at a C statement in the Source component window, hold down the left mouse button, and click the S key.

   The Breakpoints tab opens and inserts a new breakpoint in the list of breakpoints defined in the application.

2. Select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints.
3. Specify the condition for breakpoint activation in the **Condition** group Condition box. You must use ANSI-C syntax to specify the condition (for example, **counter == 7**). You can use register values in the breakpoint condition field with the following syntax: **$RegisterName** (for example, **$RX == 0x10**)
4. Close the window by clicking **OK**.

Use the Source Window Context Menu

1. Point at a C statement in the Source component window and right click to display the Source context menu.
2. Select **Set BreakPoint** from the context menu to define a breakpoint on the selected instruction.
3. Point in the Source component window and right click to display the Source context menu.

4. Select **Show Breakpoints** from the context menu. The Breakpoints tab opens and inserts a new breakpoint in the list of breakpoints defined in the application.
5. Select the breakpoint you want to modify by clicking on the corresponding entry in the list of defined breakpoints.
6. Specify the condition for breakpoint activation in the **Condition** group Condition box. You must specify the condition using the ANSI C syntax (for example, **counter == 7**). You can use register values in the breakpoint condition field with the following syntax: **$RegisterName** (for example, **$RX == 0x10**)
7. Close the window by clicking **OK**.

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

Deleting Breakpoints

The Debugger provides three ways to delete a breakpoint:

Use Delete Breakpoint from Source Context Menu

1. In the Source component window, point at a C statement where a breakpoint exists and right click. This displays the Source context menu.
2. Choose **Delete** Breakpoint from the context menu to delete the breakpoint.

Hold down the left mouse button, click the D key

1. Point at a C statement in the Source component window where a breakpoint exists.
2. Hold down the left mouse button and click the D key.

   This deletes the breakpoint.

Choosing Show Breakpoints from Source Context Menu

1. Point in the Source component window and right click. The Source context menu appears.
2. Choose **Show Breakpoints** from the context menu. The **Breakpoints Setting** dialog appears.
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 and remove the icon associated with the deleted breakpoint from the Source component.

Associate a Command with a Breakpoint

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

Choose Show Breakpoints from Source Window Context Menu

1. Point in the Source component window and right click to display the Source context menu.
2. Choose **Show Breakpoints** from the context menu. The Controlpoints Configuration window, with the Breakpoints tab displayed, appears.

In the Breakpoints tab of the Controlpoints Configuration window

1. Select the breakpoint to modify by clicking on the corresponding entry in the list of defined breakpoints.
2. 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). Associate a command file with a breakpoint using the command **CALL** or **CF** (Example: **CF breakCmd.cmd**).
3. Click **OK** to close the window.

When the breakpoint is detected, the command executes and the application stops.

The **Continue** check button of the Controlpoints Configuration window allows the application to continue after the command executes.

Demo Version Limitations

Only two breakpoints can be set.

Watchpoints

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

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

Control Watchpoints through the Watchpoints tab of the Controlpoints Configuration window. Open this window through the Memory or Data component window context menu, as described below.

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

1. Position your cursor in either the Memory or Data component window.
2. Click the right mouse button.
3. Select **Show Watchpoints** from either menu.
4. Click the left mouse button.

The ControlPoints Configuration window appears. Figure 4.6 shows the Watchpoints tab of this window.

Figure 4.4 Memory Context Menu

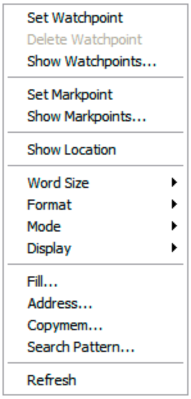
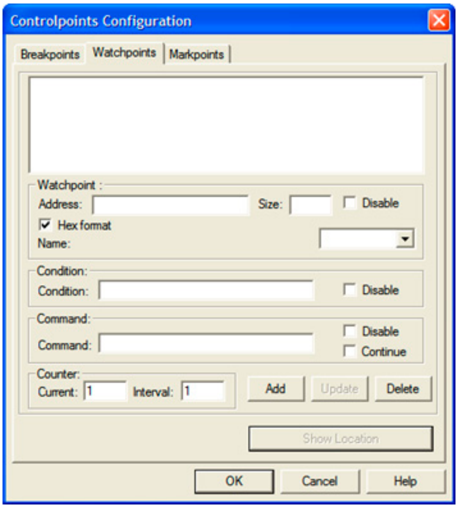


Figure 4.5 Data Context Menu

Figure 4.6 Controlpoints Configuration Window (Watchpoints Tab)



Watchpoints Tab

The Watchpoints tab of the Controlpoints Configuration window contains:

- List box that displays the list of currently defined watchpoints.
- **Watchpoint** group box that displays the address of the currently selected watchpoint, size of the watchpoint, name of the procedure or variable on which the watchpoint is set, state of the watchpoint (disabled or not), read access of the watchpoint (enabled or not) and write access of the watchpoint (enabled or not).
- **Condition** group box that displays the condition string to evaluate and the state of the condition (disabled or not).
- **Command** group box that displays the command string to execute and state of the command (disabled or continue after command execution).
- **Counter** group box that displays the current value of the counter and interval value of the counter.
- **Add** button to add new watchpoints; specify the Address in hexadecimal when **Hex format** is checked or as an expression when **Hex format** is unchecked.
- **Update** button to Update all modifications in the dialog.
- **Delete** button to remove currently selected watchpoint and select the watchpoint that is below the removed watchpoint.
- **OK** button to validate all modifications.

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

NOTE When the Interval value changes, the Counter value automatically resets to the Interval value.

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

Multiple Selections

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

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

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

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

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

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

Checking Syntax

You can enter an expression in the Condition group edit box. The debugger checks the syntax of the expression when you select another watchpoint in the list box or when you click **OK**.

If a syntax error is detected, a message box appears:

```
Incorrect Condition. Do you want to correct it?
```

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

Click **Cancel** to clear the condition edit box.

Setting Watchpoints

Watchpoints may be set in a Data or Memory window.

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

Setting a Read Watchpoint

A green vertical bar appears in front of a variable associated with a read access watchpoint. The Debugger provides two ways to define a read access watchpoint:

Use the Data Context Menu

1. Point at a variable in the Data window and right click to display the Data Context Menu.
2. Choose **Set Watchpoint** from the context menu to define a **Read/Write** watchpoint.
3. Point in the Data window and right click to display the Data context menu.
4. Choose **Show WatchPoints** from the context menu. The Controlpoints Configuration window Watchpoints tab appears.
5. Select the watchpoint you want to define as *read* access from the list.
6. Select the **Read** type in the list box to define a read access watchpoint for the selected variable.

Use the Left Mouse Button, click the R Key

1. Point at a variable in the Data window.
2. Hold down the left mouse button and click the R key.

   This defines a read access watchpoint for the selected variable.

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

Setting a Write Watchpoint

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

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

Using the Data Context Menu

1. Point at a variable in the Data window and right click. The Data context menu appears.
2. Choose **Set Watchpoint** from the context menu to define Read/Write Watchpoint.
3. Point in the Data component window and right click. The Source context menu appears.
4. Choose **Show WatchPoints** from the context menu. The Controlpoints Configuration window Watchpoints tab appears.
5. From the list, select the watchpoint you want to define as write access.
6. Select the **Write** type in the list box to define a write access watchpoint for the selected variable.

Using the Left Mouse Button and pressing the W Key

1. Point at a variable in the Data window.
2. Hold down the left mouse button and click the W key. This defines a write access watchpoint 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 appears in front of a variable associated with a read/write access watchpoint.

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

Use the Data Context Menu

1. Point at a variable in the Data window and right click. The Data context menu appears.
2. Choose **Set Watchpoint** from the context menu to define a Read/Write Watchpoint.

Use the Left Mouse Button and click the B Key

1. Point at a variable in the Data window.
2. Hold down the left mouse button and click the B key.

   This defines a read/write access watchpoint for the selected variable.

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

Defining a Counting Watchpoint

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

Use the Data Context Menu

1. Point at a variable in the Data window and right click. The Data context menu appears.
2. Choose **Set Watchpoint** from the context menu to define a read/write watchpoint.
3. Point in the Data component window and right click to display the Source context menu.
4. Choose **Show WatchPoints** from the context menu. The Controlpoints Configuration window Watchpoints tab appears.
5. Select the watchpoint you want to define as a counting watchpoint.
6. From the list box, select the type of access you want to track.
7. In the interval field, specify the interval count for the watchpoint.
8. Click **OK** to close the window and define a counting watchpoint for the selected variable.

Use the Left Mouse Button and click the S Key

1. Point at a variable in the Data window.
2. Hold down the left mouse button and click the S key. The Watchpoints tab of the Controlpoints Configuration window appears.
3. Select the watchpoint you want to define as a counting watchpoint from the list.
4. From the list box, select the type of access you want to track.

5. In the interval field, specify the interval count for the watchpoint. Click **OK** to close the window and define a counting watchpoint for the selected variable.

If you continue program execution, the **Current** field decrements each time an appropriate access on the variable is detected. When **Current** equals zero, the application stops. **Current** reloads with the value stored in the interval field to enable the counting watchpoint again.

Defining a Conditional Watchpoint

You can associate a condition with any type of watchpoint (read, write, read/write). The Debugger provides two ways to define a conditional watchpoint:

Use the Data Context Menu

1. Point at a variable in the Data window and right click. The Data context menu appears.
2. Choose **Set Watchpoint** from the context menu to define a read/write watchpoint.
3. Point in the Data window and right click. The Source context menu appears.
4. Choose **Show WatchPoints** from the context menu. The Controlpoints Configuration window Watchpoints tab appears.
5. Select the watchpoint you want to define as a conditional watchpoint.
6. From the list box, select the type of access you want to track.
7. Specify the condition for the watchpoint in the **Condition** field. The condition must be specified using the ANSI-C syntax (Example: **counter ==** 7).
8. Click **OK** to close the window and define a conditional watchpoint for the selected variable.

Use the Left Mouse Button and click the S Key

1. Point at a variable in the Data window.
2. Hold down the left mouse button and click the S key. The Watchpoints tab of the Controlpoints Configuration window appears.
3. Select the watchpoint you want to define as a conditional watchpoint.
4. From the list box, select the type of access you want to track.
5. 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**)

6. Click **OK** to close the window and define a conditional watchpoint for the selected variable.

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

Deleting a Watchpoint

The Debugger provides three ways to delete a watchpoint:

Use Delete Breakpoint from Context Menu

1. In the Data window, point to a variable where a watchpoint has been defined and right click. The Data context menu appears.
2. Select **Delete Watchpoint** from the context menu to delete the watchpoint and remove the vertical bar in front of the variable.

Use the Left Mouse Button and click the D Key

1. Point at a variable in the Data window.
2. Hold down the left mouse button and click the D key. This deletes the watchpoint and removes the vertical bar in front of the variable.

Choosing Show Watchpoints from Data Context Menu

1. Point in the Data window and right click. The Data context menu appears.
2. Choose **Show Watchpoint*s*** from the context menu. The Watchpoints tab of the Controlpoints Configuration window appears.
3. Select the watchpoint you want to delete.
4. Click **Delete**. This removes the selected watchpoint from the list of defined watchpoints.
5. Click **OK** to close the window. This deletes the watchpoint and removes the vertical bar in front of the variable.

Associate a Command with a Watchpoint

You can associate each watchpoint type (read, write, read/write, counting, or conditional) with a debugger command. Specify this command in the Watchpoints tab of the Controlpoints Configuration window. To open this window:

Choosing Show Watchpoints from Data Context Menu

1. Point in the Data component window and right click. The Data Context Menu appears.
2. Select **Show Watchpoints** from the context menu. The Watchpoints tab of the Controlpoints Configuration window appears.
3. Click on the corresponding entry in the list of defined breakpoints to select the watchpoint you want to modify.
4. 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. Associate a command file with a watchpoint using the commands CALL or CF (Example CF breakCmd.cmd).
5. Click **OK** to close the window.

   When the watchpoint is detected, the command executes and the application stops at this point. The **Continue** check button allows the application to continue after command execution.

Demo Version Limitations

Only two watchpoints can be set.

Watchpoints in Multi Core Projects

HCS12X multicore derivative debug module allows setting of watchpoint to either CPU12X or XGATE data bus. Correspondingly, for multicore HCS12X derivatives debugger sets watchpoint to either CPU12X or XGATE data bus, so when watchpoint for the variable shared between CPU12X and XGATE is set it will work only for access from the corresponding core.

Watchpoint is set to XGATE data bus in two cases:

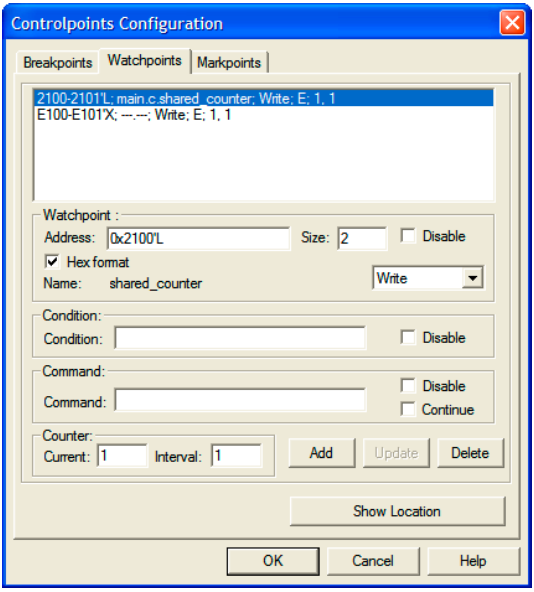
- watchpoint memory area is identified as variable defined in XGATE source code;
- watchpoint is set directly to memory area with XGATE space.

In all other cases watchpoint is set to CPU12X data bus.

Setting Watchpoint to the variable shared between CPU12X and XGATE

1. Variable shall be defined in CPU12X source code, otherwise watchpoint to CPU12X data bus can not be set;
2. Set first watchpoint to the variable or directly to the memory area with Logical or Global space. This watchpoint will be set to CPU12X data bus;
3. Set second watchpoint directly to the memory area with XGATE space. Use HCS12XAdrMap component to convert address to XGATE space. This watchpoint will be set to XGATE data bus.

Figure 4.7 Watchpoint Configuration for Shared Variable



Markpoints

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

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

Markpoints are controlled through the Markpoint tab of the Controlpoints Configuration window. Open the window with the Source, Memory or Data window context menu, as described below.

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

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

The ControlPoints Configuration window appears with the Markpoints tab of this window exposed, as shown in Figure 4.6.

Figure 4.8 Source Window Context Menu

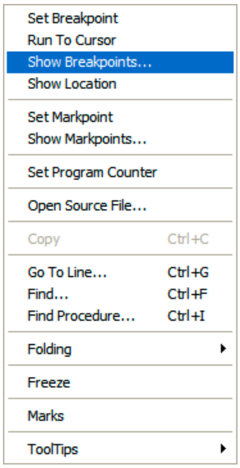


Figure 4.9 Memory Context Menu

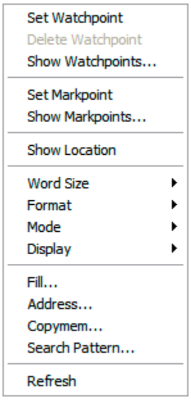
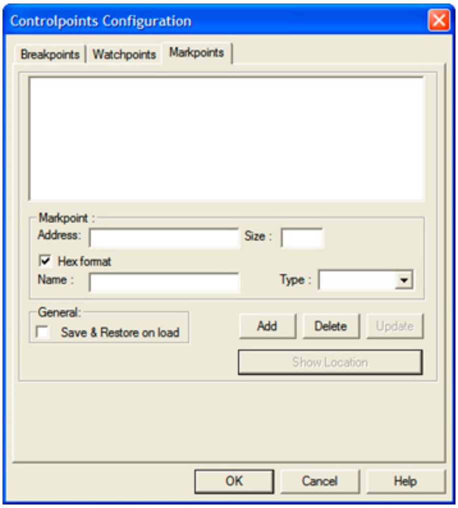


Figure 4.10 Data Context Menu

Figure 4.11 Controlpoints Configuration Window (Markpoints Tab)



Markpoints Tab

The Markpoints tab of the Controlpoints Configuration window contains:

- List box that displays the list of currently defined markpoints.
- **Markpoint** group box that displays the address of the currently selected markpoint, size of the markpoint, name of the procedure or variable on which the markpoint has been set, and type of the markpoint.
- **General** group box that contains a checkbox that allows you to save and restore the markpoint selected.
- **Add** button to add new markpoints. Specify the Address in hexadecimal when **Hex format** is checked or as an expression when **Hex format** is unchecked.
- **Delete** button to remove currently selected markpoint and select the markpoint that is below the removed markpoint.
- **Update** button to update all modifications in the window.
- **OK** button to validate all modifications.
- **Cancel** button to ignore all modifications.
- **Help** button to display help file and related help information.

Setting Markpoints

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

Setting a Source Markpoint

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

Use the Source Context Menu

1. Point at a code line in the Source window and right click. The Source Window context menu appears (see Figure 4.8).
2. Choose **Set Markpoint** from the context menu to define a markpoint at the beginning of the line.
3. Point in the Source window and right click. The Source context menu appears.
4. Choose **Show WatchPoints** from the context menu. The Controlpoints Configuration Window Markpoints Tab appears.
5. Make any modifications to any markpoints listed.
6. Click **OK** to close the window.

Setting a Data Markpoint

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

Use the Data Context Menu

1. Point at a variable in the Data window and right click. The Data context menu appears (see Figure 4.10).
2. Choose **Set Markpoint** from the context menu to define a markpoint at the beginning of the data range selected.
3. Point in the Data window and right click. The Data context menu appears.
4. Choose **Show WatchPoints** from the context menu. The Controlpoints Configuration window Markpoints tab appears.
5. Make any modifications to any markpoints listed.
6. Click **OK** to close the window.

Setting a Memory Markpoint

A blue letter L appears in front of a memory range associated with a markpoint.

To define a Memory markpoint:

Use the Memory Context Menu

1. Point at a line in the Memory window and right click. The Memory context menu appears (see Figure 4.9).
2. Choose **Set Watchpoint** from the context menu to define a Markpoint.
3. Point in the Memory window and right click. The Memory context menu appears.
4. Choose **Show WatchPoints** from the context menu. The Controlpoints Configuration window Markpoints tab appears.
5. Make any modifications to any markpoints listed
6. Click **OK** to close the window.

Deleting a Markpoint

To delete a markpoint:

Use the Left Mouse Button, click the D Key

1. Point at the markpoint variable in the Data window, the memory range in the Memory window, or the codeline in the Source window.
2. Hold down the left mouse button and click the D key.
3. This deletes the markpoint and removes the blue letter L in front of the variable, memory range or codeline.

Choose Show Markpoints from Appropriate Context Menu

1. Point in the Data, Memory or Source component window and right click. The associated context menu appears.
2. Choose **Show Markpoint***s* from the context menu. The Markpoints tab of the Controlpoints Configuration window appears.
3. In this tab's list box, select the markpoints you want to delete.
4. Click **Delete**. This removes the selected markpoint from the list of defined markpoints.
5. Click **OK** to close the window. This deletes the markpoint and removes the blue letter L in front of the variable, memory range, or code line.

Halting on a Control Point

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

Counting Control Point

If the interval property is greater than one, a counting control point has been defined. When the Debugger is running, every time code reaches the control point, its current value decrements. The Debugger halts when the value reaches zero. When the Debugger stops on the control point, a command executes (if defined and enabled).

Conditional Control Point

If a condition is defined and enabled for a control point that halts the Debugger, a command executes (if defined and enabled).

Control Point with Command

When the Debugger halts on the control point, the specified command executes.

5

Real-Time Kernel Awareness

The Debugger allows you to load and control applications on the target system or 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.

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

- Debugging any task in the system (e.g., viewing the state of any task in the system). 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 (such as scheduling information, queues, timers). Some of these data structures are of interest to operating system users and are described in this chapter.

Inspecting Task State

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

The Debugger allows you to inspect the CPU registers and stack containing all procedure activation frames (return addresses, parameters, local variables). Therefore, it must retrieve this information for each task to be debugged. The debugger reads this information from a file called `OSPARAM.PRM`, which contains the algorithm for retrieving all the needed data from the target memory task descriptors. To describe this algorithm, a simple procedural language is used. The only parameter to the algorithm is a user-specified address which identifies the task to be inspected. The result is the CPU context (CPU registers) and status of the task, which allows the debugger to display the procedure activation stack in a symbolic way.

RTK Interface

When the application halts, the debugger displays the state of the current task. To identify the task to be inspected, follow these steps:

1. Make the task descriptor, or a pointer to it, visible in any of the debugger's data windows.
2. While holding down the left mouse button on a variable of type `pointer to task descriptor`, click the **P** key.

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

Task Description Language

To perform debugging on any task, create a file named `OSPARAM.PRM` and store it in one of the directories specified in GENPATH: #include “File” Path.

`OSPARAM.PRM` contains the algorithm for collecting the context information for a specific task (the PC, SP, DL, SR and registers).

Use the following syntax (in EBNF) to specify the algorithm:

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

Table 5.1 shows the terminal symbol meanings:

Table 5.1 Terminal Symbol Meanings

Terminal Symbol	Meaning
B	Given reference to the task descriptor (initialized upon start)
a–z	Variables for intermediate storage
MB	Retrieves value of memory BYTE at given address
MW	Retrieves value of memory WORD at given address
MD	Retrieves value of DOUBLE WORD at given address
MA	Retrieves value at given address interpreted as DOUBLE WORD
PC	Program counter to be set
SP	Stack pointer to be set
SR	Status register value to be set
DL	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	Error message (must be specified if N >= 1000)

On activation of the task debugging command, the file `OSPARAM.PRM` opens and stores the selected address 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 greater than 1000, the string **MSG** expects the status.

Application Example

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

Listing 5.1 OSPARAM.PRM File

```
{ File OSParam.PRM, implementation for SOOM System/REM }
{ R0..R7 = D0..D7, R8..R15 = A0..A7 }
```

```
{ MSG = message displayed in Procedure Chain window }

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

Inspecting Kernel Data Structures

To allow the debugger to display the data structures of the operating system, the corresponding symbol information (in this case, for SOOM System/REM) must be available. To use another kernel, its source code must be available and must be compiled. However, if only the object code is available, generate the needed symbol information by describing the kernel data structures of interest using ANSI-C language, as shown in Listing 5.2.

Listing 5.2 Kernel Data Structure Description

```
   typedef struct PD {
      int status;
      struct PD *next;
```

```
    long regs[6];
} PD;
```

Define a simple task descriptor by collecting variables in a structure and assigning them to a segment (for example, OS_DATA shown in Listing 5.3). Define this structure to fit the same layout as the operating system. If necessary, use filler variables to get the correct alignment.

Listing 5.3 OS_DATA Structure

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

The linker uses a PRM file like the one shown in Listing 5.4 to place the segment at the correct address.

Listing 5.4 Linker PRM File

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

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

Compile the source file (for example, `rtk.c`) and list it in the `NAMES` section of the linker parameter file. To force linking, follow the name of the object file immediately by +. In this example the variable is linked to the address `0x1040`.

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

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

RTK Awareness Register Assignments

Table 5.2 shows the register assignments for the RTK awareness for the HC12 processor.

Table 5.2 HC12 RTK Awareness Register Assignments

Register	Register Name	Size (in bits)
R0	A	8 (high byte of D)
R1	B	8 (low byte of D)
R2	CCR	8
R6	D	16 (concatenation of A:B)
R7	X	16
R8	Y	16
R9	SP	24 (concatenation of xPAGE:SP if in banked area)
R10	PC	16
R11	PPAGE	8
R12	EPAGE	8
R13	DPAGE	8
R14	IP	24 (concatenation of PPAGE:PC if in banked area)

OSEK Kernel Awareness

The OSEK Kernel provides a framework for building real-time applications. OSEK Kernel awareness within the debugger allows you to debug your application from the operating system perspective.

The CodeWarrior Debugger supports OSEK ORTI-compliant real-time operating systems and offers dedicated kernel awareness, using the information stored in your application's ORTI file. With CodeWarrior OSEK kernel awareness, you can monitor kernel task information, semaphores, messages, queues, resources allocations, synchronization, and communication between tasks.

ORTI describes the applications in any OSEK implementation:

- A set of attributes for system objects.
- A method for interpreting the data obtained.

OSEK RTI

The OSEK Run-Time Interface (ORTI) is a development tool interface to the OSEK operating system. It is a part of the OSEK standard (see www.osek-vdx.org).

The ORTI enables the attached tool to evaluate and display information about the operating system, its state, its performance, the different task states, and different operating system objects.

ORTI File and Filename

The ORTI file name has the same name as the application file name, but with the extension `.ort`. For instance, if the application file name is `winLift_demo.abs`, the ORTI file name is `winLift_demo.ort`. Otherwise the debugger cannot use the ORTI file.

The ORTI file contains dynamic information as a set of attributes that are represented by formulas to access corresponding dynamic values. Formulas for dynamic data access are comprised of constants, operations, and symbolic names within the target file. The given formula can then be evaluated by the debug tool to obtain internal values of the required OS objects.

ORTI Aware Debugging System

Two types of data are made available to the CodeWarrior debug tool. One type describes static configuration data that remains unchanged during program execution. The second type of data is dynamic and this data is re-evaluated each time by the CodeWarrior debug tool. 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 the CodeWarrior debug tool is represented in an ORTI text file. The file describes the different objects configured in the OS and their properties. The information is presented as direct text, enumerated values, symbolic names, or an equation that may be used for evaluating the attribute.

Building the project through the OSEK System Generator generates the ORTI file. The generated file has the same name and location as the executable file but with the `.ort` extension.

ORTI File Structure

The ORTI file structure builds on the structure of the OSEK OIL file. It consists of the following parts:

- Version Section describes the version of the ORTI standard used for the current ORTI file.
- Implementation Definition Section describes the proper method for interpreting the data obtained for the value. This section may also detail the suggested display name for a given attribute.
- Application Definition Section contains information on all objects currently available for a given system. This section also describes the proper method for referencing or calculating each required attribute. Supply this information either as a static value or as a formula to calculate the required value.

OSEK RTK Inspector Component

This section describes the OSEK RTK Inspector component.

Open the Inspect window by selecting **Component > Open** and clicking on the ***Inspect*** icon in the **Open Window Component** window.

CodeWarrior RTK Inspect Window

When you select the RTK components icon in the hierarchical content of the items, the right side displays a variety of information about OSEK Awareness. The OSEK RTK Inspect Window provides access to all this information. The ORTI file definition groups objects of the same type so they can be viewed together. The following object types are accessible through the Inspect window:

- Task
- Stack
- SystemTimer
- Alarm
- Message

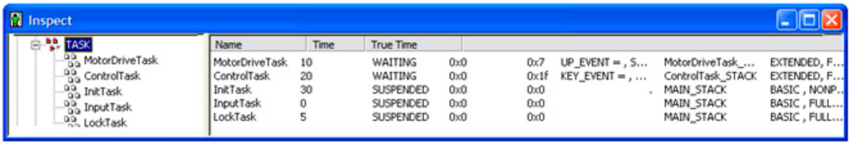
The following sections describe typical objects, their attributes and their presentation.

NOTE Objects and their attributes depend on the OSEK implementation and OSEK configuration, and therefore may differ from this description.

Inspector Task

The Task, shown in Figure 5.1, displays the current state of the OSEK task trace.

Figure 5.1 Inspector Task



When selecting a Task in the hierarchical tree on the left side of the Inspect window, additional information concerning tasks appears on the right side of the window under the following headings:

- **Name**: displays the name of the task.
- **Task Priority**: displays the priority of the task.
- **Task State**: describes the current state of the task. Possible values are READY, SUSPENDED, WAITING, RUNNING or INVALID_TASK. The ORTI file defines the different states.
- **Events State**: the event is represented by its mask. The event mask is a number in the range from 1 to 0xFFFFFFFF. Setting the event mask value to 1 activates the event. Clearing the event mask value to 0 disables the event.
- **Waited Events**: when the bit is cleared to 0, the event is not expected. When the bit is set to 1, the event is expected.
- **Task Event Masks**: describes the current task event mask.
- **Current Task Stack**: displays the name of the current stack used by the task.
- **Task Properties**: describes task properties. Possible value are BASIC, EXTENDED, NONPREMPT, FULLPREMPT, Priority value, and AUTO. The ORTI file defines the possible values.

Inspect Stack

The Stack displays the current state of OSEK stack trace.

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

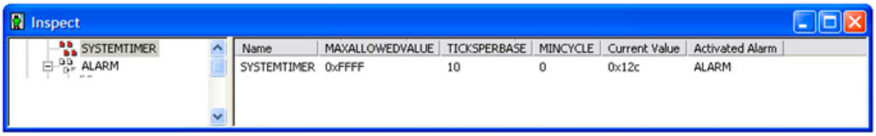
- **Name**: displays the name of the stack.
- **Stack Start Address**: displays the start address of the stack.
- **Stack End Address**: displays the end address of the stack.

- **Stack Size**: displays the size of the stack.

Inspect SystemTimer

The SystemTimer shown in Figure 5.2 displays the current state of OSEK SystemTimer trace.

Figure 5.2 Inspector SystemTimer



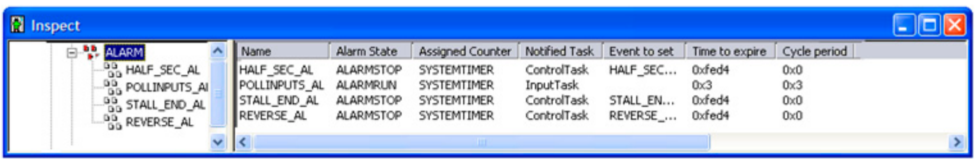
When selecting SystemTimer in the hierarchical tree on the left side, additional information concerning the timer appears on the right side of the window under the following headings:

- **Name**: displays name of the system timer.
- **MAXALLOWEDVALUE**: displays the maximum allowed counter value. When the counter reaches this value it rolls over and restarts the count 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.

Inspect Alarm

The Alarm shown in Figure 5.3 displays the current state of OSEK alarm trace.

Figure 5.3 Inspect Alarm



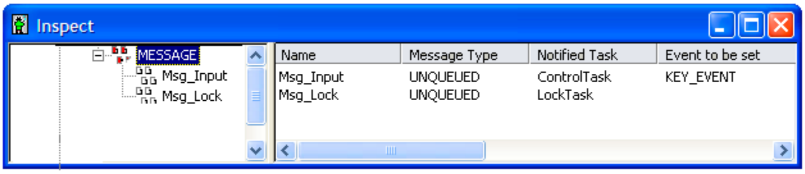
When selecting Alarm in the hierarchical tree on the left side, additional information concerning the alarm appears on the right side of the window under the following headings:

- **Name**: displays the name of the alarm.
- **Alarm State**: displays the current state of the alarm. Possible values are ALARMRUN and ALARMSTOP.
- **Assigned Counter**: based on counters, the OSEK OS offers an alarm mechanism to the application software. Assigned Counter is the name of the counter used by alarm.
- **Notified Task**: alarm management allows the user to link task activation to a certain counter value, the assignment of an alarm to a counter, and the action to be performed when an alarm expires. Notified Task defines the task to be notified (by activation or event setting) when the alarm expires.
- **Event to Set**: alarm management allows the user to link event setting to a certain counter value, the assignment of an alarm to a counter, and 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.

Inspect Message

The Message shown in Figure 5.4 displays the current state of OSEK message trace.

Figure 5.4 Inspect Message



When selecting Message in the hierarchical tree on the left side, additional information concerning task appears 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 activates when the message is sent.
- **Event to be set**: displays the event to set when the message is sent.

6

How To...

This chapter provides methods for accomplishing common tasks.

- Configuring the Debugger
- Starting the Debugger
- Switching Connections
- Using the Stationery Wizard to Create a Project
- CodeWarrior IDE Integration
- Automating Debugger Startup
- Loading an Application
- Starting an Application
- Stopping an Application
- Stepping in the Application
- Working on Variables
- Working on the Register
- Modify Content of Memory Address
- Consulting Assembler Instructions Generated by a Source Statement
- Viewing Code
- Communicating with the Application
- About startup.cmd, reset.cmd, preload.cmd, postload.cmd

Configuring the Debugger

If you have installed the Debugger under Windows® 2000 or higher, you can start the Debugger from the CodeWarrior IDE, from the desktop, from the Start menu, or from an external editor such as WinEdit or CodeWright. To work efficiently (find all requested configuration and component files), you must associate the Debugger with a working directory.

For Use from Desktop (Windows 2000)

When starting the Debugger from Windows (without WinEdit), you can define the working directory 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, set the working directory in the configuration file `MCUTOOLS.INI`.

Define the working directory, including the path, in the environment variable **DefaultDir** in the [**Options**] group or **WorkDir** [**WorkingDirectory**].

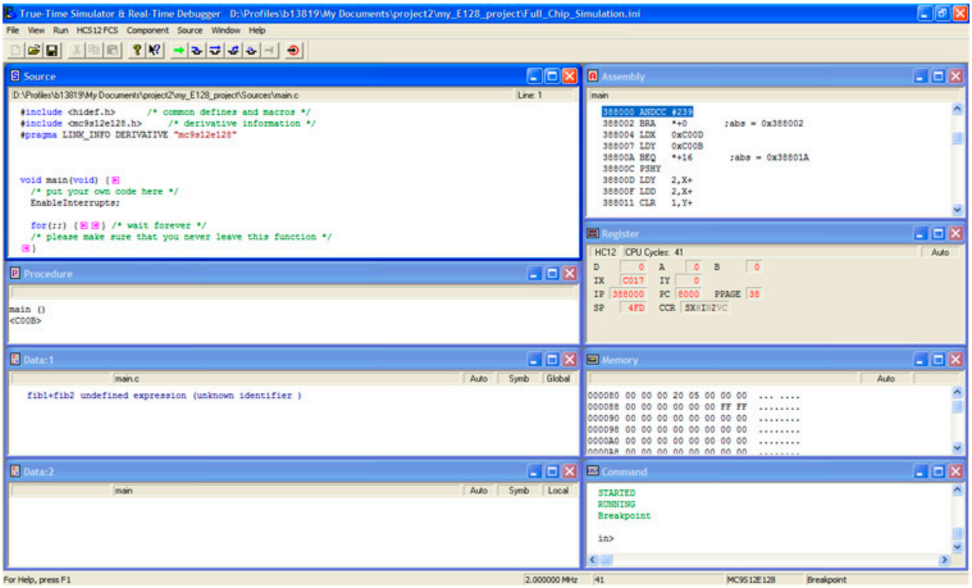
Starting the Debugger

This section explains starting the debugger using WinEdit, from within the Codewarrior IDE or from a DOS command line.

Starting with WinEdit

Start the Debugger by selecting **Project > Debug** or by clicking the Debugger icon (bug) in the WinEdit tool bar (when configured). The Debugger window looks like Figure 6.1.

Figure 6.1 Debugger after Startup



READY displayed in the status bar indicates that the simulator is ready.

Starting from within the IDE

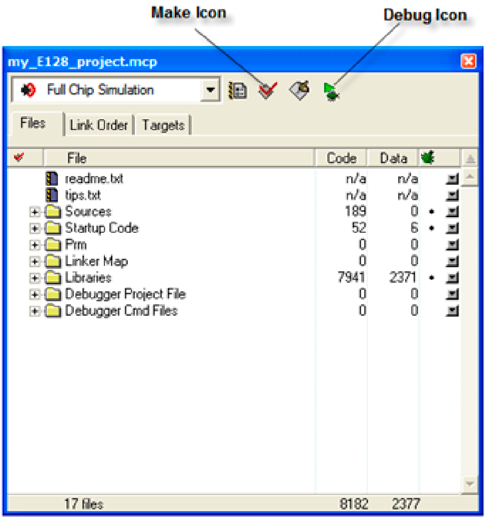
There are two ways to start the debugger from within the IDE:

- From a **Project** window icon
- From the IDE Main Window menu bar

Starting Debug from the Project Window

To start the debugger from the **Project** window, click the **Debug** icon (Figure 6.2), at the top of the Project window.

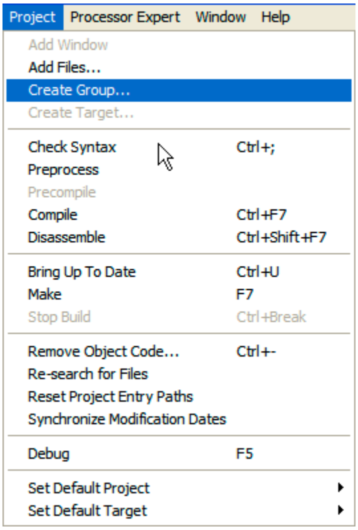
Figure 6.2 Project Window Make and Debug Icons



Starting Debug from the Main Window Menu Bar

You can also start the debugger from the main menu bar of the CodeWarrior IDE. To start the debugger from the main menu bar, select **Project** > **Debug**.

Figure 6.3 Main Window Project Menu



Debugger Command Line Start

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

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

`AbsFileName` is the name of the application to load in the debugger. Precede each option with a dash. Refer to Command Line Options for available command line commands.

Order of Commands

Commands specified by options are executed in the following order:

1. Load (activate) the project file (see below). The debugger uses `project.ini` by default, unless you specify another project file.
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

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

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

Example

```
C:\Program Files\Freescale\CWS12v5.1\Prog\DEMO\TEST.ABS -w -
d
```

Switching Connections

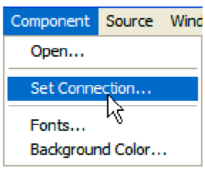
It is possible to switch connections from within an existing HC12 debugging project. The following paragraphs explain how to change the connection in debugger, although it is recommended to switch connection in project in IDE to keep consistency. If you are not using CodeWarrior IDE project then this information might be important for you.

Loading the Full Chip Simulation Connection

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

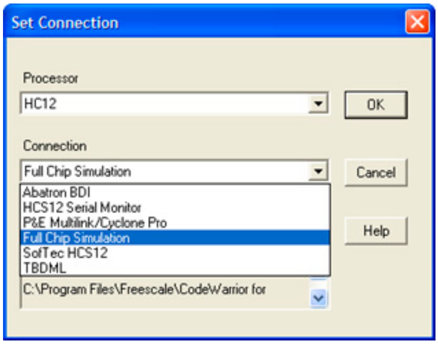
1. From the Debugger main menu, select **Component > Set Connection**, as shown below.

Figure 6.4 Component Menu



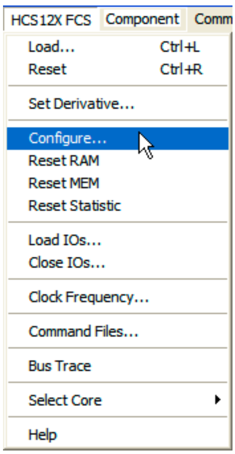
The Set Connection dialog box now appears.

Figure 6.5 Set Connection Dialog Box



2. Set the Processor as HC12 and the Connection as Full Chip Simulation.
3. Click the **OK** button. The Debugger main menu entry bar for the connection now changes to HCS12X FCS.

Figure 6.6 HCS12X FCS Menu



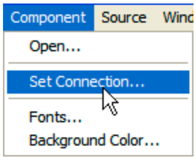
You have successfully switched connections to the FCS connection. The values and use of each HCS12X FCS menu entry is explained in the Full Chip Simulation chapter of this manual.

Loading the P&E Multilink/Cyclone Pro Connection

To load the Multilink/Cyclone Pro (ICD-12) connection from within an existing project, take the following steps:

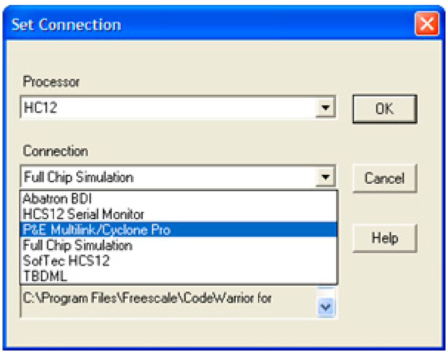
1. From the Debugger main menu, select **Component > Set Connection**, as shown below.

Figure 6.7 Component Menu



The Set Connection dialog box now appears.

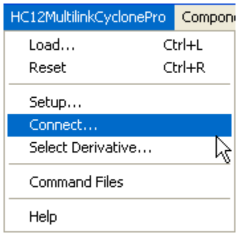
Figure 6.8 Set Connection Dialog Box - Connection Menu



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

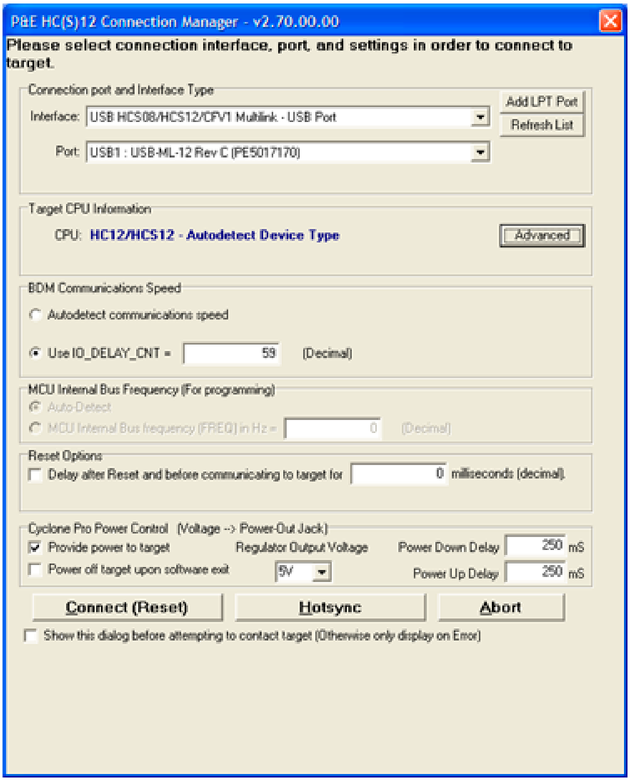
   The Connection menu selection **P&E Multilink/Cyclone Pro** loads the proper drivers, and other things for the connection.
4. In the Debugger Main window, the Connection heading has been renamed **HC12MultilinkCyclonePro**. Click on this heading to display its menu with the list of possible selections.

Figure 6.9 HC12MultilinkCyclone Pro Menu



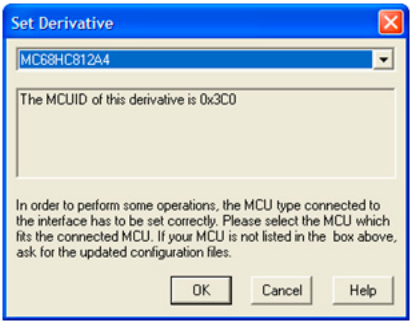
- The menu selection **HC12MultilinkCyclonePro > Load** loads an executable (`.abs`) file into connection memory. The file's program counter points to the first instruction of the startup section.
- The menu selection **HC12MultilinkCyclonePro > Reset** triggers a reset of the connection and executes the command file `reset.cmd`.
- The menu selection **HC12MultilinkCyclonePro > Connect** takes you to the P&E ICD-12, Multilink, Cyclone Pro dialog box. The two tabs of this dialog box allow you to set the Communications and Special Setup parameters for the connection.

Figure 6.10 P&E Multilink, Cyclone Pro Connection Dialog Box



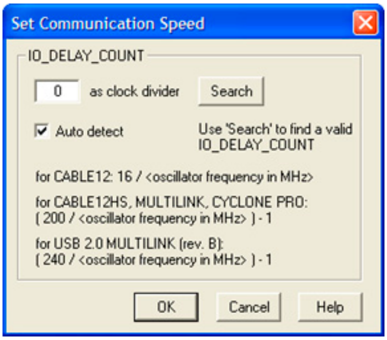
- The menu selection **HC12MultilinkCyclonePro > Select Derivative** takes you to the Set Derivative dialog box. This dialog box allows you to choose the target MCU for the connection.

Figure 6.11 Set Derivative Dialog Box



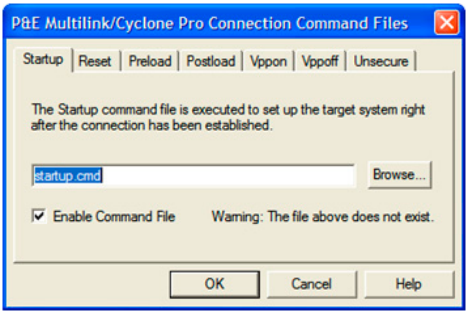
- The menu selection **MultilinkCyclonePro > Set Communication Speed** lets you control the various factors associated with communication speed for the connection.

Figure 6.12 Set Communication Speed Dialog Box



- The menu selection **MultilinkCyclonePro > Command Files** takes you to the Command Files window.

Figure 6.13 Command Files Window

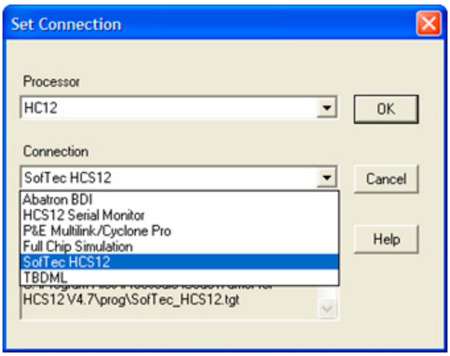


Switching to SofTec HCS12

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

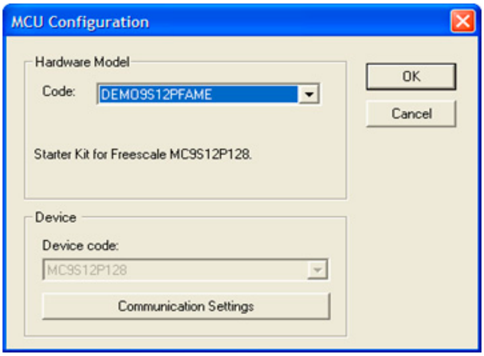
1. In the Debugger window menubar, display the Component menu.
2. Choose **Component > Set Connection** from this menu to select another connection in the Set Connection dialog box.

Figure 6.14 Set Connection Dialog Box - SofTec HCS12 Selection



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

Figure 6.15 MCU Configuration Dialog Box



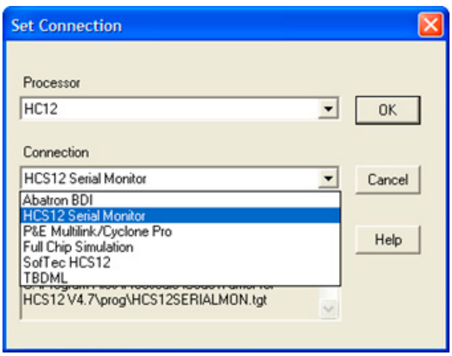
5. In the MCU Configuration dialog box, choose the correct target processor.
6. Click the **OK** button to start debugging.

Switching to HCS12 Serial Monitor Connection

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

1. In the Debugger Main window select the Component menu.
2. Choose **Component > Set Connection** to select another connection.

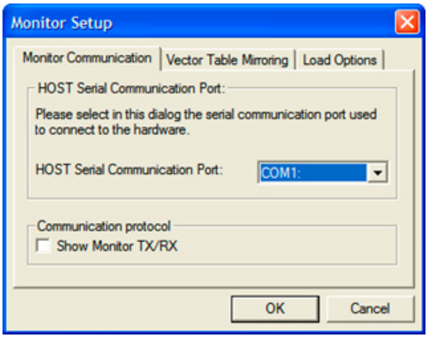
Figure 6.16 Set Connection Dialog Box - HCS12 Serial Monitor Selection



3. Select **HC12** as Processor then **HCS12 Serial Monitor** as the connection in the Set Connection dialog box and click the OK button.

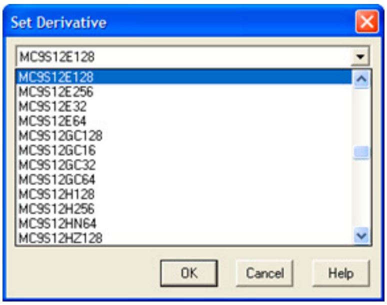
4. In the **Monitor Setup** window Monitor Communication tab, choose the correct Host serial communication port if necessary.

Figure 6.17 Monitor Setup Window - Monitor Communication Tab



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

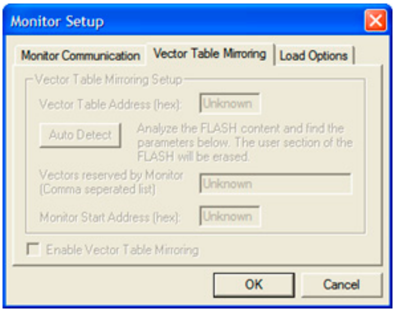
Figure 6.18 Derivative Selection Dialog Box



7. Click the OK button.

   The **Monitor Setup** window opens again, showing the Vector Table Mirroring Tab. We recommend that you use the Vector Table Mirroring feature. Otherwise, vectors cannot be programmed as captured, or protected from erasing or overwriting by the HCS12 Serial Monitor.
8. To enable this specific feature, check the **Enable Vector Table Mirroring** checkbox.

Figure 6.19 Monitor Setup Window - Vector Table Mirroring Tab



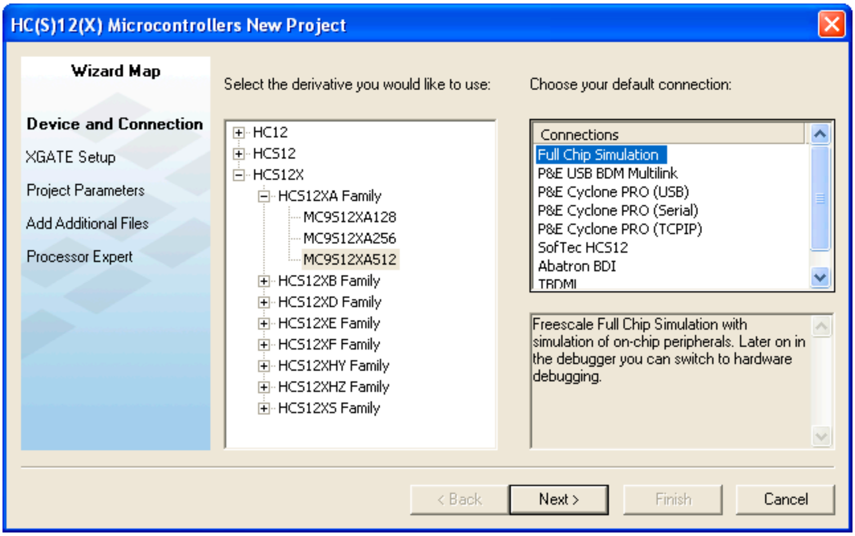
9. Click the **Auto Detect** button to make the debugger search for the vector table address and vectors reserved by the HCS12 Serial Monitor.
10. Once the auto-detection completes, click the **OK** button to start debugging.

Using the Stationery Wizard to Create a Project

Debugging HC12 code using the CodeWarrior IDE requires that a project be created or exist which specifies a connection that can be used to debug the code. To take the first steps toward debugging with CodeWarrior IDE using the stationery Wizard:

1. Run the *CodeWarrior IDE* with the shortcut created in the program group.
2. Choose **File > New Project** to create a new project from a stationery - the **HC(S)12(X) Microcontrollers New Project** wizard screen appears.

Figure 6.20 HC(S) 12(X) Microcontrollers New Project Screen



3. In the tree navigate to the family and select derivative, for example **HCS12X > HCS12XA Family > MC9S12XA512**.
4. Select the connection by clicking on the appropriate connection.

   Selecting any of the options results in the following conditions:

   - Full Chip Simulation — Connects to Freescale Full Chip Simulation with simulation of on-chip peripherals. With this selection, you can switch to hardware debugging later in the debugging session.
   - P&E USB BDM Multilink — Connect to P&E USB BDM Multilink. This development tool allows access to the Background Debug Mode (BDM) on Freescale HCS12(X) microcontrollers to directly control the target's execution, read/write registers and memory values, debug code on the processor, and program internal or external FLASH memory devices.
   - P&E Cyclone PRO (USB) — Connect to P&E Cyclone PRO via USB port. This flexible tool is designed for in-circuit flash programming, debugging, and testing of Freescale HCS12(X) microcontrollers in development and production environments. The Cyclone PRO can be operated in interactive or batch mode. Once loaded with data it can be disconnected and operated manually in stand-alone mode via the LCD menu and control buttons. The Cyclone PRO has over 3 MB of non-volatile memory, which allows the onboard storage of multiple programming images. When connected to a computer for programming or loading it can communicate via Ethernet, USB, or serial interfaces.

- P&E Cyclone PRO (Serial) — Connect to P&E Cyclone PRO via serial port. This flexible tool is designed for in-circuit flash programming, debugging, and testing of Freescale HCS12(X) microcontrollers in development and production environments. The Cyclone PRO can be operated in interactive or batch mode. Once loaded with data it can be disconnected and operated manually in stand-alone mode via the LCD menu and control buttons. The Cyclone PRO has over 3 MB of non-volatile memory, which allows the onboard storage of multiple programming images. When connected to a computer for programming or loading it can communicate via Ethernet, USB, or serial interfaces.
- P&E Cyclone PRO (TCP/IP) — Connect to P&E Cyclone PRO via Ethernet port. This flexible tool is designed for in-circuit flash programming, debugging, and testing of Freescale HCS12(X) microcontrollers in development and production environments. The Cyclone PRO can be operated in interactive or batch mode. Once loaded with data it can be disconnected and operated manually in stand-alone mode via the LCD menu and control buttons. The Cyclone PRO has over 3 MB of non-volatile memory, which allows the onboard storage of multiple programming images. When connected to a computer for programming or loading it can communicate via Ethernet, USB, or serial interfaces.
- OSBDM — Connect to Freescale Open Source BDM circuit via USB port. This on-board interface provides basic run control and internal FLASH programming support for a resident processor on an evaluation platform.
- SofTec HCS12 — Connects to any of the USB-based SofTec Microsystems tools for the HC12 (inDart-HCS12, etc.).

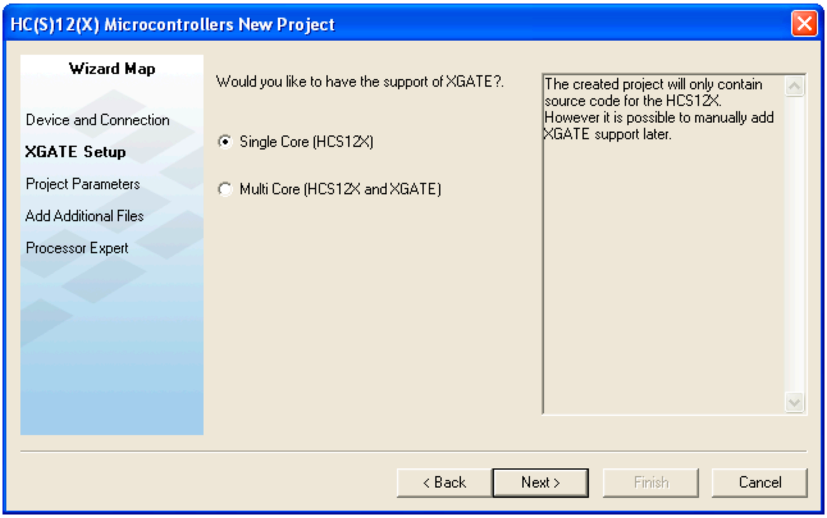
Depending on derivative selected, the following connections may also be available:

- Abatron BDI — Connect to the hardware board using Abatron hardware (BDI-HS or BDI 1000) through the BDM connection.
- TBDML — Connect to a board through Freescale TBDML (TurboBDM Light).

NOTE CodeWarrior IDE provides Change MCU/Connection wizard to easily modify a project later. For more information, refer to Change MCU/Connection Wizard section in S12(X) Build Tools Reference Manual (`C:\Program Files\Freescale\CWS12v5.1\Help\pdf`).

5. Click **Next** to display next page of the wizard. The **XGATE Setup** screen appears.

Figure 6.21 XGATE Setup Screen



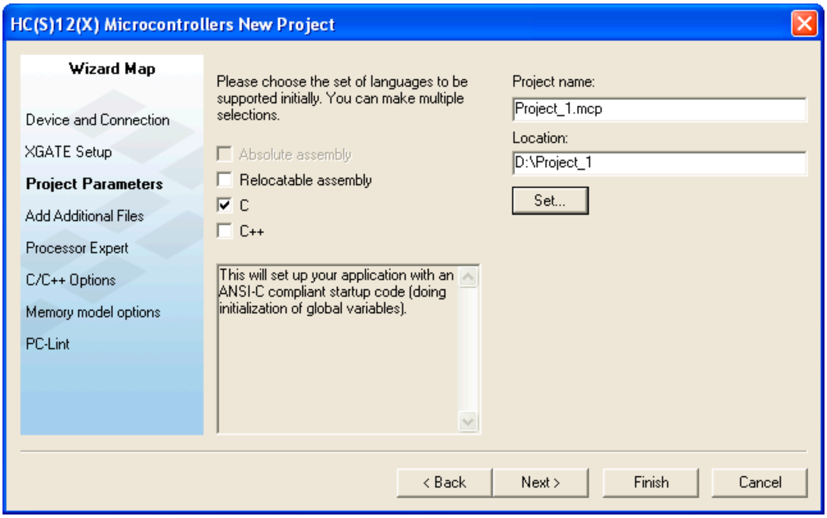
6. This screen appears only for selected derivatives that support XGATE. Unless you need XGATE support, select the **Single Core (HCS12X)** format by clicking its radio button.

   Selecting any of the options results in the following conditions:

   - Single Core (HCS12X) - The created project only contains source code for the HCS12X. However, it is possible to add XGATE support at a later date manually.
   - Multi Core (HCS12X and XGATE) - The created project contains source code for the HCS12X and the XGATE.

7. Click **Next** to continue. The **Project Parameters** screen appears.

Figure 6.22 Project Parameters Screen



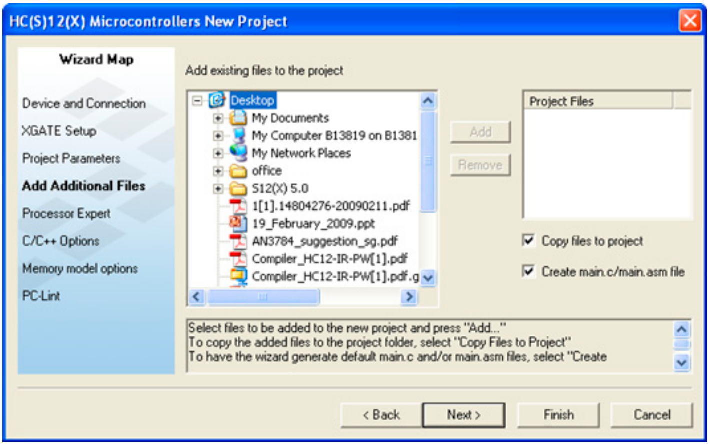
8. Select the language format by checking its checkbox.

   You can make multiple selections, creating the code in multiple formats. Selecting any of the options results in the following conditions:

   - Absolute Assembly - Using only one single assembly source file with absolute assembly. There is no support for relocatable assembly or linker.
   - Relocatable Assembly - It supports to split up the application into multiple assembly source files. The source files are linked together using the linker.
   - C - This sets up your application with ANSI C-compliant startup code, and initializes global variables.
   - C++ - This sets up your application with ANSI C++ startup code, and performs global class object initialization.

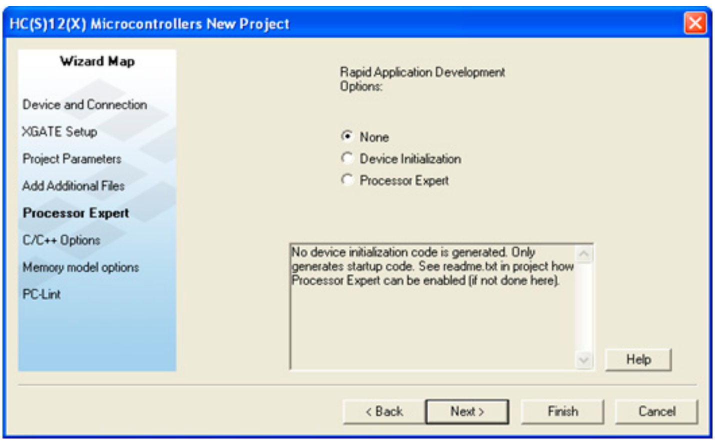
9. In the **Project name** textbox, type the name of your new project.
10. Click **Next** to continue. The **Add Additional Files** screen appears.

Figure 6.23 Add Additional Files Screen



11. Select files to be added to the new project and click **Add** button. You can also select checkbox to:
    - Copy files to project - To copy the added files to the project folder.
    - Create main.c/main.asm file - To have the wizard generate default main.c and/or main.asm files.
12. Click **Next** to continue. The **Processor Expert** screen appears.

Figure 6.24 Processor Expert Screen



13. This screen appears only for selected derivatives that offer Processor Expert support as well as it also depends on other project settings. For example

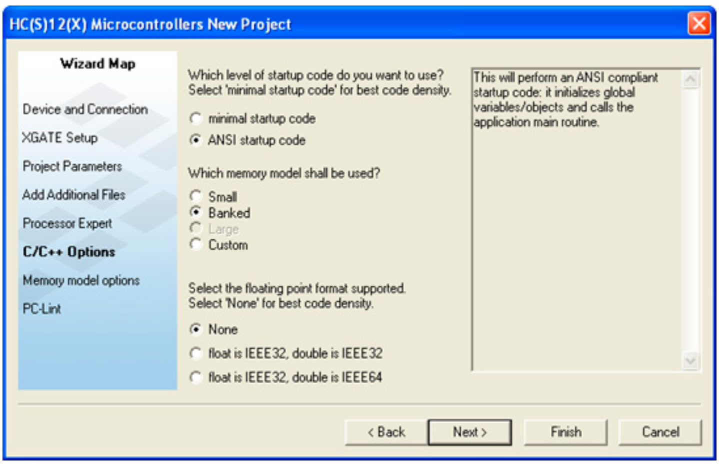
    - Processor Expert is not available for projects with XGATE, and
    - Processor Expert is not available for projects with absolute assembly or C++.

    Selecting any of the rapid application development options results in the following conditions:

    - None - No device initialization code is generated. Only generates startup code. See readme.txt in project to know how Processor Expert can be enabled (if not done here).
    - Device Initialization - The tool can generate initialization code for on-chip peripherals, interrupt vector table and template for interrupt vector service routines.
    - Processor Expert - Processor Expert can generate for you all the device initialization code. It includes many low-level drivers.

14. Click **Next** to continue. The **C/C++ Options** screen appears.

Figure 6.25 C/C++ Options Screen



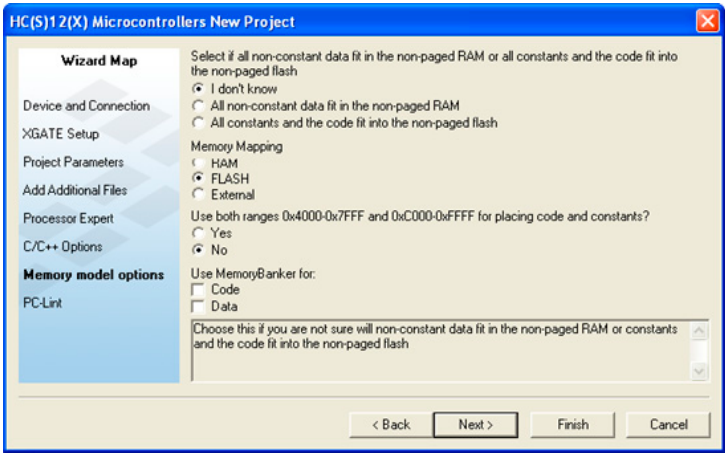
15. The C/C++ options screen lets you select the level of **Startup Code** you wish to produce. Selecting either of the options results in the following conditions:

    - Minimal startup code — This option produces the best code density. The startup code initializes the stack pointer and calls the main function. No initialization of global variables is done, giving the user the best speed/code density and a fast startup time. The application code must address variable initialization. This means this option is <u>not</u> ANSI compliant, since ANSI requires variable initialization.

- ANSI startup code — This ANSI-compliant startup code initializes global variables/objects and calls the application main routine.

16. Select the **Memory Model** by clicking the appropriate radio button. Selecting any of the options results in the following conditions:
    - Small — Use the Small memory model if both the code and the data fit into the 64-kilobyte address space. By default all variables and functions are accessed with 16-bit addresses. The compiler supports banked functions or paged variables in this memory model, but all accesses must be explicitly handled.
    - Banked — Banked memory model uses banked function calls by default, but the default data access is still 16-bit. Because the overhead of the `far` function call is not very large, this memory model suits all applications with more than 64-kilobytes of code. Data paging can be used, however all `far` objects and pointers to them must be specially declared.
    - Large — The Large memory model supports both code banking and data paging by default. However, data paging requires a lot of overhead and should be used with care. Overhead is significant with respect to both code size and speed. If it is possible to manually use `far` accesses to any data which does not fit into the 64-bit address space, then use the banked memory model instead.
    - Custom — The Custom memory model allows you to configure the project to support both code banking and data paging. It allows the build tools to optimize accesses and generate more efficient code than the Large memory model, without the need for the programmer to manually place data. Note that any application can be written using the Banked memory model instead, and the generated code will be more efficient than using the custom memory model. The cost is that the user must manually place data that does not fit in the first 64k by means of pragmas.
17. Select the floating point format by clicking the appropriate radio button. Selecting any of the options results in the following conditions:
    - None — Don’t use floating point for the HC12.
    - Float is IEEE32, double is IEEE32 — All float and double variables are 32-bit IEEE32 for the HC12.
    - Float is IEEE32, double is IEEE64 — Float variables are 32-bit IEEE32. Double variables are 64-bit IEEE64 for the HC12.
18. Click **Next** to continue. The **Memory model options** screen appears. The Memory model options are available for derivatives from HCS12X family.

Figure 6.26 Memory model options Screen



19. The Memory model options screen lets you select if all non-constant data fit in the non-paged RAM or all constants and the code fit into the non-paged flash.

- I don't know — Choose this option if you are not sure whether non-constant data fit in the non-paged RAM or constants and the code fit into the non-paged flash.
- All non-constant data fit in the non-paged RAM — Choose this in either one of the following situations:
  - Your non-constant data fit into 12k and you do not plan on accessing non-paged RAM areas through RPAGE. If you choose this and still do accesses through the RPAGE register the compiler generated code may be incorrect. Accesses through RPAGE include accesses through __rptr-qualified pointers and accessed to variables defined in __RPAGE_SEG sections.
  - You have less than 8k non-constant data. If this is the case you can also do accesses through RPAGE.

WARNING! This will induce non-ANSI behavior in the compiler. When accessing constant data by means of pointer to non-const the compiler may produce code that will not meet the required functionality.

- All constants and the code fit into the non-paged flash — Choose this in either one of the following situations:
  - Your constants and code fit into 48k of flash and you do not plan on accessing non-paged flash areas through PPAGE. If you choose this and still do accesses through the PPAGE register (e.g. calling a far function) the code may be incorrect.

– You have less than 32k constants and code. If this is the case you can also do accesses through PPAGE.

WARNING! This will induce non-ANSI behavior in the compiler: when accessing non-constant data by means of pointer to const the compiler may produce code that will not meet the required functionality. This is also true when accessing constant members of structures that are not constant by means of pointer to const.

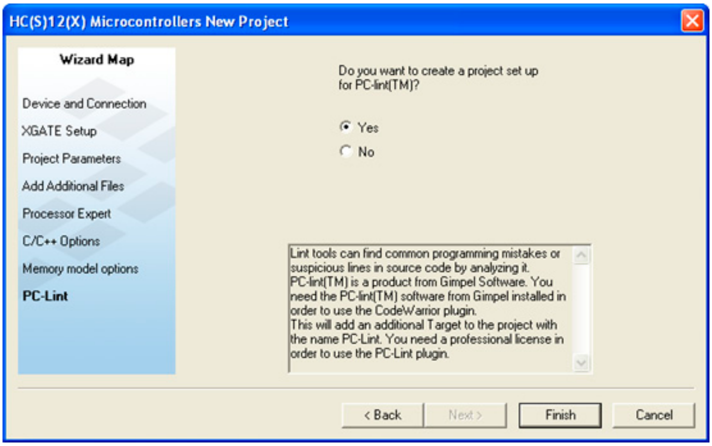
20. Select the **Memory Mapping** format by clicking the appropriate radio button. The Memory Mapping option is supported on some of HCS12X devices as new mode. The default selection is FLASH. Selecting any of the options results in the following conditions:
    - RAM — Maps accesses to 0x4000-0x7FFF to 0x0F_C000-0x0F_FFFF in the global memory space (RAM area). More RAM will be available in the local memory map.
    - FLASH — Maps accesses to 0x4000-0x7FFF to 0x7F_4000-0x7F_7FFF in the global memory space (FLASH). More flash will be available in the local memory map.
    - External — Maps accesses to 0x4000-0x7FFF to 0x14_4000-0x14_7FFF in the global memory space (external access).
21. Select the **Use MemoryBanker** by clicking the appropriate check box. MemoryBanker is an automation tool that optimizes the layout of code and data in order to minimize the application's memory footprint. In the first pass it gathers information about the application and generates the layout, while in the second pass it generates the optimized code. It can be used to generate an optimal distribution for functions, for data or for both functions and data.

    Selecting any of the options results in the following conditions:
    - Code — The project will be setup for code optimization.
    - Data — The project will be setup for data optimization. The data optimization is available for Custom memory only, this selection is disabled for Banked memory model.

NOTE It is not possible to use MemoryBanker for project with Processor Expert, XGATE or OSEK.

22. Click **Next** to continue. The **PC-lint** options screen appears.

Figure 6.27 PC-lint Options Screen



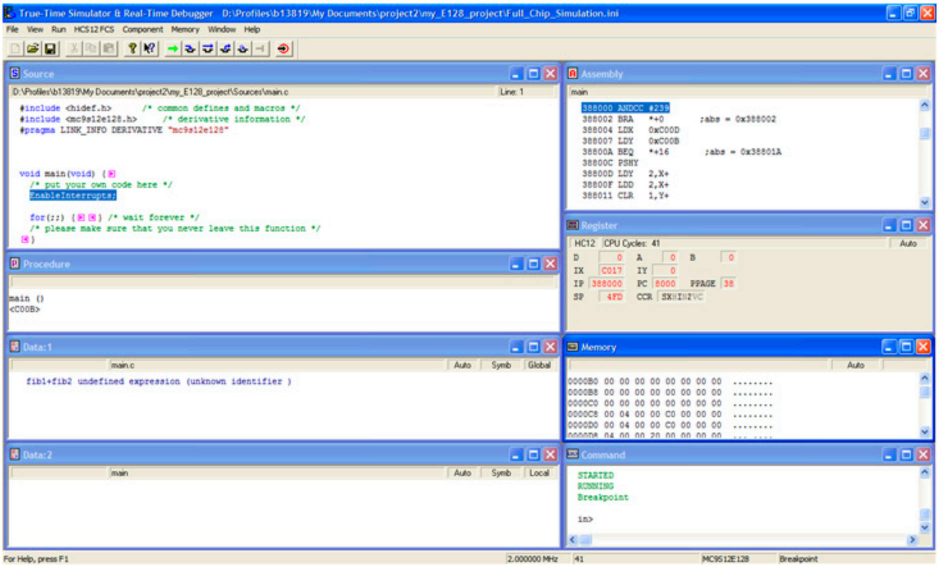
23. Unless you wish to create a project set up for PC-lint, select **No**.

    While Lint tools can find common programming mistakes or suspicious lines in source code by analyzing it, you need to install the PC-lint software from Gimpel to use the CodeWarrior plug-in. You can enable PC-lint later by manually cloning a target and changing the linker to PC-lint linker.

    Selecting the Yes option adds an additional target to the project with the name PC-Lint. Using the PC-lint plug-in requires a professional license.

24. Click the **Finish** button. The IDE opens.
25. In the IDE main window, choose **Project > Make**.
26. Now choose **Project > Debug** to start the debugger.

Figure 6.28 Your Project in Debugger Main Window



CodeWarrior IDE Integration

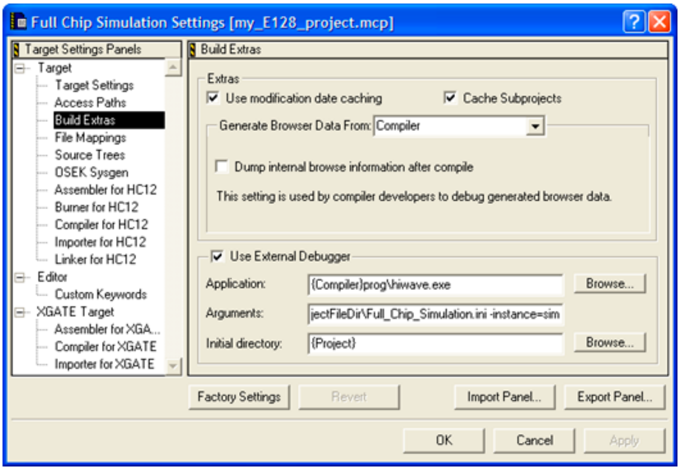
This section provides information on how to use and configure the Simulator/Debugger within the CodeWarrior IDE using the CodeWarrior IDE - HC12 version 4.5 or later.

Debugger Configuration

To configure the Simulator/Debugger in the CodeWarrior IDE, open the **Target Settings Panel** by clicking on the Targets panel of the IDE main window, then double clicking on the name of your target in the list displayed in this panel.

1. Select **Build Extras** as shown in Figure 6.29.
2. In the **Build Extras** pane check the **Use External Debugger** checkbox.
3. In the Application field, type the Debugger path, or select from the Open window by clicking the Browse button; for example: **{Compiler}prog\hiwave.exe**.
4. In the Arguments field, type the arguments; for example, **%targetFilePath -Target=sim**.
5. Click on **Apply** to validate these changes.

Figure 6.29 IDE Target Window - Build Extras Panel



Automating Debugger Startup

Often you must perform the same tasks every time you start the Debugger. Automate these tasks by writing a command file that contains all commands to be executed after startup of the Debugger, as shown in Listing 6.1.

Listing 6.1 Example of a Command File to Automate Tasks

```
load fibo.abs
bs &main t
g
```

This file above first loads an application, then sets a temporary breakpoint at the start of the function **main**, and then starts the application. The application stops 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**: Do this in the application that starts the Debugger (for example, Editor, Explorer, or Make utility).

  Example:

  ```
  \Freescale\PROG\HIWAVE.EXE -c init.cmd
  ```

When you start the Debugger with this command line, it executes the command specified in the file `init.cmd` after loading the layout (or project file).

- Call the command file from the project file (Listing 6.2). The project file, where the layout and connection component can be saved (**File > Save**), is a normal text file that contains command line commands to restore the context of a project. After creating this file with the save command, you can extend it with a call to the command file (**CALL INIT.CMD**). When this project is loaded by the **File > Open** command or by the corresponding entry in the Project file, commands in this file are executed.

Listing 6.2 Calling a Command File from the Project File

```
set Sim
CLOSE *
call \Freescale\DEMO\test.hwl
call init.cmd
```

- Call the command file when the Connection component is loaded. Most connection components execute the command file `STARTUP.CMD` once the connection component is loaded and initialized. By adding the call command file in this file (for example, **CALL INIT.CMD**), it automatically executes when the connection component loads.

NOTE Refer to section Starting the Debugger.

Loading an Application

1. Choose **HC12FCS > Load**. The **LoadObjectFile** dialog box opens.
2. Select an application (for example `FIBO.ABS`).
3. Click **OK**. The dialog box closes and the application loads 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.

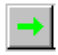
The Global Data component contains the list of global variables defined in the module containing the application entry point.

The Local Data component is empty.

The PC in the Register component is initialized with the PC value from the application entry point.

Starting an Application

There are two different ways to start an application:

- Choose **Run > Start/Continue**
- Click the **Start > Continue** icon in the debugger tool bar 

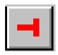
RUNNING in the status line indicates that the application is running.

The application continues execution until:

- You decide to stop the execution (See Stopping an Application).
- The application reaches a breakpoint or watchpoint.
- The application detects an exception (watchpoints or breakpoints).

Stopping an Application

There are two different ways to stop program execution:

- Choose **Run > Halt**
- 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.

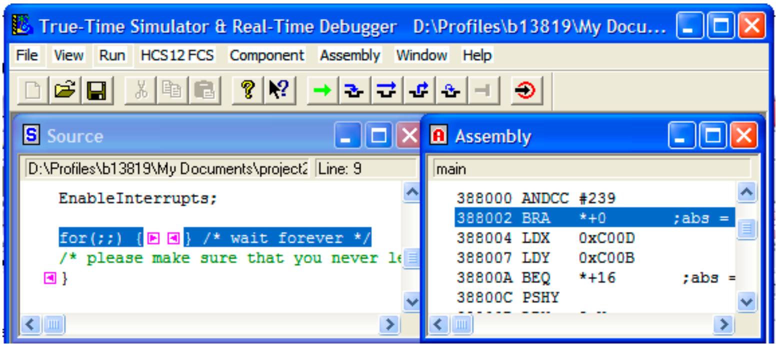
Data window with attribute **Local** displays the name and values of the local variables defined in the current procedure. The name of the procedure is specified in the Data info bar.

Stepping in the Application

The Debugger provides stepping functions at the application source level and assembler level (Figure 6.30).

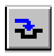
On Source Level

Figure 6.30 Stepping on Source Level



On the Next Source Instruction

The Debugger provides two ways of stepping to the next source instruction:

- Choose **Run > Single Step**
- Click the **Single Step** icon from the Debugger tool bar 

STEPPED in the status line indicates that the application is stopped by a step function.

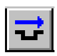
If the application was previously stopped on a subroutine call instruction, a **Single Step** stops the application at the beginning of the invoked function.

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 appear in red are the register, memory position, local or global variables, and the values that changed during execution of the source statement.

Step Over a Function Call (Flat Step)

The Debugger provides two ways of stepping over a function call:

- Choose **Run > Step Over**
- Click the **Step Over** icon from the Debugger tool bar 

STEPPED OVER (STEPOVER) or **STOPPED** (STOP) in the status line indicates that the application is stopped by a step over function.

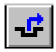
If the application was previously stopped on a function invocation, a **Step Over** stops the application on the source instruction following the function invocation.

The display in the Assembly component is always synchronized with the display in the Source component. The highlighted instruction in the Assembly component is the first assembler instruction generated by the highlighted instruction in the Source component.

Elements from Register, Memory, or Data components that appear in red are the register, memory position, local or global variables, and the values that changed during execution of the invoked function.

Step Out from a Function Call

The Debugger provides two ways of stepping out from a function call:

- Choose **Run > Step Out**
- Click the **Step Out** icon from the debugger tool bar 

STOPPED (STOP) in the status line indicates that the application is stopped by a step out function.

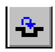
If the application was previously stopped in a function, a **Step Out** stops the application on the source instruction following the function invocation.

The display in the Assembly component is always synchronized with the display in the Source component. The highlighted instruction in the Assembly component is the first assembler instruction generated by the highlighted instruction in the Source component.

Elements from Register, Memory, or Data components that appear in red are the register, memory position, local or global variables, and the values that changed since the **Step Out** was executed.

Step on Assembly Level

The Debugger provides two ways of stepping to the next assembler instruction:

- Choose **Run > Assembly Step**
- 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 appear in red are the register, memory position, local or global variables, and the values that changed during execution of the assembler instruction.

Working on Variables

This section shows the different methods to work on variables.

Display Local Variable from a Function

The Debugger provides two different ways to see the list of local variables defined in a function:

- Using Drag and Drop
  - Drag a function name from the Procedure component to a Data component with attribute **local**.
- Using Double click
  - Double click a function name in the Procedure component.

The Data component (with attribute **local** that is neither **frozen** or **locked)** displays the list of variables defined in the selected function with their values and type.

Display Global Variable from a Module

The Debugger provides two ways to see a list of global variables defined in a module:

Opening Module Component

1. Choose **Component > Open**. The list of all available components appears on the screen.
2. Double click the entry **Module**. A module component opens, 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 Context Menu

1. Right click in a Data component with attribute **Global**.
2. Choose **Open Module** in context menu. A dialog box opens, which contains the list of all modules building the application.
3. Double click on a module name. The Data component with attribute **global**, which is neither **frozen** nor **locked**, is the destination component.

The destination Data component displays the list of variables defined in the selected module with their values.

Change Format for Variable Value Display

The Debugger allows you to see the value of variables in different formats. This is set by entries in the **Format** menu (Table 6.1).

Table 6.1 Debugger Display Format

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

The following variances apply for different variable types:

- Values for pointer variables appear in hexadecimal format.
- Values for function pointer variables appear as function name.
- Values for character variables appear in ASCII character and decimal format.
- Values for other variables appear in signed or unsigned decimal format depending on the condition of the variable.

Activate the **Format** menu as follows:

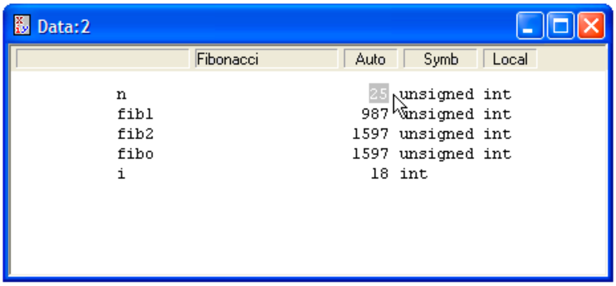
1. Right click in the Data component. The Data context menu appears on the screen.
2. Choose **Format** from the context menu. The list of all formats appears on the screen.

The format selected is valid for the whole Data component. Values from all variables in the data component appear according to the selected format.

Modify a Variable Value

The Debugger allows you to change the value of a variable, as shown in Figure 6.31.

Figure 6.31 Modifying a Variable Value



Double click on a variable. The current variable value is highlighted and can be edited.

- Formats for the input value follow the rule from ANSI-C constant values
  - Prefixed hexadecimal value with 0x
  - Prefixed octal values with 0
  - 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.
- To validate the input value you can either click the **Enter** or **Tab** key.
- If you validate an input value using the **Tab** key, the value of the next variable in the component is automatically highlighted (this value can also be edited).
- To restore the previous variable value, click the **Esc** key or select another variable.

A local variable can be modified when the application is stopped. Since these variables are located on the stack, they do not exist while the function where they are defined is inactive.

Retrieve the Variable Allocation Address

The Debugger provides you with the start address and size of a variable if you do the following:

1. Point to a variable name in a Data Component
2. Click the variable name

The start address and size of the selected variable appears in the Data information bar.

Inspect Memory Starting at a Variable Location Address

The Debugger provides two ways to dump the memory starting at a variable allocation address.

- Using Drag and Drop, drag a variable name from the Data Component to Memory component.
- Holding down the left mouse button and clicking the **A** key
  - Point to a variable name in a Data Component.
  - Hold the left mouse button down and click the **A** key.

The Memory component scrolls until it reaches the address where the selected variable is allocated. The memory range corresponding to the selected variable is highlighted in the memory component.

Load an Address Register with the Variable Address

The Debugger allows you to load a register with the address where a variable is allocated.

Drag a variable name from the Data Component to Register component. This updates the destination register with the start address of the selected variable.

Working on the Register

This section describes working with the Register component.

Change Format of Register Display

The Debugger allows you to display the register content in hexadecimal or binary format.

1. Right click in the Register component. The Register context menu appears.
2. Choose **Options** from the context menu. The list menu containing the possible formats appears.
3. Select either binary or hexadecimal format.

The format selected is valid for the entire Register component. The contents from all registers appear according to the selected format.

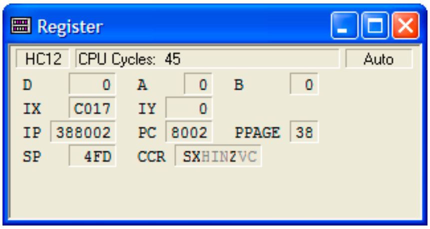
Modify a Register Content

The Debugger allows you to change the content of indexes, accumulators or bit registers.

Modify Index or Accumulator Register Content

Double click a register. The current register content is highlighted and may be edited.

Figure 6.32 Modifying Index or Accumulator Register Content



- 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.
- To validate the input value click either the **Enter** or **Tab** key, or select another register.
- If you validate an input value using the **Tab** key, the content of the next register in the component is automatically highlighted. This register can also be edited.
- To restore the previous register content, click the **Esc** key.

Modify Bit Register Content

In a bit register, each bit has a specific meaning (a Status Register (SR) or Condition Code Register (CCR)).

Mnemonic characters for bits that are set to 1 appear in black, whereas mnemonic characters for bits that are cleared to 0 appear in gray.

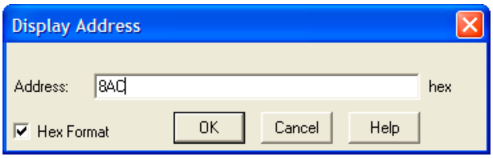
Toggle single bits inside the bit register by double clicking the corresponding mnemonic character.

Start Memory Dump at Selected Register Address

The Debugger provides two ways to dump memory starting at the address to which a register points.

- Using Drag and Drop, drag a register from the Register component to Memory component.
- Choose **Address**

Figure 6.33 Memory menu Display Address



1. Right click in the Memory component. The Memory context menu appears.
2. Choose **Address** from the context menu. The **Memory** dialog box shown in Figure 6.33 opens.
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 treated as hexadecimal numbers. Otherwise, type expressions and prefix Hex numbers with **0x** or **$**.

Modify Content of Memory Address

The Debugger allows you to change the content of a memory address. Double click the memory address you want to modify. Content from the current memory location is highlighted and can be edited.

- 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.
- 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.

- To stop editing and validate the last input value, click either the **Enter** or **Tab** key, or select another variable.
- To stop editing and restore the previous memory value, click the **Esc** key.

Consulting Assembler Instructions Generated by a Source Statement

The Debugger provides an on-line disassembly facility, which allows you to disassemble the hexadecimal code directly from the Debugger code area. Perform online disassembly in one of the following ways:

Using Drag and Drop

1. In the Source component, select the section you want to disassemble.
2. Drag the highlighted block to the Assembly component.

Holding down the left mouse button and pressing the R key

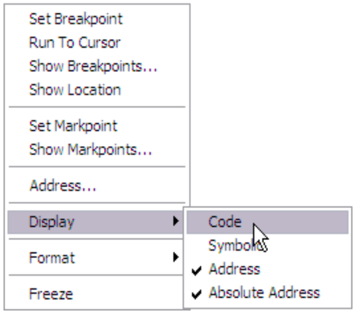
1. In the Source component window, point to the instruction you want to disassemble.
2. Hold down the left mouse button and click the R key

The disassembled code associated with the selected source instruction appears gray in the Assembly component.

Viewing Code

The Debugger allows you to view the code associated with each assembler instruction.

Figure 6.34 Viewing Code Associated with Assembler instruction.



Perform online disassembly in one of the following ways:

Using Context Menu

1. Point in the Assembly component and right click. The Assembly Context Menu appears.
2. Choose **Display > Code** (Figure 6.34).

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.

Communicating with the Application

The Debugger has a pseudo-terminal facility. Use the **TestTerm or Terminal** component window to communicate with the application using specific functions defined in the `TERMINAL.H` file and used in the calculator demonstration file.

1. Start the Debugger and choose **Open** from the Component menu.
2. Open the **TestTerm or Terminal** Component.
3. Choose **Load** from the Simulator menu.
4. Load the program `CALC.ABS`.

The target application retrieves data entered in the **TestTerm** or **Terminal** component window through the keyboard using the **Read** function. The target application sends data to the Terminal component window of the host with the **Write** function.

About startup.cmd, reset.cmd, preload.cmd, postload.cmd

The command files `startup.cmd`, `reset.cmd`, `preload.cmd`, and `postload.cmd` are Debugger system command files. All these command files do not exist automatically. They could be installed when installing a new connection.

However, the Debugger recognizes these command files and executes them.

- `startup.cmd` executes when a connection is loaded (the target defined in the **project.ini** file or loaded when you select **Component > Set Connection**).
- `reset.cmd` executes when you select ***Connection Name*** **> Reset** in the menu (**Connection Name** is the real name of the connection, such as **MMDS0508**, etc.).
- `preload.cmd` executes before loading an `.ABS` application file or S-Records file (when you select ***Connection Name*** **> Load** in the menu).
- `postload.cmd` executes after loading an `.ABS` application file or S-Records file (when you select ***Connection Name*** **> Load** in the menu).

Depending on the connection used, the Debugger recognizes other command files. Refer to the appropriate connection manual for information and properties of these command files.

7

CodeWarrior Integration

This chapter provides information on how to use and configure the Simulator/Debugger within the CodeWarrior IDE using the following software:

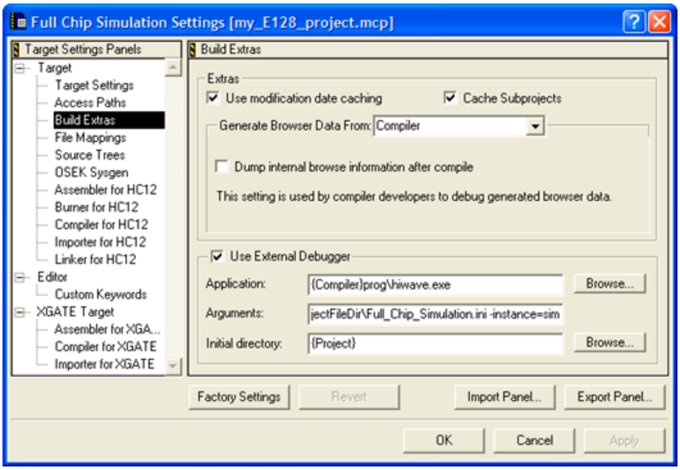
- CodeWarrior IDE - HC12 version 4.5 or later
- Debugger Configuration

Debugger Configuration

To configure the Simulator/Debugger in the CodeWarrior IDE, open the **Target Settings Panel** by clicking on the Targets panel of the IDE main window, then double clicking on the name of your target in the list displayed in this panel.

1. Select **Build Extras** as shown in Figure 7.1.
2. In the **Build Extras** pane check the **Use External Debugger** checkbox.
3. In the Application field, type the Debugger path, or select from the Open window by clicking the Browse button; for example: **{Compiler}prog\hiwave.exe**.
4. In the Arguments field, type the arguments in the Argument field; for example, **%targetFilePath -Target=sim**.
5. Click on **Apply** to validate these changes.

Figure 7.1 IDE Target Window - Build Extras Panel



8

Debugger COM Capabilities

The debugger provides the Component Object Model (COM) Interface which allows the user to control debugger using scripts or other application.

This chpater has following two sections:

- COM Implementation
- Driving Debugger through COM

COM Implementation

The debugger has COM server and client implementation.

The COM application name of the server is `Metrowerks.Hiwave`.

The Debugger COM support allows you to execute almost any command available from within the debugger (from Command line).

Driving Debugger through COM

The COM implementation in the Debugger allows you to drive it easily by using the commands from a script or application or another Hiwave instance. You can find simple script example in the `(CodeWarrior_Examples)\Scripting\PERL)` directory.

There are the following commands implemented in Hiwave to support COM - COM_START, COM_EXE, COM_EXIT.

To use the COM interface one should create the instance of Hiwave and register it as COM server, this could be done with starting it with option `-RegServer`. Once the COM server registered it is possible to execute any command available in command window.

9

Synchronized Debugging through DA-C IDE

This chapter provides information on using and configuring Freescale tools within the Development Assistant for C (DA-C) IDE. For more information on DA-C, refer to the *Development Assistant for C* documentation v 3.5.

You must be running DA-C - version 3.5 build 555 or later - (Development Assistant for C - RistanCASE).

This chapter contains the following sections:

Configuring DA-C IDE for Freescale Tool Kit

Install the DA-C software. The Freescale CD contains a demonstration version located in `\Addons\DA-C`. Run **Setup** to install the **Typical** installation.

Complete the following steps to make efficient use of Freescale Tools within DA-C IDE:

- Create a new project
- Configure the working directories
- Configure the file types
- Configure the Freescale library path
- Add files to project
- Build the Database
- Configure the tools

In the following sections, we assume that the Freescale tool kit is installed in `C:\Freescale` directory. You may need to adapt the paths to your current installation. An example configuration for the M68000 CPU is provided, which can be adapted to each CPU supported by Freescale.

Create New Project

Start `DA-C.exe` and choose **Project > New Project** from the main menu. Browse to the directory and enter a project file name, for example:

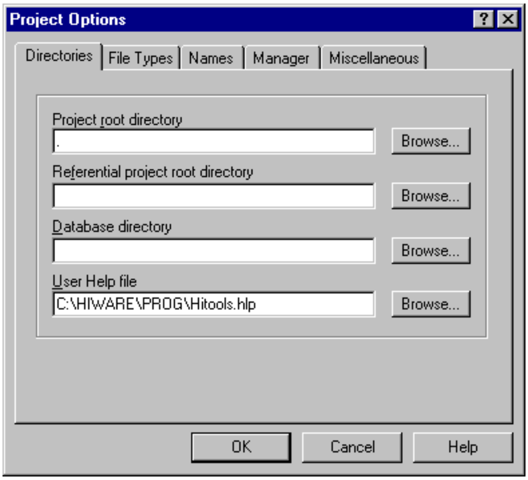
```
C:\Freescale\work\<processor>c\myproject
```

Change the `<processor>` field to your CPU. A specific project file is created with `.dcp` extension (for example `myproject.dcp`).

Configure Working Directories

Choose **Options > Project** from the main menu of DA-C. The window shown in Figure 9.1 contains options which establish project directories.

Figure 9.1 DA-C Project Options Window - Directories Tab



Project Root Directory

This text box determines the project root directory. In our case, enter a single dot to specify that the same directory in which the project file resides is the root directory. All project files are considered relative to the Project root directory, if the full file path is not given. You can also enter the full file path if desired.

Referential Project Root Directory

For the purposes of this project, leave this field empty. If used, this text box specifies an alternate Project Root Path for locating files not found in the original project path.

Filenames in the original path with referential extensions are tried before those in the referential path. The specified path may be either full or relative to project root, and it may not specify a subdirectory in the project root directory tree.

Database Directory

For the purposes of this project, leave this field empty. You can use this text box to specify the directory in which to save the symbols and software metrics database. This directory can be absolute or relative to the Project Root Directory.

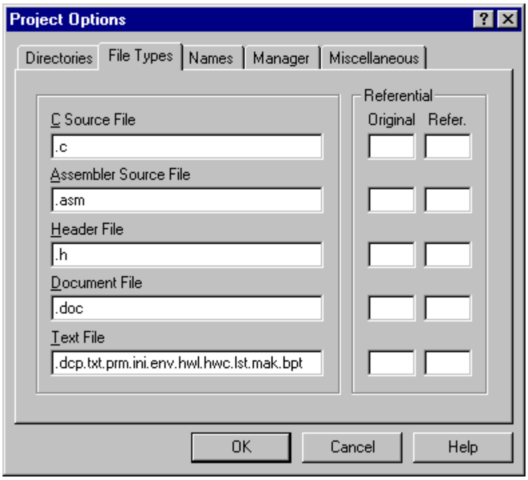
User Help File

This text box determines the user help file. For this project, browse in the `\prog` directory of your Freescale installation and select the help file matching your CPU. Define the hot key for the User Help File in the Keyboard definition file (default is Ctrl-Shift-F1).

Configure File Types

From the Project Option window of DA-C choose the **File Types** tab to configure the basic file types. Use the text boxes on this page to determine project file types (see Figure 9.2).

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

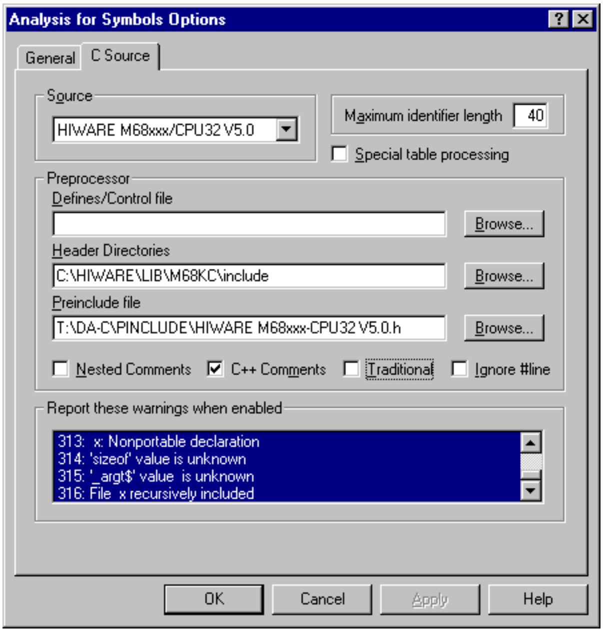


Configure Library Path

You must define an additional configuration path to specify the location of library header files (needed for DA-C symbol analysis). To do this, choose **Options > Analysis for Symbols > C Source** in the main menu of DA-C.

Use the window shown in Figure 9.3 to specify C source code analysis parameters.

Figure 9.3 Analysis for Symbols Options Window - C Source Tab



Use the following parameters for fields in this window:

- Source

  Select the supported C dialects of the C language used in the current project in this text field. In our example we chose the Freescale M68k language (adapt it to your needs).

- Preprocessor - Header Directories

  This text box determines the list of directories to search 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.

You can assign the list of header directories in a file. To do this, enter the file name (absolute or relative in relation to the project root) with the prefix @ in this field. Separate directories with a semi-colon or start a new line.

Define the library path matching your CPU (assuming Freescale tools are installed on `C:\Freescale`):

`C:\Freescale\lib\<processor>c\include.`

- Preprocessor - Preinclude File

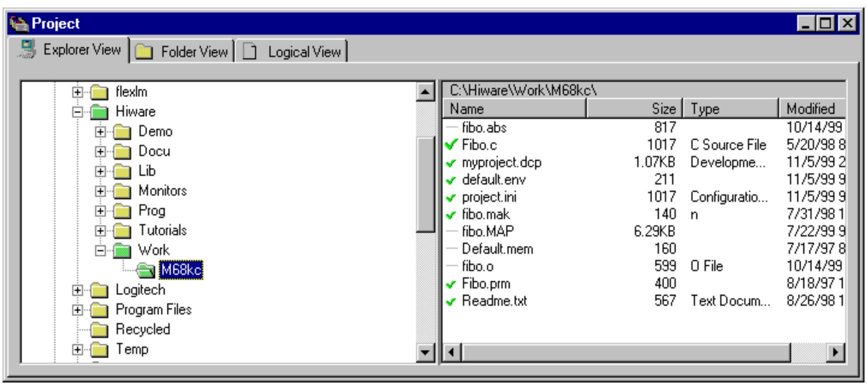
  In this text box specify the name of the file to include automatically at the beginning of every source module during analysis, as if `#include "string"` were present in the first line. Use the preinclude file to specify non-default predefined macros and variable and function declarations for a particular compiler. We selected the one corresponding to our example: M68k preinclude file (adapt it to your needs).

Add Files to Project

In the Project window, the Explorer View Tab replaces Windows Explorer and supplies you with additional information on project file directories. It also gives you the option to add files into the project. For example, we will now set all files needed to run the `fibo` example.

In the Explorer View, browse to the `>Freescale>WORK><processor>c` directory of your Freescale installation and select `fibo.c` file. Then right click the mouse and choose **Add to Project** from the context menu. This adds the file to the current project and a green mark appears in front of it (Figure 9.4).

Figure 9.4 Adding Files to Project Using Explorer Tab



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

You can also add a directory to the project in the following way:

1. Select Explorer View Tab in Project Window.
2. In the left section, select the directory containing the files to be added to the project (you may also add files from subdirectories to the project).
3. From context menu, choose **Add to project**.

You may also perform this operation from Folder view, if the directory is in the left section.

NOTE When adding an entire directory to the project, only files with extensions defined in **Options > Project > File types** (as described in Configure File Types) will be added to the project.

Build the Database

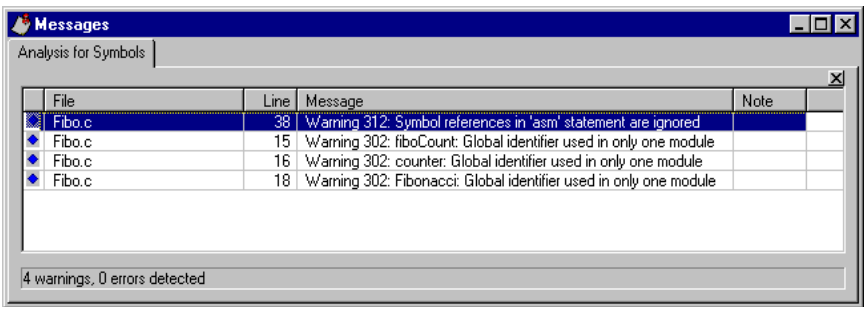
Development Assistant for C provides the static code analysis of C source files and generates various data based on the results.

Results of the analysis of the project source files and individual program modules are saved in database files on the disk. You can choose between the unconditional analysis of all project files using **Start > Build** or analysis of changed source files only, using **Start > Build database** and **Start > Update database**. When analyzing changed files only, you can optionally check modified include files used in program modules.

Data about global symbols usage, resulting from analysis, is saved in database files on the disk, enabling their use later in DA-C.

To build the database in our example, use the **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 appear in the Messages window as shown in Figure 9.5 (for `Fibo.c` sample file):

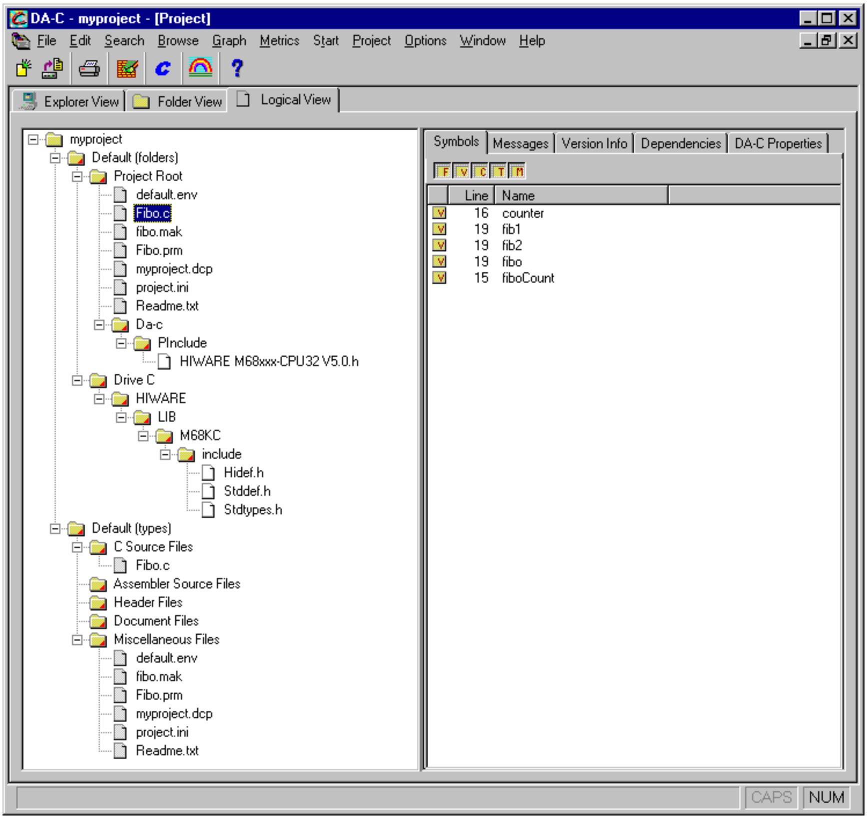
Figure 9.5 DA-C Message Window



After the analysis of all project files, the new database file containing information about global symbols is constructed. Refer to the DA-C manual for more information on using symbol information.

In the Project Manager window of DA-C, select the **Logical View** Tab shown in Figure 9.6 and unfold all fields. You now have the overview of your project.

Figure 9.6 Logical View Tab



Double click on `Fibo.c` file to open it.

Configure the Tools

We will now configure the compiler and maker into DA-C IDE. Define procedures using **Project > User Defined Actions** from the main menu of DA-C.

Compiler Configuration

First, set up a new action:

1. In **Menu "Start" Actions**, click on **New**.

   The **New Action** box appears.
2. Type `C&ompile` in the **New Action** box.
3. Click ENTER (Figure 9.7).

Next, associate a bitmap with each tool using the **Toolbar** field:

1. Click on the **Picture** radio button
2. Browse to the `\Bitmap` directory of your current DA-C installation
3. Choose `Compiler.bmp`.

   This is a default bitmap delivered with DA-C IDE. You can also add your own bitmap.

Figure 9.7 DA-C Compiler Settings

Now fill in the **Action Script** field to associate related compiler actions:

1. Copy the code shown in Listing 9.1 into the **Action Script** field
2. Change the directory in the code to your compiler directory.

Listing 9.1 Script for Compiler Action Association

```
.%If(%HasModuleExt(%CurrFile),,%Message(Not a module file!)%Cancel)
.%SaveAll
.c:\Freescale\prog\cm68k.exe %CurrFile
.%if(%Exist(edout),,%Message(No Messages found!)%Cancel)
.%ErrClr(Compiler)
.%ErrGet(edout,Compiler)
.%Reset(%CurrFile)
```

3. Click on **OK** to validate these settings.
4. Select `Fibo.c` file.
5. Click on the **Compiler** button (or from the main menu of DA-C select **Start > Compile**).

This file now compiles and generates the corresponding object file (`Fibo.o`).

Linker Configuration

In the same way, you can now configure the linker as shown in Figure 9.8:

1. In the **Menu "Start" Actions**, click on **New**.

   The **New Action** box appears.
2. Type `&Link` in the **New Action** box.
3. Validate by clicking ENTER.

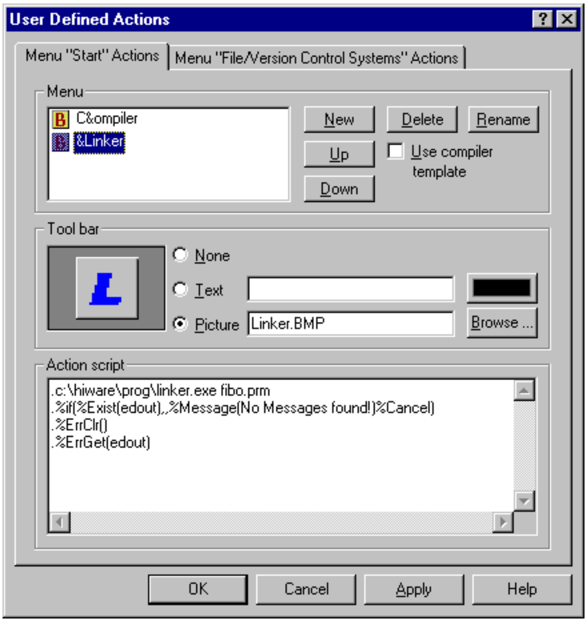
Set the corresponding Linker bitmap:

1. Copy the lines shown in Listing 9.2 into the **Action Script** field
2. Change the directory in the code to your linker directory.

Listing 9.2 Script for Linker Action Association

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

Figure 9.8 DA-C Linker Settings



Maker Configuration

Now configure the maker as shown in Figure 9.9:

1. In the **Menu "Start" Actions,** click on **New**.

   The **New Action** box appears.

2. Type `&Make` into the **New Action** box.
3. Click **ENTER**

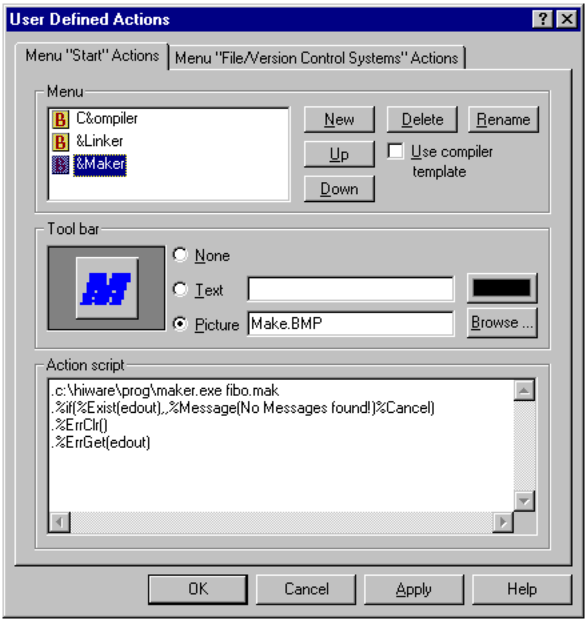
Set the corresponding Maker bitmap

1. Copy the code from Listing 9.3 into the **Action Script** field
2. Change the directory in the code to your maker directory.

Listing 9.3 Script for Maker Action Association

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

Figure 9.9 DA-C Maker Settings



Debugger Interface

DA-C v3.5 currently integrates a Debugging support Application Programming Interface (DAPI). This interface enables the DA-C to exchange messages with the Debugger. This shows that it is possible to set or delete breakpoints from within DA-C (in an editor, flowchart, graph, browser) and to execute other debugger operations. DA-C follows the Debugger in its operation, since it is always in the same file and on the same line as the debugger. Thus, usability of both the DA-C and Debugger increases. Make the following configurations to ensure efficient use of this Debugger Interface:

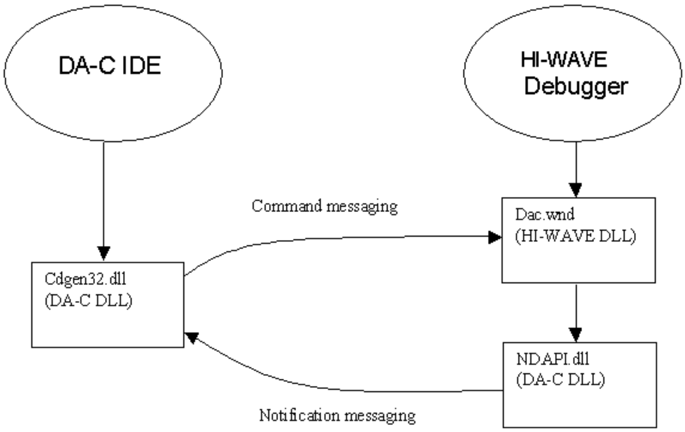
- Install communication DLL
- Configure Debugger properties
- Configure the Debugger project file

DA-C IDE and Debugger Communication

DA-C and the Debugger are both Microsoft® Windows® applications and base communication on the DDE protocol, as shown in Figure 9.10. The system contains:

- DA-C
- Debugger
- cDAPI interface implementation DLL, used by DA-C (`Cdgen32.dll`)
- nDAPI communication DLL (provided by DA-C), used by Debugger
- Debugger-specific DLL, for bridging its interface to the debugging environment and DA-C's nDAPI (`DAC.wnd`)

Figure 9.10 Communication between DA-C IDE and Debugger



Communication DLL Installation

The Debugger needs the nDAPI communication DLL (provided by DA-C IDE). This dll (called `Ndapi.dll`) installs automatically during the Freescale Tool Kit installation. However, if you install a new release of DA-C you must follow this procedure:

1. In the `\Program` directory of your DA-C installation, copy the `Ndapi32.dll` (`Ndapi32.dll` version 1.1 or later).
2. Paste it in your current `Freescale\PROG` directory (where Debugger is located).
3. Rename it to `Ndapi.dll`.

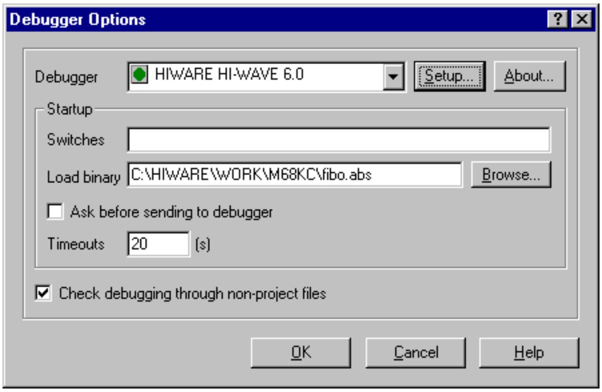
Debugger Properties Configuration

Now configure the debugger properties:

1. In the DA-C main menu, choose **Options > Debugger**.

   The dialog box shown in Figure 9.11 opens.

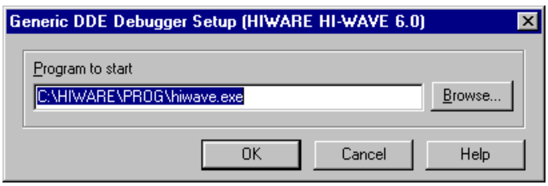
Figure 9.11 DA-C Debugger Options Dialog Box



2. In the **Debugger Options** box, select the corresponding debugger: **HI-WAVE 6.0**.
3. Now specify the binary file to open: in our example we want to debug the `fibo.abs` file.
4. Then click on the **Setup** button.

   The dialog box shown in Figure 9.12 opens.

Figure 9.12 DDE Debugger Setup Dialog Box



5. Specify the path to the `hiwave.exe` file or use the **Browse** button
6. Click **OK**.

Debugger Project File Configuration

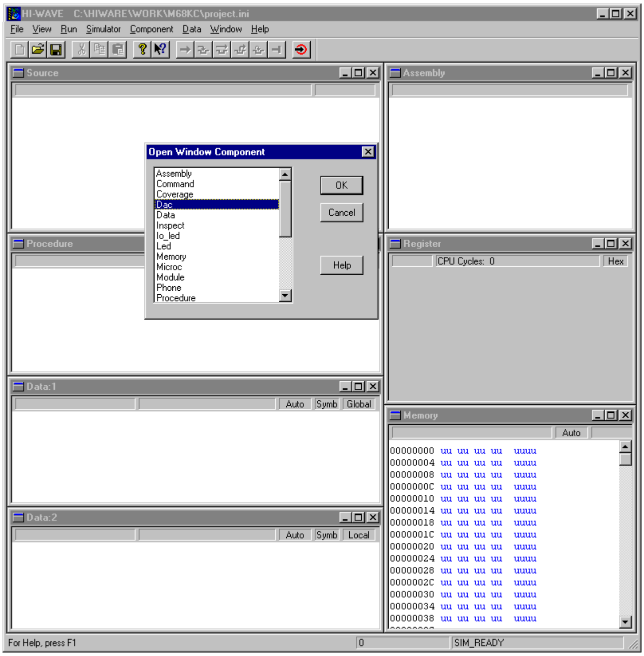
Now configure the debugger project file:

NOTE Before configuring the project file, close DA-C.

1. Open Debugger and select **File > Open Project** from the main menu bar.
2. Select the `Project.ini` file from the currently defined working directory (in our case `C:\Freescale\WORK\<processor>c\project.ini`).

   Now add in the layout of the project the Debugger DAC component (`dac.wnd`).
3. In the Debugger select **Component > Open** from the main menu bar
4. Choose **Dac**, as shown in Figure 9.13.

Figure 9.13 DA-C Component Opening



The Debugger DA-C window, needed for communication with DA-C IDE, opens (Figure 9.14).

Figure 9.14 DA-C Window

5. Save this configuration by selecting **File > Save Project** from the main menu of the Debugger.

   This component loads automatically the next time this project is called.

6. Close the Debugger.

Synchronized Debugging

We can now test the synchronization between DA-C IDE and Debugger:

1. Run `DA-C.exe` and open the previously created project.
2. Open `Fibo.c` if it's not already open.
3. Right click the mouse button on `Fibo.c` source window.
4. Select **Main** in the context menu.

   The cursor points to the `void main(void) {` statement.

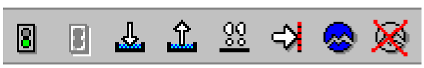
5. In the main DA-C menu, select **Debug > Set Breakpoint** (or click on the corresponding button on the debug toolbar).

   The selected line is highlighted in red, indicating that a breakpoint has been set.

6. Select **Debug > Run**.

   The Debugger starts and after a while stops on the specified breakpoint. You can debug from DA-C IDE with the toolbar, as shown in Figure 9.15, or from the Debugger.

Figure 9.15 DA-C Toolbar



NOTE If you make changes to your source code, remember to rebuild the Database when generating new binary files to avoid misalignment between the Debugger and DA-C source positions.

Troubleshooting

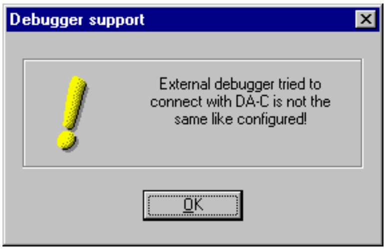
This section describes possible trouble when trying to connect the Debugger with the DA-C IDE.

Load the DA-C component into the Debugger. If the message box shown in Figure 9.16 appears, find out if the `Ndapi.dll` is located in the `\prog` directory of your current Freescale installation. If not, copy the specified `DLL` into this directory.

Figure 9.16 DA-C Component Loading Error Message

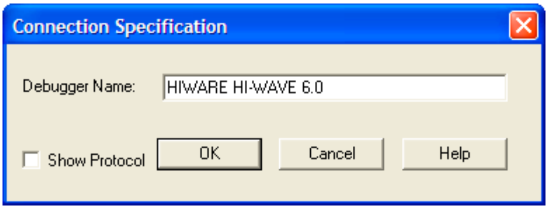
If the message box shown in Figure 9.17 appears in DA-C IDE, then the current name specified in the **Options > Debugger** main menu of DA-C doesn't match the debugger name specified in the Debugger.

Figure 9.17 DA-C Debugger Support Message



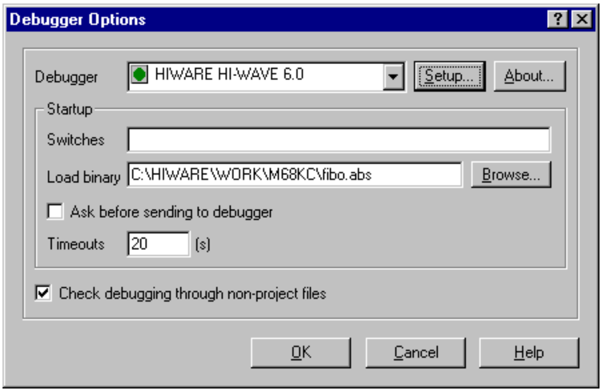
Open the setup dialog in the Debugger by clicking on the DA-C Link component and choose **DA-C Link > Setup** from the main menu. The *Connection Specification* dialog box opens (Figure 9.18).

Figure 9.18 DA-C Connection Specification Dialog Box



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

Figure 9.19 DA-C Debugger Options Dialog Box



Both must be the same. If not, change the Debugger name in the Debugger **Connection Specification** and click **OK**. This establishes a new connection and saves the **Connection Specification** in the current `Project.ini` file in the section shown in Listing 9.4.

Listing 9.4 DA-C Section in Project File

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

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

Book II Contents

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

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

10

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

This section provides information about debugging with the CodeWarrior debugger and the *HC(S)12X Full Chip Simulation* connection.

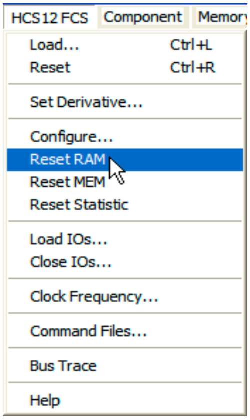
Technical Considerations

The Full Chip Simulation (FCS) connection runs a complete simulation of all processor peripherals and I/O on the user's Personal Computer. No development board is required. Each derivative has a unique simulation engine to accurately simulate the memory ranges, I/O, and peripherals for a given derivative (for more information on selecting a specific derivative, see Supported HC(S)12(X) Derivatives.

Full Chip Simulation Menu

This menu, shown in Figure 10.1, is associated with the Full Chip Simulation connection, and allows you to load an application in the FCS. Table 10.1 describes the FCS menu entries.

Figure 10.1 HCS12 FCS Menu



NOTE The menu changes slightly for a project that uses an XGATE coprocessor.

Table 10.1 Simulator Menu Entry Description

Menu Entry	Description
Load	Opens the Load Executable Window menu.
Reset	Resets the FCS.
Set Derivative	Selects the current simulated derivative.
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
Clock Frequency	Opens the Clock Frequency Setup dialog box to set the FCS real-time clock.
Command Files	Opens the Command Files Window

Table 10.1 Simulator Menu Entry Description (*continued*)

Menu Entry	Description
Bus Trace	Opens the Bus Trace dialog box to enable instructions and memory accesses recording and display recording captures.
Select Core	Selects the processor with which to communicate.

Debugger Status Bar with Full Chip Simulation

The status bar (Figure 10.2 and Figure 10.3) shows status and other information. As well as execution status, it includes a context-sensitive menu help line, and connection-specific information like the number of CPU cycles (64 bits) or the elapsed time in `hours:minutes'seconds"milliseconds(float)` format since the application started.

Figure 10.2 Debugger Status Bar with CPU Cycles

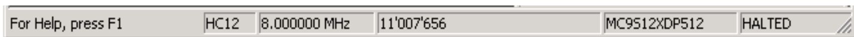


Figure 10.3 Debugger Status Bar with Elapsed Time

The selected simulated derivative or simulated “CORE” or core “SAMPLE” is shown, and the current derivative CPU frequency in MHz.

NOTE Clicking on the CPU frequency opens the Clock Frequency Setup.

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

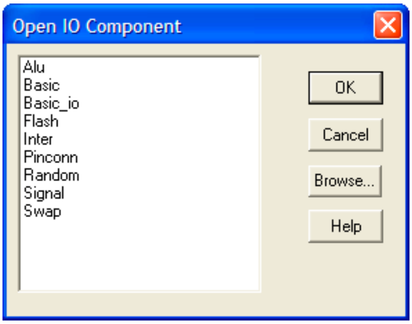
NOTE Clicking on the displayed derivative or CORE, or on the core SAMPLE opens the Set Derivative dialog box.

NOTE The CPU information in the Status Bar, such as **HC12**, might be displayed with **XGATE**, when simulating an **HCS12X** core device. The debugger indicates its current halt or step location on the core thread.

Open I/O Component Dialog Box

From the Simulator menu, choose **Load I/Os** to open the **Open I/O Component** dialog box. This dialog box, shown in Figure 10.4, allows you to open an **I/O** device (peripheral) simulation. The **Browse** button allows you to specify a location for the I/O.

Figure 10.4 Open IO Component Dialog Box



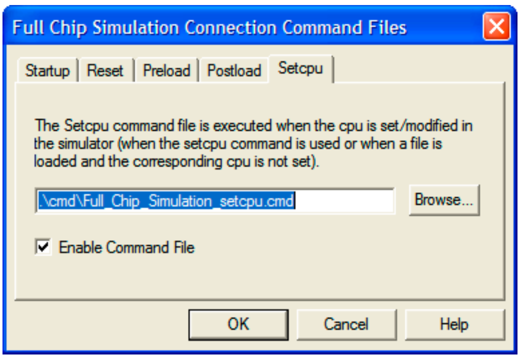
NOTE I/O simulation components are either designed by Freescale and delivered with the tool-kit installation or designed by the user with the Peripheral Builder (separate product).

Demo Version Limitations

Only two I/O components can be loaded.

Command Files Window

Figure 10.5 Full Chip Simulation Connection Command Files Window



Setcpu Command File

The **Setcpu** command file is a specific FCS command file executed by the Debugger after a CPU has been set or modified. Set or modify the CPU using either of two methods:

- By using the `setcpu` command, or
- By loading an application in the FCS when the corresponding CPU is not set.

Specify the full name and status of the **Setcpu** command file by using either the **CMDFILE SETCPU** Command Line command or the **Setcpu** property tab of the connection Command Files dialog.

The default **Setcpu** command file is `SETCPU.CMD`, located in the current project directory. Other Command Files are described in the Debugger Interface section, in the Debugger Engine book.

Memory Configuration

The memory configuration interface is an FCS advanced configuration feature, that divides the emulated memory into blocks. A memory manager handles the list of memory blocks. The memory configuration facility allows you to specify types and properties of memory blocks (such as RAM and ROM) and offers a degree of automation, but does not restrict the flexibility of manual adjustment.

The memory configuration facility uses a binary file format to read and set the FCS configuration. The extension for binary files is `.mem`; the default memory file is `default.mem`.

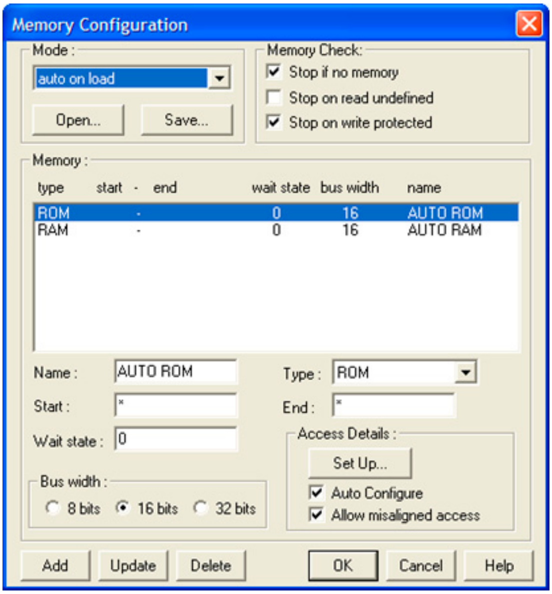
Memory Configuration Dialog Box Features

The memory configuration dialog box (Figure 10.6) lets you perform these memory-block operations interactively:

- Select the configuration mode for simulation
- Define a memory block name
- Define how the FCS verifies the memory
- Set the type of the memory: RAM, ROM, Flash, EEPROM or I/O
- Define start and end addresses
- Define the wait state (the time for each read or write access)
- Set the width of the bus that accesses the memory
- Set access details like:
  - *auto configure*: automatically computing read and write access
  - *misaligned access*: allowing misaligned access on words and longs

- Open and save memory configuration
- Add, delete, or update memory blocks

Figure 10.6 Memory Configuration Dialog Box



Memory Configuration Modes

Use the **Memory Configuration** dialog box to select the memory configuration mode: **auto configuration on access**, **auto configuration on load,** or **user defined**. Depending on your settings, the FCS component initializes the FCS memory as Table 10.2 explains.

Table 10.2 Memory Configuration Modes

Mode	Description
Auto Configuration on Access (Standard Configuration)	Defines memory as RAM of unlimited size. *Mode* combo box displays *auto on access*.

Table 10.2 Memory Configuration Modes (*continued*)

Mode	Description
Auto Configuration on Load (default)	Defines memory as RAM and ROM, according to code and data area defined in a loaded absolute file. Defines code segments as ROM. Defines data segments as RAM. (Memory outside these segments is *not implemented*; access to unimplemented locations result in error messages.) *Mode* combo box displays *auto on load*.
Manual Configuration (User Defined)	Defines memory as RAM, ROM, or non-volatile RAM, depending on configuration. Use Memory Configuration dialog box to construct definition interactively, or read it in from a file. *Mode* combo box displays *user defined*.

Memory Configuration Settings

Depending on the configuration mode, the Memory Configuration dialog box lets you redefine memory settings within certain limits. You always must set I/O devices manually.

Standard Configuration: Auto on Access: The Memory Configuration dialog box contains a single RAM entry with unspecified (*) starting and ending addresses. You cannot modify these addresses. You can adjust wait states, and other such settings, only for the whole RAM block.

Auto Configuration on Load: Initially, the dialog box lists a single RAM and a single ROM block, with unspecified (*) starting and ending addresses. You can adjust wait states, and other such settings, separately for RAM and ROM blocks.

For the ELF/DWARF object file format, the Memory Configuration dialog box lists separate RAM and ROM blocks for each data and code segment in the absolute file, once an application is loaded. The segment addresses and lengths determine the starting and ending addresses of each block; you cannot modify these addresses. Initial attributes of each code and data block come from the corresponding initial RAM and ROM blocks; you can modify these attributes independently.

Manual Configuration: The Memory Configuration dialog box lists an entry for each memory block. You can modify such entries without restriction.

NOTE To simulate an absolute file generated in Freescale object file format, you must open the Memory Configuration dialog box, set the **auto on load** mode, then add a new RAM segment. The start and end addresses of this segment must match the associated `.prm` file. Once you close the dialog box, you can load your application and start a simulation.

Open Memory Block

Click the **Open** button to load a memory blocks file. The **Open Memory blocks** standard dialog box appears. Select a memory map file, then click the **OK** button. The dialog box closes, and the system loads the memory blocks file.

The *Mode* combo box changes to indicate the mode contained in the memory map file.

The list box lists the memory blocks loaded from the file, selecting the first memory block. Appropriate data appears in the fields **Name**, **Type**, **Start**, **End**, **Wait state**, **Bus width** and **Access Details**.

Save Memory Block

Click the **Save** button to store the current memory blocks configuration. The **Save Memory blocks** standard dialog box appears. Enter a file name, then click the **OK** button. The dialog box closes, and the system stores the memory block configuration into the file.

Memory Check Options

The Memory Check group box consists of three checkboxes, all checked when you bring up the Memory Configuration dialog box:

- Stop if no memory — Check this box to have the FCS stop when an access to non-existent memory occurs. Clear this box to ignore this condition.
- Stop on read undefined — Check this box to have the FCS stop when a read of undefined memory occurs. Clear this box to ignore this condition.
- Stop on write protected — Check this box to have the FCS stop when a write to read-only (write-protected) memory occurs. Clear this box to ignore this condition.

Memory Configuration Module Startup

Memory configuration is a dynamically loaded facility. That is, the new entry **Configure** appears in the **Simulator** menu upon loading the FCS (the FCS dll). Selecting **Configure** opens the **Memory Configuration** dialog box, so that you can configure memory.

Memory Block Setting

You must set memory blocks within the available memory, and each block must cover a certain range. The *start address* and *end address* define each memory block.

Memory Block Properties

Table 10.3 lists the properties you may specify for a memory block.

Table 10.3 Memory Block Properties

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

Memory Configuration Dialog Box Command Buttons

The command buttons of the Memory Configuration dialog box are:

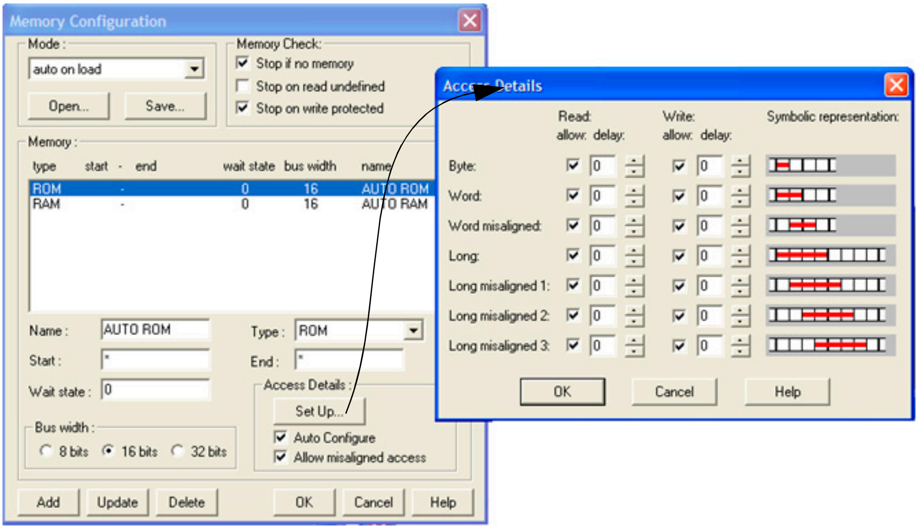
- **Add** — Fills a new memory block according to the current data in the **Name**, **Type**, **Start**, **End**, **Bus width**, and **Access Details** controls. This new memory block appears at the end of the list box. If there are any errors in this new block (such as an improper field value), a message box appears that informs you of the problem.
- **Update** — Updates the current memory block according to the current data of the **Name**, **Type**, **Start**, **End**, **Bus width**, and **Access Details** controls.
- **Delete** — Removes the currently selected memory block from the list box. The list box contents adjust to reflect this deletion.
- **OK** — Closes the dialog box and validates the list of modified memory blocks. The parent class can access this list, updating its own list.

- **Cancel** — Closes the dialog box, canceling your modifications.
- **Help** — Opens the dialog box help file.

Access Details Dialog Box

Figure 10.7 shows the **Access Details** dialog box, which lets you change read and write access values for seven types.

Figure 10.7 Memory Configuration Dialog Box - Access Details Dialog Box



Follow this guidance to use the **Access Details** dialog box:

- A check box indicates if an access kind is allowed or not.
- To modify the value of each read or write type, change the value of the associated spin box.
  - The lowest possible value is 0.
  - The highest possible value is 127.
- To store changes into currently selected memory block, click the **OK** button. The **Access Details** dialog box disappears, and the system clears the **Auto Configure** checkbox.
- To abandon changes, click the **Cancel** button. The **Access Details** dialog box disappears; the system discards your changes.
- To bring up appropriate help information, click the **Help** button.

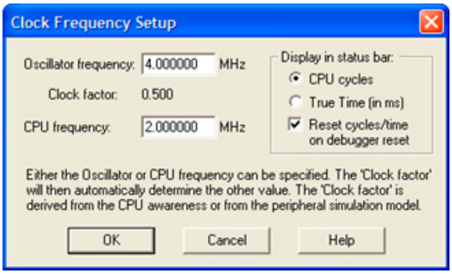
Output

You can save the current memory configuration into the file you defined at the outset.

Clock Frequency Setup

The FCS provides **true time information**. It is possible to provide an oscillator clock frequency to the debugger. The debugger CPU awareness and IO modules provide the clock factor to apply to this input frequency to derive the CPU cycle frequency.

Figure 10.8 Clock Frequency Setup Dialog Box



Derivative specific IO simulations caring of bus speed change (multiply or divide) through PLL modules dynamically update the clock factor, i.e. while the application simulation is running.

Accumulated elapsed time will not be affected and a new cycle time is applied to next simulated instructions in real time.

Open the Clock Frequency Setup dialog (**Simulator > Clock Frequency** menu entry) to set, enter, or edit either the oscillator frequency or the CPU frequency. However, the frequency saved in the project is the **oscillator** frequency. Two radio buttons allow you to choose whether cycles or true-time displays in the debugger status bar.

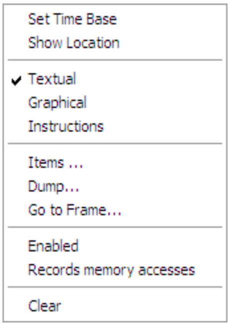
Unchecking **Reset cycles/time** makes the debugger accumulate cycles/time other than CPU reset. The true-time unit is the microsecond. The `TRUETIME` debugger command line command gives the time as a number in microseconds. The `OSCFREQUENCY` variable displays/sets the oscillator frequency.

Bus Trace

The FCS can record all executed instructions and memory accesses in the Trace component, up to one million frames. To enable recording, open the Trace component and use the Trace menu/context-sensitive menu.

NOTE Refer to the *HCS12 (or HCS12X) Onchip DBG Module* manual for Trace window common functionality and common menu entries.

Figure 10.9 Trace Window Context Menu

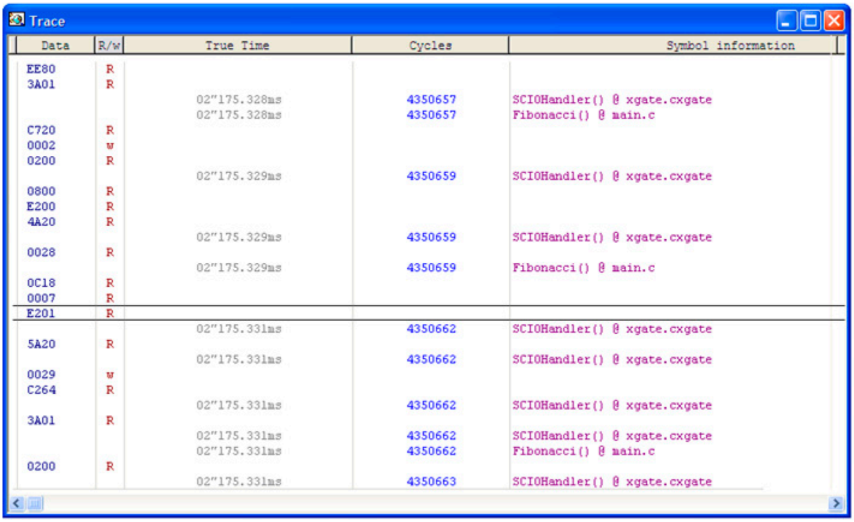


By default, the FCS records instructions only (faster). Check **Record memory accesses** and choose **Textual** mode in the Trace menu/context-sensitive menu to record memory accesses.

Many types of information can be retrieved from the Trace window, including:

- instructions and instruction addresses,
- data addresses, data values and read/write access type,
- true time, cycles and total simulation cycles for each instruction,
- function name and module name for each instruction,
- variable name and module name for each global variable data access.

Figure 10.10 Bus Trace Data Access Symbolic Information



Full Chip Simulation Warnings

By default, the FCS generates warning messages when the application accesses on-chip registers that are not implemented for the selected derivative. These warnings appear in the Command window.

For example, the following messages can be repeated indefinitely in the Command window:

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

```
HALTED
```

Warning message IDs usually belong to a group of registers from the same simulated block, such as the Flash `CONTROL` registers block in the listing above. Therefore, any access to unimplemented Flash `CONTROL` registers generate the same kind of message.

The debugger provides a set of commands to hide specific ID messages, to stop the FCS automatically, or to display a warning message box. Execute these commands from a `POSTLOAD` command file. These commands are volatile and not saved in current project. For a list of commands and their uses, see Full Chip Simulation Connection Commands.

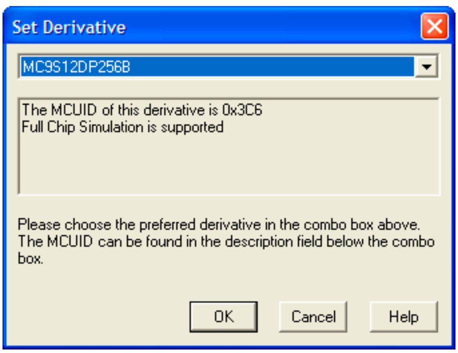
FCS and Silicon On-Chip Peripherals Simulation

FCS not only simulates the core instruction set but also the on-chip I/O devices, such as `CRG`, `PWM`, or `ECT`. Supported HC(S)12(X) Derivatives lists the supported I/O devices for each supported derivative.

Generating a new project with the **New HC(12) Project Wizard** and the **Full Chip Simulation** connection sets everything up to run the project with FCS support.

Use the menu option **Simulator > Set Derivative** to change the derivative to simulate. In addition to the derivatives, there are special entries: HC(S)12(X) CORE and HC(S)12(X) SAMPLE. The CORE entries allow you to simulate the chip without FCS support (plain instructions only) and the SAMPLE entries allow you to simulate a chip with a minimal set of commonly available on-chip peripherals, like Register Block, Memory Expansion Registers, Clock and Reset Generator, Serial Communication Interface 0 (SCI0) and PortB.

Figure 10.11 ‘Set Derivative Dialog Box



The status bar shows the current mode of Simulation (SAMPLE, CORE or MCU derivative). You can access the **Set Derivative** dialog by double clicking on the FCS support entry in the status bar. See Debugger Status Bar with Full Chip Simulation.

Supported HC(S)12(X) Derivatives

Please refer to release notes section for Full Chip Simulation to get detailed information on supported derivatives and modules simulated.

`<CodeWarrior install directory>\Release_Notes\HC12\FCS_Notes`

NOTE To simulate unlisted derivatives, either use a derivative with similar on-chip peripherals, or use the FCS SAMPLE or CORE mode.

Communication Modules

The communication modules available on the HC(S)12(X) debugger are described in the corresponding derivative simulation release notes. The following I/O devices are not simulated unless it is defined otherwise.

- Byteflight (BF)
- J1850 Bus (BLCD)
- Scalable CAN (MSCAN)
- Universal Serial Bus Module (USB20D6E2F)
- Inter-IC Bus (IIC)

Serial Communication Interface

This I/O device simulates the Serial Communication Interface (SCI). The unmapped registers `SCIInput/SCIInputH` and `SerialInput` serve to send characters to the SCI Module. The unmapped registers `SCIOutput/SCIOutputH` and `SerialOutput` contain the characters sent from the SCI Module.

Table 10.4 Simulated SCI Registers

Register Acronym	Full Register Name	Simulated Fields
SC0BDH	SCI Baud Rate Register High	SBR12:8
SC0BDL	SCI Baud Rate Register Low	SBR7:0

Table 10.4 Simulated SCI Registers (*continued*)

Register Acronym	Full Register Name	Simulated Fields	
SC0CR1	SCI Control Register 1	M	ILT
SC0CR2	SCI Control Register 2	TIE TCIE RIE ILIE	TE RE SBK
SC0SR1	SCI Status Register 1	TDRE TC RDRF	IDLE OR
SC0SR2	SCI Status Register 2	RAF	
SC0DRH	SCI Data Register High	R8/T8	
SC0DRL	SCI Data Register Low	R7:0/T7:0	

Registers not Mapped to Memory

Table 10.5 shows the SCI registers that are not mapped to memory.

Table 10.5 SCI Registers not Mapped to Memory

Register	Description
SCIInput	Sends a character to the SCI. Value received from the SCI; can be read via a read access to the SCDR. Ninth bit is taken from `SCIInputH` register. Read access to `SCIInput` has no specified meaning. Bits 7–0 characters sent to the SCI.
SCIInputH	Sends a character to the SCI, containing the ninth bit associated with `SCIInput`. Must be written before writing the `SCIInput` register. Read access to `SCIInputH` has no specified meaning. Bit 0 (ninth bit) sent to the SCI.
SCIOutput	Receives a character sent from the SCI. Value received in `SCIOutput` is triggered by a write access to the SCDR. Ninth bit is written to the `SCIOutputH` register. Write access to `SCIOutput` has no specified meaning. Bit 7–0 characters sent from the SCI.

Table 10.5 SCI Registers not Mapped to Memory (*continued*)

Register	Description
SCIOutputH	Receives a character sent from the SCI. Contains the ninth bit associated with `SCIOutput`. Write access to `SCIOutputH` has no specified meaning. Bit 0 (ninth bit) sent from SCI.
SerialInput	Alias for `SCIInput` register. `SerialInput` connects SCI to terminal window. Ninth bit is not supported. Read access to `SerialInput` has no specified meaning. Bit 7–0 data sent from terminal window to SCI.
SerialOutput	Alias for `SCIOutput` register. `SerialOutput` connects SCI to terminal window. Ninth bit is not supported. Write access to `SerialOutput` has no specified meaning. Bit 7–0 data sent from SCI to terminal window.

Serial Peripheral Interface

Table 10.6 details the simulated Serial Peripheral Interface (SPI) registers.

Table 10.6 Simulated SPI Registers

Register Acronym	Full Register Name	Simulated Fields	
SPICR1	SPI Control Register 1	SPIE SPE MSTR	CPOL CPHA LSBFE
SPICR2	SPI Control Register 2	SPISWAI	SPC0
SPIBR	SPI Baud Rate Register	SPPR2:0	SPR2:0
SPISR	SPI Status Register	SPIF SPTEF	MODF
SPIDR	SPI Data Register	SPIDR7:0	

Registers not Mapped to Memory

Table 10.7 shows the registers that are not mapped to memory.

Table 10.7 SPI Registers not Mapped to Memory

Register	Description
SPIValue	Sends and receives (swaps) a character from and to the SPI. Bit 7–0 data sent from/to SPI

Analog to Digital Converter Module

This I/O device simulates the Analog to Digital Converter (ADC). FCS supports eight- and 16-channel versions of the ADC module. Access the analog inputs (PAD0 to PAD7/PAD15) separately through the object pool. For ADC module 1, PAD0 input corresponds to PAD8/PAD16 pin of the microcontroller.

Conversion Results

The analog inputs of ADC module are simulated as 8-bit logic values. Therefore, the simulation of the conversion itself only has a limited interest. The conversion results are an image of the simulated input.

For the unsigned right-justified 8-bit conversion, the result displayed in the corresponding data register is the exact image of the input.

Simulation is accurate on the conversion delays and the modifications that affect the input (8-10 bits, left/right justified, signed/unsigned). The data registers in which to transfer the conversion results give a precise image on how to configure the ADC modules for the proper conversion process.

Table 10.8 Simulated ADC Registers

Register Acronym	Full Register Name	Simulated Fields	
ATDCTL2	ATD Control Register 2	ADPU AFFC AWAI ETRIGLE	ETRIGP ETRIGE ASCIE ASCIF
ATDCTL3	ATD Control Register 3	S8C S4C	S2C S1C
ATDCTL4	ATD Control Register 4	SRES8 SMP1:0	PRS4:0

Table 10.8 Simulated ADC Registers (*continued*)

Register Acronym	Full Register Name	Simulated Fields	
ATDCTL5	ATD Control Register 5	DJM DSGN SCAN MULT	CC CB CA
ATDSTAT0	ATD Status Register 0	SCF ETORF	FIFOR CC2:0
ATDSTAT1	ATD Status Register 1	CCF7:0	
ATDDIEN	ATD Input Enable Register (8 Channel)	IEN7:0	
ATDDIEN0	ATD Input Enable Register (16 Channel)	IEN15:8	
ATDDIEN1	ATD Input Enable Register (16 Channel)	IEN7:0	
PORTAD	Port Data Register (8 Channel)	PTAD7:0	
PORTAD0	Port Data Register (16 Channel)	PTAD15:8	
PORTAD1	Port Data Register (16 Channel)	PTAD7:0	
ATDDRx	ATD Conversion Result Registers	Entire register	

Registers not Mapped to Memory

The following ADC registers are not mapped to memory:

- PADx
  - The PADx registers are eight registers not mapped to memory that serve as the measured values for the ATD. The format of the PADx registers is IEEE32. To set up a PAD use the following command:

```
ATDx_SETPAD <CHANNEL> <VOLTAGE AS FLOAT>
```

Memory Modules

These memory modules are not simulated:

- EEPROM (EETS)
- Flash (FTS)

Miscellaneous Modules

The following miscellaneous modules are not simulated:

- Voltage Regulator (VREG)
- Compact Flash Host Controller (CFHC)
- Memory Stick Host Controller (MSHC)
- Secure Digital Host Controller (SDHC)
- ATA5HC Module (ATA5HC)
- Integrated Queue Module (IQUE)
- Ethernet Media Access Controller (EMAC)
- Ethernet Physical Transceiver (EPHY)
- Debug Module (DBG)

S12X_INT

Table 10.9 shows the simulated S12X_INT registers.

Table 10.9 Simulated S12X_INT Registers

Register Acronym	Full Register Name	Simulated Fields
IVBR	Interrupt Vector Base Register	Entire register
INT_XGPRIO	XGATE Interrupt Priority Configuration Register	Entire register
INT_CFADDR	Interrupt Request Configuration Address Register	Entire register
INT_CFDATA0:7	Interrupt Request Configuration Data Registers 0–7	All registers

XGATE

Table 10.10 shows the simulated XGATE registers.

Table 10.10 Simulated XGATE Registers

Register Acronym	Full Register Name	Simulated Fields
XGMCTL	XGATE Module Control Register	Entire register
XGCHID	XGATE Channel ID Register	Entire register

Table 10.10 Simulated XGATE Registers (*continued*)

Register Acronym	Full Register Name	Simulated Fields
XGVBR	XGATE Vector Base Address	Entire register
XGIF	XGATE Interrupt Flag Vector	Entire register
XGSWT	XGATE Software Trigger Register	Entire register
XGSEM	XGATE Semaphore Register	Entire register
XGCCR	XGATE Condition Code Register	Entire register
XGPC	XGATE Program Counter	Entire register
XGR1	XGATE Register 1	Entire register
XGR2	XGATE Register 2	Entire register
XGR3	XGATE Register 3	Entire register
XGR4	XGATE Register 4	Entire register
XGR5	XGATE Register 5	Entire register
XGR6	XGATE Register 6	Entire register
XGR7	XGATE Register 7	Entire register

Port I/O Modules

The following Port I/O modules are not simulated:

- External Bus Interface (EBI)

Module Mapping Control (MMC)

Table 10.11 shows the simulated MMC registers.

Table 10.11 Simulated MMC Registers

Register Acronym	Full Register Name	Simulated Fields
GPAGE	Global Page Index Register	Entire register
DIRECT	Direct Page Register	Entire register

Table 10.11 Simulated MMC Registers (*continued*)

Register Acronym	Full Register Name	Simulated Fields
RPAGE	RAM Page Index Register	Entire register
EPAGE	EEPROM Page Index Register	Entire register
PPAGE	Program Page Index Register	Entire register

The following MMC registers are not simulated:

- Miscellaneous System Control Register
- MTSTO (Reserved Test Register Zero)

Multiplexed External Bus Interface (MEBI)

This I/O device simulates the Multiplexed External Bus Interface (MEBI). The MEBI block is part of the Core and its description can be found in the Core manual. This block controls the behavior of ports A, B, E and K, the IRQ and XIRQ signals, and the operating mode of the Core (normal/extended/special).

FCS simulates only single-chip mode, therefore ports A and B cannot be used as external bus lines.

Except for port E, FCS simulates only the I/O behavior of the ports. The IRQ and XIRQ functionality going through port E pins 0 and 1 are simulated together with the various I/O enabling conditions of the port E pins described in the PEAR register. When a port E pin is not selected as an I/O pin, it stays at 0. Other functionalities are not simulated.

Port Integration Module (PIM)

This I/O device simulates the Port Integration Module (PIM). The PIM controls all the ports that are not directly associated to the CORE. All registers present in the PIM are port specific apart from the MODRR register that affects ports S, P, M, J and H. All port-specific registers are implemented together with the associated interrupt logic.

Timer Modules

This section describes the simulated timer modules and specifies which modules, blocks, and features are not simulated.

Clock and Reset Generator (CRG)

This I/O device simulates the PLL, RTI and COP features of the Clock and Reset Generator (CRG). Additional features of the CRG such as oscillator system hardware failures are not simulated.

The PLL output clock frequency is `(PLLCLK) = 2 OSCCLK ? (SYNR + 1)/ (REFDV + 1)`. FCS considers the PLL block a frequency converter. FCS ignores other PLL functionalities in the hardware.

Reference Clock

The CRG module reference clock is CLK24, given at the output. The CLK3 and CLK23 clocks are not simulated.

When you clear PLLSEL to 0, the oscillator clock frequency (used by the RTI and COP) is the same as the reference clock frequency.

When you set PLLSET to 1, `OSCCLK frequency = CLK24 * (REFDV + 1) / (2 * (SYNR + 1))`.

Since some systems do not work with a CLK24 frequency less than the hardware OSCCLK frequency, the simulation does not accept CLK24 frequencies less than the hardware OSCCLK frequency and generates a warning message.

Any OSCCLK frequency greater than the CLK24 frequency has the same frequency as CLK24.

Blocks

The CRG PLL Control Register (PLLCTL) is not simulated.

The following blocks are simulated:

- Phase Lock Loop (PLL)

  The simulated PLL clock divider functionality includes the REFDV and the SYNR registers and the PLLSEL bit in the CLKSEL register.

  Changing the value of PLLSEL automatically updates the COP and the RTI events. This may cause cycle irregularities as described in the manual. For proper use of the COP and RTI, change PLLSEL before enabling these modules.

  The simulated PLL stabilization time ranges from 100 to 1500 clock cycles, after modifying the REFDV or SYNR registers. Setting PLLSEL to 1 before this stabilization time elapses generates a warning message. The FCS operates properly but the corresponding program may not work on the hardware.

- Real-Time Interrupt (RTI) and COP

  Both RTI and COP use CLK24 as a reference clock. If OSCCLK is not equal to CLK24, the simulator adapts the RTI and COP period to the clock difference.

Table 10.12 Simulated RTI and COP Registers

Register Acronym	Full Register Name	Simulated Fields
SYNR	CRG Synthesizer Register	SYN5:0
REFDV	CRG Reference Divider Register	REFDV3:0
CRGFLG	CRG Flags Register	RTIF
CRGINT	CRG Interrupt Enable Register	RTIE
CLKSEL	CRG Clock Select Register	PLLSEL
RTICTL	CRG RTI Control Register	RTR6:0; Also RTDEC if supported by derivative.
COPCTL	CRG COP Control Register	WCOP RSBCK CR2:0
ARMCOP	CRG COP Timer Arm/Reset Register	Entire register

Enhanced Capture Timer (ECT)

This I/O device simulates the Enhanced Capture Timer (ECT). The various functionalities are cycle accurate up to 99%. Instruction pipelining simulation may differ from the hardware; some interruptions might be raised with a one-instruction delay.

The functions with errors detected in the hardware are not simulated. One operation mode is used as default. Further information is given for unimplemented features.

The Delay Counter Control Register (DLYCT) is not simulated.

Modes of Operation

NORMAL and STOP mode are implemented; entering FREEZE or WAIT mode causes the system to behave like STOP mode.

Table 10.13 Simulated ECT Registers

Register Acronym	Full Register Name	Simulated Fields	
TIOS	Timer Input Capture/Output Compare Select Register	IOS7:0	
CFORC	Timer Compare Force Register	FOC7:0	
OC7M	Output Compare 7 Mask Register	OC7M7:0	
OC7D	Output Compare 7 Data Register	OC7D7:0	
TCNT	Timer Count Register	Partly simulated; Not writable in test mode	
TSCR1	Timer System Control Register 1	TEN	TFFCA
TTOV	Timer Toggle On Overflow Register 1	TOV7:0	
TCTL1/TCTL2	Timer Control Register 1 and 2	OM7:0	OL7:0
TCTL3/TCTL4	Timer Control Register 3 and 4	EDG7B EDG7A EDG6B EDG6A EDG5B EDG5A EDG4B EDG4A	EDG3B EDG3A EDG2B EDG2A EDG1B EDG1A EDG0B EDG0A
TIE	Timer Interrupt Enable Register	C7I C6I C5I C4I	C3I C2I C1I C0I
TSCR2	Timer System Control Register 2	TOI TCRE	PR2:0
TFLG1	Main Timer Interrupt Flag 1	C7F C6F C5F C4F	C3F C2F C1F C0F
TFLG2	Main Timer Interrupt Flag 2	TOF	
TCx	Timer Input Capture/Output Compare Registers 0:7	All registers	

Table 10.13 Simulated ECT Registers (*continued*)

Register Acronym	Full Register Name	Simulated Fields
PACTL	16-Bit Pulse Accumulator A Control Register	PAEN PAOVI PEDGE
PAFLG	Pulse Accumulator A Flag Register	PAOVF
PACN3, PACN2	Pulse Accumulators Count Registers 3 and 2	All registers
PACN1, PACN0	Pulse Accumulators Count Registers 1 and 0	All registers
MCCTL	16-Bit Modulus Down-Counter Control Register	MCZI FLMC MODMC MCEN RDMCL MCPR1:0 ICLAT
MCFLG	16-Bit Modulus Down-Counter FLAG Register	MCZF POLF3:0
ICPAR	Input Control Pulse Accumulators Register	PA3EN PA1EN PA2EN PA0EN
ICOVW	Input Control Overwrite Register	NOVW7:0
ICSYS	Input Control System Control Register	SH37 TFMOD SH26 PACMX SH15 BUFEN SH04 LATQ
PTPSR	Precision Timer Prescaler Select Register	Entire register if derivative supports it
PTMCPSR	Precision Timer Module Counter Prescaler Select Register	Entire register if derivative supports it
PBCTL	16-Bit Pulse Accumulator B Control Register	PBEN PBOVI
PBFLG	Pulse Accumulator B Flag Register	PBOVF
PA3H–PA0H	8-Bit Pulse Accumulators Holding Registers 3–0	Entire register

Table 10.13 Simulated ECT Registers (*continued*)

Register Acronym	Full Register Name	Simulated Fields
MCCNT	Modulus Down-Counter Count Register	Entire register
TC0H-TC3H	Timer Input Capture Holding Registers 0–3	Entire register

Registers not Mapped to Memory

The following registers are not mapped to memory:

- Port T (PORTT)

  The functionality linking the PWM module and port T are simulated using the Port T I/O Register (PTT).

- PORTTBitx

  The pins are simulated as ‘not memory mapped’ and can be accessed one by one through the object pool (PORTTBit0 to PORTTBit7).

Periodic Interrupt Timer (PIT)

The Periodic Interrupt Timer (PIT) I/O device is not simulated.

Pulse Width Modulator (PWM)

This I/O device simulates the Pulse Width Modulator (PWM). Simulation of both 6- and 8-channel PWMs is supported. The 6-channel PWM is a subset of the 8-channel PWM, with fewer registers, and in some registers, using fewer bits.

The simulation is accurate up to one instruction due to instruction pipelining differences between the hardware and the simulation. However, the simulation strictly respects the period and the duty time of the generated pulses.

Changing control registers while the counters are running causes irregularities on the hardware outputs and cycle duration, as well as in the simulation, although not the same irregularities as in the hardware. For proper use of the module, disable channels (PWME register) and reset the counter (PWMCNTx registers) before modifying the corresponding control register (clock selection, period settings etc.) as described in the manual.

Clock Select

Scalers and prescalers are simulated for the clock selection. Changing clock control bits while channels are operating can cause irregularities that affect the time until the next end of a period (and duty) and the value displayed in the PWN counter registers.

Polarity, Duty and Period

It is important to note the information given in the inspector component concerning the various events. The two types of event used in the PWM module are the **Duty** and **Period** events.

In left-aligned mode:

- The *End of Period Time* represents the number of bus clock cycles remaining before the counter is reset.
- The *End of Duty Time* represents the number of bus clock cycles remaining before the output changes state.

In center-aligned mode:

- The *End of Period Time* represents the number of bus clock cycles remaining before the counter changes state. This means that the event period is half the effective period of the centered output waveform.
- The *End of Duty Time* represents the number of bus clock cycles remaining before the output changes state. An *End of Duty Time* is set after the end of each *Period Event*.

Table 10.14 Simulated PWM Registers

Register Acronym	Full Register Name	Simulated Fields
PWME	PWM Enable Register	PWME7:0
PWMPOL	PWM Polarity Register	PPOL7:0
PWMCLK	PWM Clock Select Register	PCLK7:0
PWMPRCLK	PWM Prescale Clock Select Register	PCKB2:0 PCKA2:0
PWMCAE	PWM Center Align Enable Register	CAE7:0

Table 10.14 Simulated PWM Registers (*continued*)

Register Acronym	Full Register Name	Simulated Fields
PWMCTL	PWM Control Register	CON45 CON23 CON01; PFRZ is not simulated but system acts as if PFRZ is always set to 1
PWMSCLA	PWM Scale A Register	Entire register
PWMSCLB	PWM Scale B Register	Entire register
PWMCNTx	PWM Channel Counter Registers 0-5/7	Entire register
PWMPERx	PWM Channel Period Registers 0-5/7	Entire register
PWMDTYx	PWM Channel Duty Registers 0-5/7	Entire register
PWMSDN	PWM Shutdown Register	PWMIF PWMIE PWMRSTRT PWMLVL PWM7IN PWM7INL PWM7EN
PORTP	Port P	Functionality linking the PWM module and port P are simulated using the PTP (Port P I/O Register)

PWMoutx

As in the hardware, writing to PTP has no effect. The input pins are simulated as ‘not memory mapped’ and can be accessed one by one through the object pool (`PWMout0` to `PWMout7`). Only `PWMout7` can be configured as an input. Writing to the other pins has no effect.

Timer Module (TIM)

This I/O device simulates the Timer Module (TIM). This module can be viewed as a subset of the ECT module. The TIM for example has only two Pulse Accumulator Count Registers called `PACNT_H` and `PACNT_L`. Both registers are simulated. For more information see Enhanced Capture Timer (ECT).

Legacy HC12 (CPU12) Derivatives Simulation

MC68HC812A4

This section explains the simulated features of the MC68HC812A4 derivative. The FCS implements the on-chip peripherals listed here.

Register Block

Table 10.15 shows the register block functionality. You can move all I/O registers according to the INITRG (Register Block Mapping) at offset $11 inside the register block.

Table 10.15 MC68HC12A4 Register Block

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

Lite Integration Module

FCS simulates many functions of the Lite Integration Module (LIM), including:

- Interrupt handling
- Watchdog
- Periodic Interrupt

General restrictions:

- FCS does not distinguish normal from special mode. Accordingly, it allows all write accesses, as if the chip were in special mode.
- Table 10.16 shows restrictions relative to special registers and single bits of registers.

LIM Simulated Registers

Table 10.16 shows the LIM Simulated Registers.

Table 10.16 LIM Simulated Registers

Register Name	Register Address	Initial Value	Remarks
CLKCTL	0x0047	0x00	LCKF, PLLON, PLLS, BCSC, BCSB, BCSA: These CLKCTL bits control PLL settings. FCS does not simulate the PLL; values of these bits have no effect.
RTICTL	0x0014	0x00	RSWAI: FCS does not support the CPU Clock stop; this bit has no effect. RSBCK: FCS does not simulate background mode; this bit has no effect.
RTIFLG	0x0015	0x00	
COPCTL	0x0016	0x0F	CME, FCME, FCM: FCS does not support these COPCTL bits; writing to these bits has no effect.
COPRST	0x0017	0x00	
INTCR	0x001E	0x60	FCS does not distinguish normal from special mode. IRQE: The implementation allows any write access. In normal mode, write to this register once only. In special mode, system ignores the first write access.
HPRIO	0x001F	0xF2	System may write to HPRIO register if I mask in CPU condition code register (CCR) is set. FCS does not simulate this register.

Standard Timer Module (TIM)

FCS simulates all functions of TIM.

General restrictions:

- The HPRIO register [$001F] may be written if the I mask in the CPU CCR is set. This is not simulated.
- The external timer output occurs at the PORTT register for testing purposes only.
- Restrictions considering special registers and single bits of registers are covered in Table 10.17.

TIM Simulated Registers

Table 10.17 shows all simulated TIM registers:

Table 10.17 TIM Simulated Registers

Register Name	Register Address	Initial Value	Remarks
TIOS	0x0080	0x00	
CFORC	0x0081	0x00	
OC7M	0x0082	0x00	
OC7D	0x0083	0x00	
TCNT_H	0x0084	0x00	
TCNT_L	0x0085	0x00	
TSCR	0x0086	0x00	TSWAI: FCS does not support the CPU Clock stop; setting this bit has no effect. TSBCK: FCS does not simulate background mode; this bit has no effect.
TQCR	0x0087	0x00	
TCTL1	0x0088	0x00	
TCTL2	0x0089	0x00	
TCTL3	0x008A	0x00	
TCTL4	0x008B	0x00	
TMSK1	0x008C	0x00	

Table 10.17 TIM Simulated Registers (*continued*)

Register Name	Register Address	Initial Value	Remarks
TMSK2	0x008D	0x30	TPU: This bit controls a pull-up resistor or a pin. Since the FCS has no real pins, setting this bit has no effect. TDRB: This bit controls the output drive of a pin. Since the FCS has no real pins, setting this bit has no effect.
TFLG1	0x008E	0x00	
TFLG2	0x008F	0x00	
TC0_H	0x0090	0x00	
TC0_L	0x0091	0x00	
TC1_H	0x0092	0x00	
TC1_L	0x0093	0x00	
TC2_H	0x0094	0x00	
TC2_L	0x0095	0x00	
TC3_H	0x0096	0x00	
TC3_L	0x0097	0x00	
TC4_H	0x0098	0x00	
TC4_L	0x0099	0x00	
TC5_H	0x009A	0x00	
TC5_L	0x009B	0x00	
TC6_H	0x009C	0x00	
TC6_L	0x009D	0x00	
TC7_H	0x009E	0x00	
TC7_L	0x009F	0x00	
PACTL	0x00A0	0x00	
PAFLG	0x00A1	0x00	

Table 10.17 TIM Simulated Registers (*continued*)

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

Serial Communication Interface (SCI)

Implement the SCI module as a separate class, because there are several nearly-identical instances of this class.

Supported Features

Table 10.18 shows the supported SCI features.

Table 10.18 SCI Supported Features

Feature Acronym	Full Feature Name	Comments
SBRx	Baud Rate	Bit transmittal follows current baud rate settings
BTST	Reserved for internal tests	Ignored
BSPL	Reserved for internal tests	Ignored
BRLD	Reserved for internal tests	Ignored
LOOP	LOOP Mode	LOOP mode determines SCI connection to outer world. As this SCI is simulated, there is no connection to simulate.
WOMS	Wired Or Mode	Special feature of LOOP mode; not simulated
RSRC	Receiver Source	Special feature of LOOP mode; not simulated

Table 10.18 SCI Supported Features (*continued*)

Feature Acronym	Full Feature Name	Comments
M Mode	8 or 9 data bits	Supported (different timing, ninth bit)
WAKE	Wakeup by Address Mark/Idle	Not supported
ILT	Idle Line Type	Considered in the Idle Line Detection
PE	Parity Enabled	Not simulated
PT	Parity Type	Not simulated
TIE	Transmit Interrupt Enable	Supported
TCIE	Transmit Complete Interrupt Enable	Supported
RIE	Receive Interrupt Enable	Supported
ILIE	Idle Line Interrupt Enable	Supported
TE	Transmitter Enable	Transmission process stops if this bit is clear
RE	Receiver Enable	Receive process stops if this bit is clear. As the input register is not part of the simulation, it still receives stimuli.
RWU	Receiver Wake Up Control	Not supported
SBK	Send Break	When first set, transmitter sends ten (11 if M bit is set) 0 values. Counter is set only if flag was previously cleared. After the counter sends the required number of 0 bits, it continues sending 0 bits as long as the SBK flag remains set.
TDRE	Transmit Data Register Empty Flag	Set when value to be transmitted moves from transmit data register to serial shift register.
TC	Transmit Complete Flag	Set when the transmission of one value ends, but no other value is yet in the transmit data register.

Table 10.18 SCI Supported Features (*continued*)

Feature Acronym	Full Feature Name	Comments
RDRF	Receive Data Register Full Flag	Set upon the complete read of a value and the clearing of RDRF.
IDLE	Idle Line Detection Flag	Set after a period without any input. The system considers the ILT flag.
OR	Overrun Error Flag	Set if the receipt of value ends, but the processor has not yet read the value.
NF	Noise Error Flag	Not supported; no physical transmission takes place.
FE	Framing Error Flag	Not supported; no physical transmission takes place.
PF	Parity Error Flag	Not supported; no physical transmission takes place.
RAF	Receiver Active Flag	Supported and cleared only when going into idle mode. Detection of a false start bit does not clear this flag, as no physical transmission takes place.
R8	Receive Bit 8	Supported
T8	Transmit Bit 8	Supported
Rx/Tx	Receive/Transmit Bit x	Supported, with autoclear feature

FCS uses non-memory-mapped registers to simulate SCI connection to the outer world. FCS buffers all values sent to the input registers, then simulates receipt from another SCI (with maximum speed and no transmission errors). If the buffer contains no values, FCS simulates an empty input line. All the sent values are available in the output registers, listed in Table 10.19. Other modules can subscribe to these registers to receive the sent values.

Table 10.19 Input, Output, Serial Output Registers

Name	Meaning	Comments
Input	Adds a value to be received. The system takes the ninth bit from the last value written to InputH.	Read has no specified meaning
InputH	Ninth Input bit; must be written before Input.	Read has no specified meaning
Output	Contains the last value sent. Notification is sent every time a new value is written.	Write has no specified meaning
OutputH	Ninth Output bit. Must be read immediately after Output.	Write has no specified meaning
SerialInput	Alias for Input for SCI 0; connects SCI 0 to terminal window.	Only available in SCI 0. Only supports eight bits.
SerialOutput	Alias for Output for SCI 0; connects SCI 0 to terminal window.	Only available in SCI 0. Only supports eight bits.

Serial Peripheral Interface (SPI)

Table 10.20 describes the SPI interface.

Table 10.20 SPI Interface

Acronym	Full Name	Comments
Control Register 1		
SPIE	Interrupt Enable	Implemented
SPE	System Enable	If set, FCS supports SPI functions
SWOM	Port S Wired-OR Mode	Not simulated; no physical transmission takes place.
MSTR	Master Slave Mode Select	Select Master or Slave mode
CPOL	Clock Polarity	Not simulated; no physical transmission takes place.
CPHA	Clock Phase	Not simulated; no physical transmission takes place.

Table 10.20 SPI Interface (*continued*)

Acronym	Full Name	Comments
SSOE	Slave Select Output Enable	Not simulated; no physical transmission takes place.
LSBF	LSB First Enable	Not simulated; no physical transmission takes place.
Control Register 2		
PUPS	Pull Up Port S Enable	Not simulated; no physical transmission takes place.
RDS	Reduce Drive of Port S	Not simulated; no physical transmission takes place.
SPC0	Serial Pin Control 0	Selects Normal or Bidirectional transmission mode
SPRx	Baud Rate Register	Baud rate of the SPI transmission
SPIF	Interrupt Request	System sets SPIF after the eighth SCK cycle in a data transfer. Status Register read followed by a read or write access to the SPI data register, clears SPIF.
WCOL	Write Collision Status Register	Set upon the writing of new data to the Data Register, during a serial data transfer.
MODF	Mode Error Interrupt Status Flag	Not simulated; no physical transmission takes place.
SP0DR	Data Register	8-bit Data Register for SPI data.
PORTS	Port S Data Register	Not simulated; no physical transmission takes place.
DDRSx	Port S Data Direction for Bit x	Direction of Data. Only bits 4 and 5 have any effect.

Virtual register Value simulates the data register of a second SPI device. This permits simulated communication with a second SPI device. The transmission can be in Normal or Bidirectional Mode; the device can be set as Master or Slave. See also the *MC68HC812A4 Technical Summary (MC68HC812A4TS/D)*.

Key Wakeups

Table 10.21 defines the Key Wakeups.

Table 10.21 Key Wakeups

Acronym	Full Name	Implemented Meaning
PORTD	Port D Register	Implemented
DDRD	Port D Data Direction Register	Implemented
KWIED	Port D Interrupt Enable Register	Implemented
KWIFD	Port D Flag Register	A falling edge on the associated pin sets each flag, provided that corresponding DDRD Register bit is reset. Clear flag by writing 1 to corresponding bit of KWIFD register.
PORTH	Port H Register	Implemented
DDRH	Port H Data Direction Register	Implemented
KWIEH	Port H Interrupt Enable Register	Implemented
KWIFH	Port H Flag Register	A falling edge on the associated pin sets each flag, provided that corresponding DDRH Register bit is reset. Clear flag by writing 1 to corresponding bit of KWIFH register.
PORTJ	Port J Register	Implemented
DDRJ	Port J Data Direction Register	Implemented
KWIEJ	Port J Interrupt Enable Register	Implemented
KWIFJ	Port J Flag Register	A falling edge on the associated pin sets each flag, provided that corresponding DDRJ Register bit is reset. Clear flag by writing 1 to corresponding bit of KWIFJ register.
KPOLJ	Port J Polarity Register	Implemented

Table 10.21 Key Wakeups (*continued*)

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

The FCS does not implement Port D register mapping in wide expanded modes, or in special expanded narrow mode with the MODE Register bit EMD set.

Memory-Mapped Page Registers

Table 10.22 describes the memory-mapped page registers.

Table 10.22 Memory Mapped Page Registers

Acronym	Full Name	Implemented Meaning
Port F Register		
CS	Chip Select/General-Purpose I/O (Bits 0:6)	Not implemented; no physical outputs.
Port G Register		
ADDR	Memory Expansion/ General-Purpose I/O (Bits 0:5)	Not implemented; no physical outputs.
DDRF	Port F Data Direction Register (Bits 0:6)	Not implemented; no physical outputs.
DDRG	Port G Data Direction Register (Bits 0:5)	Not implemented; no physical outputs.
PDA	Data Page	Selects the data page
PPA	Program Page	Selects the program page
PEA	Extra Page	Selects the extra page
Window Definition Registers		
DWEN	Data Window Enable	Enables paging of data space

Table 10.22 Memory Mapped Page Registers (*continued*)

Acronym	Full Name	Implemented Meaning
PWEN	Program Window Enable	Enables paging of program space
EWEN	Extra Window Enable	Enables paging of extra space
A21E-A16E	Memory Expansion Assignment/ General-Purpose I/O	Not simulated; no physical outputs.

Non-Supported Modules

A/D Converter Device (ADC).

Register Block Address Map

Table 10.23 shows the Register Block Address mapping.

Table 10.23 Register Block Address Map

Register Block Address	Description	Remarks
$0000-$000D	Port access	Not simulated; memory configuration controls correct timing of memory accesses.
$000E-$000F	Reserved	
$0010	Internal RAM mapping	Register not simulated. Use the memory configuration dialog box to specify simulated memory configuration.
0x0011	Register Block mapping	Completely simulated
$0012-$0013	ROM/EEPROM mapping	Registers not simulated. Use the memory configuration dialog box to specify simulated memory configuration.
$0014-$0017	Clock Function Control	Completely simulated
$001E-$001F	Interrupt Control & Highest Priority I Interrupt	Completely simulated

Table 10.23 Register Block Address Map (*continued*)

Register Block Address	Description	Remarks
$0020-$002E	Key Wakeup Control	Completely simulated
$002F	Reserved	
$0030-$0033	Port Registers	Currently not simulated
$0034-$0038	PAGE & memory configuration Registers	Page Registers are simulated
$0039-$003B	Reserved	
$003C-$003F	Chip select control registers	Currently not simulated
$0040-$0043	PLL divider registers	Currently not simulated
$0044-$0046	Reserved	
$0047	Clock Control Register	Completely simulated
$0048-$005F	Reserved	
$0060-$0069	Analog to Digital Converter	Currently not simulated
$006A-$006E	Reserved	
$006F	PORTAD	Currently not simulated
$0070-$007F	ADRxH/reserved	Currently not simulated
$0080-$009F	Timer Registers	Completely simulated
$00A0-$00A3	Pulse Accumulator Control Registers	Completely simulated
$00A4-$00AC	Reserved	
$00AD-$00AF	Timer Test, Timer Port	Completely simulated
$00B0-$00BF	Reserved	

Table 10.23 Register Block Address Map (*continued*)

Register Block Address	Description	Remarks
$00C0-$00C7	SCI0	Completely simulated
$00C8-$00CF	SCI1	Completely simulated
$00D0-$00D3	SPI	Completely simulated
$00D4	Reserved	
$00D5-$00D7	SPI, PORTS	Completely simulated
$00D8-$00EF	Reserved	
$00F0-$00F3	EEPROM Control	Currently not simulated
$00F3-$01FF	Reserved	

Related Documentation

The following documents are available from Freescale:

- *MC68HC812A4TS/D*, Technical Summary for MC68HC812A4 16-Bit Microcontroller, 1996
- *CPU12 Reference Manual (CPU12RM/AD)*

HC912DG128x, HC912DT128x

This section explains derivative simulated mechanisms and implemented features that match the real HC12 derivatives. It also explains simulation limitations. (For technical specifications of all I/O mechanisms, see *MC68HC912DA128/MC68HC912DG128 16-Bit Microcontroller Technical Summary (MC68HC912DA128TS/D)*.)

Register Block

You can reassign the 1-kilobyte register block to any 2-kilobyte boundary within the standard 64-kilobyte address space.

Related Register

INITRG Initialization of Internal Register Position Register, simulated.

Memory Expansion Register

The system fully simulates this mechanism within CALL and RTC instructions for **banked memory model**.

Related Register

Program Page Register PPAGE: PIX2/PIX1/PIX0 bits memory defined but NOT updated.

Enhanced Capture Timer

The 16-Bit Modulus Down-Counter is fully simulated, and contains the following:

- Eight Input Capture/Output Compare channels

  All channels are non-buffered and identical, except channel 7, with TCRE (Timer Counter Reset Enable) also implemented.

- **PORTT** pins

  Configure individually as standard, parallel-port I/O pins, or as timer pins. For standard parallel I/O pins, reading and writing are transparent, behaving like reading/writing in typical RAM. For this configuration, assign the value 1 to the channel x bit IOSx, in the TIOS register (for compare mode). Assign the value 0 to the OMx and OLx bits of the TCL1 or TCTL2 register for **Timer disconnected from output pin logic** mode/output action.

  – Capture Stimulation on PORTT.

    You can simulate rising- and falling-edge input signals on PPORT with the Stimulate component (I/O Stimulation). In this case, PORTT is bit accessible via non-memory-mapped I/O registers PORTTBit0 through PORTTBit7.

    The stimulation example below periodically stimulates the PORTT bit 5 to simulate an input capture.

```
def a = TIMER.PORTTBit5;
PERIODICAL 4000, 500:
   1000 a = 1;
   3000 a = 0;
END
```

    Other user-designed I/O components also can set the PORTT bit value. Use the `OP_SetValue("RegisterBlock.PORTTBit5",&parameter, NO_UPDATE);` function (with `parameter.n = 0 | 1`).

16-Bit Modulus Down-Counter

Table 10.24 shows the simulated registers and fields of the 16-bit modulus down-counter.

Table 10.24 16-Bit Modulus Down-Counter Related Registers

Register Acronym	Full Register Name	Simulated Fields
MCCTL	16-bit modulus down counter control register	All bits except ICLAT
MCCNT	Modulus down-counter count register	All
Capture / Compare Timer		
TIOS	Timer input capture/output compare select	All
CFORC	Timer compare force register	All
TCNT	Timer count register	All
TCTL1/TCTL2	Timer control register - output	All
TCTL3/TCTL4	Timer control register - input	All
TMSK1	Timer interrupt mask 1	All
TMSK2	Timer interrupt mask 2	TOI TCRE PR2:0
TFLG1/TFLG2	Main timer interrupt flags	All
TC0 to **TC7**	Timer input capture/output compare registers	All

Serial Communication Interface (SCI)

This I/O Device simulates the two SCI signals SCI0 and SCI1. The non-memory-mapped registers `SCIInput/SCIInputH` and `SerialInput` send characters to the SCI Module. The non-memory-mapped registers `SCIOutput/SCIOutputH` and `SerialOutput` contain the characters sent from the SCI Module.

Table 10.25 Serial Communication Interface Related Registers

Register Acronym	Full Register Name	Simulated Fields
SC0BDH/ SC1BDH	SCI Baud Rate Register High	SBR (bits 4:0) simulated BTST, BSPL, and BRLD (bits 7:5) reserved for test functions
SC0BDL/ SC1BDL	SCI Baud Rate Register Low	SBR (bits 7:0) simulated
SC0CR1/ SC1CR1	SCI Control Register 1	M (bit 4) simulated ILT (bit 2) simulated LOOPS, WOMS, RSRC, WAKE, PE, and PT (bits 7:5, 3, 1:0) not simulated
SC0CR2/ SC1CR2	SCI Control Register 2	TIE, TCIE, RIE, ILIE, TE, RE, SBK (bits 7:2, 0) simulated; RWU (bit 1) not simulated
SC0SR1/ SC1SR1	SCI Status Register 1	TDRE TC RDRF IDLE OR (bits 7:3) simulated; NF, FE, and PF (bits 2:0) not simulated
SC0SR2/ SC1SR2	SCI Status Register 2	RAF (bit 0) simulated; bits 7:1 unused
SC0DRH/ SC1DRH	SCI Data Register High	R8 T8 (bits 7:6) simulated
SC0DRL/ SC1DRL	SCI Data Register Low	R7:0 T7:0 simulated

Table 10.26 contains information about the SCI input and output registers.

Table 10.26 Input, Output, Serial Output Registers

Name	Meaning	Comment
SCIInput	Non-memory-mapped register that sends a character to the SCI. Read access to the SCDR can read this value. System takes the ninth bit from the SCIInputH register. Read access to SCIInput has no specified meaning.	bits 7:0 — character sent to the SCI
SCIInputH	Non-memory-mapped register that sends a character, the ninth bit, to the SCI. Write this register value before writing the SCIInput register value. Read access to SCIInputH has no specified meaning.	bits 7:1 — unused bit 0 — ninth bit sent to the SCI
SCIOutput	Non-memory-mapped register that receives a character sent from the SCI. Write access to the SCDR triggers the value that the SCIOutput receives. SCIOutputH register receives the ninth bit. Write access to SCIOutput has no specified meaning.	bits 7:0 — character sent from the SCI
SCIOutputH	Non-memory-mapped register that receives a character, the ninth bit, sent from the SCI. Write access to SCIOutput has no specified meaning.	bits 7:1 — unused bit 0 — ninth bit sent from the SCI
SerialInput	Non-memory-mapped register is an alias for the SCIInput register. Connects the SCI to the terminal window, but does not support the ninth bit. A read access to SerialInput has no specified meaning.	bits 7:0 — data sent from terminal window to SCI
SerialOutput	Non-memory-mapped register is an alias for SCIOutput register. Connects the SCI to the terminal window, but does not support the ninth bit. Write access to SerialOutput has no specified meaning.	bits 7:0 — data sent from SCI to terminal window

FCS Visualization Utilities

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

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

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

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

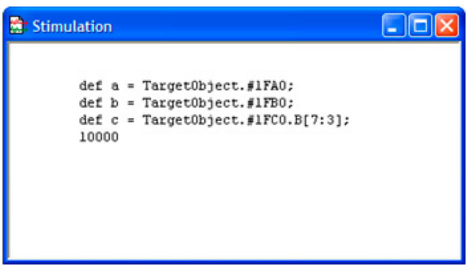
Stimulation Component

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

NOTE the True-Time I/O Stimulation section describes in detail and with example how to take advantage of this component.

The Stimulation window component shown in Figure 10.12 provides the basic FCS functionality. It serves to execute timed action and raise exception events. The Stimulation component displays and executes I/O stimulation described in a text file.

Figure 10.12 Stimulation Window



Stimulation Context Menu

Figure 10.13 shows functions associated with the Source component. Table 10.27 describes these functions.

Figure 10.13 Stimulation Context Menu

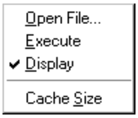


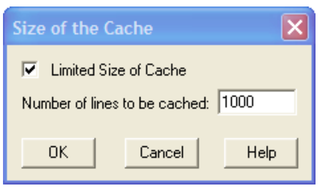
Table 10.27 Stimulation Context Menu Description

Menu Entry	Description
Open File	Opens a dialog box to load a stimulation file.
Execute	Starts execution of the input file.
Display	Switches display of stimulated file on or off.
Cache size	Opens the **Size of the Cache** dialog box.

Cache Size

The **Size of the Cache** dialog box, shown in Figure 10.14, allows you to define the number of lines displayed in the Stimulation component. Clear the **Limited Size of Cache** checkbox to have an unlimited number of lines. Check the **Limited Size of Cache** check box to limit the number of lines to the value displayed in the edit box. Specify a value between 10 and 1,000,000. By default, the number of lines is 1000.

Figure 10.14 Size of the Cache Dialog Box



NOTE Increasing the cache size may slow performance.

Example of a Stimulation File

Using an editor, open the file named `IO_VAR.TXT` located in the project directory. Listing 10.1 shows an example file.

Listing 10.1 Stimulation File Example

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

PERIODICAL 200000, 50:
     50000 a = 128;
    150000 a = 4;
END
10000000 a = 0;
```

The first line defines the stimulated object, 1 byte wide, and located at address 0x210.

This code accesses the memory location 0x210 periodically 50 times, once 200000 cycles have been executed (line 3). First the memory location is set to 128, and then 100000 cycles later, it is set to 4.

NOTE The True-Time I/O Stimulation section describes in detail and with examples how to take advantage of this component.

Drag Out

Nothing can be dragged out.

Drop Into

Nothing can be dragged in.

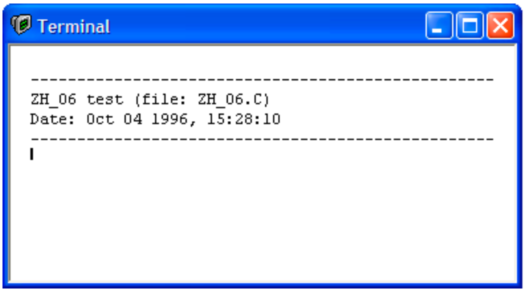
Demo Version Limitations

Generates only 15 interrupts and memory accesses.

Terminal Component

Use the Terminal component window shown in Figure 10.15 to simulate input and output. It can receive characters from several input devices and send them to other devices.

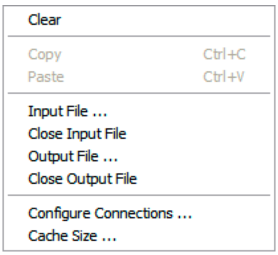
Figure 10.15 Terminal Window



You can use a virtual SCI port provided by the framework for communication with the target, but it is also possible to use the keyboard, the display, files or even the serial port of your computer as I/O devices.

To control and configure a terminal component use the Terminal menu of the terminal shown in Figure 10.16.

Figure 10.16 Terminal Menu and Context Menu

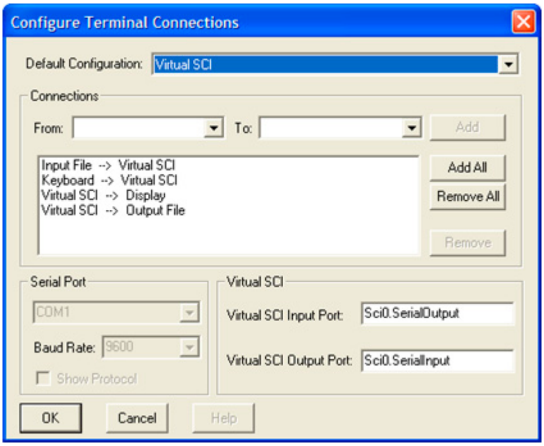


To open the context menu, right click in the terminal window.

Configure Terminal Connections

Using the terminal window, you can redirect characters received from any available input device to any available output device. Specify these connections by choosing **Configure Connections** in the context menu of the terminal component. This opens the dialog box shown in Figure 10.17.

Figure 10.17 Configure Terminal Connections Dialog Box



You can 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 click 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 click 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 FCS 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.

To use a file as an input stream, make sure there is at least one connection from the input file to any output device. Then open an input file by choosing **Input File** from the context menu. As soon as you click the **OK** button in the **File Open** dialog, input from the file starts. The file closes as soon as the end of file is reached or as soon as you choose **Close Input File** from the context menu.

When the input file reaches the end a CTRL-Z character (ASCII code 26 decimal) is sent to all output devices receiving characters from the input file to notify them that the file transfer is complete.

Redirecting input devices to an output file requires a similar process. Make sure that you have chosen your connections from input devices to the output file. Then open or create your output file by choosing **Output File** from the context menu. If the file does not exist it is created. Otherwise you can choose to overwrite or append the existing file. To stop writing to the output file choose **Close Output File** from the context menu.

File Control Commands

It is also possible to open and close input and output files through special Escape sequences in the data stream from serial port or virtual SCI. Table 10.28 illustrates the possible commands and associated Escape sequences in which *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).

Table 10.28 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; suppress output to terminal display.

You can give these commands 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. Listing 10.2 shows the source code in `terminal.c`.

Listing 10.2 TERM_Direct Source Code

```
void TERM_Direct(TERM_DirectKind what, const char* fileName) {
  /* sets direction of the terminal */
  if (what < TERM_TO_WINDOW || what > TERM_APPEND_FILE) return;
  TERM_Write(ESC); TERM_Write('h');
  TERM_Write((char)(what + '0'));
  if (what != TERM_TO_WINDOW && what != TERM_FROM_KEYS) {
    TERM_WriteString(fileName); TERM_Write(CR);
  }
}
```

In the example, the parameter `what` is one of the following constants:

- `TERM_TO_WINDOW`: send output to terminal window
- `TERM_TO_BOTH`: send output to file and window
- `TERM_TO_FILE`: send output to file `fileName`
- `TERM_FROM_KEYS`: read from keyboard (close input file)
- `TERM_FROM_FILE`: read input from file `fileName`
- `TERM_APPEND_BOTH`: append output to file and window
- `TERM_APPEND_FILE`: append output to file `fileName`

See also `terminal.h` for further details.

How to Use Virtual SCI

In its default **Virtual SCI** configuration the terminal component accesses the target through the Object Pool interface.

To make the terminal component work in this default configuration, the target must provide an object with the name **Sci0**. If no **Sci0** object is available, no input or output happens. It is possible to check, through the Inspector component, if the environment currently provides an **Sci0** object.

NOTE Only some specific FCS components currently have an **Sci0** object. For all other FCS 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.

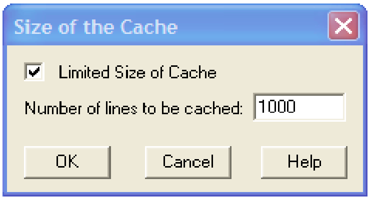
For implementations of this register with help of the **IOBase** class, use the **I**`OB_NotifyAnyChanges` flag. Otherwise only the first of two identical characters are received.

It is also possible to configure the terminal to use another object in the Object Pool instead of **Sci0** with which to communicate. Refer to Configure Terminal Connections for information about where you can do this.

Cache Size

The item **Cache Size** in the context menu allows you to set the number of lines in the terminal window with the dialog shown in Figure 10.18.

Figure 10.18 Size of the Cache Dialog Box



True-Time I/O Stimulation

Use the FCS I/O Stimulation component to trigger I/O events. With the Stimulation component loaded, interrupts and register modifications invoked by the hardware can be simulated. This tutorial introduces and explains examples of stimulation files.

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

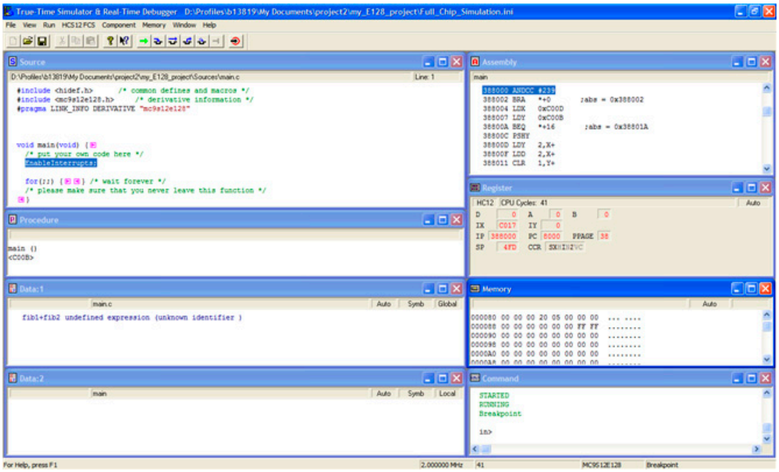
The following examples demonstrate several uses of true-time I/O stimulation.

Running an Example Program Without Stimulation

1. Run the debugger with the FCS connection.

   Figure 10.19 shows the Main window.

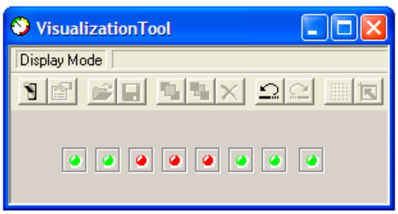
Figure 10.19 FCS I/O-Simulation Main Window



2. Choose **Simulator > Set > Sim**.
3. Choose **Component > Open > Visualizationtool.**

   Figure 10.20 shows the LED instruments within Visualization Tool component.

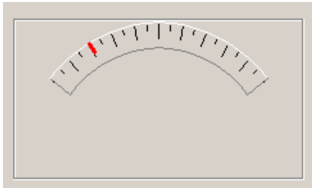
Figure 10.20 Configure LED Instrument



4. Choose **Visualization Tool > Add New Instrument > Analog** instrument.

Figure 10.21 shows the Analog instrument.

Figure 10.21 Analog Instrument



5. Choose **Simulator > Load io_demo.abs**.
6. Choose **Run > Start/Continue** or click the green arrow icon.
7. If the program halts in startup, click the **Start/Continue** command again.
8. Choose **Run > Halt** to stop execution after a few seconds.

The Analog instrument 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 graphic.

The **IO_Show** procedure in `io_demo.c`, shown in Listing 10.3, this value is incremented or decremented, depending on the raise direction. The raise direction depends on a global variable `dir` that is returned when the top or bottom value is reached.

Listing 10.3 IO_Show Procedure in io_demo.c

```
static void IO_Show(void) {
   for (;;) {                          // endless loop
      dir = 1;
      do {
         Delay();
         PORT_DATA++;
      } while ((dir == 1) && (PORT_DATA != 255));
      dir = -1;
      do {
         Delay();
         PORT_DATA--;
      } while ((dir == -1) && (PORT_DATA != 0));
   }
}
```

Example Program with Periodical Stimulation of a Variable

1. Choose **Simulator > Reset**.
2. Choose **Simulator > Load Io_demo.abs**.
3. Choose **Component > Open > Stimulation**

Figure 10.22 shows the Stimulation component.

Figure 10.22 Stimulation Component Window

4. Activate Stimulation Window by clicking on it.
5. Choose **Stimulation > Open File io_var.txt**.
6. Choose **Stimulation > Execute**.
7. Choose **Run > Start/Continue**.

The **Stimulation** component executing `io_var.txt` accesses TargetObject at address **0x210** associated with **PORT_DATA** in the source. You can observe this by watching the Template component. The arrow is not continuously rising, but jumping around. The value of **PORT_DATA** is now handled from the Stimulation component.

Using an editor, open the file named `io_var.txt` in the FCS demo directory. This file looks like Listing 10.4.

Listing 10.4 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. */
```

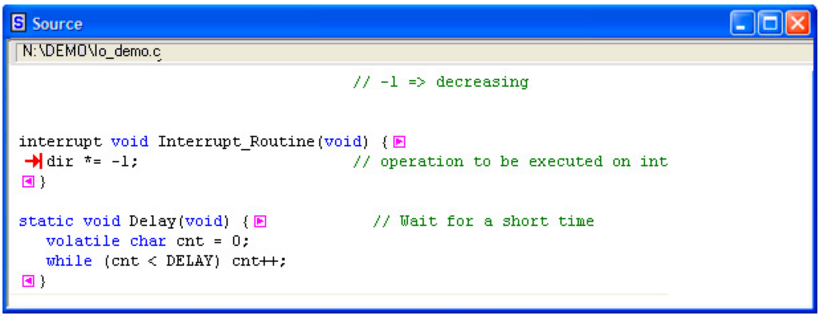
First, the simulated object is defined. This object is located at address 0x210 and is 1 byte wide. Once 200,000 cycles have been executed, the memory location 0x210 is accessed periodically 50 times. First the memory location is set to 128, then 100,000 cycles later, it is set to 4.

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.

Figure 10.23 shows the Source component window.

Figure 10.23 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 FCS stops on the breakpoint in the interrupt routine and the corresponding source line is highlighted. The interrupt has been called. Start the FCS. It stops approximately each 100,000 cycles on the same breakpoint. Restart and repeat these actions until 1,200,000 cycles. Start again; the FCS runs until 10,000,000 cycles and stops on the breakpoint. Start the FCS. It continues to run. The stimulation is finished.

The interrupts have been invoked by the Stimulation component source `io_int.txt`. Listing 10.5 shows the listing of the Stimulation file.

Listing 10.5 io_int.txt

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

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

10000000 RAISE 7, 3, "test_interrupt";
```

In the first line, the stimulated object is defined. The interrupt is raised periodically 10 times. The **RAISE** command takes the number of the interrupt in the interrupt vector map as the first argument. This number **7** in our example is arbitrarily chosen. To export this example to a different target processor, take a look at the interrupt vector map in the technical data manual of the matching MCU. Using an editor, open the `io_demo.prm` file in the same demo directory. You can see at the end of this file how to set the interrupt vector (adapt it to your needs).

```
VECTOR 7 Interrupt_Function  /* set vector on Interrupt 7  */
```

If the interrupt vector address is not specified in the `prm` file, the FCS stops when an interrupt is generated. The exception mnemonic (matching the interrupt vector number) is displayed in the FCS status bar.

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. Ten 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 10.6 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 `LED` instrument. Definitions of **b**, **c** and **pbits** are only here for illustration. Remove these definition lines and the lines that refer to them, if the example presented here is not executable.

Listing 10.6 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;
```

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

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

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

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

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

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

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

```
def pbits = Leds.Port_Register.B[7:3];
```

This code defines `pbits` as bits 5, 6 and 7 in the byte (.B) register named **Port_Register** in **LEDs**. (In the Full Chip Simulation, names of target objects can be specified. In our example, it is the name of the port register defined by the IO-LED component).

```
#10000 pbits = 3;
```

This code sets the three bits of **LEDs**. **Port_Register** accessed with **pbits** to binary **011**. Other bits are unaffected. The new value of **Port_Register** will be 0x75, if the initial value was 0x55. Values outside the valid BitRange of pbits are truncated (in this example only values from 0 to 7 are allowed for pbits). The # means that the time of execution of the instruction is 10000 cycles after the start of the simulation.

```
20000 a = 0;
```

This code 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;
```

This code reads `pbits` (three 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:
```

This code executes the following block10 times.

```
   10000 a = 128;
   30000 RAISE 7, 3, "test_interrupt";
```

Execution starts 100000 cycles after the start of the simulation.

```
10000 a = 128;
```

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

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

This code 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
```

This code indicates the 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";
```

This code 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

This section details the input file syntax required by the FCS.

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

- Omitting **bitRange** affects all bits of the object register. Specifying **bitRange** applies the mask defined by this **bitRange** to the value calculated with the **Expression**. This value affects only the bits of the object register defined in the **bitRange**.
- Bits are numbered from right to left (in a byte, bit 7 is the left-most bit). So in **bitRange**, **noOfBits** is always less than or equal to **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 FCS began.
- If “+” is specified, the time is relative to the previous time specification.
- If nothing is specified, time is the number of cycles since execution of the Stimulation file.
- If size is omitted, the default size is byte (B).
- The periodical event is sent for the first time at initial time + start + time specified in periodical timed event.
- In the **PerTimedEvent** declaration, the “+” is optional. If specified or not, the following time is interpreted exactly the same way.
- The periodical events are not displayed in the stimulation screen.

Electrical Signal Generators and Signals Application to Device Pins

This section describes the FCS-relevant signal generators and device pins.

Signal IO Component

This **Signal IO** is the first implementation of a **Signal Generator** reading a file describing (electrical) levels, in real debugger time. Levels are applied and available at a virtual IO pin called **SignalPin** as **float** value.

Levels are programmed one after the other during the debugger internal FCS Event queue.

If level durations are smaller then cycle time or smaller than cycles, **undersampling** is performed in the signal file.

Up to 16 Signal Generators can be run at the same time.

Signal Description File EBNF

This section shows the signal file format and some example signal files.

Signal File Format

```
FILELOOP=<INF| nbr of file loops to perform> {signal block}*
EOF
```

Signal Block Description

```
{signal header}
{signal data}
```

Signal Header Description

```
LOOP=<INF| nbr of file loops to perform>
TIMEUNIT=<NONE|CYCLES|SECONDS>
TIMEFACTOR=<double value>
GAIN=<double value>
DCOFFSET=<double value>
OPTION=NORMAL|ONLYPOSITIVE|ONLYNEGATIVE|ABSOLUTE
```

Signal Data Description

```
{<level double value> [<time double value (duration in
seconds or cycles)>]}*
```

File Example 1

```
FILELOOP=INF
```

```
LOOP=4
TIMEUNIT=SECONDS
TIMEFACTOR=0.5
GAIN=1
DCOFFSET=0
OPTION=NORMAL
0.000000e+000 3.051758e-005
3.051758e-005 3.051758e-005
6.103516e-005 3.051758e-005
9.155273e-005 3.051758e-005
1.220703e-004 3.051758e-005
1.525879e-004 3.051758e-005
1.831055e-004 3.051758e-005
LOOP=16
TIMEUNIT=SECONDS
TIMEFACTOR=3.6
GAIN=-4.2
DCOFFSET=2.5
OPTION=NORMAL
2.136230e-004 3.051758e-005
2.441406e-004 3.051758e-005
2.746582e-004 3.051758e-005
3.051758e-004 3.051758e-005
3.356934e-004 3.051758e-005
3.662109e-004 3.051758e-005
EOF
```

File Example 2

```
FILELOOP=INF
LOOP=INF
TIMEUNIT=NONE
TIMEFACTOR=0.5
GAIN=1
DCOFFSET=0
OPTION=NORMAL
-5
5
2
8
-0.4e-3
300
123
EOF
```

File Parameters

Table 10.29 shows the available file parameters.

Table 10.29 Signal Description File Parameters

Parameter	Description
`LOOP/FILELOOP`	`INF` means infinite loop. If a block is `INF`, scanning stays in this block till the IO is closed or `CLOSESIGNALFILE` command is executed. If a number is specified, scanning loops through the block that number of times.
`TIMEUNIT`	`CYCLES` means that the second data field is specified in cycles. `SECONDS` means that the second data field is specified in seconds. -`NONE` means that the second data field does not exist and the data time unit is forced to 1s. Adjust the data time unit by the `TIMEFACTOR` parameter.
`TIMEFACTOR`	At event programming, multiplies the number of cycles or time duration by this factor.
`GAIN`	At Pin level setup, multiply the level by this gain.
`DCOFFSET`	At Pin level setup, level offset specified after gain is applied.
`OPTION`	`NORMAL`: do nothing. `ONLYPOSITIVE`: at Pin level setup, after gain and offset, set 0 if signal level < 0. `ONLYNEGATIVE`: at Pin level setup, after gain and offset, set 0 if signal level > 0. `ABSOLUTE`: at Pin level setup, after gain and offset, set `|signal level|`.

Signal IO Usage

The Signal IO can handle 16 signals at the same time. Each signal block is independent in parameters and options from other blocks. Open the Signal component within the Open I/O Component Dialog Box or with the `openio signal` command. Release the Signal component within the same dialog or with the `close signal` command. See Signal Commands for specific Signal IO commands.

Base Signal Files Provided

You can reuse the base signal files shown in Table 10.30 to create more complex signal descriptions. These files are usually stored in the `prog\FCSsignals` folder of the debugger installation path.

Table 10.30 Base Signal Files

File	Properties
`saw_11bit_0_5v_1Hz.txt`	• Sawtooth signal • 11-bit sampling definition • Scaled on a 1 Hz frequency • 0 to 5 Volts voltage range.
`saw_8bit_0_5v_1kHz.txt`	• Sawtooth signal • 8-bit sampling definition • Scaled on a 1000 Hz frequency • 0 to 5 Volts voltage range.
`sinus_11bit_0_5v_1Hz.txt`	• Sinus signal • 11-bit sampling definition • Scaled on a 1 Hz frequency • 0 to 5 Volts voltage range.
`sinus_8bit_0_5v_1kHz.txt`	• Sinus signal • 8-bit sampling definition • Scaled on a 1000 Hz frequency • 0 to 5 Volts voltage range.
`square_1_5v_1Hz.txt`	• Pure square signal • Scaled on a 1 Hz frequency • 1 volt at low level • 5 volts at high level.
`square_1_5v_1kHz.txt`	• Pure square signal • Scaled on a 1000 Hz frequency • 1 volt at low level • 5 volts at high level.

Virtual Wire Connections with the Pinconn IO Component

This section explains making virtual wire connections using the Pinconn IO Component.

Pinconn IO

Use the Pinconn IO component to create virtual links/shortcuts between processor device pins, like a simple wire. Open the Pinconn component within the Open I/O Component Dialog Box or with the `openio pinconn` command. Release the Pinconn component within the same dialog or with the `close pinconn` command. See Pinconn Commands for the Pinconn IO commands.

WARNING! It is up to the user to properly connect input pins to output pins without bus or level conflicts.

Command Set to Apply Signal on ATD Pin

The following example loads the Pinconn and Signal IO components, and creates a signal generator generating the signal described in `square_1_5v_1kHz.txt`. The generator output signal pin is connected to the on-chip ADC via the PAD0 pin.

```
openio Pinconn
openio Signal
setsignalfile 0 "square_1_5v_1kHz.txt"
connect "SignalGenerator0.SignalPin","Atd0.PAD0"
```

FCS Tutorials

This chapter contains a tutorial on how to use the Full Chip Simulation. The tutorial is split up into small steps. After completing the last step a fully functional example exists.

This chapter contains the following sections:

- Guess the Number
- PWM Channel 0

Guess the Number

In this tutorial, we create, step by step, the demonstration run in the executive tutorial. The application uses the SCI and a terminal window from the debugger. At the end the user can guess a number between 0 and 9, using the Terminal window. The final application runs on real hardware as well.

Environment Setup

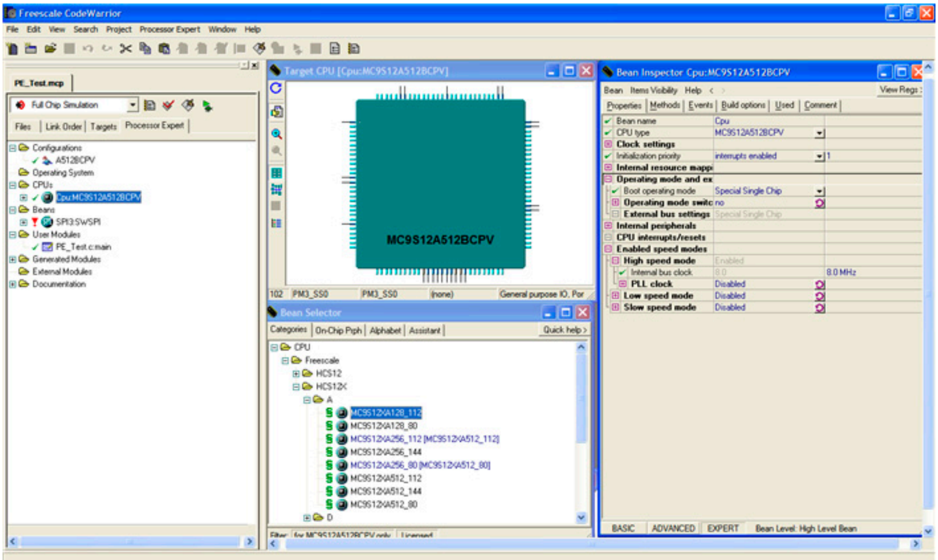
- The tutorial uses Processor Expert. You can get a free Processor Expert license (Special Edition) at `www.codewarrior.com`.
- To run the example on real hardware, you need a serial cable. This cable must connect COM1 (PC) with the SCI0 (Hardware Board).

Create the project

1. Launch the **CodeWarrior IDE**.
2. In the CodeWarrior menu, Select **File** > **New Project**.
3. Select **HCS12 > HCS12D Family > MC9S12DP256B** derivative in HC(S)12X Microcontrollers New Project window.
4. Select **Full Chip Simulation** connection and click **Next** to proceed.
5. Select **C** for the language and enter a project name like **MyGuessTheNumber**.
6. Change the directory if you want (**Location > Set**) and click **Next**.
7. Add existing files to the project if required and click **Next**.
8. Select **Processor Expert** radio button from the rapid application developement options and click **Next**.
9. Select **ANSI startup code, Banked memory model, and float is IEEE 32 and double is IEEE 32** and click **Next**.
10. Select **No** for **PCLint** support and click **Finish**.

    This creates a new project with Processor Expert available. Several windows are visible:

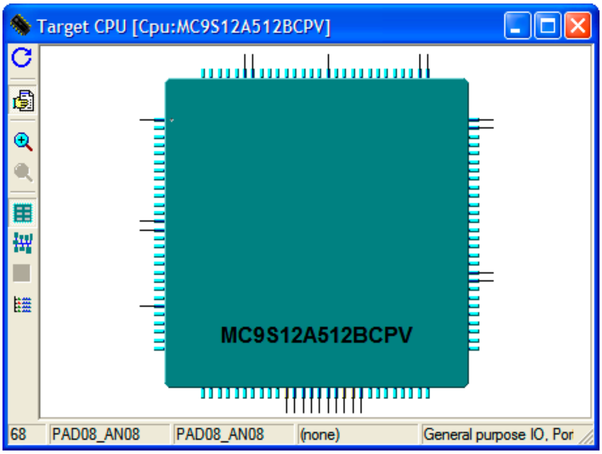
Figure 10.24 Created Project



Target CPU Window

The **Target CPU** window in the center shows a footprint of the processor selected for the development. In the device, we see the different on-chip modules such as CPU, Timer, and ADC. Modules with an icon attached to them are modules used by the application. The pins used to connect external functions are indicated by a line and an icon symbol of the function attached (CPU and Port A).

Figure 10.25 Target CPU Window



Optional:

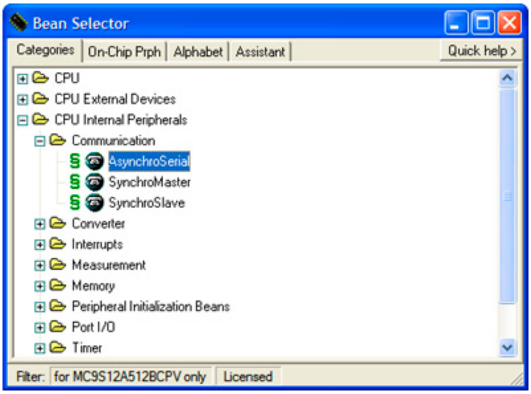
- Place the cursor on the pins to see a description of their functions.
- Enlarge the **Target CPU** window to see different on-chip modules.

Bean Selector Window

The **Bean Selector** window offers the developer a list of beans to add to the project. Some of the beans may not be usable with some versions of the CodeWarrior IDE. The Standard and Professional Editions offer a wider range of hardware and software beans than the Special Edition.

- Select **Bean Categories > CPU internal peripherals > Communication > AsynchroSerial**.

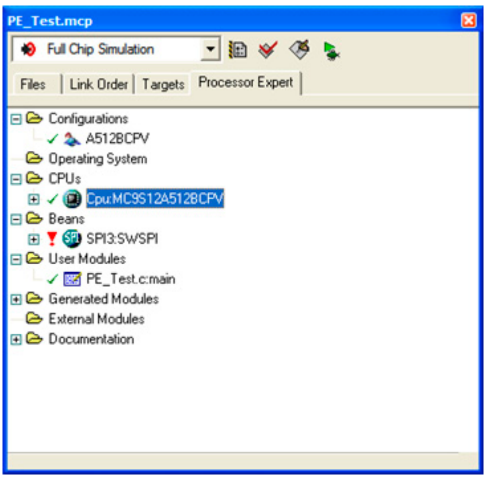
Figure 10.26 Bean Selector Window - Selection of AsynchroSerial Bean



Project Panel Window

The **Project Panel** window shows and keeps track of the beans that have been created for this application. **This Panel is a tab of the Project Manager window**. A click on the [+] next to a bean shows a list of methods and/or events related to the bean. A green checkmark indicates whether the named methods or event is selected and a red cross indicates that code has not been generated.

Figure 10.27 Project Window - Processor Expert Tab



The **Beans** folder shows the previously created bean with the name **AS1:AsynchroSerial**.

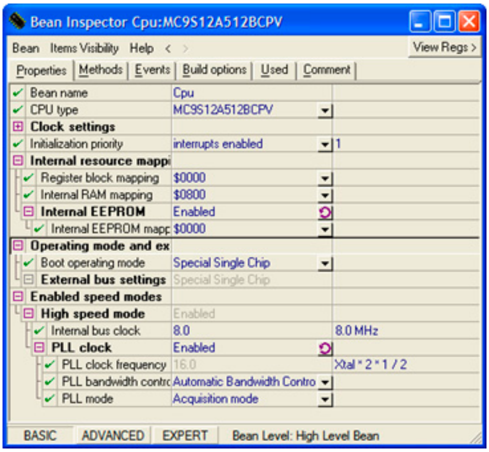
Bean Inspector AS1:AsynchroSerial Window

In this window you can modify the behavior of the bean to your needs. Use the **Properties** tab to make general settings. Use the **Methods** and **Events** tabs to modify software drivers.

1. Select the **Properties** tab
2. Enter a proper **baud rate**.

   To run on real hardware, check your board manual for the right value. To run on FCS only, enter **9600**.

Figure 10.28 Bean Inspector Window



Generation of Driver Code

Next, generate the code for the I/O drivers and the files for the user code.

- Select the **Make** icon in the Project Manager window (or the menu bar **Project > Make** or **[F7]**).

Processor Expert shows several messages. One message indicates that we have started the code generation. The second message shows the progress with the information processed and the code generated. Another window shows compiling and linking progress.

Verification of Files Created

We can verify the folders created by Processor Expert.

User Modules

A file called `MyGuessTheNumber.C` is the placeholder for the main procedure and any other procedure desired by the user. You can also place these other procedures in additional files.

Generated Code

The `.C` files for the code associated with the beans are added to the project. This includes initialization, input, output and the declarations necessary for the use of the functions.

Entering User Code

1. Open the user module `MyGuessTheNumber.C`
2. Insert the following code **before** the main routine:

```
#include <stdlib.h>
void PutChar(unsigned char c) {
  while (AS1_SendChar(c) == ERR_TXFULL) {
    // could wait a bit here
  }
}
void PutString(const char* str) {
  while (str[0] != '\0') {
    PutChar(str[0]);
    str++;
  }
}

void GuessTheNumber(void) {
  int ran = rand() / (RAND_MAX / 9);
  AS1_Init();

  PutString("Guess a Number between 0 and 9\n");
  PutString("Number: ");
  for (;;) {
    unsigned char c;
    if (AS1_RecvChar(&c) == ERR_OK) {
      PutChar(c); PutChar(' ');
      if(c < '0' || c > '9') {
        PutString("not a number, try again\n");
      } else if(c == ran + '0') {
        PutString("\nCongratulation! You have found the number!");
```

```
        PutString("\nGuess a new number\n");
        ran = rand() / (RAND_MAX / 9);
      } else if(c > ran + '0') {
        PutString("lower\n");
      } else {
        PutString("greater\n");
      }
      PutString("Number: ");
    } else {
      // could wait a bit here
    }
  } // for
}
```

3. Call the function **GuessTheNumber** in the main routine.

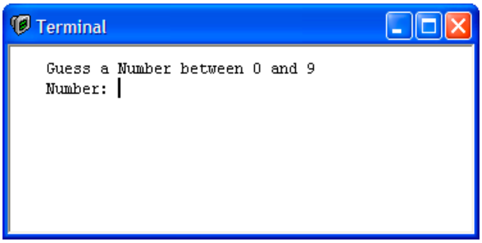
```
void main(void) {
  /*** Processor Expert internal initialization. DON'T REMOVE THIS
CODE! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.      ***/

  /*Write your code here*/
  GuessTheNumber();

  /*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!
***/
    for(;;);
  /*** Processor Expert end of main routine. DON'T WRITE CODE BELOW!
***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT! ***/
```

Run the Application

The application is now finished and we can launch it. Make sure you have chosen the FCS connection.

1. Select the **Debug** icon in the Project Manager window (or the menu bar **Project > Debug** or **[F5]**).
2. Select **Component > Open** in the debugger and open the **Terminal** component.
3. Select the **Save** icon in debugger (or the menu bar **File > Save Configuration**) to save the window layout.
4. Select the **Debug** icon in debugger (or the menu bar **Run > Start/Continue** or **[F5]**).

Figure 10.29 Debugger Main Window - Final Application



PWM Channel 0

We are going to create, step by step, the demo run in the executive tutorial. The application makes use of the Pulse Width Accumulator (PWM). With the final application you will be able to change the period and duty time of the PWM and see the changes displayed in a chart.

Environment Setup

- The tutorial uses Processor Expert. You can get a free Processor Expert license (Special Edition) from `www.codewarrior.com`.

Creating the Project

1. Launch the **CodeWarrior IDE**.
2. In the CodeWarrior menu, Select **File > New Project**.
3. Select **HCS12 > HCS12D Family > MC9S12DP256B** derivative in HC(S)12X Microcontrollers New Project window.
4. Select **Full Chip Simulation** connection and click **Next** to proceed.
5. Select **C** for the language and enter a project name like **MyPWMChannel0**.
6. Change the directory if you want (**Location > Set**) and click **Next**.
7. Add existing files to the project if required and click **Next**.
8. Select **Processor Expert** radio button from the rapid application developement option and click **Next**.
9. Select **ANSI startup code, Banked memory model,** and **None** for floating point support and click **Next**.
10. Select **No** for **PCLint support** and click **Finish**.

    The IDE creates a new project with the Processor Expert available. Several windows will be visible:

Target CPU Window

The *Target CPU* window in the center shows a footprint of the processor selected for the development. In the device, we see the different on-chip modules such as CPU, Timer, and ADC. Modules with an icon attached to them are modules used by the application. The pins that are used to connect external functions are indicated by a line and an icon, symbol of the function attached (CPU and Port A).

Optional:

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

Creating PWM Bean

- Select **Bean Categories > CPU internal peripherals > Timer > PWM**

Project Panel Window

The **Project Panel** window shows and keeps track of the beans that have been created for this application. **This Panel is a tab of the Project Manager window**. A click on the [+] next to a bean shows a list of methods and/or events related to the bean. A green checkmark indicates if the named methods or event is selected and a red cross indicates that code has not been generated.

Locate the previously created bean, with the name **PWM8:PWM**, under *Beans*.

Bean Inspector PWM8.PWM

In this window you can modify the behavior of the bean to your needs. Modify general settings in the **Properties** tab. Modify software drivers under the **Methods** tab and the **Events** tab.

1. Select **Properties** tab
2. Select **Period** and enter **100 ms**
3. Select **Starting pulse width** and enter **10 ms**

Generate Driver Code

Now we will generate the code for the I/O drivers and the files for the user code.

- Select the **Make** icon in the Project Manager window (or the menu bar **Project > Make** or **[F7]**).

Processor Expert displays several messages. One message indicates that we have started the code generation. The second message shows the progress with the information processed and the code generated. Another window shows compiling and linking progress.

Verification of Files Created

We can verify the folders created by Processor Expert.

User Modules

A file `MyPWMChannel0.C` is the placeholder for the main procedure and any other procedure desired by the user. These other procedures can be placed in additional files.

Generated Code

The `.C` files for the code is associated with the beans added to the project. This includes initialization, input, output and the declarations necessary for the use of the functions.

Entering User Code

- Open the user module `MyPWMChannel0.C`
- Replace the main routine with the following **code**:

```
volatile static byte pwmChannel[1];
volatile static unsigned int pwmRatio= 6939;
void main(void) {
  /*** Processor Expert internal initialization. DON'T REMOVE THIS
CODE! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.    ***/

  /*Write your code here*/
  for(;;) {
    pwmChannel[0]= PTP_PTP0;
    void)PWM8_SetRatio16(pwmRatio);
  }

  /*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!
***/
    for(;;);
  /*** Processor Expert end of main routine. DON'T WRITE CODE BELOW!
***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT! ***/
```

Run the Application

The application is now finished and we can launch it. Make sure you have chosen the FCS connection.

1. Select the **Debug** icon in the Project Manager window (or the menu bar **Project > Debug** or **[F5]**).
2. Select **Component > Open** in the debugger and open the **VisualizationTool** component.

VisualizationTool Properties

In the following paragraphs we create a visualization for our project. Make all changes in the **VisualizationTool** window. Make sure that you are in the **Edit mode** (switch with **Right mouse click** > **Edit Mode** or **[Ctrl-E]**).

1. **Right mouse click** > **Properties**
2. For **Refresh Mode** select **CPU Cycles**
3. For **Cycle Refresh Count** select **10000**

Chart Properties

Now add a chart, in which we can see the changing value of the channel in a graphic.

1. **Right mouse click > Add New Instrument > Chart**
2. **Double click** on the Chart to see the Chart Properties
3. Select **Expression** for **Kind of Port**
4. Select **pwmChannel[0]** for **Port to Display**
5. Select **2** for **High Display Value**
6. Select **Target Periodical** for **Type of Unit**
7. Select **1000** for **Unit Size**
8. Select **1000** for **Numbers of Units**

Leave all others on default.

Period Bar Properties

With the bar we can change the period value of the PWM channel 0.

1. **Right mouse click** > **Add New Instrument** > **Bar**
2. **Double click** on the Bar to see the Bar Properties.
3. Select **Variable** for **Kind of Port**

4. Select **_PWMPER01.Overlap_STR.PWMPER0STR.Byte** for **Port to Display**

Leave all others on default.

You might add labels with **Right mouse click > Add New Instrument > Static Text**.

Duty Time Bar Properties

Now add a bar to change the duty time.

1. **Right mouse click > Add New Instrument > Bar**
2. **Double click** on the Bar to see the Bar Properties.
3. Select **Variable** for **Kind of Port**
4. Select **pwmRatio** for **Port to Display**
5. Select **65535** for **High Display Value**

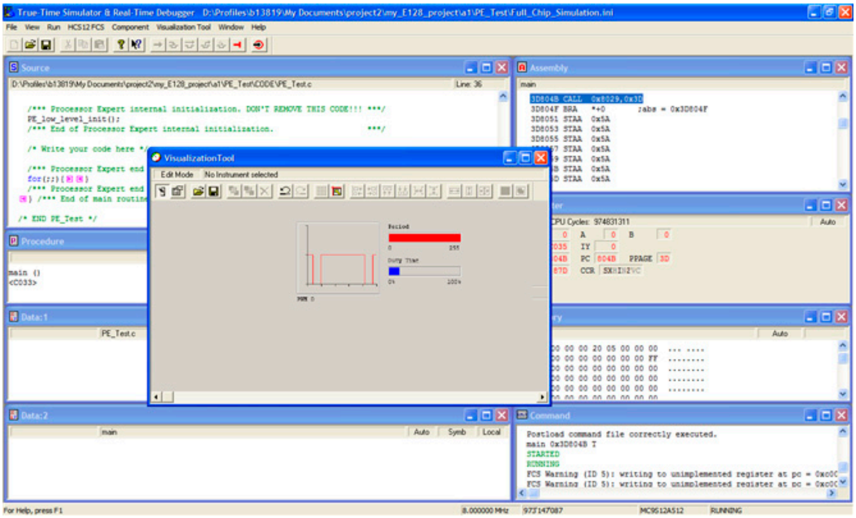
Leave all others on default.

Run the Application

Now let's leave the Edit mode and run the final application. First we save the window layout.

1. **Right mouse click > Edit Mode** (or **[Ctrl-E]**)
2. Select the **Save** icon in debugger (or the menu bar **File > Save Configuration**) to save the window layout.
3. Select the **Debug** icon in debugger (or the menu bar **Run > Start/Continue** or **[F5]**).

Figure 10.30 Debugger Main WIndow - Final Application



11

P&E Multilink/Cyclone Pro Connection

This section contains information to assist you with the *P&E Multilink/Cyclone Pro* connection.

P&E Multilink/Cyclone Pro Technical Considerations

P&E Microcomputer Systems supplies many of the debug cables that can be used to connect the 8/16-bit debugger (and the CodeWarrior IDE) to the HCS12 hardware. When the debugger runs the **P&E Multilink/Cyclone Pro** connection, it communicates and debugs **CPU12 (HC12), HCS12, HCS12X** and **XGATE** core-based hardware connected through the P&E USB BDM Multilink, P&E Cyclone Pro (via USB, Serial and TCP/IP ports).

NOTE All recent P&E hardware interfaces, such as the P&E USB BDM Multilink and Cyclone Pro, are fast enough to debug devices with a bus speed of up to 40Mhz.

To use the P&E USB BDM Multilink, the drivers from P&E must be installed on the host computer.

NOTE The P&E Multilink/Cyclone Pro connection also works with Freescale development boards with an integrated debug circuit based on Open Source BDM.

Connection Menu

Setting the ***P&E Multilink/Cyclone Pro*** connection changes the connection menu entry in the debugger main toolbar to ***HC12MultilinkCyclonePro***.

HC12MultilinkCyclonePro Menu Options

Figure 11.1 shows MultilinkCyclonePro menu options and Table 11.1 describes menu entry use.

Figure 11.1 MultilinkCyclonePro Menu Options

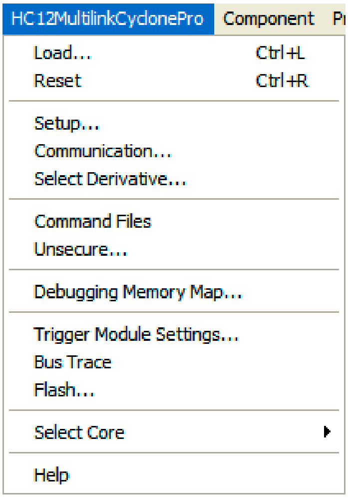


Table 11.1 MultilinkCyclonePro Menu Description

Menu Entry	Description
Load	Displays Load Executable File Dialog Box (see Load Executable File Window).
Reset	Resets project hardware and software.
Setup...	Displays P&E Multilink/Cyclone Pro Setup dialog box (see Setup Menu Option).
Communication ...	Displays P&E HC(S)12 Connection Manager dialog box (see Communication/Connect Menu Option).
Set Derivative	Displays Set Derivative dialog box, which allows you to choose target MCU for P&E Multilink/Cyclone Pro connection (see Figure 11.2).

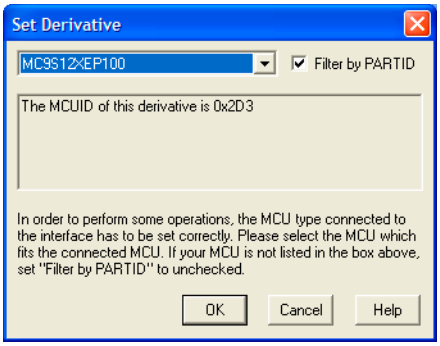
Table 11.1 MultilinkCyclonePro Menu Description (*continued*)

Menu Entry	Description

Command Files	Displays Connection Command Files window. Each tab allows access to a different set of connection command files:
	`.\cmd\P&E_Multilink_CyclonePro_startup.cmd`
	`.\cmd\P&E_Multilink_CyclonePro_reset.cmd`
	`.\cmd\P&E_Multilink_CyclonePro_preload.cmd`
	`.\cmd\P&E_Multilink_CyclonePro_postload.cmd`
	`.\cmd\P&E_Multilink_CyclonePro_vppon.cmd`
	`.\cmd\P&E_Multilink_CyclonePro_vppoff.cmd`
	`.\cmd\P&E_Multilink_CyclonePro_erase_unsecure_hcs12.cmd`

Unsecure	Runs project's Unsecure command file script. This script mass erases target device, then programs target device security byte to an unsecure state to enable BDM communication.
Debugging Memory Map	Displays Debugging Memory Map dialog box for target MCU (see Debugging Memory Map).
Trigger Module Settings	**D**isplays Trigger Module Settings dialog box (see On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms).
Bus Trace	Displays a Trace component window (see On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms).
Flash	Displays Non-Volatile Memory Control dialog box (see Flash Programming).
Select Core	Use to synchronize debugger windows to a specific core when several cores are available. Appears only when several cores are available for debugging.
Help	Displays P&E Multilink/Cyclone Pro User Manual section.

Figure 11.2 Set Derivative Dialog Box



Setup Menu Option

The Communication/Connect menu option displays the P&E Multilink/Cyclone Pro Setup dialog box. This dialog box contains the Communication tab and the Debug Options tab.

Figure 11.3 Communication Tab

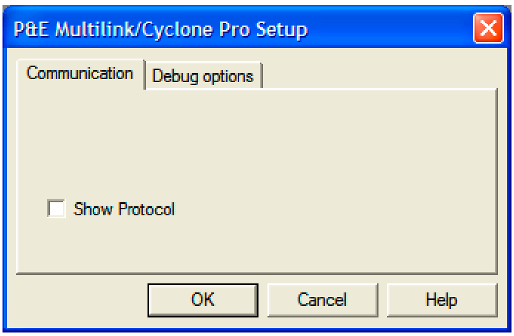


Figure 11.4 Debug Options tab

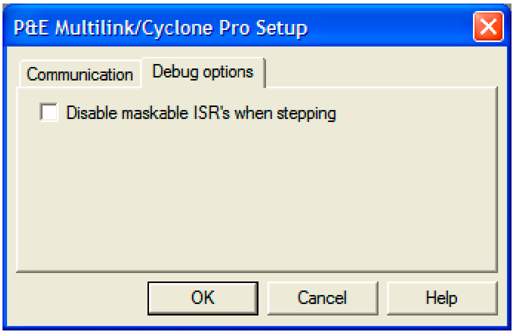


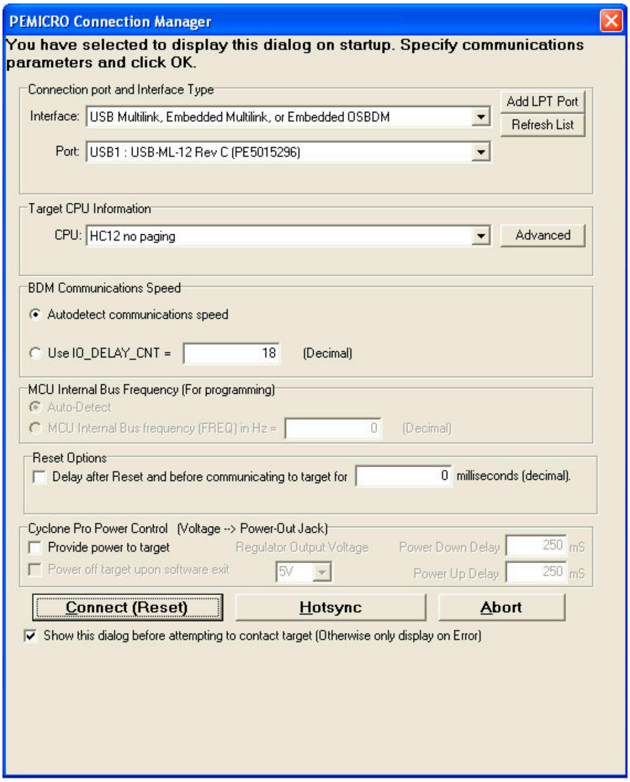
Table 11.2 P&E Multilink/Cyclone Pro Setup Description

Checkbox	Description
Show Protocol	Checked: enables a debug protocol displayed in the Command window. Enable this option only when requested by the Freescale Support team to resolve debugger issues.
Disable maskable ISRs when stepping	Checked: Automatically sets `I` bit in device core CCR and disables maskable interrupts. Debugger automatically calculates final `I` flag value according to stepped instruction.

Communication/Connect Menu Option

The Communication/Connect menu option displays the P&E HC(S)12 Connection Manager dialog box.

Figure 11.5 P&E HC(S)12 Connection Manager dialog box



NOTE The Connection Manager dialog box will only show after attempting to automatically contact to the target using the default connection settings. Connection settings can be changed during Hi-wave by going to MultilinkCyclonePro -> Communications, however, the debug session must be restarted to see the changes.

Figure 11.6 P&E HC(S)12 Connection Manager Advanced Options dialog box

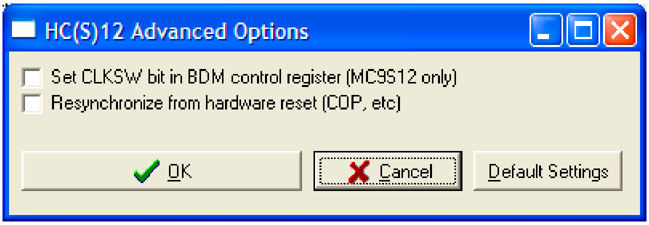


Table 11.3 P&E HC(S)12 Connection manager Description

Element	Description
Interface listbox	Selects the P&E hardware interface that is used to connect to the target.
Port listbox	After the interface is chosen, this specifies which port should be used. The available selections are dependent on the selected interface.
Add LPT Port button	This button guides the user through the series of steps needed to add a LPT port to the Connection Manager. This is necessary if additional LPT ports are added through the use of expansion cards, such as PCI.
Refresh List button	This button attempts to detect all P&E hardware devices specified by the Interface selection. This is necessary if these devices are reset or need to be plugged in after the Connection Manager has been shown.
Advanced button	Opens HC(S)12 Advanced Options dialog box which allows the user to modify the following settings: • set CLKSW=0 bit in BDM control register; • resynchronize from hardware reset.
Set CLKSW bit in BDM control register (MC9S12 only) checkbox in P&E HC(S)12 Connection Manager Advanced Options dialog box	Checked: uses BDM status register CLKSW bit to set bus clock. May be required for first BDM module revisions when: • using BDM constant clock source (CLKSW=0), • with PLL engaged (PLLSEL=1), and • PLL multiplier greater than or equal to 2 Under these conditions, set CLKSW=1 before engaging PLL or BDM may lose communication with host system. Default: Cleared; BDM clock runs at EXTAL/2.

Element	Description
Resynchronize from hardware reset (COP, etc.) checkbox in P&E HC(S)12 Connection Manager Advanced Options dialog box	Checked: Forces debugger to review and resynchronize BDM communication and control, if BDM communication is still active. If BDM mode is disabled, debugger inquires whether an internal core reset has occurred, and if so, reactivates BDM mode for debugging purposes. Rebuilds the S12X-series program flow. Default: Clear; option disabled.
BDM Communications Speed	The BDM Communication Speed defines the frequency at which the P&E hardware interface communicates with the target microprocessor. • **Autodetect communications speed** This is the recommended setting for the first time that a connection is made to the target microprocessor. The appropriate BDM communication speed is automatically calculated. • **Use IO_DELAY_CNT** Once a successful connection is made using the “Autodetect communications speed” setting, the IO_DELAY_CNT (which stores the appropriate information about the BDM communication speed) is saved into this field for all future sessions. This setting reduces the amount of time needed to connect to the microprocessor, since the correct BDM communication speed is known. • **OSBDM connection** OSBDM automatically detects communication speed with the target, thus IO_DELAY_CNT settings are disabled for this hardware interface
MCU Internal Bus Frequency (For programming)	These settings are currently unused.

Element	Description
Reset Options	• **Delay after Reset** The target microprocessor is reset during the initial startup sequence in order to establish communications. In some cases, the rise delay of the Reset signal is considerable. This setting allows the user to specify an amount of time to wait for the Reset signal to rise to logic high levels before trying to communicate with the target microprocessor.
Cyclone Pro Power Control	The P&E Cyclone Pro, if used, can directly provide power to the target microprocessor, eliminating the need for a separate external power supply. Please note that the appropriate jumper settings must first be configured on the Cyclone Pro. Refer to the user manual for more details. • **Provide power to target** This setting must be checked if the Cyclone Pro should provide power to the target microprocessor • **Power off target upon software exit** If checked, the Cyclone Pro will stop providing power to the target microprocessor when the software is closed. • **Regulator Output Voltage** Specifies the voltage generated by the Cyclone Pro internal regulator. The available options are 2V, 3V, and 5V. • **Power Down Delay** Specifies the power down delay in milliseconds • **Power Up Delay** Specifies the power up delay in milliseconds
Connect (Reset) button	Connects to the target microprocessor by resetting the device and entering background mode.

Element	Description
Hotsync button	Connects to the target microprocessor without resetting the device. If the device is running, it will still be running after this connection is made. Hotsync command will not work when a Cyclone Pro is used to power up the target. A Cyclone Pro will always attempt to power cycle the device upon connection.
Abort button	Cancels the connection.

TIP **When debugging with several cables and debuggers:** The debugger registers the serial numbers of USB cables automatically registered in each project when you press Connect in the Connection Manager dialog. The next debug session opens the cable matching the registered serial number. This feature is not available with Serial, LPT/parallel and TCP/IP connections.

12

OSBDM Connection

This section contains information to assist with use of the Open Source BDM (OSBDM) hardware with Codewarrior.

OSBDM Technical Considerations

Freescale supplies certain development boards with an integrated debug circuit based on Open Source BDM. This allows the development board to be debugged from the PC via the USB bus without requiring external debug hardware, such as the Cyclone Pro or USB Multilink. The development board also derives its power from the USB Bus.

The Open Source BDM circuit design (OSBDM-JM60) is an open source, community-driven design. It has been published on Freescale's website, and full documentation can be found in the Community Forums.

CodeWarrior Integration

Integration with CodeWarrior is handled via the "P&E Multilink/Cyclone Pro" connection. P&E has integrated the Open Source BDM support into the same connection that supports both the USB Multilink and the Cyclone Pro. Refer to the P&E Multilink/Cyclone Pro Connection chapter of this manual for more details on this connection. All of the dialogs which affect operation of these hardware interfaces function in the same manner when using OSBDM (albeit at a lower data rate).

Minimum Firmware Version

The integration with Codewarrior via the "P&E Multilink/Cyclone Pro" will require a minimum OSBDM firmware version of at least Build 27. If the OSBDM hardware doesn't have a sufficient firmware version, the software will pop up an error dialog notifying the user that the firmware needs to be upgraded. Future revisions of Codewarrior will update the firmware automatically.

For this release, if you receive this error, go to http://www.pemicro.com/osbdm and download the OSBDM Firmware Update Utility.

Support and Licensing

Open Source BDM is not supported by Freescale; it is open source. Any bugs, enhancements, or support questions should be addressed through the Open Source BDM forum. Open Source BDM has been thoroughly tested, but there is no guarantee of error-free operation. All the source files are available to anyone under the GNU LESSER GENERAL PUBLIC LICENSE. For more information on this license, refer to the COPYING_LGPL.txt file located in the root directory of the OSBDM-JM60 package, which may be downloaded from the Freescale website. Open Source BDM is a derivative project of the TBDML project.

13

SofTec HCS12 Connection

This section contains information to assist you with the *SofTec HCS12* connection.

SofTec HCS12 Technical Considerations

SMH Technologies (earlier known as SofTec Microsystems) supplies the SofTec HCS12 in-circuit debugger/programmer units which can be used with the 8/16-bit debugger (and the CodeWarrior IDE) to work with the HCS12 hardware. When the debugger uses the **SofTec HCS12** connection, it communicates and debugs **HCS12, HCS12X** and **XGATE** core-based hardware through the SofTec in-circuit debugger/programmer units:

- SMH Technologies HCS12 ISP Debuggers/Programmers (inDART Series)
- Starter Kits (AK/SK/PK/ZK and newer Series).

Refer to the *inDART®-HCS12 In-Circuit Debugger/Programmer for Freescale HCS12 Family Flash Devices User's Manual* from SMH Technologies for communication hardware requirements and SMH Technologies product installation.

Connection Menu

Setting the ***SofTec HCS12*** connection changes the connection menu entry in the debugger main toolbar to ***inDART-HCS12***.

inDART-HCS12 Menu Entries

Figure 13.1 shows the inDART-HCS12 menu entries and Table 13.1 describes their use.

Figure 13.1 inDART-HCS12 Menu Entries

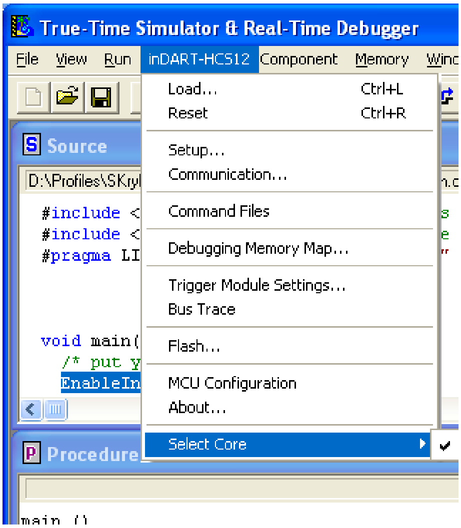


Table 13.1 inDART-HCS12 Menu Entry Descriptions

Menu Entry	Description
Load	Displays Load Executable File dialog box (see Load Executable File Window).
Reset	Resets project hardware and software.
Setup	Displays SofTec HCS12 Setup dialog box.
Communication	Displays Communication Settings dialog box (Figure 13.3) allows you to fine-tune critical parameters by choosing BDM clock source: either system bus frequency or alternate frequency dependent on target. Access from MCU Configuration Settings dialog box also.

Table 13.1 inDART-HCS12 Menu Entry Descriptions (*continued*)

Menu Entry	Description

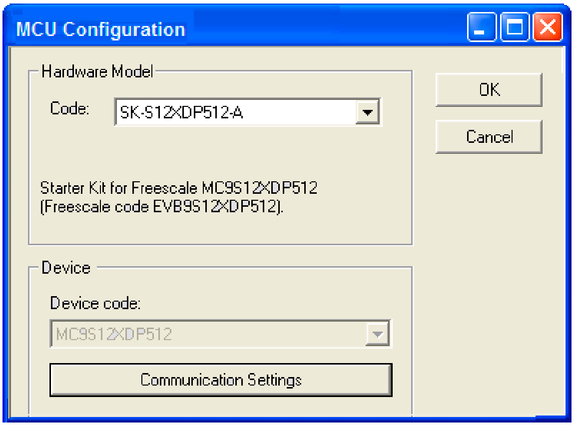
Command Files	Displays Connection Command Files window. Each tab allows access to a different set of connection command files:
	`\cmd\SofTec_HCS12_startup.cmd`
	`\cmd\SofTec_HCS12_reset.cmd`
	`\cmd\SofTec_HCS12_preload.cmd`
	`\cmd\SofTec_HCS12_postload.cmd`
	`\cmd\SofTec_HCS12_vppon.cmd`
	`\cmd\SofTec_HCS12_vppoff.cmd`

Debugging Memory Map	Displays Debugging Memory Map Dialog Box for target MCU (see Debugging Memory Map).
Trigger Module Settings	Displays Trigger Module Settings dialog box (see On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms).
Bus Trace	Displays a Trace component window (see On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms).
Flash	Displays Non-Volatile Memory Control dialog box (see Flash Programming).
MCU Configuration	Displays MCU Configuration dialog box (Figure 13.2). See MCU Configuration Dialog Box.
About	Displays About dialog box, which provides information about ***inDART_HCS12.dll*** release and version.
Select Core	Use to synchronize debugger windows with a specific core. Appears only when several cores are available for debugging.

MCU Configuration Dialog Box

Figure 13.2 shows the MCU Configuration dialog box.

Figure 13.2 MCU Configuration Dialog Box



Use the Hardware Model list menu to select another type of debug interface.

Use the Device Code list menu to select another HCS12 derivative.

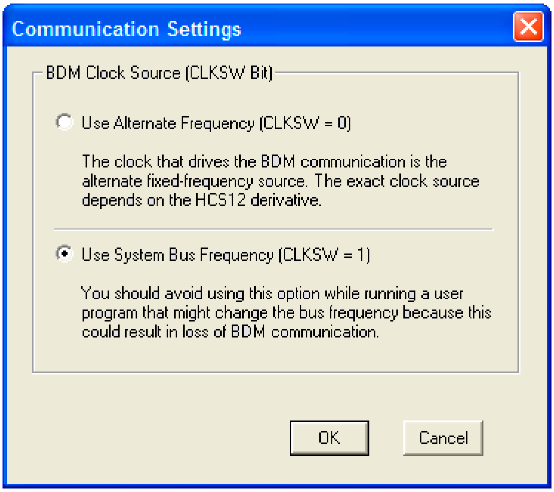
If your hardware supports stopping the application while running, the IRQ vector requires an additional interrupt service routine.

See the *inDART®-HCS12 In-Circuit Debugger/Programmer for Freescale HC12 Family Flash Devices User's Manual* from SofTec for more details

Communication Settings Dialog Box

Clicking the Communication Settings button in the MCU Configuration dialog box displays the Communication Settings dialog box (Figure 13.3). Using this dialog box, you can fine-tune critical parameters for proper operation with the target microcontroller by choosing the BDM clock source: either the system bus frequency or an alternate frequency dependent on the target.

Figure 13.3 Communication Settings Dialog Box



14

HCS12 Serial Monitor Connection

This section provides information about debugging with the CodeWarrior IDE and the *HCS12 Serial Monitor* connection.

Serial Monitor Technical Considerations

When the debugger runs the HCS12 Serial Monitor connection, it can communicate and debug hardware running the HCS12 Serial Monitor in full compliance with the Freescale Application Note **AN2548** specifications, and AN2548SW1 and AN2548SW2 software. Refer to this Application Note for communication hardware requirements.

CodeWarrior IDE and Serial Monitor Connection

You can access the HCS12 Serial Monitor connection in two different ways when using Codewarrior IDE. You either:

- Use the Stationary Wizard at the start of the project to set the connection, or
- Set the connection from within an existing project.

These paths are explained in the Debugger Interface chapter of this manual.

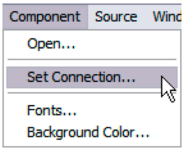
HCS12 Serial Monitor Interface

Follow these steps to use the HCS12 Serial Monitor connection:

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

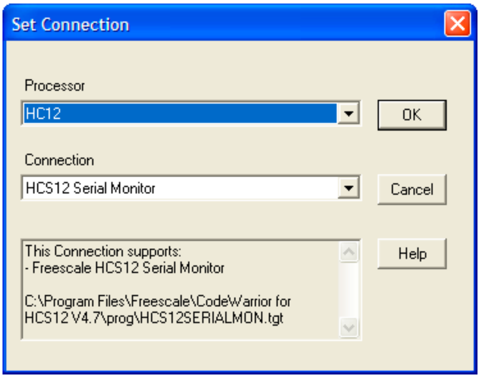
   Debugger Main window opens.

Figure 14.1 Debugger Main Window - Component Menu



4. In the debugger main window, choose **Component > Set Connection** to select another connection.

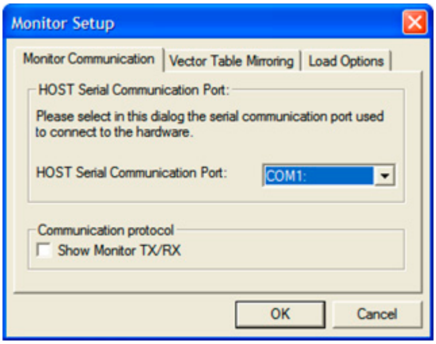
Figure 14.2 Set Connection Dialog Box - HCS12 Serial Monitor Selection



5. Select ***HC12*** as Processor, then ***HCS12 Serial Monitor*** as connection in the Set Connection dialog box.
6. Click the **OK** button.

   The Monitor Setup dialog box appears.
7. In the Monitor Setup dialog box Monitor Communication tab, choose the correct Host serial communication port if necessary.

Figure 14.3 Monitor Setup Window - Monitor Communication Tab



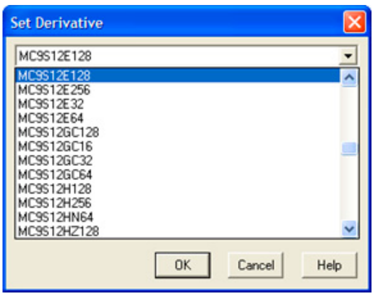
8. Click the OK button.

   The HCS12 Serial Monitor connection reads the device silicon ID and opens the Derivative Selection dialog box. The device silicon ID can match several derivatives.

9. Select the derivative that matches your hardware in the Derivative Selection dialog box.

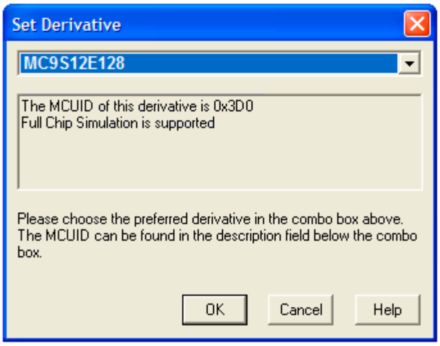
   If debugger is aware about PARTID returned by the connected derivative then you can filter derivative list by PARTID.

Figure 14.4 Set Derivative Dialog Box



If debugger has no information about PARTID returned by the connected derivative then CPU specific derivative list is used.

Figure 14.5 Set Directive Dialog Box



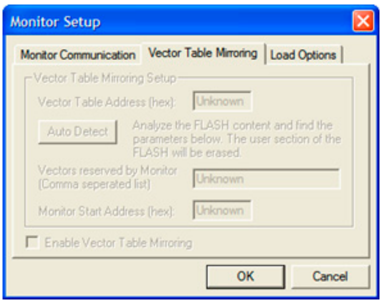
10. Click the OK button.

    The **Monitor Setup** dialog box opens again.

11. Click on the Vector Table Mirroring tab.

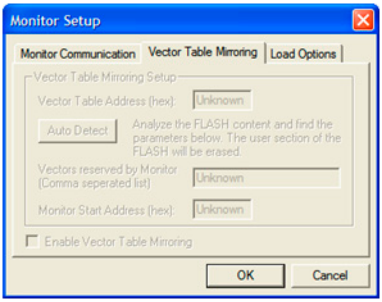
NOTE We recommend that you use the Vector Table Mirroring feature. Otherwise, you cannot program vectors as captured or protect them from erasure or overwriting by the HCS12 Serial Monitor.

Figure 14.6 Monitor Setup Dialog Box - Vector Table Mirroring Tab



12. To enable this feature, check the **Enable Vector Table Mirroring** checkbox.

Figure 14.7 Monitor Setup Dialog Box - Vector Table Mirroring Tab



13. Click **Auto Detect** to make the debugger search for the vector table address and vectors reserved by the HCS12 Serial Monitor.
14. Once automatic detection succeeds, click **OK** to start debugging.

MONITOR-HCS12 Menu Options

Once you set the ***HCS12 Serial Monitor*** connection, *MONITOR-HCS12* appears in the Debugger menu, as shown below.

Figure 14.8 MONITOR-HCS12 Menu Options

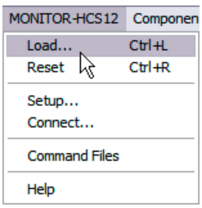


Table 14.1 MONITOR-HCS12 Menu Options

Menu Entry	Description
Load	Displays Load Executable File dialog box (see Load Executable File Window).
Reset	Resets connection hardware and software.
Setup	Displays Setup dialog box.

Table 14.1 MONITOR-HCS12 Menu Options (*continued*)

Menu Entry	Description
Connect	Displays Monitor Setup dialog box, containing Monitor Communication tab and Load Options tab. • In Monitor Communication tab (Figure 14.9), set or modify current serial communication from *HOST Serial Communication Port* list box. Check **Show Monitor TX/RX** to report all low-level communication frames between host computer and HCS12 Serial Monitor in Command Line window. • In Load Options tab (Figure 14.10), use checkbox to enable or disable automatic erase of Flash memory on loading.

Command Files	Opens Connection Command Files dialog box. Each tab allows access to a different set of connection command files:
	`\cmd\HCS12_Serial_Monitor_startup.cmd`
	`\cmd\HCS12_Serial_Monitor_reset.cmd`
	`\cmd\HCS12_Serial_Monitor_preload.cmd`
	`\cmd\HCS12_Serial_Monitor_postload.cmd`

Debugging Memory Map	Opens Debugging Memory Map dialog box for target MCU (see Debugging Memory Map).
Trigger Module Settings	Opens Trigger Module Settings dialog (see On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms).
Bus Trace	Opens Trace component window within the main window (see On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms).

Figure 14.9 Monitor Setup Dialog Box - Monitor Communication Tab

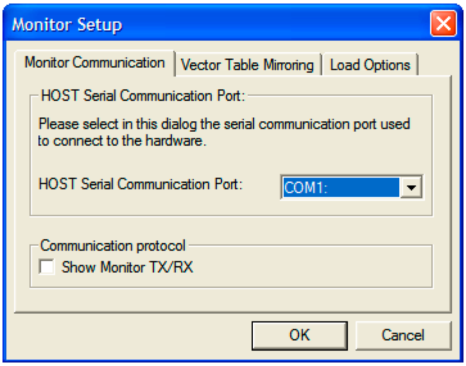
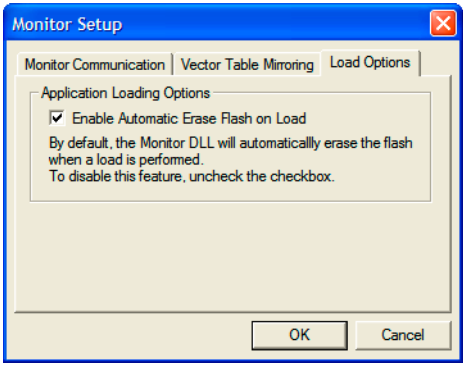


Figure 14.10 Monitor Setup Dialog Box - Load Options Tab



15

Abatron BDI Connection

This section guides you through the *Abatron BDI* connection.

Abatron BDI Technical Considerations

The **ABATRON AG** company supplies many of the debug cables used to connect the 8/16-bit debugger (and the CodeWarrior IDE) to HCS12 hardware.

When the debugger runs the **Abatron BDI** connection, it can communicate and debug **CPU12 (HC12), HCS12, HCS12X and XGATE** core-based hardware connected through Abatron **BDI**, **BDI-S**, **BDI-HS** and **BDI1000** debug interfaces.

Abatron BDI Highlights

The *Abatron BDI* Connection currently supports the Background Debug Interfaces (BDI) designed by **ABATRON AG**, including the **BDI-HS** on CPU12 and HCS12, and the **BDI1000** on the CPU12, HCS12 and HCS12X cores.

Abatron BDI Requirements

Ensure that your hardware target board incorporates a Background Debug Mode (BDM) port for CPU background interfacing with the BDI interface and the debugger. Check the technical specifications provided by the ABATRON User Manuals and Freescale.

Communicating with the BDI interface requires one free serial communication port of your computer. You may need to set it up even if you use Ethernet communication instead of an RS-232 serial communication.

Abatron BDI Connection Introduction

Use the *Abatron BDI* Connection to load different connections which implement the interface with target systems. This section describes the specific features of the Abatron BDI Connection.

With this interface, you can download an executable program from the debugger environment to an external target system based on a Freescale MCU for execution. You also have the feedback of the real target system behavior to the debugger.

The debugger supervises and monitors the target system MCU (i.e. controls program execution). You can read and write in internal/external memory (even while the application is running), single-step/run/stop the application, and set breakpoints and watchpoints in the code.

Interfacing Abatron BDI and Your System

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

The BDI interface connects to the host computer either by a serial communication link or by an Ethernet connection. You can use any available communication port of your host system. The communication protocol between the BDI and your target system is handled by the BDI Target driver, which loads automatically with the Abatron BDI Target Component. You can adapt your target system to the BDI interface.

The BDI-to-target system communication uses a single-wire serial connection. You must equip the target system with a BDM connector/port (see the *BDI User Manual* from ABATRON).

- Make sure that your hardware target board has a Background Debug Mode (BDM) port for background interfacing with the BDI interface and the debugger. Check the technical specifications provided by ABATRON User Manuals and Freescale.
- Make sure your computer has one free serial communication port to communicate with the BDI interface. You may need to set it up even if you use Ethernet communication instead of an RS-232 serial communication medium.

BDI Interface Software Setup

The Abatron BDI connection is delivered during installation and contains all required files, demo projects to use the Abatron BDI debugger, and some BDI setup (init list) files. Install the drivers delivered with the BDI interface to make sure you are using the latest drivers. These files are delivered on a disk from ABATRON.

Set up the target MCU through the BDI interface, according to your hardware configuration. Copy all files from the ABATRON disk to a new directory on your computer.

ABATRON provides an `.EXE` configuration application and a set of configuration files for specific evaluation boards and processors. These files contain microprocessor/microcontroller initialization data, vectors, chip selects for internal/external ROMs/RAMs, running modes, etc. They also contain information bound to the MCU and MCU version used, and information bound to the MCU environment on the board (RAM, ROM, PIA, ACIA, etc.). Each of these files is very specific.

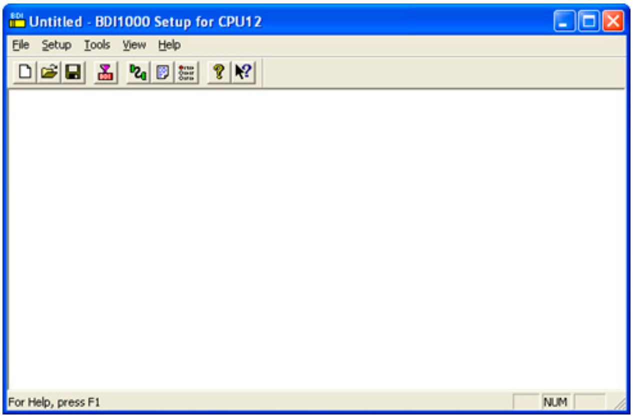
Running the ABATRON Configuration Tool

You can run the configuration program (e.g. `B10C12.EXE` for CPU12 processor with BDI1000, or `BDIHSHCI.EXE` for CPU16/CPU32 with BDI-HS) within the debugger is you browse for it using **Abatron BDI > Configure BDI Box** or specify the tool path in the **Abatron BDI > Setup** dialog box. Otherwise, run the configuration tool directly from the File Manager or Windows Explorer.

Example with B10C12.EXE Configuration Tool

NOTE Refer to the *ABATRON User Manual* for further details about the BDI interface and BDI setup.

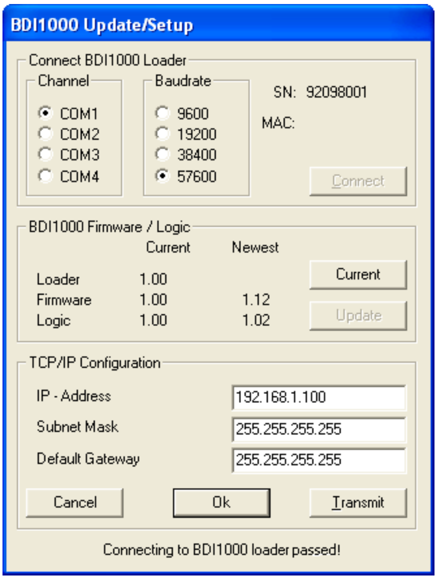
Figure 15.1 BDI1000 Setup for CPU12 Dialog Box



Firmware Loading

In the dialog box shown in Figure 15.1, select **Setup > Firmware** to open the firmware dialog.

Figure 15.2 BDI Update/Setup Dialog Box



In the Update/Setup dialog box, set the communication port and the baud rate according to your installation and click the *Connect* button. If the connection passes, the current BDI firmware/logic appears. If the dialog box shows that the **Current** firmware or logic is **unknown**, load new firmware by clicking the *Update* button.

To use Ethernet communication between your computer and the BDI interface, set the IP address reserved for the BDI, then click the *Transmit* button. Quit the dialog by clicking the *Ok* button.

Initialization List (Startup Init List) Loading

Select **File > Open** to load a configuration file (e.g. `HC912DA128.BDI`).

Figure 15.3 File Menu

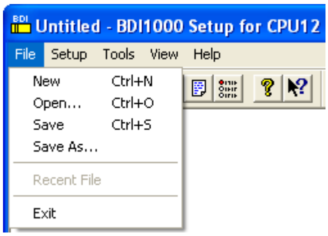
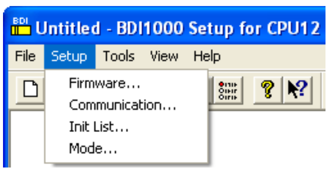


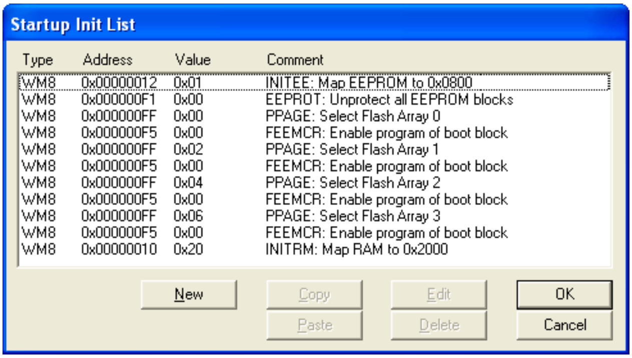
Figure 15.4 Setup Menu



Select **Setup > Init List** to see and edit (if necessary) the content of this configuration file.

This displays the Startup Init List/configuration file in the Startup Init List dialog box. You can edit, add, or remove memory write instructions in this dialog box to configure your MCU and MCU environment.

Figure 15.5 Startup Init List Dialog Box

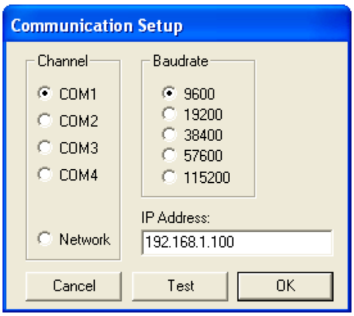


Quit this dialog box by clicking **OK** and save the settings if necessary.

Communication with the Debugger Setup

Select **Setup > Communication** to open the Communication Setup dialog box.

Figure 15.6 Communication Setup Dialog Box

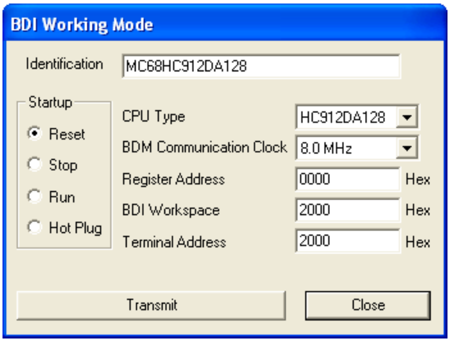


In this dialog box, set the communication for using the BDI with the debugger. Make sure these settings are identical to debugger communication settings made in the Communication Setup dialog box (see Figure 15.6). Click the Test button to check the setup, then click *OK* to quit this dialog. Save the settings if necessary.

BDI Working Mode and Setup/List Transmission

Select **Setup > Mode** to open the BCI Working Mode dialog box (Figure 15.7). Set the required parameters and click *Transmit* to download the configuration to the target board.

Figure 15.7 BDI Working Mode Dialog Box



Loading the Abatron BDI Connection

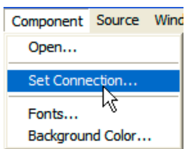
Use the `Target=Abatron BDI` in the `[Environment Variables]` section of the project file to set the target.

The *Abatron BDI* Connection automatically detects whether the target is connected to your system. If no target is detected (the target is not connected or is connected to a different port), the Communication Device Specification Dialog Box appears.

If no target or a different target is set, load the *Abatron BDI* Connection as described below.

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

Figure 15.8 Debugger Component Menu

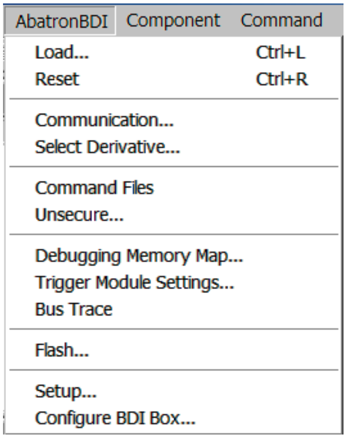


The Set Connection dialog box appears. Select *Abatron BDI Connection* in the list of connections and click **OK**.

Figure 15.9 Set Connection Dialog Box

After successfully loading the target, the Debugger main window target menu item is replaced by *Abatron BDI*.

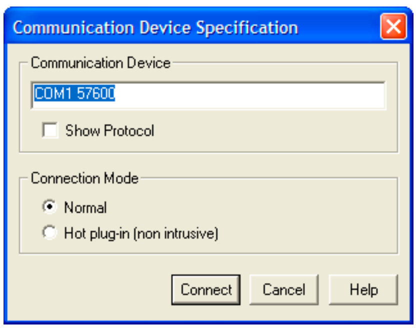
Figure 15.10 Abatron BDI Menu



You can change the communication parameters (baud rate and port) by selecting **Abatron BDI > Communication**.

If you cannot establish communication with the BDI Interface, an error message appears, followed by the Communication Device Specification dialog box.

Figure 15.11 Communication Device Specification Dialog Box

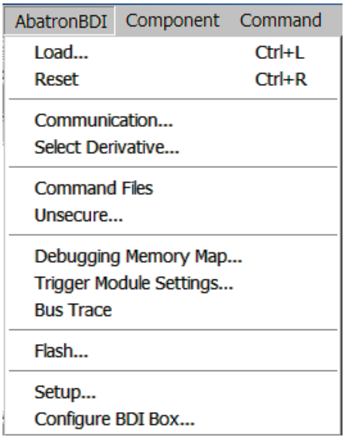


In this dialog box, you can modify the device specification (e.g. Communication Port and baud rate). These settings are saved in the current project and are used again in future sessions.

Abatron BDI Connection Menu Entries

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

Figure 15.12 Debugger Abatron BDI Menu



If the target connection fails, **Connect** replaces the entry **Communication** in the *Abatron BDI* menu.

Table 15.1 describes the *Abatron BDI* menu entries:

Table 15.1 Abatron BDI Menu Entries

Menu Entry	Description
Load	Loads the application to debug, i.e. an `.ABS` file. See Load Executable File Window.
Reset	Executes the Reset Command File and resets the hardware target. The BDI interface automatically processes the initialization list (startup init list) stored in the interface.
Communication (or Connect)	Displays the Communication Device Specification Dialog Box. If the connection to the target has failed, *Connect* appears instead of *Communication*.
Select Derivative	Displays the Set Derivative dialog box, and allows you to choose the target MCU for the Abatron BDI connection.
Command Files	Displays the Target Interface Command Files Window.

Table 15.1 Abatron BDI Menu Entries (*continued*)

Menu Entry	Description
Unsecure Option	Runs the project's Unsecure command file script. The Unsecure script mass erases the target device, then programs the target device security byte to "unsecure" state to enable BDM communication. Available only when debugger Flash Programmer is enabled. See Setup Dialog Box.
Debugging Memory Map	Displays Debugging Memory Map dialog box for target MCU. See Debugging Memory Map.
Trigger Module Settings	Displays Trigger Module Settings dialog box. See On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms.
Bus Trace Option	Displays a Trace component window. See On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms.
Flash	Displays Non-Volatile Memory Control Dialog Box. See Flash Programming. Available only when Flash Programmer is enabled. See Setup Dialog Box.
Setup	Opens the Setup Dialog Box. Use to set the link to the ABATRON configuration tool, set the download mode, and set the *Continue on illegal break (banked hardware breakpoint)* option (only available for HC12/CPU12 derivatives).
Configure BDI Box	Opens the configuration tool. If no application tool path is currently set (see Setup Dialog Box), the Select BDI Box Configuration Tool dialog box opens to create a link to the configuration tool application. Save the link in the Setup dialog box.
Select Core	Use to synchronize the debugger windows with a specific core when several cores are available. Available only when several cores can be debugged.
Help	Opens the *Abatron BDI* Connection Help File.

Abatron BDI Connection Dialog Boxes

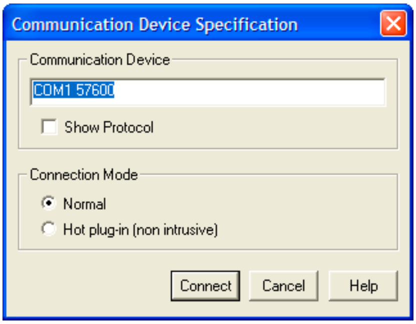
This section describes the dialog boxes specific to the *Abatron BDI* Connection. Those dialogs are:

- Communication Device Specification Dialog Box
- Setup Dialog Box

Communication Device Specification Dialog Box

The Communication Device Specification dialog box appears automatically if the *Abatron BDI* Connection fails to establish the communication with the BDI interface. You can also open this dialog box using **Abatron BDI > Communication** or **Abatron BDI > Connect**.

Figure 15.13 Communication Device Specification Edit Box



If you open the dialog using **Abatron BDI > Communication**, but the connection attempt was successful, you cannot modify the **Communication Device** edit box. Only the **Show Protocol** check box can be modified.

If the connection to the BDI box failed, and the dialog box opens automatically (or you use **Abatron BDI > Connect**), you can modify the **Communication Device Specification** edit box.

The Communication Device specification edit box contains the communication settings to connect to the BDI box. The syntax of the initialization string is:

- **`COM`***`n baudrate`*
  - `n` is the COM port number, such as 1, 2, or 3
  - `baudrate` is 9600, 19200, 38400, 57600, or 115200, according to the ABATRON configuration application setup (e.g. **`COM1 57600)`**.

For communication via an Ethernet and bdiNet, use the following initialization string syntax:

- **`NETWORK`** *`ip_address port`*
  - `ip_address` is the IP address of the BDI box or bdiNet in the form xxx.xxx.xxx.xxx
  - `port` is the bdiNet port, usually "1" for BDI1000 and BDI2000 (e.g., `NETWORK 151.120.25.101`)

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

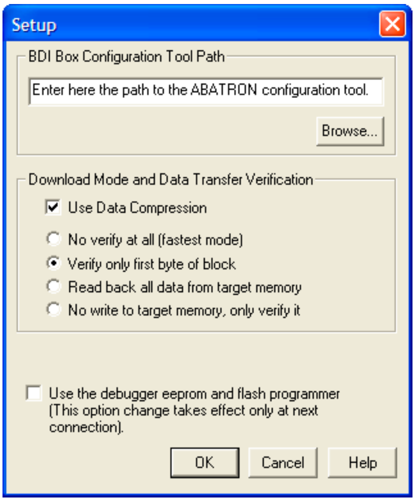
NOTE The Show Protocol checkbox is a useful debugging feature if a communication problem exists. The settings in the Communication Device Specification edit box are stored for a later debugging session in the `[Abatron BDI]` section of the project file.

In Connection Mode section you can select **Normal** or **Hot plug-in** (non intrusive) mode. In Hot plug-in mode debugger does not reset the target if target is running.

Setup Dialog Box

Open the Setup dialog box using **Abatron BDI > Setup**.

Figure 15.14 Setup Dialog Box



The BDI Box Configuration Tool Path edit box is set up with the path and application name of the configuration tool from ABATRON. Select **Abatron BDI > Configure BDI Box** to automatically browse for the application tool. Otherwise, click the **Browse** button to look for the tool. An example of the edit box contents is `C:\tmp\B10c12.exe`.

In the *Download Mode and Data Transfer Verification* section, you can set different options to transfer data from the computer to the BDI box. The default setting is *Verify only first byte of block*. Choose a different option to improve transfer speed or security. By default, data compression is enabled for asynchronous communication channels. With older computers, download speed may be faster without data compression.

HC12/CPU12 derivatives only: Use the *Continue on illegal break (banked hardware breakpoint)* check box to overcome the 2-byte address size on-chip break module, which does not handle PPAGE. Internally, the target halts at the hardware breakpoint (in Flash memory), compares that breakpoint with the breakpoint you set, then relaunches if not (bank) matching.

NOTE This feature is available as an option. Setting this option prevents code execution break handling and illegal code execution detection. **Use this option carefully**.

The *Use the debugger eeprom and flash programmer* check box allows you to activate the debugger internal Flash Programmer engine and GUI instead of the Abatron on-board non-volatile memory programmer (the Abatron on-board NVM programmer is the default, except for HCS12X-core devices).

Terminal Emulation

The Abatron BDI Connection supports terminal emulation for **CPU12(HC12), HCS12** and **HCS12X** cores. This allows the target application to write into the debugger ***Terminal*** component. Also, you can direct the characters typed on the host's keyboard to the target application. To use terminal emulation, open the Terminal component in the debugger by choosing **Component > Open > Terminal**.

To simulate the terminal I/O, a 4-byte work space is needed. Configure the work space address with the setup program from ABATRON.

For more information, see the *Terminal* section in the ABATRON *User Manual* and check the `termbgnd.c` source file for communication primitives on the BDI installation disk from ABATRON. Refer to Terminal Component.

Example

The following structure is located in unpaged data memory on the target:

```
0x00 RX - Flag (Byte)
```

`0x01 RX - Char (Byte)`

`0x02-0x03 TX - String Pointer (Word)`

The address of this structure is defined during BDI box setup. The **TermData** structure address (0x0800) must match with the software setup of the BDI, and exactly match the **Terminal Address** in the BDI Working Mode dialog of the ABATRON tool. Refer to the BDI Interface Software Setup section.

While the target is running, the BDI periodically checks to make sure `TX - String Pointer` is not zero. The BDI writes characters received from the host to `RX - Char`, and sets `RX - Flag`.

The following is a possible target implementation:

Listing 15.1 CPU12 Target Implementation

```
typedef struct {
  unsigned char rxFlag;
  unsigned char rxChar;
  char* txBuffer;
} TermDataT;

#define TermData (*((TermDataT*)(0x0800)))

static char txBuffer[2];

char GetChar(void)
{
  char rxChar;
  while (TermData.rxFlag == 0); /* wait for input */
  rxChar = TermData.rxChar;
  TermData.rxFlag = 0;
  return rxChar;
}
void PutChar(char ch)
{
  txBuffer[0] = ch;
  txBuffer[1] = 0;
  TermData.txBuffer = txBuffer;
  while (TermData.txBuffer != 0); /*wait for output buffer empty*/
}
void PutString(char *str)
{
  TermData.txBuffer = str;
  while (TermData.txBuffer != 0); /*wait for output buffer empty*/
}
```

16

TBDML Connection

This section contains information to assist you with the *TBDML* connection.

TBDML Technical Considerations

The 8/16-bit debugger (and the CodeWarrior IDE) may connect to HCS12 hardware using the TBDML cable. When the debugger runs the **TBDML** connection, it communicates and debugs **CPU12 (HC12), HCS12, HCS12X** and **XGATE** core-based hardware through the **Turbo BDM Light** debug cable.

You can retrieve the cable design schematics and cable driver from the Freescale forum, as the cable interface is open source technology. Search for Turbo BDM light or TBDML in a web browser.

The CodeWarrior IDE is compliant with the TBDML firmware and does not require any cable driver installation.

NOTE The TBDML connects to the PC via a USB port.

Connection Menu

Once you set the TBDML connection, the Connection menu entry in the debugger main toolbar changes to **TBDML HCS12**.

TBDML HCS12 Menu Entries

The following figure shows the TBDML HCS12 menu. Table 16.1 describes the menu entries.

Figure 16.1 TBDML HCS12 Menu Options

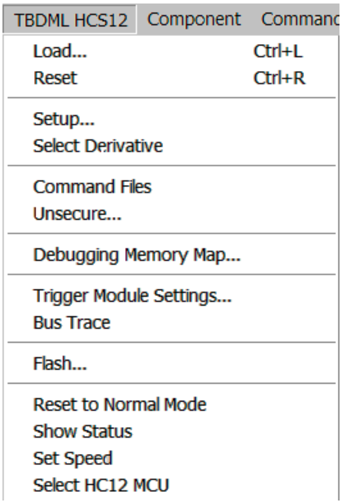


Table 16.1 TBDML HCS12 Menu Entries

Menu Entry	Description
Load Option	Displays the Load Executable File dialog box. See Load Executable File Window.
Reset Option	Resets the project hardware and software.
Setup	Displays the TBDML Setup dialog box: select appropriate TBDML cable for current debug session. **When debugging with several cables and debuggers:** Cable number is relative to connection order to PC's USB hub. At each new debug session, check which device the project addresses, as there is no way to ensure addressing the same cable with the same project.
Select Derivative	Displays Set Derivative dialog box. Allows you to choose target MCU for TBDML connection.

Table 16.1 TBDML HCS12 Menu Entries (*continued*)

Menu Entry	Description

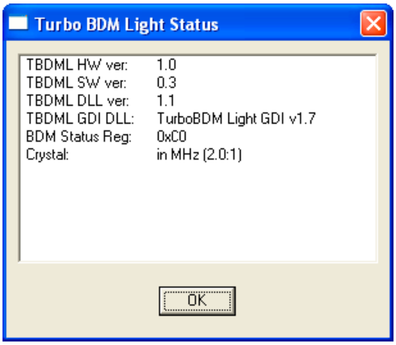
Command Files	Displays Connection Command Files window. Each tab allows access to a different set of connection command files:
	`\cmd\TBDML_startup.cmd`
	`\cmd\TBDML_reset.cmd`
	`\cmd\TBDML_preload.cmd`
	`\cmd\TBDML_postload.cmd`
	`\cmd\TBDML_vppon.cmd`
	`\cmd\TBDML_vppoff.cmd`
	`\cmd\TBDML_erase_unsecure_hcs12.cmd`

Unsecure	Runs the Unsecure command file script of the project. The Unsecure script mass erases target device, and programs target device security byte to an unsecure state, to enable BDM communication.
Debugging Memory Map	Displays Debugging Memory Map dialog box for target MCU. See Debugging Memory Map.
Trigger Module Settings	Displays Trigger Module Settings dialog box. See On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms.
Bus Trace	Displays a Trace component window. See On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms.
Flash	Displays Non-Volatile Memory Control dialog box. See Flash Programming.
Reset To Normal Mode	Resets target device to normal mode. Debugger continues running as target device resets and runs from reset vector in normal mode. Application stopping and debugging remain available.
Show Status	Displays Turbo BDM Light Status dialog box; displays revision versions of drivers and firmware, BDM status register value, and chosen target crystal (Figure 16.2).
Set Speed	Displays Set Target Speed dialog box; use to specify target crystal to synchronize TBDML cable BDM communication with silicon. Crystal/ BDM frequency ratio can be tuned when not matching with default value (2).

Table 16.1 TBDML HCS12 Menu Entries (*continued*)

Menu Entry	Description
Select HC12 MCU	Displays HC12 Derivative Selection box; enables you to debug legacy HC12 (CPU12) devices that cannot be identified with any on-chip device ID (PARTID registers). Selecting **HCS12(X) Autodetect** returns debugger to automatic identification of device ID (PARTID) mode (default debugger setup).
Select Core	Synchronizes debugger windows with a specific core when several cores are available. Available only when several cores are available for debugging.

Figure 16.2 Turbo BDM Light Status Box



Book III - HC(S)12(X) Debugger Common Features

Book III Contents

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

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

17

On-Chip DBG Module for S12, S12S, S12G, S12P, S12X Platforms

The HCS12 derivatives featuring an on-chip DBG module require a graphical user interface to set up this module and take full advantage of this feature. This chapter describes the DBG module features and user interface. The description is provided in generic way with highlighted specifics for particular families.

Several HCS12 debugger connections, such as the P&E Multilink/Cyclone Pro, Abatron BDI, HCS12 Serial Monitor and SofTec inDART-HCS12, provide a complete graphical user interface through a trigger setup dialog box combined with context-sensitive menus. These interfaces are available in Source, Assembly, Data, and Memory component windows to enable you to set the on-chip DBG module and triggers.

This DBG module support is available according to the user-selected derivative features.

DBG Features

The on-chip DBG module provides the user with the following features:

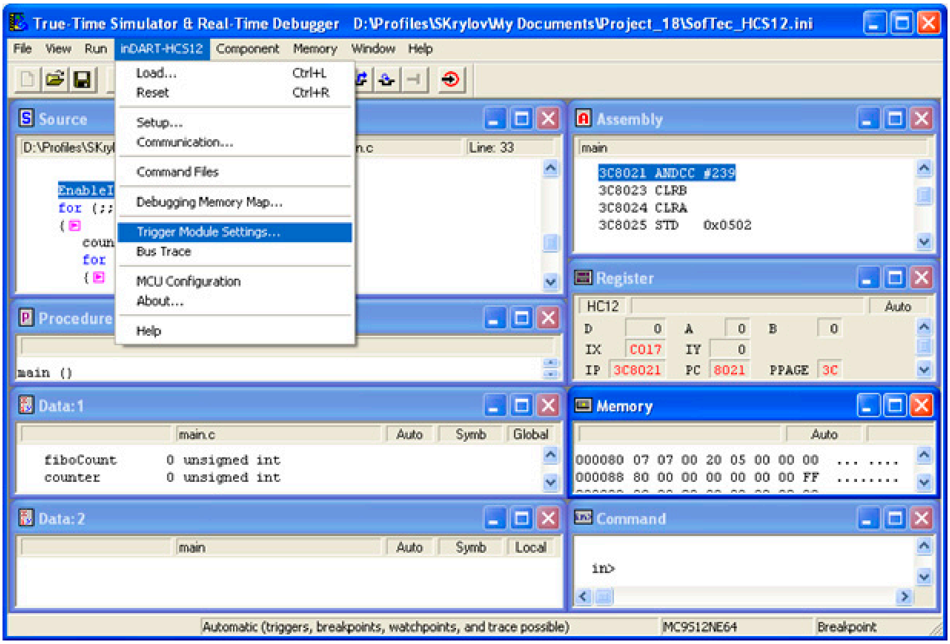
- Regular hardware breakpoints and watchpoints
- Predefined, preset Instruction triggers, Memory Access triggers, or Capture triggers, a wide set of complex hardware breakpoints and watchpoints and data bus recording
- Expert Triggers, as powerful as predefined preset triggers[1]
- Code program flow rebuild from DBG data capture within the Trace window component
- Real-time program code profiling and coverage within the Profiler and Coverage window components

1. S12 platform only

Specific Connection Menu Options

The menu displays two additional entries as soon as the DBG module acknowledges the debugger target processor: **Trigger Module Settings** and **Bus Trace**. Figure 17.1 shows an example with the SofTec (inDART-HCS12) connection.

Figure 17.1 Connection Menu - Added DBG Options



Choose **Trigger Module Settings** to open the Trigger Module Settings Window.

Choose **Bus Trace** to open the Trigger Module Settings Window.

Context Menu Entries

This section describes the functions available from the Source, Assembly, Data, and Memory windows.

Source and Assembly Windows

Source and Assembly windows have menu entries to set or delete triggers, a Trigger Settings entry to set the DBG module triggers, and a Trigger Module Usage entry to set the DBG module functionality globally. Figure 17.2 shows the context menu available in the Source and Assembly windows.

Figure 17.2 Source Context Menu - Added Options, S12 Platform

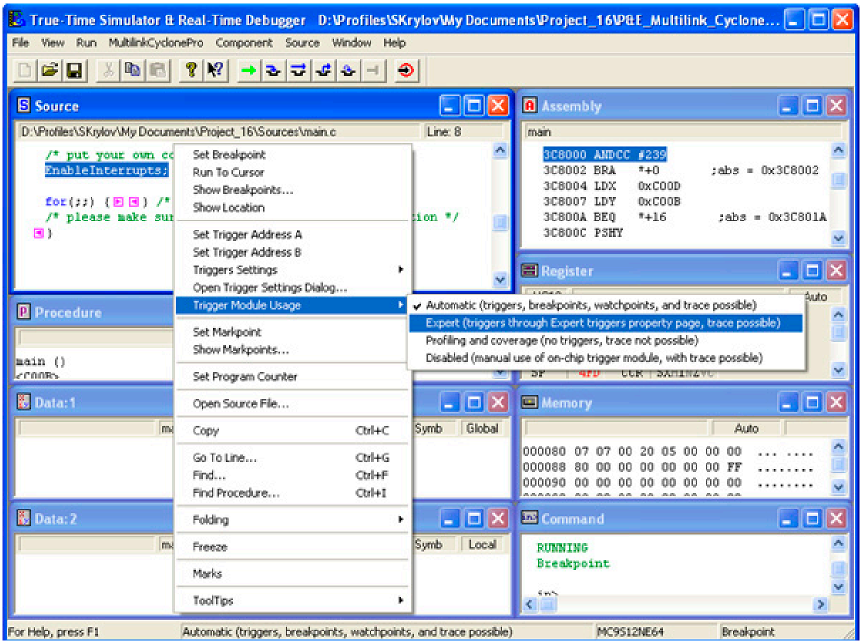
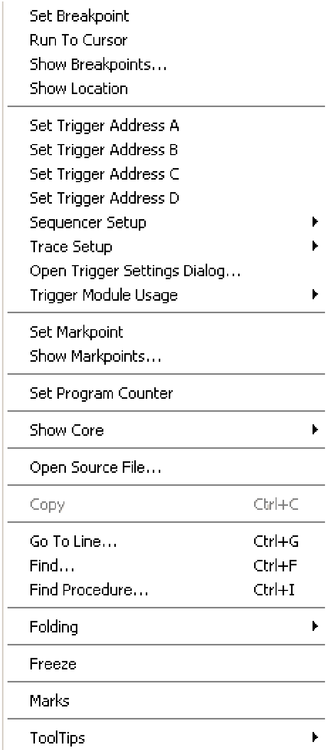


Figure 17.3 Source Context Menu - Added Options, S12X Platform

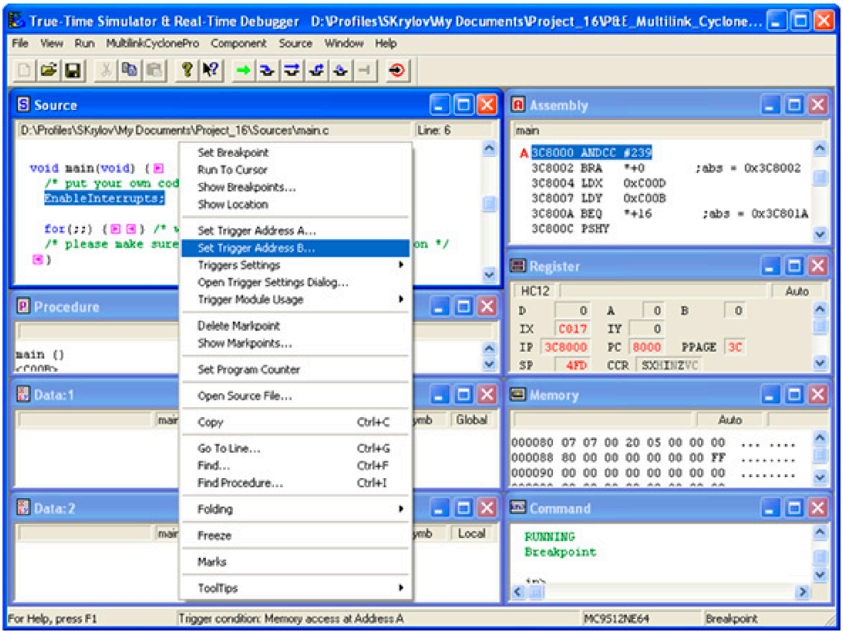


You can set a trigger instead of setting a breakpoint. Setting **Trigger A**, **Trigger B**[1] in various combinations and with various conditions increases programming and debugging flexibility (see DBG Module Mode Setup).

Set TriggerAddress sets a trigger at the selected source location/address (Figure 17.4).

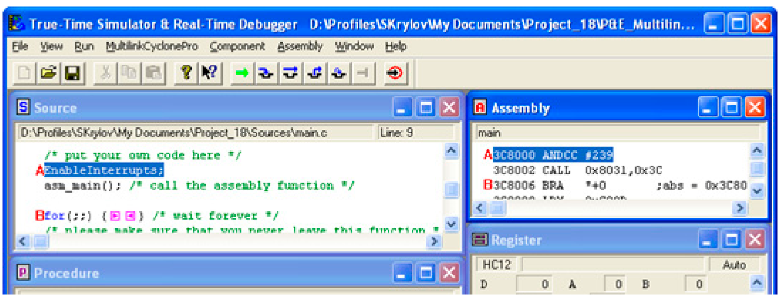
[1] The number of available triggers depends on the device: A,B on S12 platform; A,B,C on S12S/P; A,B,C,D on S12X.

Figure 17.4 Set Trigger Address B Option, S12



The trigger appears in the Source window and at the corresponding address in the Assembly window, just like a breakpoint icon. To be distinguishable from breakpoints, trigger A is marked with a red **A** icon and trigger B with a red **B** icon (Figure 17.5). [1]

Figure 17.5 Triggers Set in Source and Assembly Windows



Once you set a trigger, you can delete it by opening any context-sensitive menu that contains the **Delete Trigger Address** options.

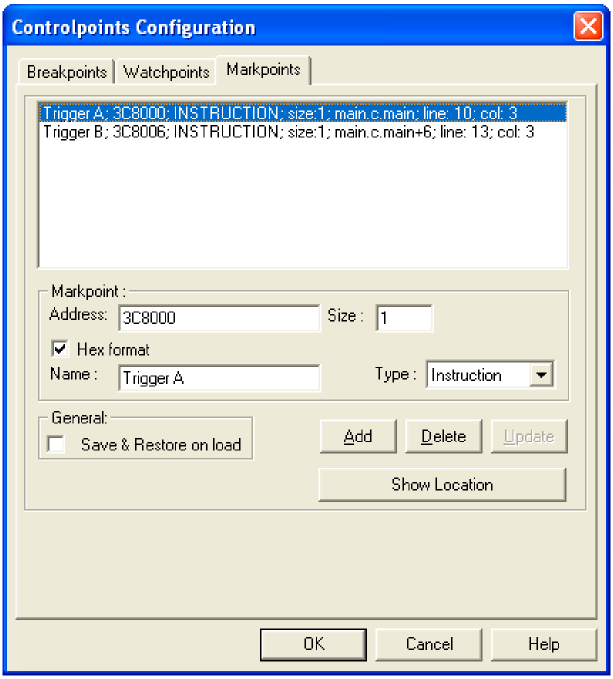
[1] A,B on S12 platform; A,B,C on S12S/P; A,B,C,D on S12X.

Storing Triggers as Markpoints

The debugger stores triggers as special **Trigger A** and **Trigger B**[1] markpoints. You can view markpoints by choosing **Show Markpoints** in the menu. The markpoint names are reserved by the debugger. When you set the trigger from the Source or Assembly window, the debugger automatically selects the Instruction markpoint type.

Selecting **Show Markpoints** from the Source window context menu opens the **Controlpoints Configuration** window with its **Markpoints** tab displayed (Figure 17.6).

Figure 17.6 Controlpoints Configuration Window - Markpoints Tab



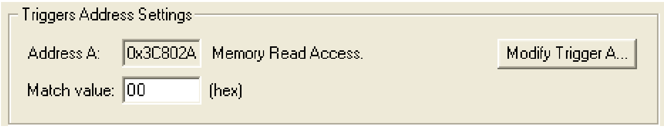
Use the **Save and Restore on load** option to save the application with the DBG module setup and trigger positions. This option is also available with breakpoints and watchpoints.

1. A,B on S12 platform; A,B,C on S12S/P; A,B,C,D on S12X.

Trigger Editing

Use the Triggers Address Settings area (Figure 17.7) of the Trigger Settings tab when specifying trigger addresses, or match or mismatch values. You can also use the context-sensitive menus to set trigger addresses or types.

Figure 17.7 Triggers Address Settings Dialog Box



Pressing a **Modify Trigger** button opens a trigger editor dialog box called Browse for Trigger (see Figure 17.8).

Figure 17.8 Browse for Trigger A Dialog Box

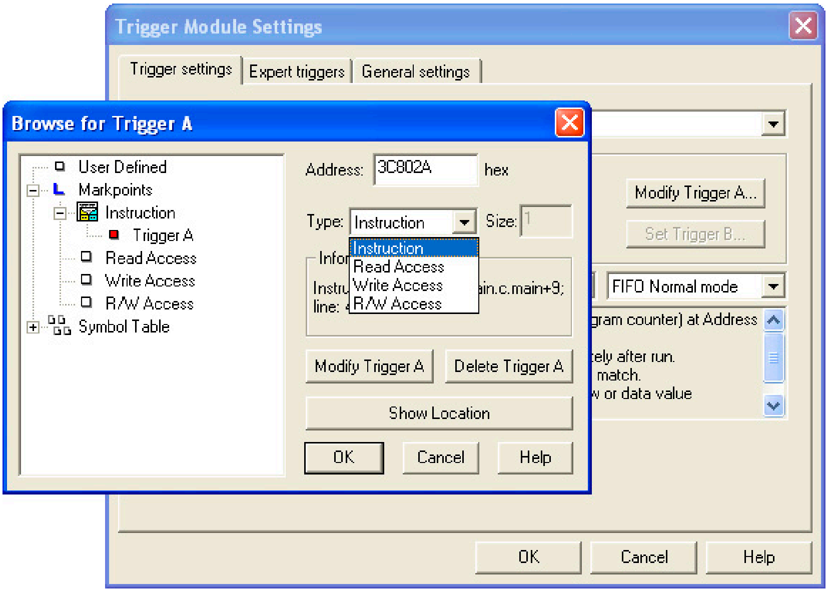


Table 17.1 describes the options available in the Browse for Trigger dialog box.

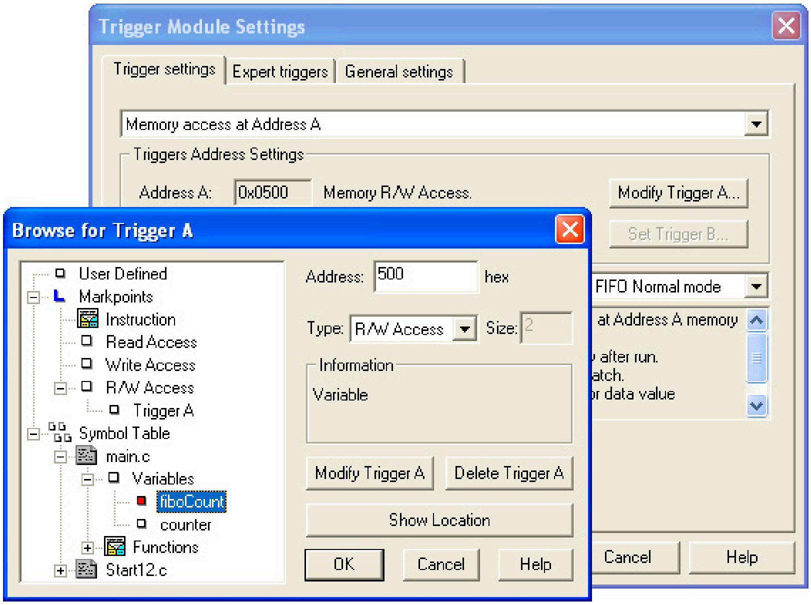
Table 17.1 Browse for Trigger A Options

Option	Description
Address	Contains initial and final trigger address value. Set by typing directly in edit box.
Type	Use to select or change trigger type. Use **Instruction** for Instruction triggers. Use **Read**, **Write** and **R/W Access** for Memory Access and Capture triggers.
Modify Trigger	Modifies and records trigger in trigger database (see Storing Triggers as Markpoints).
Delete Trigger	Removes trigger from trigger database (see Storing Triggers as Markpoints). Trigger address is **undefined**.
Show Location	Shows trigger location (as program code location or program data) in Source, Data, Assembly and Memory windows.

NOTE Pressing OK in this dialog box does NOT update the trigger database. You must press the **Modify Trigger** button in the Trigger Module Settings window before closing the dialog box to update the trigger database.

Use the panel on the left to find a trigger address in the debugger symbol database. Select a variable to copy the variable address into the **Address** edit box. Select a function to copy the entry point of the function into the **Address** edit box. Select regular markpoints from the markpoint list to copy the address of the markpoint into the **Address** edit box.

Figure 17.9 Finding Trigger Address in Editor Dialog Box



Expert Triggers in Source and Assembly Windows[1]

NOTE Expert Triggers are available for S12 devices only

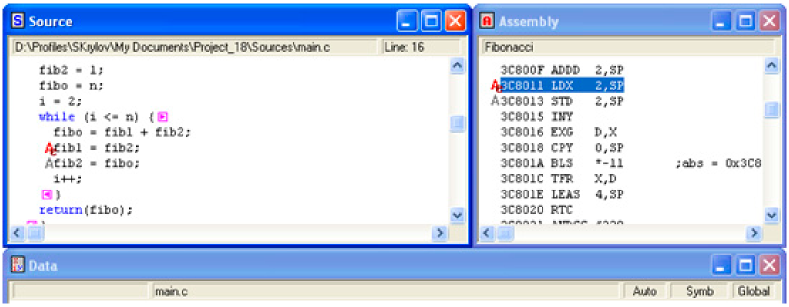
Expert Mode offers a different set of triggers and trigger options. To completely separate Expert mode from Automatic mode, the debugger provides a unique set of Expert triggers. Expert triggers are independent of normal regular triggers.

As shown in Figure 17.10, Expert triggers appear in Source and Assembly windows with a small additional **e** character, and different colors in the Memory component. When you set the trigger from the Source or Assembly window, the debugger automatically selects the markpoint type **INSTRUCTION**.

NOTE Setting Expert mode grays out preset Instruction, Memory Access, or Capture trigger designs. Setting automatic mode or a predefined preset trigger grays out the Expert mode trigger designs.[2]

1. S12 platform only.
2. S12 platform only.

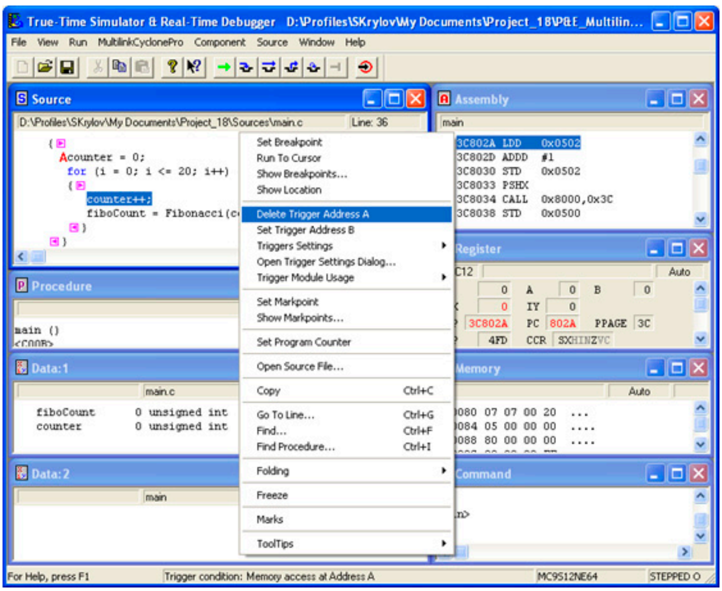
Figure 17.10 Expert Triggers in Source, Assembly, Memory and Data Windows



Data and Memory Windows

From the Data and Memory windows context menus, you can set or delete Memory Access Triggers A, B[1], set the DBG module triggers settings, and globally set the DBG module functionality.

Figure 17.11 Data Window Context Menu - Delete Trigger A Option, S12



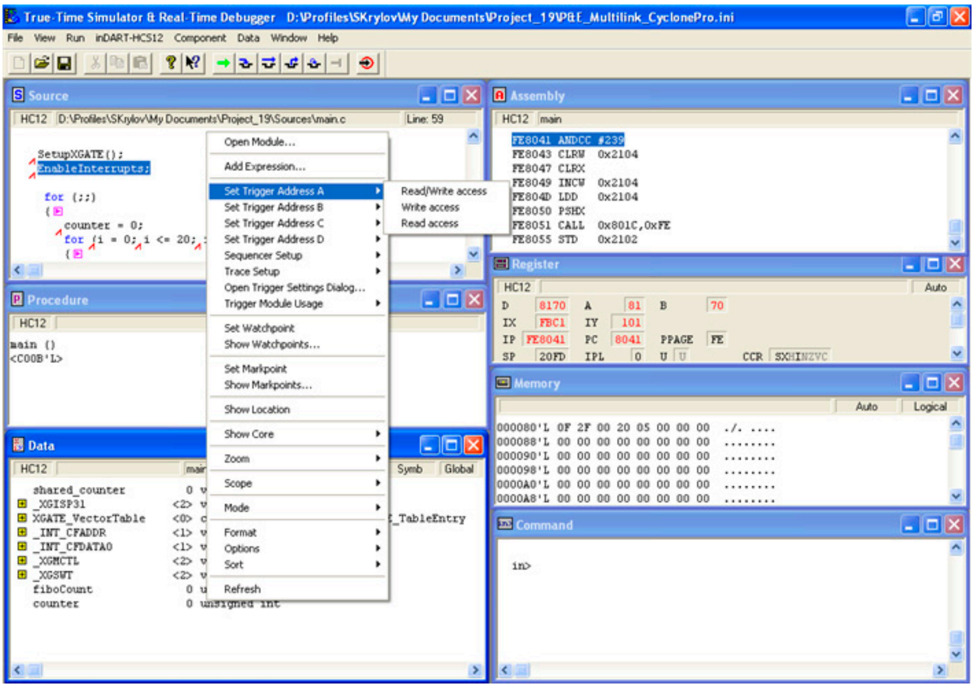
[1] A,B on S12 platform; A,B,C on S12S/P; A,B,C,D on S12X.

You can set or delete a trigger in the Data window. The on-chip DBG module provides combinations of trigger conditions, and according to the number of triggers defined[1], you can choose a different type of trigger (see DBG Module Mode Setup).

To set a trigger, select a location, choose one of the **Set Trigger Address** options and select the kind of access (**Read, Write, Read/Write**). Setting the trigger from the Data or Memory window automatically selects the corresponding **READACCESS**, **WRITEACCESS** or **READWRITEACCESS** markpoint type.

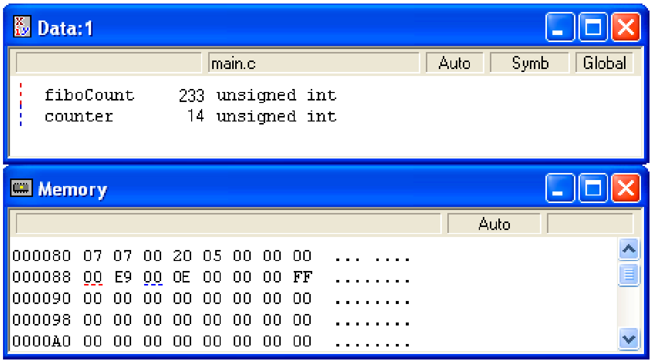
The trigger appears in the Data window and at the corresponding address in the Memory window. To distinguish triggers from watchpoints, trigger A appears with a dotted red vertical line and trigger B with a dotted blue vertical line (Figure 17.13).

Figure 17.12 Data Window Context Menu - Set Trigger A Option, S12X



1. A,B on S12 platform; A,B,C on S12S/P; A,B,C,D on S12X.

Figure 17.13 Triggers Set in Data and Memory Windows



Expert Triggers in Data and Memory Windows[1]

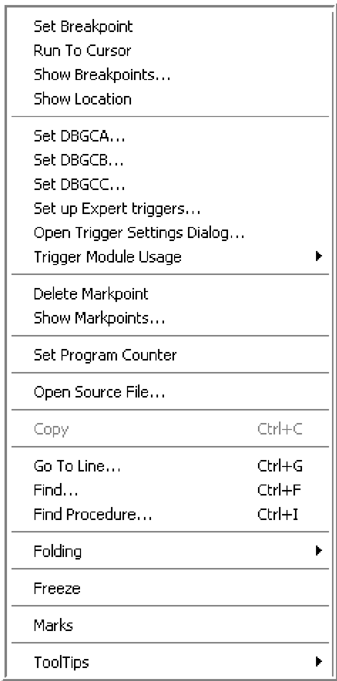
NOTE Expert Triggers are available for S12 devices only.

Expert Mode offers a different set of triggers and trigger options. To completely separate Expert mode from Automatic mode, the debugger provides a unique set of Expert triggers. Expert triggers are independent of normal regular triggers.

In the Data and Memory windows, context-sensitive menu entries for Expert Triggers contain a **Set DBGCA** entry and a **Set DBGCB** entry. Expert Mode requires a thorough knowledge of the DBG module, register usage, and debugging. For more details on Expert mode and Expert triggers, see Expert Triggers Tab.

[1] S12 platform only

Figure 17.14 Source Window Context Menu - Expert Trigger Options, S12



NOTE Setting Expert mode grays out preset Instruction, Memory Access, or Capture trigger designs. Setting automatic mode or a predefined preset trigger grays out the Expert mode trigger designs.

The Markpoints tab of the Controlpoints Configuration window stores expert triggers as **DBGCA** and **DBGCB** markpoints. These markpoint names are therefore reserved by the debugger.

Use the **Save and Restore on load** option to automatically save the application with the current DBG module setup and trigger positions.

Trigger Settings[1]

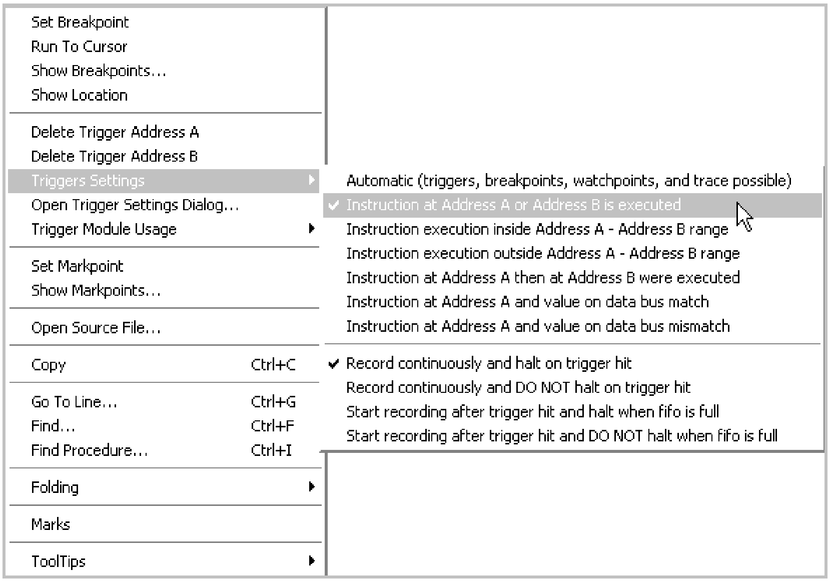
You can use the Trigger Settings option of a context menu to set all kinds of triggers without opening the Trigger Module Settings Window. However, the amounts and combinations of trigger types are **dynamic**, depending on how many triggers are defined, which triggers are defined, and the trigger type (Instruction, Read Access, Read/Write

1. S12 platform only

Access, Write Access). The menu displays only those trigger conditions and combinations that are currently available.

You can also directly edit the DBG Module Options.

Figure 17.15 Triggers Setting Menu Option - Extended Menu, S12



Trigger Module Usage

Use this menu entry to set the DBG module functionality globally, without opening the Trigger Module Settings window (Figure 17.16).

Figure 17.16 Source Window Extended Menu, S12

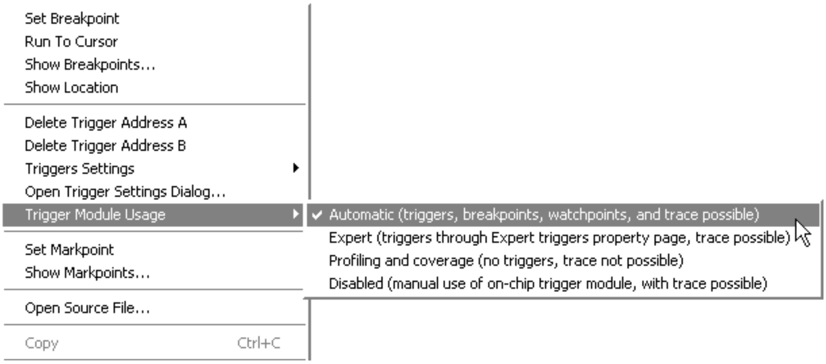
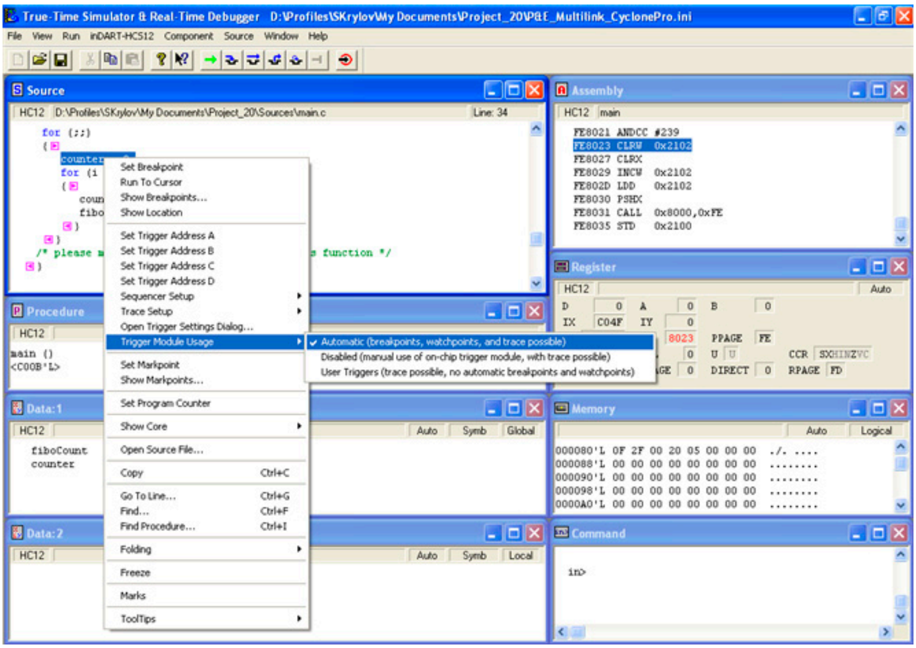


Figure 17.17 Source Window Extended Menu, S12X



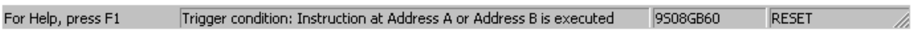
DBG Support Status Bar Item

A specific DBG support debugger status bar item appears as soon as the debugger target processor features the DBG module. Clicking on this item opens the Trigger Module Settings window.

Figure 17.18 Status Bar Item

The status bar displays the current DBG module mode setup (as shown above) or the current preset Instruction triggers, Memory Access triggers or Capture triggers in use.

Figure 17.19 Status Bar Item



Trigger Module Settings Window

You can open this window from context-sensitive menus in the Source, Data, Memory and Assembly component windows, from the Connection menu, or by clicking on a Status Bar item.You can fully control the on-chip DBG module from this window.

- S12 DBG Module Tabs
- S12G, S12P, S12S DBG Module Tabs
- S12X DBG Module Tabs

S12 DBG Module Tabs

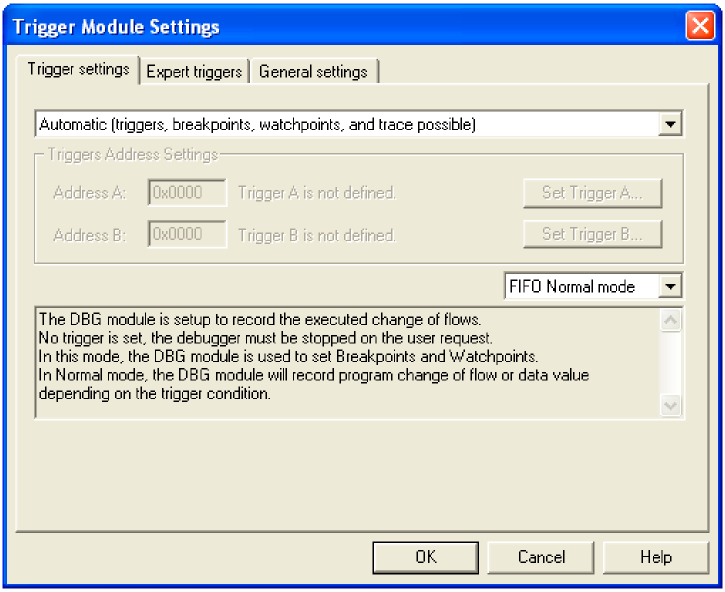
It consists of:

- Trigger Settings Tab
- Expert Triggers Tab

Trigger Settings Tab

Use the Triggers Settings tab to set the trigger mode and trigger address (if this option is available in the selected mode).

Figure 17.20 Trigger Module Settings Window - Trigger Settings Tab, S12



DBG Module Mode Setup

The on-chip DBG module provides some exclusive debugging features. Four modes are available:

- Automatic
- Expert
- Profiling and Coverage
- Disabled

Four types of triggers are available:

- Memory Access Triggers
- Instruction Triggers
- Capture Triggers
- Expert Triggers

All modes and triggers are available through the Trigger Settings tab. Open the top list menu to display all available modes, triggers, and trigger conditions (Figure 17.21). Table 17.2 describes the modes and trigger settings.

NOTE Encountering any Memory Access or Instruction triggers causes the Trace Component window to switch to Instructions Display mode and display the program flow rebuild (see Trigger Module Settings Window and Instructions Display for more information).

NOTE Encountering any Capture trigger causes the Trace Component window to switch to Recorded Data Display mode and display the captured byte data (see Trigger Module Settings Window and Recorded Data Display for more information).

NOTE Expert Mode is described in Expert Triggers Tab.

Figure 17.21 Trigger Settings Tab Listbox, S12

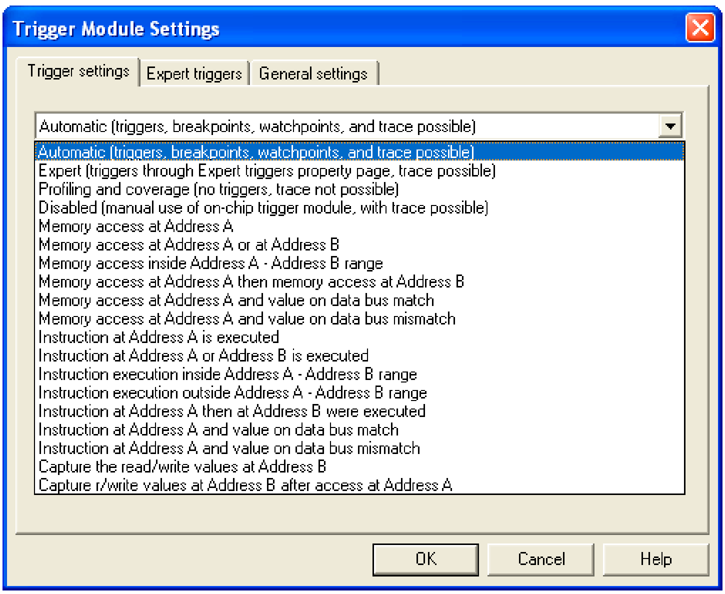


Table 17.2 Trigger Modes

Mode	Description
Automatic Mode (Default)	Set up **three** regular hardware breakpoints or one watchpoint, or set up triggers selected from list or menu. Set trigger conditions and addresses from Source, Assembly, Memory or Data components using Set Trigger A or Set Trigger B menu entry, or within this dialog. DBG module records executed change of flows. Since no triggers are set in automatic mode, the debugger stops on the typical breakpoints/watchpoints or by user request.
Expert Mode	Enables the Expert Triggers tab (see Expert Triggers Tab).
Profiling and Coverage Mode	Sets DBG module to code execution profiling and code execution coverage. Open Profiler and/or Coverage components to display results. Uses a periodic real-time fetch from debugger program counter to DBG module. Not all program counters are represented with each fetch. Improve accuracy and precision by using a longer run time and test period. Use software breakpoints; triggers and DBG-based control points have no effect. User must request the debugger to stop.
Disabled Mode	Requires advanced knowledge of on-chip DBG module. Use to set hardware breakpoints, watchpoints, and triggers. Requires user to handle trigger comparator addresses and DBG control registers through Memory component or using command line commands, without a dedicated GUI to access DBG module register. Use selected flags within DBG control registers to enable or arm DBG module. On halt, debugger automatically protects FIFO content from unexpected reads, analyzes FIFO content, and disarms DBG module (can be disabled by user). Stopping an application does not reset DBG module. Debugger does not set DBG module.

Table 17.3 Trigger Types

Type	Description
Memory Access at Address A	Set trigger on a program instruction read or write at memory location labeled Address A.
Memory Access at Address A or Address B	Set trigger on a program instruction read or write at memory location labeled Address A or Address B.
Memory Access Inside Address A - Address B Range	Set trigger on a program instruction read or write that occurs within Address A to Address B memory range.
Memory Access at Address A then Memory Access at Address B	Set trigger on a program instruction sequence that first reads or writes at Address A memory location, then reads or writes at Address B memory location.
Memory Access at Address A and Value on Data Bus Match	Set trigger on a program instruction read or write of a specific matching byte value at Address A memory location. Uses **trigger B** address as **match value** rather than address location. Select this trigger type without setting match value and an error message prompts for match value. Replaces standard trigger editing dialog box with Triggers Address Settings dialog box (see Figure 17.7). Match Value edit boxes replace Trigger Editing dialog.
Memory Access at Address A and Value on Data Bus Mismatch	Set trigger on a program instruction read or write of a non-matching byte value at Address A memory location. Uses **trigger B** address as **mismatch value** rather than address location. Select this trigger type without setting the mismatch value and an error message prompts for value. Replaces standard trigger editing dialog box with Triggers Address Settings dialog box (see Figure 17.7). Match Value edit boxes replace Trigger Editing dialog.
Instruction at Address A Is Executed	Set a trigger on a program instruction execution (program counter) occurring at Address A.
Instruction at Address A or Address B Is Executed	Set a trigger on a program instruction execution (program counter) occurring at either Address A or Address B.
Instruction Execution Inside Address A - Address B Range	Set a trigger on a program instruction execution (program counter) occurring within Address A to Address B range.

Table 17.3 Trigger Types (*continued*)

Type	Description
Instruction Execution Outside Address A - Address B Range	Set a trigger on a program instruction execution (program counter) occurring outside Address A to Address B range. With the **HCS12 Serial Monitor via GDI connection**, monitor code may interfere with this trigger type. Debugger may fail for executed code not belonging to user application.
Instructions at Address A then at Address B Were Executed	Set a trigger on a program instruction execution (program counter) occurring first at Address A, then Address B.
Instruction at Address A and Value on Data Bus Match	Set a trigger on a program instruction execution at Address A with an instruction opcode that matches a specific byte value at Address A memory location. Uses **trigger B** address as **match value** rather than an address location. Select this trigger type without setting match value and an error message prompts for match value. Replaces standard trigger editing dialog box with Triggers Address Settings dialog box (see Figure 17.7). Match Value edit boxes replace Trigger Editing dialog.
Instruction at Address A and Value on Data Bus Mismatch	Set a trigger on a program instruction execution of a non-matching byte value at Address A memory location. Uses **trigger B** address as **mismatch value** rather than an address location. Select this trigger type without setting match value, and an error message prompts for match value. Replaces standard trigger editing dialog box with Triggers Address Settings dialog box (see Figure 17.7). Match Value edit boxes replace Trigger Editing dialog.
Capture Read/Write Values at Address B	Use to capture data used in a read or write access to address location specified by **trigger B**. Typically a data or memory address rather than program code address (program counter).
Capture Read/Write Values at Address B After Access at Address A	Use to capture data used in a read or write access to address location specified by **trigger A** and **trigger B**. Typically a data or memory address rather than program code address (program counter). Capture of values at Address B begins only after accessing Address A.

DBG Module Options

The DBG module includes options to record and change the program flow when using instruction and memory access triggers. The following options are available:

- Record continuously and halt on trigger hit
- Record continuously and DO NOT halt on trigger hit
- Start recording after trigger hit and halt when the FIFO is full
- Start recording after trigger hit and DO NOT halt when the FIFO is full

When using Capture triggers, the following data recording options are available:

- Halt when the FIFO is full
- Do not halt when the FIFO is full

Recording Program Code Change of Flow and Data Recording Options describe these options.

Recording Program Code Change of Flow

Use the recording options with Instruction and Memory Access triggers. Use the Trigger Module Settings window's Trigger Settings tab list box (Figure 17.22) to control the recording options described in Table 17.4:

Figure 17.22 Change of Flow Recording Control

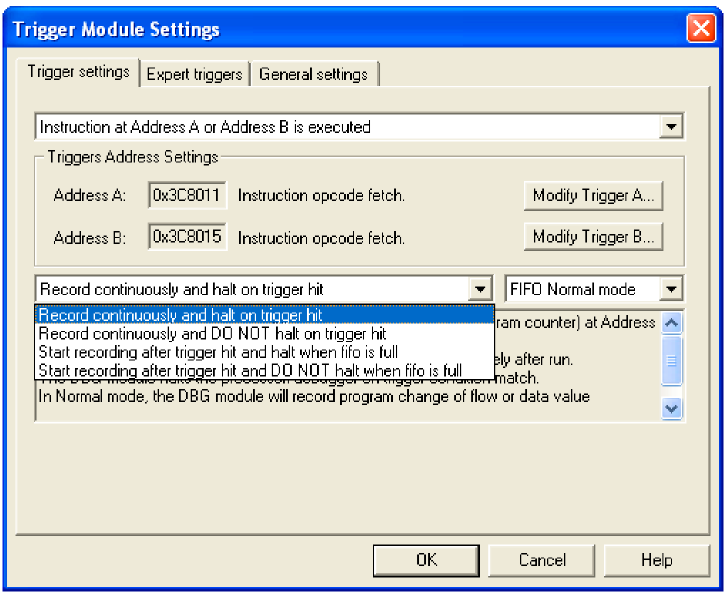


Table 17.4 Recording Options in Trigger Settings Tab

Option	Description
Record continuously and halt on trigger hit	Begins recording program flow information immediately after run. Halts processor/debugger when trigger condition match occurs.
Record continuously and DO NOT halt on trigger hit	Begins recording program flow information immediately after run. Does not halt the processor/debugger when trigger condition match occurs
Start recording after trigger hit and halt when the FIFO is full	Begins recording program flow information when trigger condition match occurs and halts processor/debugger when capture buffer is full.
Start recording after trigger hit and DO NOT halt when the FIFO is full	Begins recording program flow information when trigger condition match occurs. Does not halt processor/ debugger.

Data Recording Options

Use the data recording options with Capture triggers. Select these options from the list box in the Trigger Settings tab of the Trigger Module Settings window (Figure 17.23). Table 17.5 shows the available data recording options.

Figure 17.23 Data Recording Control

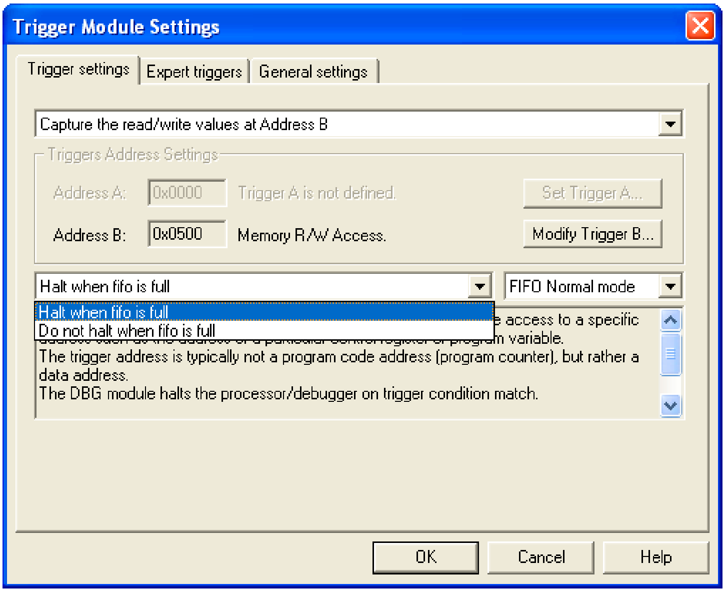


Table 17.5 Data Recording Options for Capture Triggers

Option	Description
Halt when the FIFO is full	Continuously records data accesses and halts processor/ debugger when capture buffer is full.
Do not halt when the FIFO is full	Continuously records data accesses but does not halt processor/debugger when the capture buffer is full.

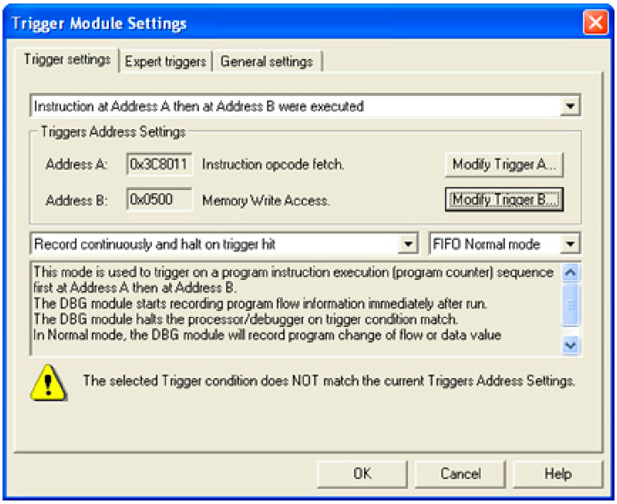
Display Information

A large gray box provides dynamic information about the current triggers and selected options.

As context-sensitive menus only display triggers matching the number and type of currently set triggers, the debugger checks the current trigger settings against the trigger mode. If one or more triggers do not match the trigger mode selection, a warning icon and message appears on the bottom of the dialog.

The display field in Figure 17.24 shows that the **Memory Write Access** type does not match the Instruction trigger type selected in the list.

Figure 17.24 Trigger Settings Tab Information Display, S12



Expert Triggers Tab

The Expert triggers tab gives you access to most of the on-chip DBG module registers. You can set trigger types directly from the **DBGT - Debugger Trigger Register** list menu.

Using Expert Mode and Expert triggers requires thorough knowledge and understanding of the on-chip DBG module, registers, and flags. Use this mode to synchronize code program flow rebuild and data recording and display the results in Trigger Module Settings Window.

To set Expert triggers, use the Trigger Module Settings window **ExpertTriggers** tab. Select **Expert** mode in the list menu (Figure 17.25) to enable the Expert Triggers tab (Figure 17.26).

Figure 17.25 Trigger Settings Tab - Expert Mode Information

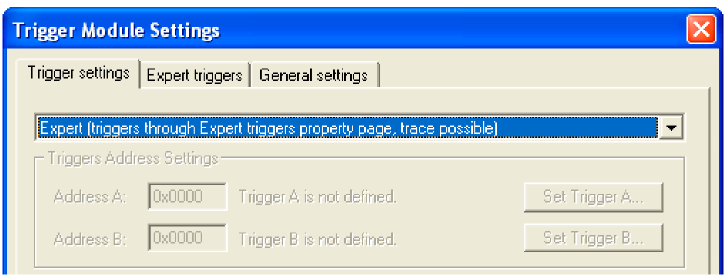
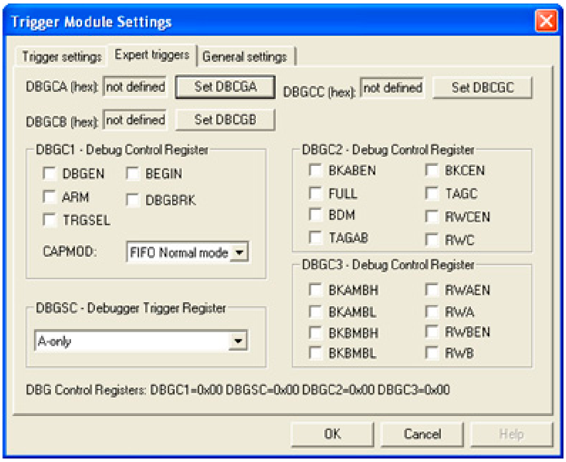


Figure 17.26 Trigger Module Settings Window - Expert Triggers Tab



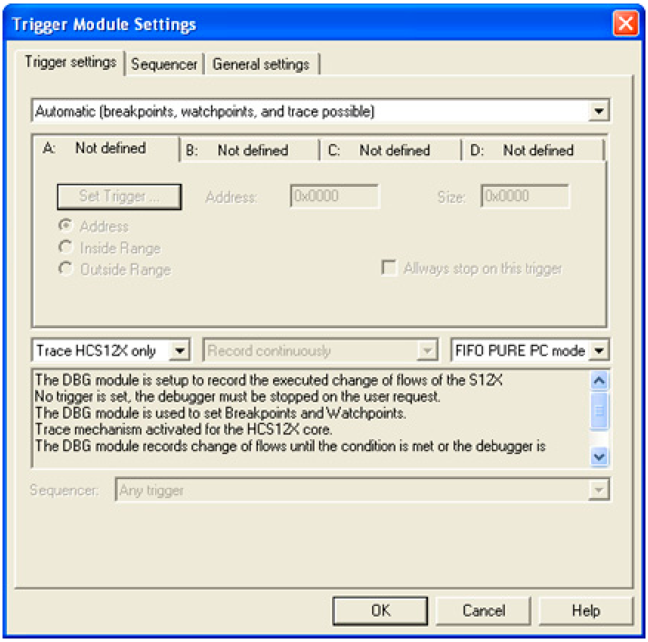
Use **Set DBGCA** or **Set DBGCB** to set the triggers comparator addresses from the Source, Assembly, Memory and Data component windows. The debugger sets the DBG module. DBG module enabling and arming depend on selected flags set within the DBG register control registers (through the Expert triggers tab). In this mode, the debugger writes all settings to the hardware right before the application runs. The debugger resets the DBG module when the application stops.

NOTE Refer to the core reference manual for detailed information on specific registers.

S12G, S12P, S12S DBG Module Tabs

The user can modify or control all of the available options using the Trigger Module settings window, with its three tabs, Trigger Settings, Sequencer and General Settings.

Figure 17.27 Trigger Settings Tab Listbox



Switch DBG modes by displaying the list menu, as shown above, and selecting any one of the three modes. With some modes, some of the Trigger Settings tab options are not available.

Use the sub-tabs of the Trigger Settings tab of this window, with their text fields, radio buttons and check boxes, to set up the individual triggers.

Trigger Settings Tab

Use the Triggers Settings tab to set the trigger mode and trigger address (if this option is available in the selected mode).

DBG Module Mode Setup

Three modes are available:

- Automatic
- Disabled

- User triggers

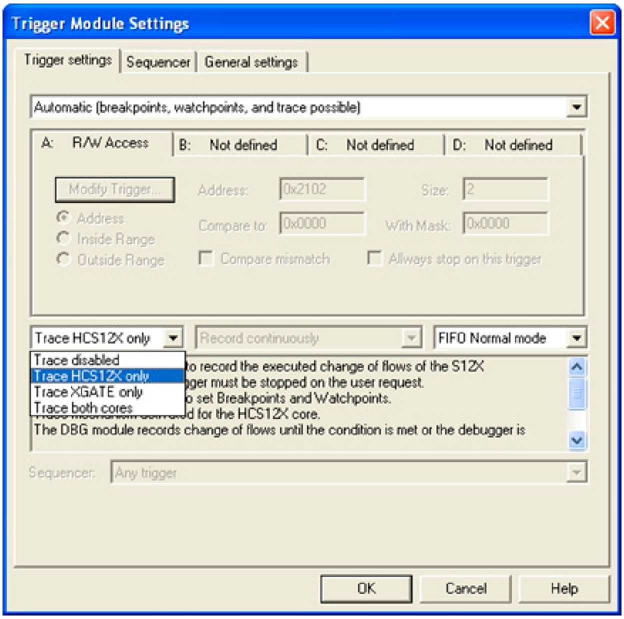
Table 17.6 Trigger Modes

Mode	Description
Automatic Mode (Default)	Use DBG Module to set up **three** hardware breakpoints or two hardware breakpoints and one watchpoint. Set triggers by selecting from the list or from a context-sensitive menu. Automatic mode is the default selection when no triggers are currently set. Set trigger condition and trigger addresses from debugger Source, Assembly, Memory and Data components using Set Trigger A or Set Trigger B context-sensitive menu entry, or from Trigger Settings dialog. DBG module records executed change of flows. Since no triggers are currently set, stopping the debugger requires a user request, a breakpoint, or watchpoint.
Triggers Disabled Mode	User makes trigger settings manually. Debugger analyzes DBG module before and after the run to build up Status and Trace information, but does not set up DBG module before running. Requires advanced knowledge of on-chip DBG module. Use to set hardware breakpoints, watchpoints, and triggers. User must handle trigger comparator addresses and DBG control registers through Memory component or by using command line commands, without a dedicated GUI to access DBG module register. Use selected flags within DBG control registers to enable or arm DBG module. By default, when debugger halts it automatically protects FIFO content from unexpected reads, analyzes FIFO content, and disarms DBG module (user can disable this function). Stopping an application does not reset DBG module, nor does debugger set DBG module.
User Triggers Mode	User must define trigger type. Provides a 4-Stage Sequencer for trace buffer control. List menu provides predefined sequences. User can define their own sequences. In this mode: • Debugger does not set triggers as watchpoints or breakpoints automatically • User can define three triggers and their conditions • User can use the Sequencer to decide how to stop the debugger.

Trace Setup Control

Use the options made available in the Trace list menu of the Trigger Settings tab to set up DBG trace in the Trace component window.

Figure 17.28 Trigger Settings Tab - Trace Drop List

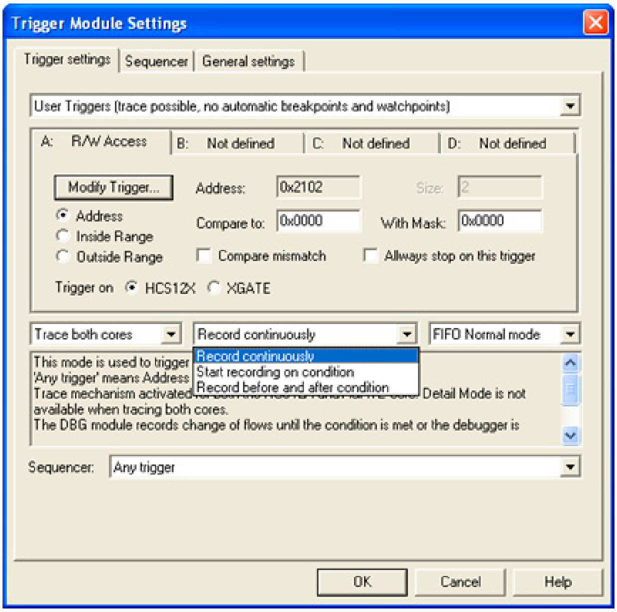


The list options allow you to enable or disable tracing.

Table 17.7 Trigger Settings Tab - Trace Drop List

Option	Description
Trace disabled	Disable trace
Trace HCS12X only	Trace the HCS12X core only
Trace XGATE only	Trace the XGATE core only
Trace both cores	Trace both the cores

Figure 17.29 Trigger Align



Using the this list it is possible to align the trigger with the end or the beginning of a tracing session.

Table 17.8 Trigger Align

Options	Description
Record continuously	Starts recording program flow information immediately after run. Halts the processor/ debugger when capture buffer is full.
Start recording on condition	Starts recording program flow information on trigger condition match. Halts the processor/ debugger when capture buffer is full.
Record before and after condition	Starts recording program flow information on trigger condition match. Does not halt the processor/debugger on trigger condition match; halts when capture buffer is full.

Figure 17.30 Data Recording Control

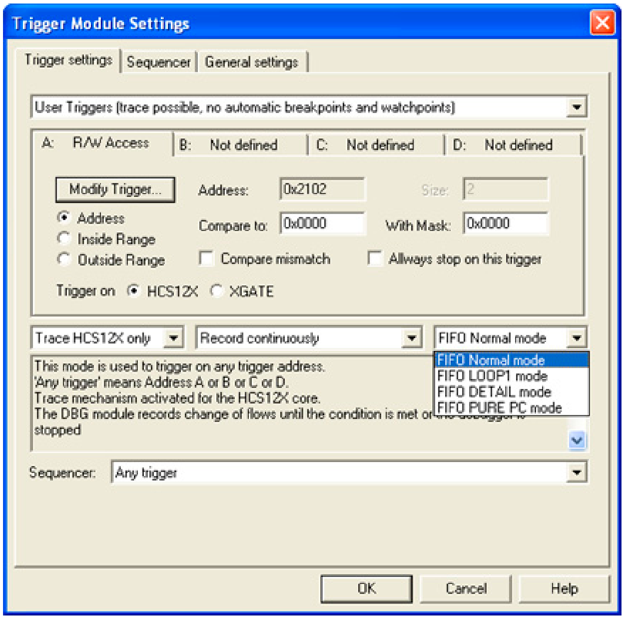


Table 17.9 Data Recording Options

Option	Description
FIFO normal mode	Change of flow (COF) program counter (PC) addresses are stored, debugger rebuilds program flow.
FIFO LOOP1 mode	Loop1 Mode, similarly to Normal Mode also stores only COF address information to the trace buffer, it however allows the filtering out of redundant information. Loop1 Mode only inhibits consecutive duplicate source address entries that would typically be stored in most tight looping constructs. It does not inhibit repeated entries of destination addresses or vector addresses, since repeated entries of these would most likely indicate a bug in the user's code that the DBG module is designed to help find. The debugger rebuilds program flow.

Table 17.9 Data Recording Options

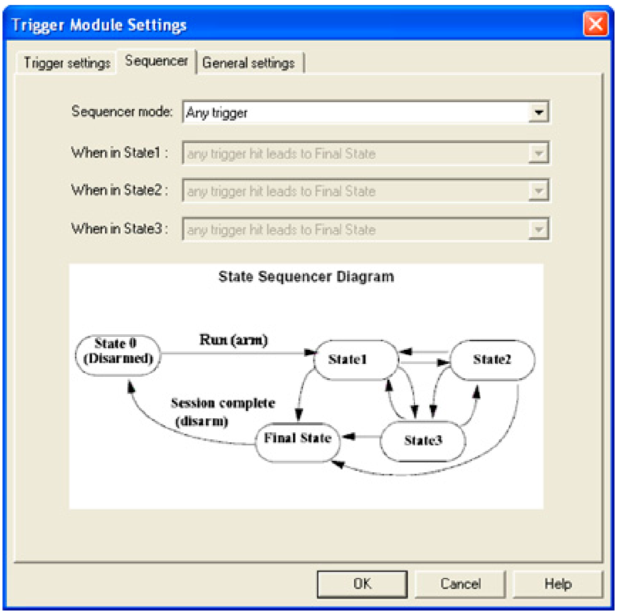
Option	Description
FIFO DETAIL mode	In Detail Mode, address and data for all memory and register accesses is stored
Pure PC mode	Default recording mode when available. Debugger decodes pure PCs recorded by module. Does not perform program flow rebuild. In Compressed Pure PC Mode[1], the PC addresses of all executed opcodes are stored. A compressed storage format is used to increase the effective depth of the trace buffer.

1. S12G, S12P platform

Sequencer Tab

You can choose or change the DBG sequence in the Sequencer tab of the Trigger Module Settings window. The sequencer tab reflects transitions that occur when you select a predefined sequencer mode.

Figure 17.31 Trigger Module Settings Window - Sequencer Tab



You can modify the transitions in the Sequence tab only when the DBG is in User Sequencer setup mode.

S12X DBG Module Tabs

Trigger Settings Tab

Use the Triggers Settings tab to set the trigger mode and trigger address (if this option is available in the selected mode).

DBG Module Mode Setup

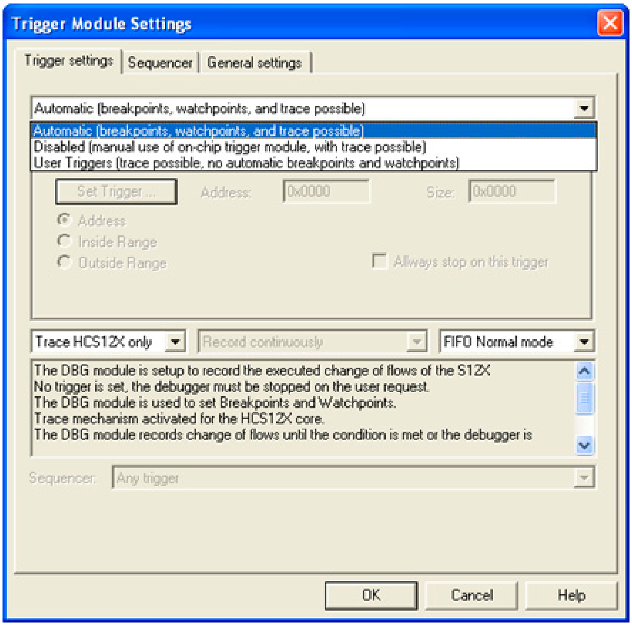
The on-chip DBG module provides some exclusive debugging features. Three modes are available:

- Automatic
- Disabled
- User triggers

All three modes offer:

- Code rebuilding from Change of Flow, in the Trace component window,
- Breakpoint, watchpoint and trigger setting for either the HCS12X core or the XGATE core.

Figure 17.32 Trigger Settings Tab Listbox



Trigger Types

This section describes the types of triggers and how to use them. Three types of triggers are available:

- Memory Access Triggers
- Instruction Triggers
- Capture Triggers

All triggers are available in Automatic and User Triggers modes. Select either of these modes to enable access to trigger options. Table 17.2 describes the trigger settings.

NOTE Encountering any Memory Access or Instruction triggers causes the Trace Component window to switch to Instructions Display mode and display the program flow rebuild (see Trace Component Window and Instructions Display for more information).

NOTE Encountering any Capture trigger causes the Trace Component window to switch to Recorded Data Display mode and display the captured byte data (see Trace Component Window and Recorded Data Display for more information).

Table 17.10 Trigger Modes

Mode	Description
Automatic Mode (Default)	Use DBG Module to set up **four** hardware breakpoints or two hardware breakpoints and one watchpoint. User can see and set HCS12X trace. Setting a trigger automatically switches to User Triggers mode. Set triggers by selecting from the list or from a context-sensitive menu. Automatic mode is the default selection when no triggers are currently set. Set trigger condition and trigger addresses from debugger Source, Assembly, Memory and Data components using Set Trigger A or Set Trigger B context-sensitive menu entry, or from Trigger Settings dialog. DBG module records executed change of flows. Since no triggers are currently set, stopping the debugger requires a user request, a breakpoint, or watchpoint.

Table 17.10 Trigger Modes (*continued*)

Mode	Description
Triggers Disabled Mode	User makes trigger settings manually. Debugger analyzes DBG module before and after the run to build up Status and Trace information, but does not set up DBG module before running. Requires advanced knowledge of on-chip DBG module. Use to set hardware breakpoints, watchpoints, and triggers. User must handle trigger comparator addresses and DBG control registers through Memory component or by using command line commands, without a dedicated GUI to access DBG module register. Use selected flags within DBG control registers to enable or arm DBG module. By default, when debugger halts it automatically protects FIFO content from unexpected reads, analyzes FIFO content, and disarms DBG module (user can disable this function). Stopping an application does not reset DBG module, nor does debugger set DBG module.
User Triggers Mode	User must define trigger type. Provides a 4-Stage Sequencer for trace buffer control. List menu provides predefined sequences. User can define their own sequences. In this mode: • Debugger does not set triggers as watchpoints or breakpoints automatically • User can define four triggers and their conditions • User can use the Sequencer to decide how to stop the debugger.

Table 17.11 Memory Access Triggers Available

Mode	Description
Memory Access at Address A	Use to set a trigger on a program instruction read or write at memory location labeled Address A.
Memory Access at Address A or Address B	Use to set a trigger on a program instruction read or write at memory location labeled Address A or Address B.
Memory Access Inside Address A - Address B Range	Use to set a trigger on a program instruction read or write that occurs within Address A to Address B memory range.
Memory Access at Address A then Memory Access at Address B	Use to set a trigger on a program instruction sequence that first reads or writes at Address A memory location, then reads or writes at the Address B memory location.

Table 17.11 Memory Access Triggers Available (*continued*)

Mode	Description
Memory Access at Address A and Value on Data Bus Match	Use to set a trigger on a program instruction read or write of a specific matching byte value at Address A memory location. Uses **trigger B** address as a **match value** rather than an address location. If you select this trigger type using a context menu without setting the match value, an error message prompts for the match value. Replaces the standard trigger editing dialog box with Triggers Address Settings dialog box (see Figure 17.7). Match Value edit boxes replace Trigger Editing dialog.
Memory Access at Address A and Value on Data Bus Mismatch	Use to set a trigger on a program instruction read or write of a non-matching byte value at Address A memory location. Uses **trigger B** address as a **mismatch value** rather than an address location. If you select this trigger type using a context menu without setting the match value, an error message prompts for the match value. Replaces the standard trigger editing dialog box with Triggers Address Settings dialog box (see Figure 17.7). Match Value edit boxes replace Trigger Editing dialog.
Instruction at Address A Is Executed	Use to set a trigger on a program instruction execution (program counter) occurring at Address A.
Instruction at Address A or Address B Is Executed	Use to set a trigger on a program instruction execution (program counter) occurring at either Address A or B.
Instruction Execution Inside Address A - Address B Range	Use to set a trigger on program instruction execution (program counter) occurring within Address A to Address B range.
Instruction Execution Outside Address A - Address B Range	Use to set a trigger on a program instruction execution (program counter) occurring outside Address A to Address B range.
Instructions at Address A then at Address B Were Executed	Use to set a trigger on program instruction execution (program counter) occurring first at Address A, then Address B.

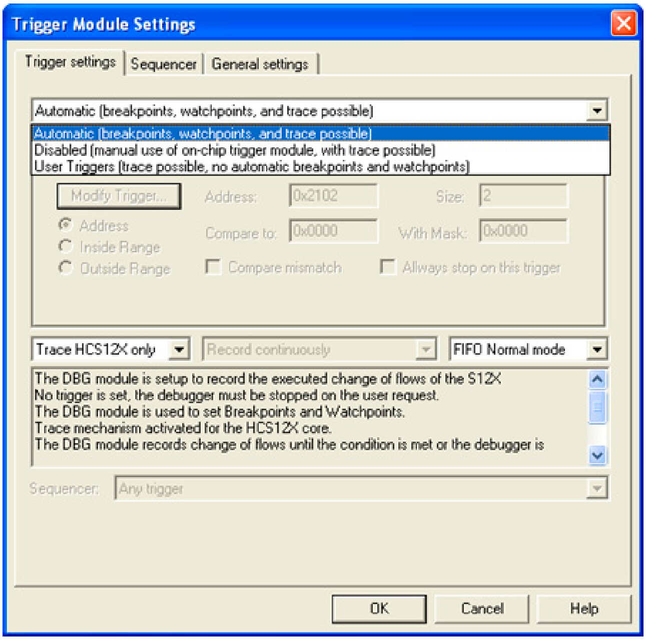
Table 17.11 Memory Access Triggers Available (*continued*)

Mode	Description
Instruction at Address A and Value on Data Bus Match	Use to set a trigger on a program instruction execution at Address A with an instruction opcode that matches a specific byte value at the Address A memory location. Uses **trigger B** address as a **match value** rather than an address location. If you select this trigger type using a context menu without setting the match value, an error message prompts for match value. Replaces the standard trigger editing dialog box with the Triggers Address Settings dialog box (see Figure 17.7). Match Value edit boxes replace Trigger Editing dialog.
Instruction at Address A and Value on Data Bus Mismatch	Use to set a trigger on a program instruction execution of a non-matching byte value at Address A memory location. Uses **trigger B** address as a **mismatch value** rather than an address location. If you select this trigger type using a context menu without setting the match value, an error message prompts for the match value. Replaces standard trigger editing dialog box with Triggers Address Settings dialog box (see Figure 17.7). Match Value edit boxes replace Trigger Editing dialog.
Capture Read/Write Values at Address B	Use to capture data used in a read or write access to address location specified by **trigger B**. Typically a data or memory address rather than program code address (program counter).
Capture Read/Write Values at Address B After Access at Address A	Use to capture data used in a read or write access to address location specified by **trigger A** and **trigger B**. Typically a data or memory address rather than program code address (program counter). Capture of values at Address B begins only after accessing Address A.

On-Chip DBG Control Options

The user can modify or control all of the available options using the Trigger Module settings window, with its three tabs, Trigger Settings, Sequencer and General Settings.

Figure 17.33 Trigger Module Settings Window - Trigger Settings Tab



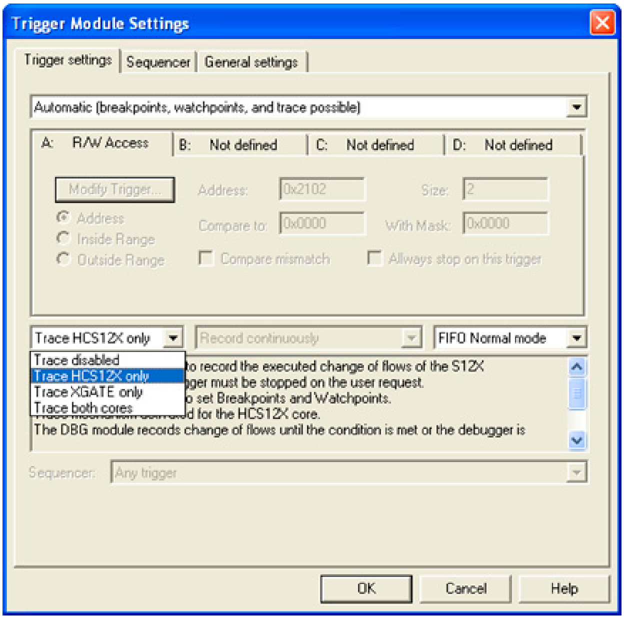
Switch DBG modes by displaying the list menu, as shown above, and selecting any one of the three modes. With some modes, some of the Trigger Settings tab options are not available.

Use the four sub-tabs of the Trigger Settings tab of this window, with their text fields, radio buttons and check boxes, to set up the individual triggers.

Trace Setup Control

Use the options made available in the Trace list menu of the Trigger Settings tab to set up DBG trace in the Trace component window. Configure DBG Trace to record HCS12X only, XGATE only, or both cores.

Figure 17.34 Trigger Settings Tab - Trace Drop List



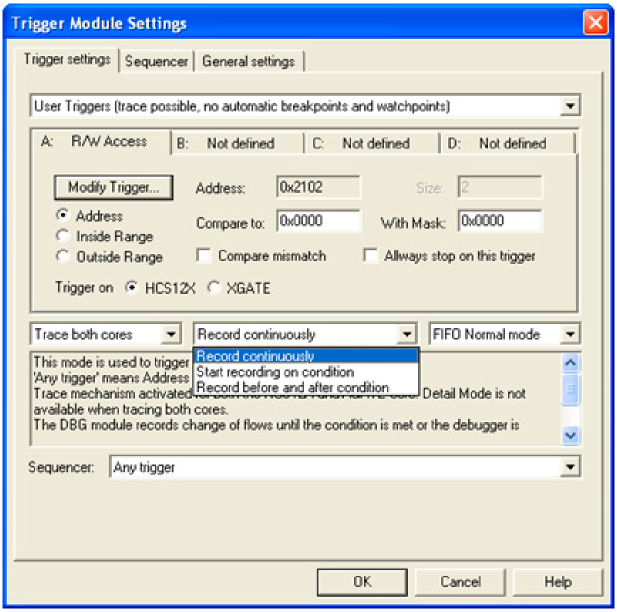
The list options allow you to:

- Disable trace
- Trace the HCS12X core only
- Trace the XGATE core only
- Trace both cores.

Program Code Change of Flow Recording

The program code change of flow options are available for Instruction and Memory Access triggers and controlled through the Trigger Module Settings window Trigger Settings list menu.

Figure 17.35 Change of Flow Recording Control



Recording can start either when the sequencer final state is reached and then stop, stop when the sequencer final state is reached, or provide trace information both before and after the final state is reached. Table 17.12 describes the available recording options.

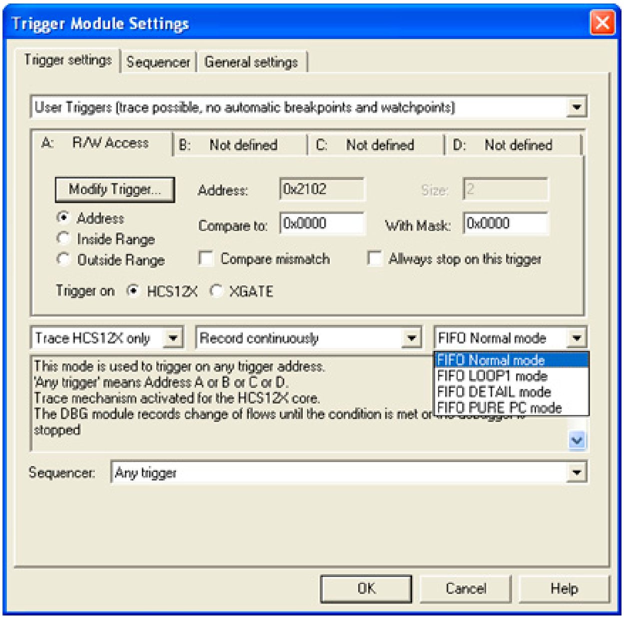
Table 17.12 Recording Options

Option	Description
Record continuously	Starts recording program flow information immediately after run. Halts the processor/debugger when capture buffer is full.
Start recording on condition	Starts recording program flow information on trigger condition match. Halts the processor/debugger when capture buffer is full.
Record before and after condition	Starts recording program flow information on trigger condition match. Does not halt the processor/debugger on trigger condition match; halts when capture buffer is full.

Data Recording Control

The data recording options are available for Capture triggers only. Select these options from the list box in the Trigger Settings tab.

Figure 17.36 Data Recording Control



FIFO Normal and LOOP1 modes record the change of flow (allowing program rebuild), while DETAIL mode only records data accesses. In Pure PC mode the debugger decodes pure PCs only. Table 17.13 describes the available data recording options.

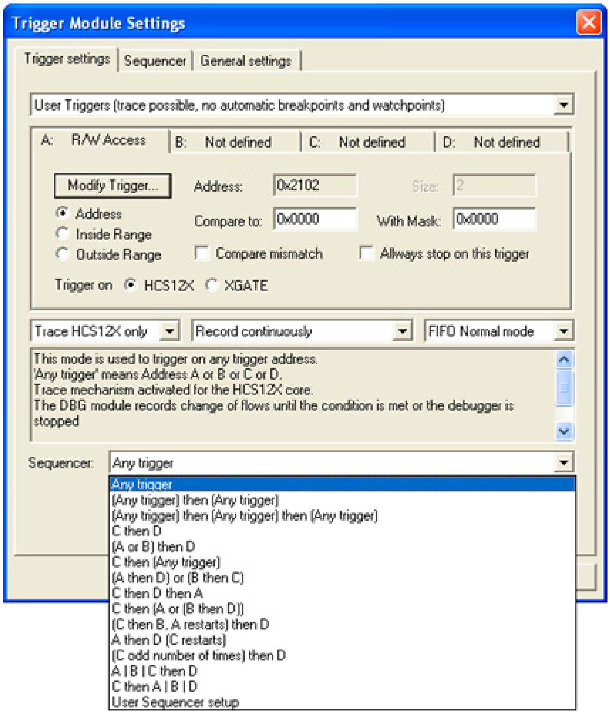
Table 17.13 Data Recording Options

Option	Description
FIFO normal mode	Continuously records data accesses. Halts the processor/ debugger when capture buffer is full.
FIFO LOOP1 mode	Continuously records data accesses. Does not halt the processor/ debugger when capture buffer is full.
FIFO DETAIL mode	Continuously records data access details. Halts the processor/ debugger when capture buffer is full.
Pure PC mode	Default recording mode when available. Debugger decodes pure PCs recorded by module. Does not perform program flow rebuild.

Trigger Sequence Control

Select the Trigger Sequence control options from the list box in the Trigger Settings tab of the Trigger Module Settings window shown below. You can also choose or change the DBG sequence in the Sequencer tab of the Trigger Module Settings window. Version 3 and later devices incorporate an extended trigger sequencer, with preset sequences providing more complex trigger combinations.

Figure 17.37 Trigger Sequence Control



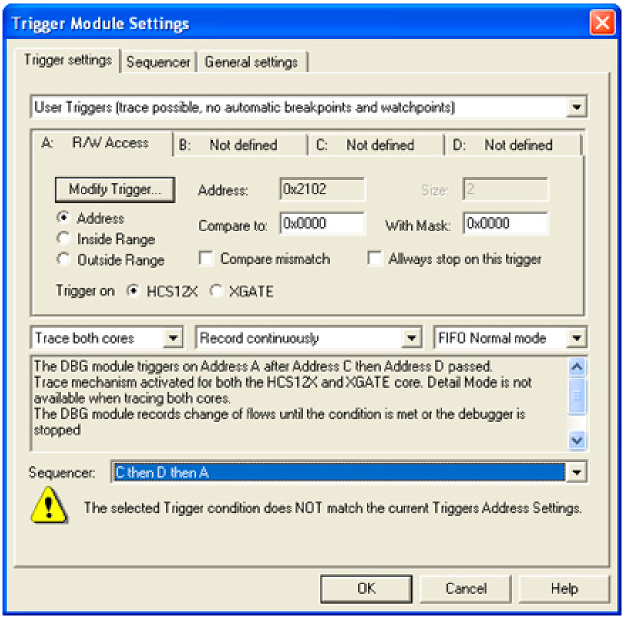
Display Information

A large gray box provides dynamic information about the current triggers and selected options.

As context-sensitive menus only display triggers matching the number and type of currently set triggers, the debugger checks the current trigger settings against the trigger mode. If one or more triggers do not match the trigger mode selection, a warning icon and message appears on the bottom of the dialog.

The display field in Figure 17.38 shows that the **Memory Write Access** type does not match the Instruction trigger type selected in the list.

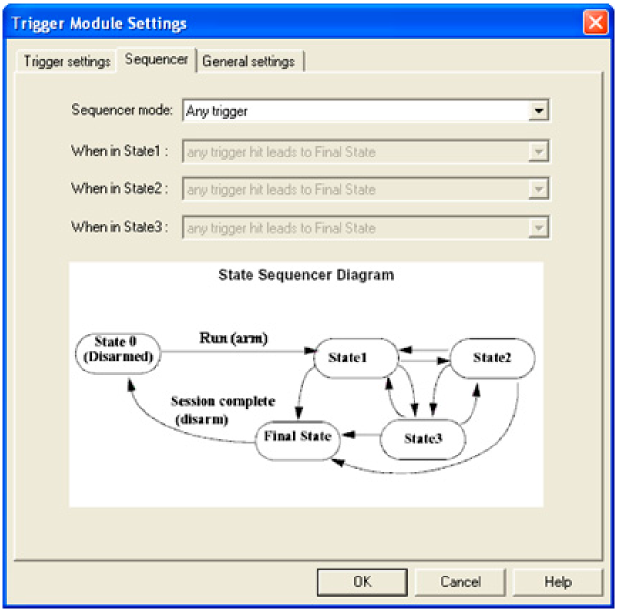
Figure 17.38 Trigger Settings Tab Information



Sequencer Tab

You can choose or change the DBG sequence in the Sequencer tab of the Trigger Module Settings window. The sequencer tab reflects transitions that occur when you select a predefined sequencer mode.

Figure 17.39 Trigger Module Settings Window - Sequencer Tab



You can modify the transitions in the Sequence tab only when the DBG is in User Sequencer setup mode.

General Settings Tab

The settings in the General Settings tab indicate the default settings of the DBG user interface (see Figure 17.40). Usually there is no need to change these settings. However, in some cases, you may wish to disable some automated background processes. The following checkboxes are available in this tab:

- Automatically analyze the FIFO content
- Disarm automatically the module when the debugger stops
- Protect DBG FIFO content from unexpected reads
- When starting, automatically step if a trigger is set at PC address (otherwise: warn)

Table 17.14 describes these options. Refer to Trigger Module Settings Window for additional information.

Figure 17.40 Trigger Module Settings Window - General Settings Tab

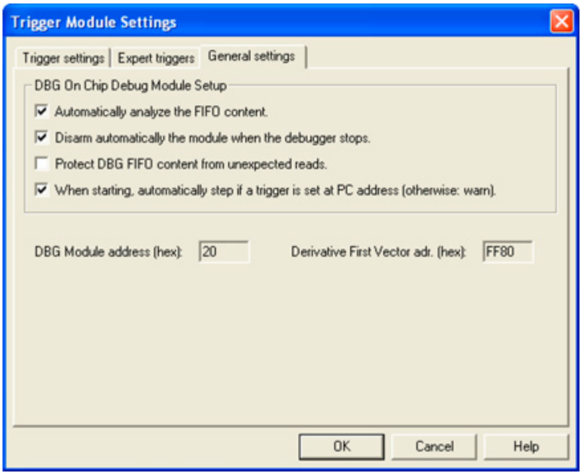


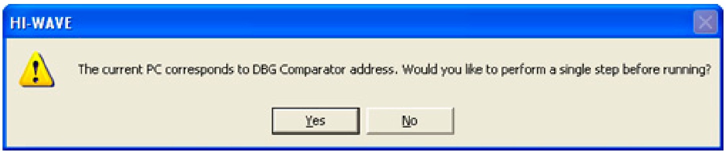
Table 17.14 On-Chip Debug Module Setup Options

Option	Description
Automatically analyze the FIFO content	When debugger halts with Trace component window open, debugger analyzes DBG module results and displays them in Trace window. If Trace window is closed, DBG user interface performs no result analysis except trigger flags reported in status bar. Clear to limit analysis to reporting trigger flags in status bar, even when Trace window is open.
Disarm automatically the module when the debugger stops	Target processor halt due to user break (not trigger) leaves on-chip DGB module armed. Check to disarm on-chip DGB module on halt to retrieve data from DBG FIFO. If clear, DBG FIFO/buffer information cannot be retrieved until module is disarmed.

Table 17.14 On-Chip Debug Module Setup Options

Option	Description
Protect DBG FIFO content from unexpected reads	Debugger retrieves FIFO data from **DBGFH-DBGFL** registers, performing several reads to retrieve entire buffer. When debugger halts, it may also read target processor memory at same location, reading the first FIFO data buffer and shifting the buffer, and corrupting user interface FIFO data. Enabling this option protects FIFO content from debugger reads.
When starting, automatically step if a trigger is set at PC address (otherwise: warn)	After encountering trigger, debugger must clear current trigger match condition and avoid being locked by trigger. Instruction triggers require a single step to escape. Check this option to step out of trigger automatically. When clear, a dialog box appears to validate stepping (see Figure 17.41).

Figure 17.41 Trigger Escape Dialog Box



Trace Component Window

Use the Trace component to display the internal database in the Trace window. Set up the context-sensitive menu from the connection (or the GDI DLL) using the component.

All debugger connections, including the DBG user interface, are synchronized with the Trace component.

It is not necessary to open the Trace component window to use the DBG user interface triggers. However, several triggers collect code program flow information or access data information. Open the Trace window from specific connection menus, from context menus, and from the DBG Support Status bar. Save this window in the debugger layout by pressing the debugger Save icon.

NOTE The debugger may run faster with the Trace component window closed, because this allows the debugger to discard the code program flow rebuild.

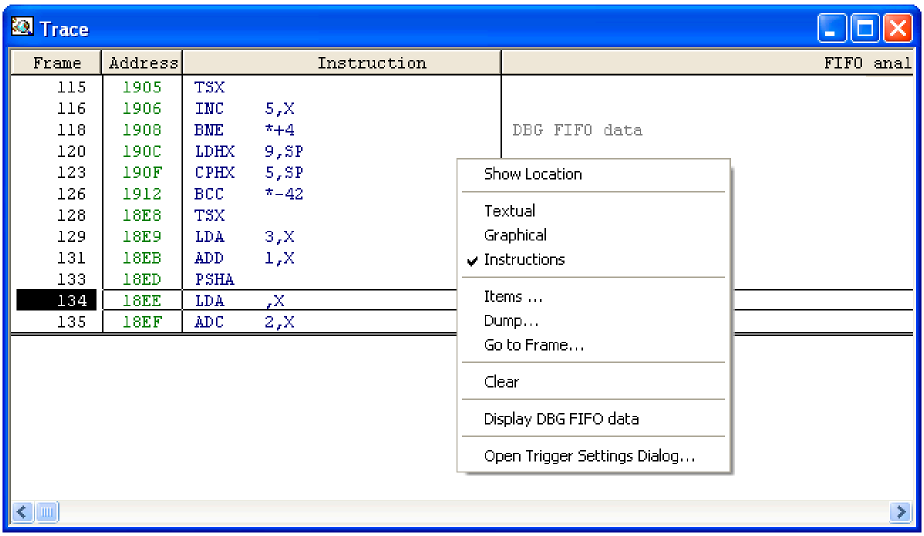
The Trace component window displays either instructions or recorded data, depending on the type of triggers activating the window.

Instructions Display

Using Instruction triggers and Memory Access triggers automatically activates this display mode. It is also the default display in Automatic mode. In this mode, the Trace component window displays four columns:

- **Frame**: This column shows a number representing an information item stored in the Trace component database.
- **Address**: This column shows the instruction program counter.
- **Instruction**: This column shows the code program flow instruction disassembly.
- **FIFO Analyze remark**: This column shows debugger information
  - **DBG FIFO data** means that the on-chip DBG module recorded this data
  - **Traced** means an item or instruction obtained by debugger or user single step or assembly step.
  - **Program flow rebuild gap** means that the debugger was unable to completely track the code program flow between two frames.

Figure 17.42 Trace Window - Context Menu Options

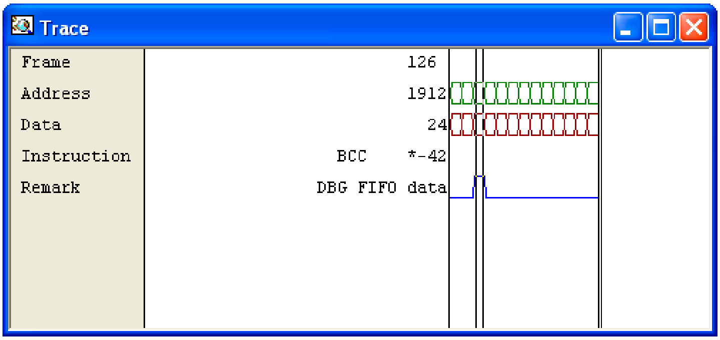


Selecting **Show Location** in the Trace window context-sensitive menu displays the frame matching source and assembly code in the Source and Assembly windows.

Graphical Display

Activate this display mode by selecting **Graphical** in the Trace window context-sensitive menu. It provides a graphical representation of the same information.

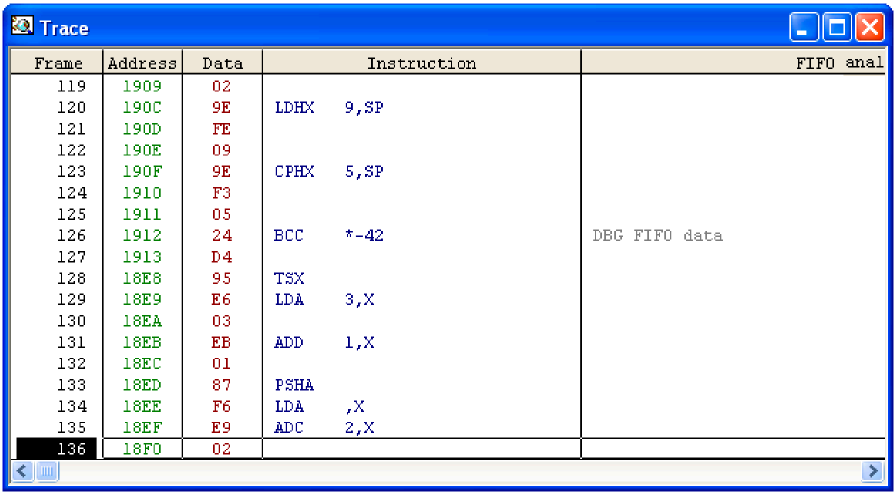
Figure 17.43 Trace Window - Graphical Display



Textual Display

Activate this display mode by selecting **Textual** in the Trace window context-sensitive menu, when using Instruction or Memory Access triggers. When using this mode, the DBG module does not record read or write accesses while program change of flow information is recorded. Textual display mode simply expands instruction assembly code in the Trace window.

Figure 17.44 Trace Window - Textual Display



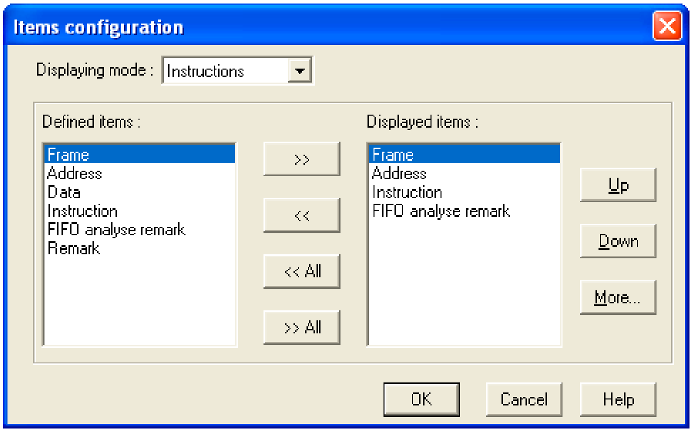
Trace

Frame	Address	Data	Instruction	FIFO anal
119	1909	02		
120	190C	9E	LDHX 9,SP	
121	190D	FE		
122	190E	09		
123	190F	9E	CPHX 5,SP	
124	1910	F3		
125	1911	05		
126	1912	24	BCC *-42	DBG FIFO data
127	1913	D4		
128	18E8	95	TSX	
129	18E9	E6	LDA 3,X	
130	18EA	03		
131	18EB	EB	ADD 1,X	
132	18EC	01		
133	18ED	87	PSHA	
134	18EE	F6	LDA ,X	
135	18EF	E9	ADC 2,X	
136	18F0	02		

Column Display and Moving

Select **Items** in the Trace window context-sensitive menu to open a configuration dialog to set up the columns to view in each display mode. You can open the **Displaying mode** list to make column display modifications in *Textual*, *Instructions* or *Graphical* mode. Use the right arrow to move items to the Displayed Items list, and the left arrow to hide the item. Moving the item Up in the list moves it to the left in the Trace component window. Select More for more options. Select OK to save your changes.

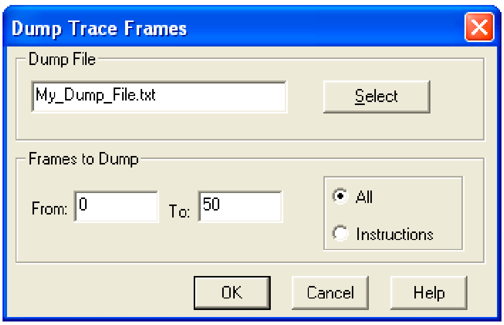
Figure 17.45 Items Configuration Dialog Box



Dumping Frames to File

Selecting **Dump** in the Trace window contest-sensitive menu opens a dialog that allows you to specify the number of Trace component frames to save, and the name of the text file to which to save the frames.

Figure 17.46 Dump Trace Frames Dialog Box



Go to Frame

Selecting **Go to Frame** in the Trace window context-sensitive menu opens a Search Frame dialog to allow you to look for a specific frame in the Trace window.

Clearing Frames

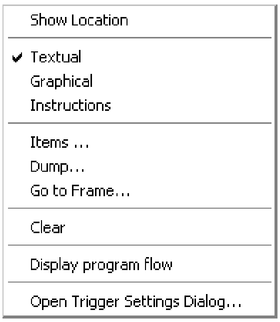
Selecting **Clear** in the Trace window context-sensitive menu flushes the frames in the Trace window. It also flushes the database in the background.

DBG Module FIFO/Buffer Display

This menu entry toggles between **Display DBG FIFO data** and **Display program flow**. Selecting **Display DBG FIFO data** in the Trace window contest-sensitive menu displays data information retrieved from the on-chip DGB module FIFO/buffer. Selecting **Display program flow** in the Trace window context-sensitive menu reverts the display back to the code program flow. The following columns appear in the Display DBG FIFO data window:

- **FIFO Depth**: This is a number representing the depth in the DBG/FIFO of the word data value. The first frame (Depth 1) is the oldest value.
- **DBG FIFO Data**: This is the word value retrieved from the DBG FIFO/buffer from DBGFH and DBGFL DBG on-chip module registers.

Figure 17.47 Trace Window - FIFO Display

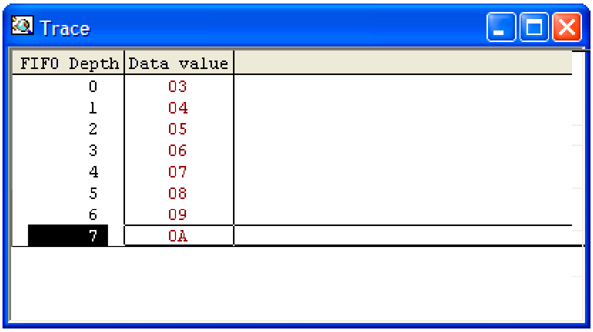


Recorded Data Display

Using Capture triggers automatically activates this display mode. The following columns appear in the Recorded Data Display window:

- **FIFO Depth**: A number representing the depth in the DBG/FIFO of the byte data value. The first frame (Depth 1) is the oldest value.
- **Data value**: The byte value retrieved from the DBG FIFO/buffer from the DBGFL DBG on-chip module register.

Figure 17.48 Trace Window - Recorded Data Display



Demonstration Mode Limitations

- During code program reconstruction, the Trace window displays a limited number of frames.
- Real-time code Profiling and code Coverage are disabled.
- Preset/Predefined Instruction, Memory Access or Capture Triggers are not available. Only Expert triggers can be set.

18

Debugging Memory Map

The Debugging Memory Map (DMM) is a software manager for all debugger accesses to device or chip memory and also for memory data caching.

The DMM provides a global approach for all different CPU families and cores, each family having its own method for memory access and its own on-chip memory layout and memory address range priorities.

The DMM gets all memory read and write calls from the debugger, and also uses the low-level function read/write primitives to call third-party cable drivers required for devices such as BDM pods and Monitors.

The debugger provides the DMM with core-specific read/write access methods, called **Types**, and core specific priority rules, called **Priority**, within the DMM GUI.

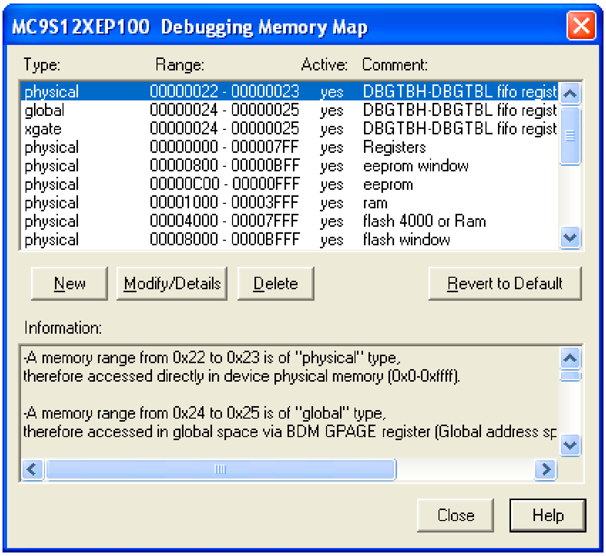
The DMM GUI allows you to change the memory access method at any time.

Debugging Memory Map GUI

The graphical user interface (GUI) is flexible and easy to use, and displays live diagnostics within the dialog. At any time, you can revert to the default (factory) setup. Most of the time, you do not need to edit or change settings within the GUI.

Open the DMM GUI by choosing the **Debugging Memory Map** option in the connection menu in the debugger main window. This opens the DMM Window.

Figure 18.1 Debugging Memory Map Window



The DMM GUI shows a list of memory address ranges, called **Modules** in this manual, defined to access device memory.

- The **Type** column displays the memory type for a given range, corresponding to the memory address range in the **Range** column.
- The **Range** column gives the memory address range.
- The **Active** column indicates whether the defined range is active, or mapped, by the DMM. If **No**, the DMM considers the range undefined.

NOTE The DMM considers all undefined memory ranges as inaccessible or unimplemented. The debugger displays some dashes (– –) in the Memory window in that case. The DMM NEVER attempts to read or write unimplemented memory.

- The **Comment** column contains information about the defined memory address range.

The scrollable **Information** window gives a general diagnostic of the DMM. This diagnostic has less information then the edit mode diagnostic.

Clicking the **New** button opens the Debugging Memory Map dialog box to create a new memory address range.

Clicking the **Modify/Details** button opens the Debugging Memory Map dialog of the selected memory address range to modify it. More memory range information appears in the dialog, and an enhanced diagnostic is also displayed.

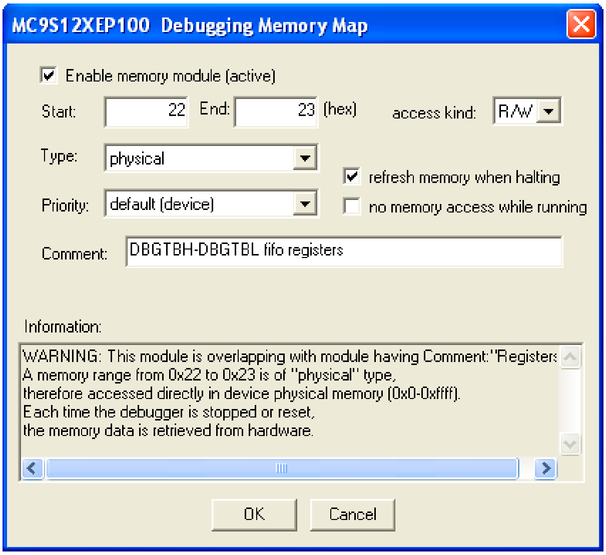
Clicking the **Delete** button leads to memory range removal, after a warning dialog.

Cliking the **Revert to default** button removes (after a warning dialog) the current setup (usually saved in the current project) and retrieves the default (factory) setup from an internal database.

Enabling the Memory Module and Changing the Memory Range

Figure 18.2 shows the DMM Memory Map dialog box.

Figure 18.2 Debugging Memory Map Dialog Box



The **Enable memory module** option checkbox maps the module/memory range in the debugger. Unchecking this option makes the module completely transparent for the DMM and the debugger.

The **Start** edit box contains the first address of a memory range and the **End** edit box contains the last address of a memory range.

Range boundaries are always limited to an overlapped range with a higher priority. For example, if two bytes are defined in a range which overlaps another range, these two bytes are accessed using the type and rules of this 2-byte range. The memory on both sides of these two bytes is accessed using the type and rules of the overlapped range.

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

Access kind

The **Access Kind** list menu provides a way to indicate that the memory range is read/write (R/W), read only, write only or none of these.

- When defined as read only, the range is never written by the debugger.
- When defined as write only, the range is never read by the debugger.
- When defined as none, the range is never read or written by the debugger. This is internally equivalent as not defining the range in the DMM dialog.

Access Size

When available, the **Access Size** list menu provides a way to define if the memory range is accessed as byte (1), short (2) or long (4).

NOTE The memory range must be size aligned. For example, a module defined with access size **2** must start with an even address and finish on an odd address. A module defined with access size **4** must start with an address with the least significant byte in 0, 4, 8, C, and finish with an address with the least significant byte in 3, 7, B, F.

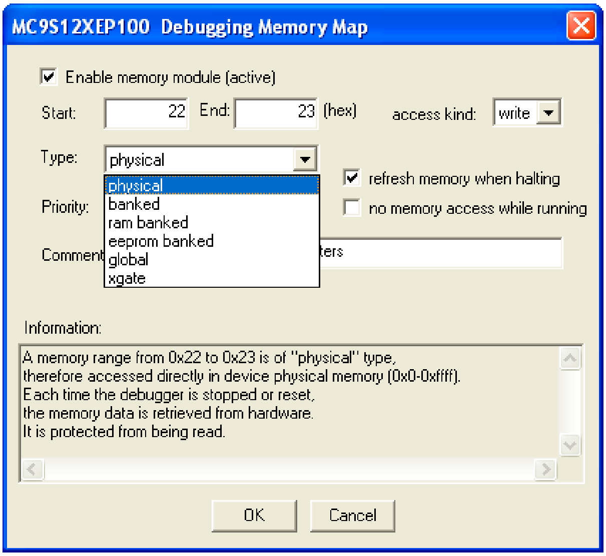
NOTE A memory range overlapping (in priority) another memory range can only have the same or a higher access size.

Types

The **Type** list menu provides all kinds of memory type available for the processor displayed in the title bar of the dialog. For some connections, the CPU core might be displayed instead of the processor name.

Types are internal rules to read and write a kind of memory. For example, the HCS12 banked type requires, first, setting a register called PPAGE to read the memory, then restoring this value as it was before reading. Also this banked type does not physically provide a memory access while running. Memory access while running is possible in physical memory (RAM, registers).

Figure 18.3 Debugging Memory Map Dialog Box - Type List Menu



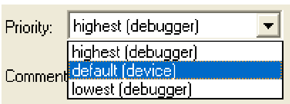
NOTE CPU core-specific memory types and Priorities are listed at the end of this manual section.

Priorities

The **Priority** list menu provides all of the memory overlap priority available for a type of processor core. The debugger can have a higher priority (highest debugger) to set up an upper address range that can overlap an on-chip address range, thus making a debugger display filter (for a Memory window), for example, when creating a **No read access while running** memory address range.

A *Flat* memory architecture (i.e. without memory blocks moving feature) provides the following Priority list menu (e.g. HCS12X core):

Figure 18.4 HCS12X Core overlap priorities



The default is that the CPU sees all memory blocks with the same priority.

Memory Read Caching

The **Refresh memory when halting** option controls the debugger memory cache. When this option is checked, internal images/caches of memory data are always deleted and the data is always retrieved from hardware when required by the debugger. When unchecked (the default for Non-Volatile Memory areas), the DMM keeps a copy of the data and does not read/retrieve the data from hardware until next application loading/programming.

NOTE Each declared memory address range in the GUI has its own private code cache monitored by the DMM.

The `DMM CACHINGOFF` command can fully disable the caching feature for the entire DMM (i.e., for all defined memory ranges). The `DMM CACHINGON` command re-enables the caching feature.

Access While Running

Use the **No memory access while running** option to discard debugger access to a memory range which the debugger can typically access while running. Use this feature to protect on-chip I/O Register flags from being triggered by debugger memory reads due to display refreshes.

Remarks

It is possible to create as many memory ranges of any size as desired, down to a single byte.

Deleting Default/Factory ranges generates warning dialogs. Some settings are required for the debugger to debug and removing ranges leads to erroneous debugging information.

All GUI settings can be done by debugger commands.

Settings and DMM changes are saved in the current user project. You can always restart from default by clicking the **Revert to Default** button.

You can disable automatic DMM range remapping with a debugger command.

The default settings are retrieved from a complete database describing each derivative, or, in some cases, describing the CPU core (when not necessary to go to derivative level).

CPU Core Priorities and Types

This section describes the various memory priorities and types for the different CPU cores.

HC12 (CPU12) Core

The following priorities and types are specific to the HC12 (CPU12) core:

Priorities

- All derivatives except MC68HC12A4
  - **Highest (debugger)**: a high debugger priority that you can use or define for the debugger; typically to protect a memory area from being read.
  - **Internal register space**: refer to device specifications.
  - **RAM memory block**: refer to device specifications.
  - **EEPROM memory block**: refer to device specifications.
  - **On-chip Flash-EEPROM**: refer to device specifications.
  - **Remaining external space**: refer to device specifications.
  - **Lowest (debugger)**: a low debugger priority that you can use or define for the debugger typically to protect a memory area from being read. This priority is of poor usage but can still be used for display purposes on chip unimplemented memory range.
- MC68HC812A4 derivative
  - **Highest (debugger)**: a high debugger priority that you can use or define for the debugger. Typically used to protect a memory area from being read.
  - **Internal register space**: refer to MC68HC812A4 specifications.
  - **RAM memory block**: refer to MC68HC812A4 specifications.
  - **EEPROM memory block**: refer to MC68HC812A4 specifications.
  - **E space (external)**: refer to MC68HC812A4 specifications.
  - **CS space (external)**: refer to MC68HC812A4 specifications.
  - **P space (external)**: refer to MC68HC812A4 specifications.
  - **D space (external)**: refer to MC68HC812A4 specifications.
  - **Remaining external space**: refer to MC68HC812A4 specifications.
  - **Lowest (debugger)**: a low debugger priority that you can use or define for the debugger typically to protect a memory area from being read. This priority is of little value but can still be used for display purposes on chip unimplemented memory range.

Types

- All derivatives:
  - **Read protected**: legacy, replaced by **physical** memory type, with “Write Only” access kind.
  - **Write protected**: legacy, replaced by **physical** memory type, with “Read Only” access kind.
  - **R/W protected**: legacy, replaced by **physical** memory type, with “None” access kind.
  - **Physical**: this sets the memory range as physical, (i.e. with linear 16-bit address bus access) as performed by the CPU when reading and writing the on-chip memory.
- Additional Types for MC68HC812A4 Derivatives:
  - **Extra banked**: this type handles the EPAGE register when accessing the Extra page banked data, typically data in $400-$7FF window.
  - **Banked**: this type handles the PPAGE register when accessing the Program page banked data, typically program code in $8000-$BFFF address range window.
  - **Data banked**: this type handles the DPAGE register when accessing the Data page banked data, typically variables in $7000-$7FFF address range window.
- Additional Types for MC68HC912xx128 Derivatives:
  - **Banked**: this type handles the PPAGE register when accessing the Program page banked data, typically program code in on-chip Flash in $8000-$BFFF address range window.

HCS12 Core

The HCS12 core provides memory block moving, with overlap priorities. These overlap rules are handled by the DMM, and rules handle the Memory Expansion Registers (MER), i.e., INITRM, INITRG, INITEE.

On each debugger halt, the MER Registers are read, and if necessary, the DMM offsets internal range addresses.

NOTE The debugger does not poll the MER registers while running. Also the debugger performs remapping only on factory-defined memory range, not on user-defined memory ranges.

Execute the `DMM HCS12MERHANDLINGOFF` command to disable the MER Registers tracking. Execute the `DMM HCS12MERHANDLINGON` command to re-engage this feature.

NOTE Factory/default setup protects the HCS12 DBG12 FIFO Registers to reserve DBG12 FIFO Reading for the debugger DBG interface. Removing this protection causes incorrect program flow rebuild.

Priorities

- **Highest (debugger)**: a high debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read.
- **Internal register space**: refer to device specifications.
- **RAM memory block**: refer to device specifications.
- **EEPROM memory block**: refer to device specifications.
- **On-chip Flash-EEPROM**: refer to device specifications.
- **Remaining external space**: refer to device specifications.
- **Lowest (debugger)**: a low debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read. This priority is of poor usage but can still be used for display purposes on chip unimplemented memory range.

Types

- **Read protected**: legacy, replaced by **physical** memory type, with “Write Only” access kind.
- **Write protected**: legacy, replaced by **physical** memory type, with “Read Only” access kind.
- **R/W protected**: legacy, replaced by **physical** memory type, with “None” access kind.
- **Physical**: this sets the memory range as physical, i.e. with linear 16-bit address bus access as performed by the CPU when reading and writing the on-chip memory.
- **Banked**: this type handles the PPAGE register when accessing the Program page banked data, typically program code in on-chip Flash in $8000-$BFFF address range window.
- **Registers**: This type cares if the I/O Registers block and Memory Expansion Registers change, including I/O Registers block moving.

HCS12X Core

These priorities and types are specific to the HCS12X core.

Priorities

- **Highest (debugger)**: a high debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read.
- **Default (device)**: default CPU visibility of the entire device/memory with a same priority, as no memory range can be moved to overlap another memory range.
- **Lowest (debugger)**: a low debugger priority that can be used by the user or defined for the debugger typically to protect a memory area from being read. This priority is of poor usage but can still be used for display purposes on chip unimplemented memory range.

Types

- **Read protected**: legacy, replaced by **physical** memory type, with “Write Only” access kind.
- **Write protected**: legacy, replaced by **physical** memory type, with “Read Only” access kind.
- **R/W protected**: legacy, replaced by **physical** memory type, with “None” access kind.
- **Physical**: this sets the memory range as physical, i.e. with **linear 16-bit address bus access as performed by the CPU when reading and writing the on-chip memory**.
- **Banked**: this type handles the PPAGE register when accessing the Program page banked data, typically program code in on-chip Flash in $8000-$BFFF address range window.
- **RAM banked**: this type covers accessing $1000-$1FFF RAM data window (the user application accesses via RPAGE) in global address space. Important: All accesses are cast by the DMM to global memory which should therefore be defined for the matching range.
- **EEP banked** or **D-flash banked**: these types cover accessing $800-$BFF EEPROM or D-flash data window (the user application accesses via EPAGE) in global address space. Important: All accesses are cast by the DMM to global memory which should therefore be defined for the matching range.
- **Global**: this type covers accessing of the global memory space via BDM GPAGE register (Global address space). The Memory window with Address Space set to Global displays the global space memory of the device.

- **xgate**: this type covers accessing of the XGATE memory space as the **XGATE** core would see it. The Memory window with Address Space set to XGATE displays the XGATE space memory of the device. When existing, the Flash/RAM XGATE memory split is internally evaluated by the DMM.

NOTE Factory/default setup protects the HCS12X DBG12X FIFO Registers to reserve the DBG12X FIFO reading for the debugger DBG interface. Removing this protection leads to incorrect program flow rebuild.

Except physical and protected access types, all types are routed to Global memory when **reading** from the device. However, for Non-Volatile Memory programming reasons, **EEP banked** and **banked** types are routed to logical paged when writing to the device.

DMM Commands

All DMM GUI settings can be done by debugger command line commands.

Debugging Memory Map Manager Command Set

The commands provide the possibility to fully script the debugging device memory mapping. However, the usage of these commands should be limited to special debugging purposes, as the default mapping is typically sufficient, and a script setup being complex and possibly leading to debugger disfunctions.

List of Commands

```
DMM
DMM ADD <parameters>
DMM DEL <module handle>
DMM SAVE <mcuid>
DMM DELETEALLMODULES
DMM RELEASECACHES
DMM CACHINGON|CACHINGOFF
DMM WRITEREADBACKON|WRITEREADBACKOFF
DMM HCS12MERHANDLINGON|HCS12MERHANDLINGOFF
DMM OPENGUI [mcuid]
```

```
DMM SETAHEADREADSIZE <front size when halted> <back size
when halted> <front size when running> <back size when
running>
```

For detailed descriptions of the available DMM commands, see DMM Commands.

19

Flash Programming

Writing to Flash modules, EEPROMs, or other non-volatile memory modules requires special algorithms. Before you write to Flash devices, you must erase them. Many Flash devices need initialization to become accessible; some devices may need write protection removed.

This chapter explains The Non-Volatile Memory Control (NVMC) utility, an extension component that lets you control the on-chip Flash devices for all Debugger connections.

The NVMC utility is very flexible. This flexibility comes from a generic Debugger component, which calls a graphical user interface, then loads an MCU-specific module. The module provides the appropriate information (such as structure, access algorithms, and location) for that MCU.

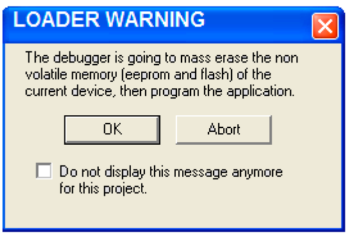
The NVMC utility lists all non-volatile memory devices, indicating their structure, state, and location. You can change the state (enabled/disabled, blank, programmed, protected/unprotected) and program data into the modules.

Automated Application Programming

The debugger can program an application without making use of the NVMC dialog/GUI, which remains useful for specific operations only. Currently, CodeWarrior projects created with the wizard may be programmed or flashed immediately. The debugger displays a warning dialog to get user acceptance before mass erasing then programming the application.

Use the Flash-specific command (`FLASH NOUNSECURE`) to incorporate device security byte programming in user code.

Figure 19.1 Flash Programming Loader Warning Dialog Box



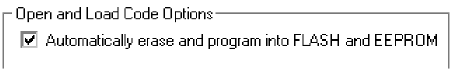
Select the **OK** button to launch background Flash commands to arm programming, load/program an application file, then disarm programming.

Check the **Do not display** checkbox to remove the Warning message for the current project (saved in project under the project variable: `AEFWarningDialog=FALSE`).

Setup

The Open and Load Code (Executable File) dialog box opens when you choose the **Load** menu entry in the debugger main window's connection menu.

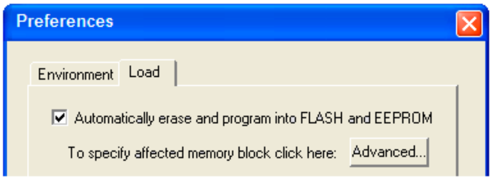
Figure 19.2 Open and Load Code Options Dialog Box



Checking this checkbox engages the automated device mass erasing and application programming into non-volatile memory, i.e., Flash and/or EEPROM.

To set this option permanently, use the **Load** tab in the debugger Preferences window (**File > Configuration**).

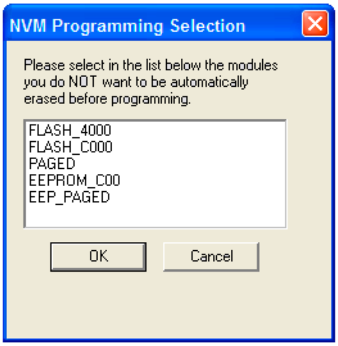
Figure 19.3 Preferences Window - Load Tab



Advanced Options: Erase Prevention

Clicking the **Advanced** button in the **Load** tab of the debugger Preferences window opens the NVM Programming Selection list box.

Figure 19.4 NVM Programming Selection List Box



The list box lists all the Non-Volatile Memory modules registered by the debugger for the current selected processor device.

Clicking once on a line selects an item (highlighted in blue) and clicking the line again deselects it.

Erasing is skipped for all selected modules. If all modules are selected, the debugger simply programs the application without erasing non-volatile memory on the device.

CAUTION The debugger ignores pre-programmed modules and the user is responsible for reprogramming limitations, risks and impossibility. However, the debugger displays a warning message when a programmed (i.e. not blank) “not automatically erased” module is going to be written. You can disable the displayed warning message.

TIP When available on-chip, EEPROM type modules are by default **not** selected for automatic erasing.

The **NVM Programming Selection** list box does not give many details about the listed blocks. Type the `Flash` command in the Command window to display more information, or open the Non-Volatile Memory Control dialog box.

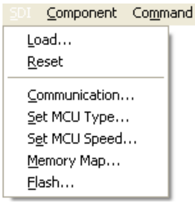
The **NVM Programming Selection** list box is closely associated with the `Flash AEFSKIPERASING` command of the debugger.

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

NVMC Graphical User Interface

The NVMC utility is integrated into the Debugger, as an extension of certain debugger connections. If the NVMC utility is available, your connection menu includes a **Flash** selection, as shown below.

Figure 19.5 SDI Connection Menu Options



Modules and Module States

If an on-chip module consists of several independent blocks, the NVMC dialog box might list all of these blocks. Typically the NVMC groups all non-volatile on-chip blocks under one single listed module, separates relevant and important non-volatile memory blocks (like mirrored non-banked memory range), and provides individual/selective modules for the individual modules.

NOTE See Hardware Considerations for more information about the Flash modules of your CPU derivative.

Table 19.1 describes module states which may appear in the NVMC dialog box list.

Table 19.1 NVMC Module States

State	Description
Enabled	Currently active on the chip. It is possible to read (as a ROM) or program an enabled module.
Disabled	Currently inactive, so programming and reading are not possible. Normally, you enable or disable a module by setting/clearing a flag in a special register. Some modules cannot be disabled.
Blank	Empty of code. You can program its full address range. Each blank byte contains the value 0xFF or 0x00, depending on hardware.
Programmed	Partially programmed (not all bytes contain 0xFF or 0x00). You must keep track of the areas still available for programming, if any.

Table 19.1 NVMC Module States

State	Description
Protected	Partially protected from erasure or programming. Normally, you protect a module by setting/clearing a flag in a special register. Some modules can never be protected.
Unprotected	Can be erased and programmed.

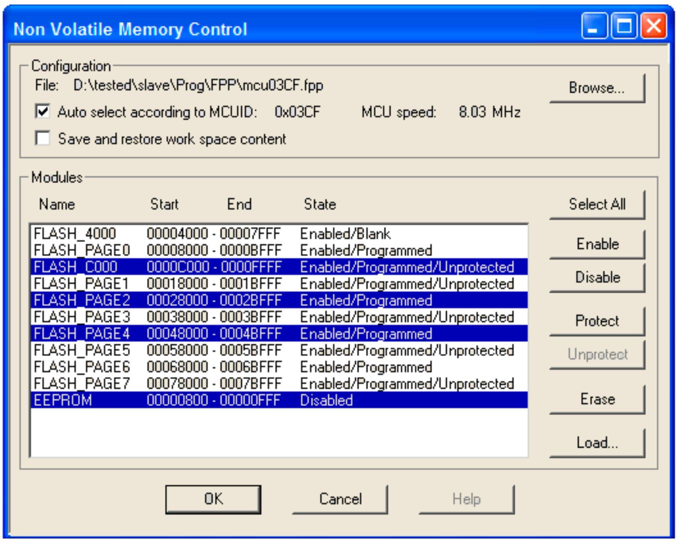
To select a module or other list item, left click the module. To deselect a module, click the **<Ctrl>** key and left click. For multiple selections or deselections, use the Shift key.

NVMC Dialog Box

The NVMC dialog box lists all the Flash or EEPROM modules of a CPU derivative. Depending on the derivative, there may be one or multiple on-chip Flash modules.

NOTE The dialog box does not have a **Select** or **Deselect** button, as you merely click on a module in the list to select it. Selecting and deselecting are not automatic from the command line. Before you use the command line to perform any operation on a module, you must use the `SELECT` command to select the module.

Figure 19.6 Non Volatile Memory Control Dialog Box



For each block, the dialog box has a line composed of the following fields:

- **Name** — the module name.
- **Start** — the module start address.
- **End** — the module end address.
- **State** — the modules states, such as *disabled, enabled, blank, programmed, protected, unprotected.*

Possible state combinations are:

- **Bad Device** (the interface could not detect a correct device)
- **Disabled** (one or all modules are disabled)
- [Enabled] / <Blank | Programmed> / [Unprotected | Protected]

The NVMC dialog box displays only meaningful states. For example, it displays *Enabled* only if it is possible to *disable* a module. It displays *Unprotected* only if it is possible to *protect* a module.

The Configuration group identifies the current `.FPP` parameter file. This group also includes the **Auto select according to MCUID** checkbox; the Configuration: FPP File Loading section explains this option.

The second checkbox of the Configuration group is **Save and restore workspace content**. If this checkbox is clear, Flash programming applications overwrite any data in RAM. To save the current RAM data, check this box. Saving RAM data slows down the NVMC; checking this checkbox is equivalent to entering the `SAVECONTEXT` and `LOADCONTEXT` commands.

Flash Module Handling

Flash parameter files (which have the extension `.FPP`) contain MCU-specific parameters, as well as programs to handle internal Flash modules. See Configuration: FPP File Loading for additional information about `.FPP` files. The `.FPP` files also include code-applet descriptions of Flash operations.

You also may use the Command Line component to handle Flash operations. The NVMC Commands explains the corresponding commands.

The NVMC dialog box has buttons for commands you can apply to each block. These buttons are dynamic: active if the operation is possible for at least one selected item, disabled if the operation is not possible. Table 19.2 describes these buttons.

Table 19.2 NVMC Dialog Box Buttons

Button Name	Associated Action
Select All/ Unselect All	Selects all modules in list box. When clicked, button changes to **Unselect All**. Click to deselect all current selections.
Enable/Disable	Enables all selected modules currently disabled. **Disable** disables all selected modules currently enabled. Ability to enable or disable Flash module depends on MCU features and context.
Protect/Unprotect	Protects all selected modules currently unprotected. **Unprotect** unprotects all selected modules currently protected. Ability to protect or unprotect a Flash module depends on MCU features and context. For some MCUs, protection is possible only for Boot section and boot routines, not entire module. See Hardware Considerations.
Erase	Removes programming from all selected Flash modules by assigning the value 0xFF or 0x00 to each byte. Changes module status to Blank. If all selected modules are already blank, **Erase** is disabled.
Load	Arms all selected modules, executes a `LOAD` command, then disarms modules. Click **Load** without selecting any Flash modules to make the NVMC utility select and load all modules. Click on a module to select or use **Select All**/**Unselect All** buttons to adjust selection. Selecting and unselecting are not automatic from the command line. Before using command line to perform any operation, use `SELECT` command to select module. See FLASH.

MCU Speed Information

The displayed MCU speed is the device bus speed/clock sensed by the Flash Programmer, the same value as the one returned by the `FLASH` command.

CAUTION A non-relevant displayed speed is symptomatic of a Flash Programmer diagnostic problem. In that case, close the dialog, check the hardware and reset the connection.

Configuration: FPP File Loading

When the dialog box is open, the NVMC utility loads the `.FPP` configuration file according to this algorithm:

1. The utility reads the `NV_PARAMETER_FILE` entry from the connection-specific section of the `project.ini` file. `[Freescale ESL]` is a connection-specific section.

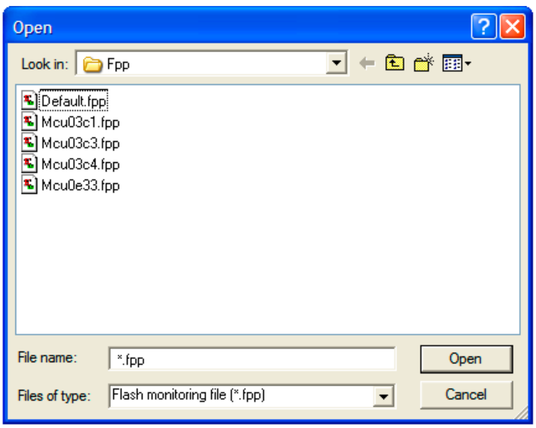
   Example:

   ```
   [Freescale ESL]
   NV_PARAMETER_FILE=C:\MYINSTALL\PROG\FPP\mcu03C4.fpp
   ```

2. If the utility retrieves a valid `.FPP` file name, it loads the file.
3. If the utility cannot find a valid `.FPP` file name, it displays an appropriate error message.
4. If the utility does not find an entry, or if it finds an empty entry, the utility automatically checks the **Auto select according to MCUID:** checkbox. Then the utility loads the parameter file from the `\FPP` subdirectory of the installation, according to the MCUID.
5. If the utility finds a file that has the wrong format, it displays an appropriate error message.
6. The utility always displays the MCUID, if the ID is available from the connection.

Another way to load an `.FPP` parameter file is by clicking the **Browse** button. This brings up a standard **Open** dialog box, which you can use to select the file. When you do so, the **Open** dialog box disappears, and the NVMC utility loads the file, automatically clearing the **Auto select according to MCUID:** checkbox. In case of any error during loading, the utility displays an appropriate message.

Figure 19.7 Open Dialog Box



If you check the **Auto select according to MCUID:** checkbox, the NVMC utility searches for and loads the corresponding `.FPP` parameter file.

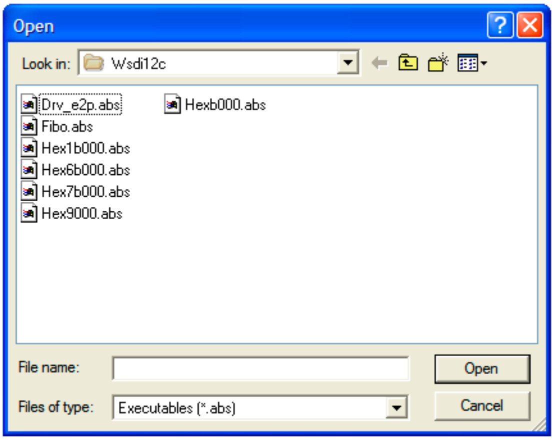
Click the **OK** button to close the NVMC dialog box. If the **Auto select according to MCUID:** checkbox is clear, the NVMC utility saves the name of the selected configuration file under the `NV_PARAMETER_FILE` entry of the `project.ini` file. If you check this checkbox, the utility does not save the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

Click the **Cancel** button to close the dialog box without saving changes.

Loading an Application in Flash

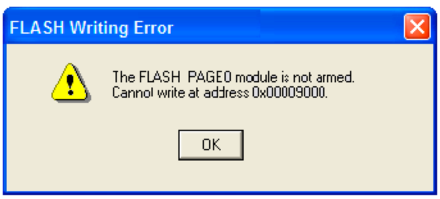
The **Load** button and the **Load** selection of the connection-specific menu function identically. Using either of these controls brings up the Load Executable File dialog box, which lets you select the file to be loaded. The Load Executable File dialog box lists the executable files that relate to blocks selected in the NVMC dialog box.

Figure 19.8 Load Executable File Dialog Box



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

Figure 19.9 FLASH Writing Error Message Box



This means that you tried to load a program into an unselected section. The NVMC utility's selecting/unselecting feature reduces the risk of overwriting, erasing, or unprotecting valuable data.

Preparing and Loading an Application

To prepare an application and load it into Flash, use either:

- The NVMC dialog box, explained in the NVMC Dialog Box section
- Flash commands within a command file. Connection-Specific Commands explains these commands.

If necessary, link your application with the appropriate memory model. The example below shows a `.PRM` file for an HC12DG128 application. The default ROM is in pages 2 and 4; the application uses the banked memory model. Make sure that your code location is within a Flash address range.

Listing 19.1 Loading an Application in Flash

```
LINK my_appli.abs
NAMES my_appli.o ansib.lib start12b.o END
SECTIONS
  MY_RAM = READ_WRITE 0x2010 TO 0x23FF;
  MY_ROM = READ_ONLY  0xC000 TO 0xFEFF;
  PAGE_2 = READ_ONLY 0x28000 TO 0x2BFFF;
  PAGE_4 = READ_ONLY 0x48000 TO 0x4BFFF;
PLACEMENT
  _PRESTART, STARTUP,
  ROM_VAR, STRINGS,
  NON_BANKED, COPY              INTO  MY_ROM;
  DEFAULT_RAM                   INTO  MY_RAM;
  MyPage, DEFAULT_ROM           INTO  PAGE_2, PAGE_4;
END
STACKSIZE 0x50
VECTOR ADDRESS 0xFFFE _Startup /*set reset vector IN FLASH on _Startup
*/
```

Follow the loading command example in Connection-Specific Commands or follow these instructions:

1. From the Debugger menu bar, open the connection-specific menu (such as `SDI`). Select **Flash** — the NVMC dialog box appears.
2. If you are sure about the absolute location of your application, you do not need to select a module. But if you program in a protected area (boot block), make sure that the matching module is unprotected.
3. Click the **Load** button — the NVMC utility selects all modules and opens the Load Executable File dialog box.
4. Select the `.ABS` file to be loaded into Flash. Loading begins and a progress bar appears. When loading is finished, the NVMC dialog box displays the new state of the modules.
5. This completes loading. You can close the NVMC dialog box and run your application. For some hardware, however, you first must do a connection reset, by clicking the reset button of the Debugger.

Hardware Considerations

This section consists of hardware-specific information about current `.FPP` files for HC12 (CPU12) CPU devices and HCS12 and HCS12X CPU devices.

NOTE The Flash Programming release note, in the on-line documentation of your toolkit installation, contains the latest information about `.FPP` files.

HC12 (CPU12) CPU Devices

The HC12B32, the HC12D60, and the HC12DG128 CPU devices and Flash module information appears below.

HC12B32

- **fpp file name:** `mcu03c1.fpp`
- **Number of Flash modules:** 1
  - applet code currently not relocatable, loaded at 0x800, using 0x400 bytes.

Table 19.3 HC12B32 Flash Module Details

Module Name	Module Number	Remarks
FLASH_B32	0	32 Kilobytes Flash located in 0x8000-0xFFFF or 0x0000-0x7FFF (both handled according to MAPROM bit in MISC register). Boot sector unprotectable/protectable (2 Kilobytes in range 0xF800-0xFFFF or 0x7800-0x7FFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register). Flash enable/disable via ROMON bit in MISC register.

HC12D60

- **fpp file name:** `mcu03c3.fpp`
- **number of Flash modules:** 2
  - Applet code currently not relocatable, loaded at 0x400, using 0x400 bytes.

Table 19.4 HC12D60 Flash Module Details

Module Name	Module Number	Remarks
FEE28	0	28 Kilobytes Flash located in 0x1000-0x7FFF or 0x9000-0xFFFF (both handled, according to MAPROM bit in MISC register). Boot sector protection off/on (8 Kilobytes in range 0x6000-0x7FFF or 0xE000-0xFFFF). Use BOOTP bit in FEE28MCR register and LOCK bit in FEE28LCK register. Flash enable/disable using ROMON28 bit in MISC register.
FEE32	1	32 Kilobytes Flash located in 0x8000-0xFFFF or 0x0000-0x7FFF (both handled, according to MAPROM bit in MISC register). Boot sector protection off/on (8 Kilobytes in range 0xE000-0xFFFF or 0x6000-0x7FFF) Use BOOTP bit in FEE32MCR register and LOCK bit in FEE32LCK register. Flash enable/disable using ROMON32 bit in MISC register.

HC12DG128

- **fpp file name:** `mcu03c4.fpp`
- **number of Flash modules:** 10
  - Applet code currently not relocatable, loaded at 0x2000, using 0x400 bytes.
  - All Flash modules enable/disable at same time using ROMON bit in MISC register.

Table 19.5 HC12DG128 Flash Module Details

Module Name	Module Number	Remarks
FLASH_ 4000	0	16 Kilobytes unpaged Flash located in 0x4000–0x8000 also matches 11FEE even page (6), that is, FLASH_PAGE6.
FLASH_ PAGE0	1	16 Kilobytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to 00FEE Flash even page (0).
FLASH_ C000	2	16 Kilobytes unpaged Flash located in 0xC000-0xFFFF also matches 11FEE odd page (7),that is, FLASH_PAGE7. Boot sector unprotectable/protectable (8 Kilobytes in range 0xE000-0xFFFF or paged range 0xA000-0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).
FLASH_ PAGE1	3	16 Kilobytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to 00FEE Flash odd page (1). Boot sector unprotectable/protectable (8 Kilobytes in range 0xA000-0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).
FLASH_ PAGE2	4	16 Kilobytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to 01FEE Flash even page (2).
FLASH_ PAGE3	5	16 Kilobytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to 01FEE Flash odd page (3). Boot sector unprotectable/protectable (8 Kilobytes in range 0xA000-0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).
FLASH_ PAGE4	6	16 Kilobytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to 10FEE Flash even page (4).

Table 19.5 HC12DG128 Flash Module Details (*continued*)

Module Name	Module Number	Remarks
FLASH_ PAGE5	7	16 Kilobytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to 10FEE Flash odd page (7). Boot sector unprotectable/protectable (8 Kilobytes in range 0xA000-0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).
FLASH_ PAGE6	8	16 Kilobytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to 11FEE Flash even page (6). Also equivalent to FLASH_4000 module.
FLASH_ PAGE7	9	16 Kilobytes paged Flash accessed in bank window 0x8000-0xBFFF, equivalent to 11FEE Flash odd page (7). Also equivalent to FLASH_C000 module. Boot sector unprotectable/protectable (8 Kilobytes in range 0xA000-0xBFFF) (via BOOTP bit in FEEMCR register and LOCK bit in FEELCK register).

HCS12 and HCS12X CPU Devices

All protections are fully removed when erasing and programming. The security byte at $FF0F is always reprogrammed to unsecure **when erasing** (due to aligned-word programming, $FF0E-FF0F is programmed to #$FFFE). The debugger asserts aligned word programming as specified in FTSxxxK and FTXxxxK specifications.

HCS12 and HCS12X device `fpp` files having been simplified to increase programming speed, as devices may have up to 1 Megabyte of on-chip Flash. Changing programming methods for each Program Page (64 PPAGEs on MC9S12XEP100) slows down the programming.

As a result, only relevant on-chip Flash blocks have their own listed module. The list below gives an overall availability for all HCS12 and HCS12X devices.

Table 19.6 HCS12 and HCS12X Module Usage

Module Name	Range	Comments
FLASH_4000	On-chip Flash in $4000-$7FFF; Mirror of PPAGE $3E on HCS12 devices and $FD on HCS12X devices	Provided to allow you to design non-banked code, such as ISR code or startup code.
FLASH_C000	On-chip Flash in $C000-$FFFF; Mirror of PPAGE $3F on HCS12 devices and $FF on HCS12X devices.	Provided to allow you to design non-banked code, such as ISR code or startup code, and vectors.
ALL_PPAGES (previously PAGED)	The entire on-chip Flash memory.	Erasing this module also erases FLASH_4000 and FLASH_C000 modules.
FLAT8000_Pxx or FLASH_8000 (HCS12X) and EEPROM_800 (HCS12X)	$4000 to $FFFF. Reset default page (Pxx) visible in $8000-$BFFF may vary from one HCS12 device to another, and be the same as the $3E PPAGE on HCS12X devices.	Allows you to design linear source code to be programmed from address $4000 to $FFFF. Use to evaluate a 48-Kilobyte application across several devices, although you may not have full control of current PPAGE. If PPAGE changes (by program CALL or by accidentally writing the PPAGE register), program code stored in window range $8000-$BFFF does not execute properly. For this reason, it is best not to use entire capacity of Flash. To ensure backward compatibility, these modules can be programmed, but not erased. Erasing is available but has no effect.
ALL_EPAGES (previously called EEP_PAGED) (HCS12X only)	The entire on-chip Flash memory.	Erasing this module erases all other EEPROM modules.

Table 19.6 HCS12 and HCS12X Module Usage

Module Name	Range	Comments
PFLASH	Range is specified by MODULEREMAP parameter of .fpp file	Universal module with adjustable memory range. Operates with any P-FLASH block.
DFLASH	Range is specified by MODULEREMAP parameter of .fpp file	Universal module with adjustable memory range. Operates with D-FLASH memory.

HCS12 EEPROM Relocation

HCS12 devices provide some hidden EEPROM memory that can only be accessed when changing the Memory Expansion Register called `INITEE`. This EEPROM is hidden or visible under the following conditions:

- Fully Hidden EEPROM
  - The EEPROM is fully blank checked.
  - The `FPP` file uses `INITEE` to automatically remap the EEPROM to $2000, on the condition that the user did not relocate the EEPROM, and changes `INITEE`. In that case, the `FPP` driver accesses the EEPROM at the user-specified location.
- Partially Visible EEPROM in $400-7FF or $400-FFF
  - The EEPROM is fully blank checked.
  - If the EEPROM is not at the reset location, the EEPROM size and location are automatically updated.
  - The EEPROM size in the NVMC dialog is automatically updated if the RAM does not overlap the EEPROM module.

EB386 Compliance and RAM Moving

NVMIF2 (format) new `FPP` drivers can be relocated in RAM. This format **for HCS12** devices is based on PIC code runtimes. Therefore, the NVM handling runtime can be moved in RAM if necessary.

First you must type the `FLASH` command in a Command window to verify that the `FPP` file is `NVMIF2`.

Execute the `FLASH NVMIF2WORKSPACE` to relocate the driver workspace in RAM, according to an eventual user-specified RAM relocation using `INITRM`, set up with a debugger `WB` command. See NVMC Commands.

This provides more flexibility for EB386 **Example 1 Layout** device RAM memory relocation. However, if the application itself performs the relocation, using FPP relocation has no effect, as programming is performed with the default location of the RAM.

CAUTION The `FPP` files/drivers do not support HCS12 on-chip Registers block moving from default/reset position.

HCS12X Emulated EEPROM

Currently the debugger does not support handling of these memory types.

Legacy Flash Programming Commands in Preload and Postload Command Files

The legacy Flash commands created by the project wizard to program an application automatically are given below.

Listing 19.2 In *xxxx*_Preload.cmd file

```
// reset the device to get default settings
RESET
// initialize Flash programming process
FLASH
// select the Flash modules
FLASH SELECT
// erase the Flash modules
FLASH ERASE
// arm the Flash for programming
FLASH ARM
```

Listing 19.3 In *xxxx*_Postload.cmd file

```
// The following commands must be enabled to terminate the programming
process with the ICD12

// disarm the Flash modules
FLASH DISARM
// unselect the Flash modules
FLASH UNSELECT
// reset the target board
RESET
```

TIP You can replace this Legacy implementation by using the Automated Application Programming feature. Clean or disable both command files, then engage the **Automatically erase and program** option in debugger Preferences.

S12G, S12P, S12X, S12XE, S12XS D-Flash memory

D-Flash memory is fully supported on these platforms. Please refer to TN263 for details here.

`C:\Program Files\Freescale\CWS12v5.1\Help\PDF.`

20

Unsecure HCS12 Derivatives

HCS12 derivatives include a security circuitry to prevent unauthorized access to contents of Flash, EEPROM and RAM memory when background debugging.

The *HC12MultilinkCyclonePro* Target interface provides an Unsecure function.

The **HC12MultilinkCyclonePro > Unsecure** menu command (and corresponding command line command CHIPSECURE UNSECURE) allows the debugger to connect to the target through the Information Required to Unsecure the Device Dialog Box and to execute the Unsecure Command File to unsecure the connected derivative.

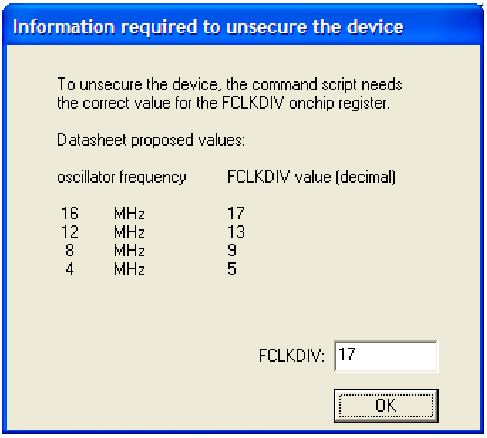
NOTE Some of the HCS12 derivatives cannot be unsecured while in Special mode (this is not possible with all MC9S12DP256 derivatives masks). Check the appropriate user manual for the connected derivative.

Information Required to Unsecure the Device

To unsecure a device, the debugger needs to know the value of the flash divider register to correctly mass erase the device and program the security byte. The Information required to unsecure the device dialog box provides an register value to the Unsecure Command File script.

Select **HC12MultilinkCyclonePro > Unsecure** to display the *Information Required to Unsecure the Device dialog* box.

Figure 20.1 Information Required to Unsecure the Device Dialog Box

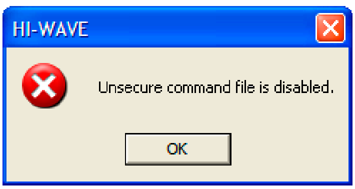


The information required for unsecuring the device can differ depending on the device family, the dialog box will prompt accordingly.

Once you enter the correct clock divider value, click OK to start the unsecure process executing the Unsecure Command File.

CAUTION If the Unsecure Command File has not been set up in the Target Interface Command Files dialog, the warning shown in Figure 20.2 appears.

Figure 20.2 Unsecure command file warning



The unsecure process checks the security byte to see if the device is unsecured, according to a mask and a compare value: `if (((value in security byte)& mask) == compare value))` then the chip is secured.

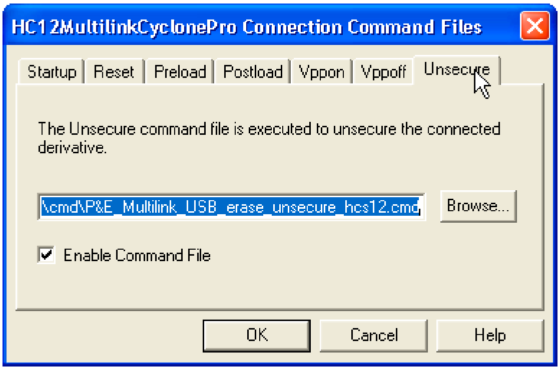
NOTE Modify the address of the security register, the mask and the compare value using the CHIPSECURE SETUP command. Those parameters are then stored in the project file.

Unsecure Command File

Set up the *Unsecure* command file using the *HC12MultilinkCycloneProTarget Interface Command Files* dialog. Choose **HC12MultilinkCyclonePro > Command Files** and click the *Unsecure* index tab. The project created with New Project Wizard will have appropriate unsecure command file configured.

Execute this command file to unsecure a secured HCS12 derivative (using **HC12MultilinkCyclonePro > Unsecure** menu entry).

Figure 20.3 Unsecure Command FIle tab



Listing 20.1 is an example of command file to unsecure an HCS12 derivative.

Listing 20.1 Example command file

```
// HCS12 Core erasing + unsecuring command file:
// These commands mass erase the chip then program the
// security byte to 0xFE (unsecured state).
// Evaluate the clock divider to set
// in ECLKDIV/FCLKDIV registers:
// An average programming clock of 175 kHz is chosen.
// If the oscillator frequency is less than 10 MHz,
// the value to store in ECLKDIV/FCLKDIV is equal to
// " oscillator frequency(kHz) / 175 ".

// If the oscillator frequency is higher than 10 MHz,
// the value to store
// in ECLKDIV/FCLKDIV is equal to
// " oscillator frequency (kHz) / 1400  + 0x40
// (to set PRDIV8 flag)".

// Datasheet proposed values:
```

```
//
// oscillator frequency
// ECLKDIV/FCLKDIV value (hexadecimal)
//
//  16 MHz     $49
//   8 MHz     $27
//   4 MHz     $13
//   2 MHz     $9
//   1 MHz     $4

define CLKDIV 0x49

FLASH MEMUNMAP   // do not interact with regular flash programming
monitor

//mass erase flash
wb 0x100 CLKDIV  // set FCLKDIV clock divider
wb 0x103 0       // FCFNG select block 0
wb 0x102 0x10    // set the WRALL bit in FTSTMOD
                 // to affect all blocks
wb 0x104 0xFF    // FPROT all protection disabled
wb 0x105 0x30    // clear PVIOL and ACCERR in FSTAT register
ww 0x108 0xD000  // write to FADDR address register
ww 0x10A 0x0000  // write to FDATA data register
wb 0x106 0x41    // write MASS ERASE command in FCMD register
wb 0x105 0x80    // clear CBEIF in FSTAT register
                 //to execute the command
wait 20          // wait for command to complete

//mass erase eeprom
wb 0x110 CLKDIV  // set ECLKDV clock divider
wb 0x114 0xFF    // EPROT all protection disabled
wb 0x115 0x30    // clear PVIOL and ACCERR in ESTAT register
ww 0x118 0x0400  // write to EADDR eeprom address register
ww 0x11A 0x0000  // write to EDATA eeprom data register
wb 0x116 0x41    // write MASS ERASE command in ECMD register
wb 0x115 0x80    // clear CBEIF in ESTAT register
                 // to execute the command
wait 20          // wait for command to complete

reset

//reprogram Security byte to Unsecure state
wb 0x100 CLKDIV  // set FCLKDIV clock divider
wb 0x103 0       // FCFNG select block 0
wb 0x104 0xFF    // FPROT all protection disabled
wb 0x105 0x30    // clear PVIOL and ACCERR in FSTAT register
ww 0xFF0E 0xFFFE // write security byte to "Unsecured" state
```

```
wb 0x106 0x20     // write MEMORY PROGRAM command
                  // in FCMD register
wb 0x105 0x80     // clear CBEIF in FSTAT register
                  // to execute the command
wait 20           // wait for command to complete

reset

FLASH MEMMAP      // restore regular flash programming monitor
undef CLKDIV      // undefine variable
```

21

On-Chip Hardware Breakpoint Module

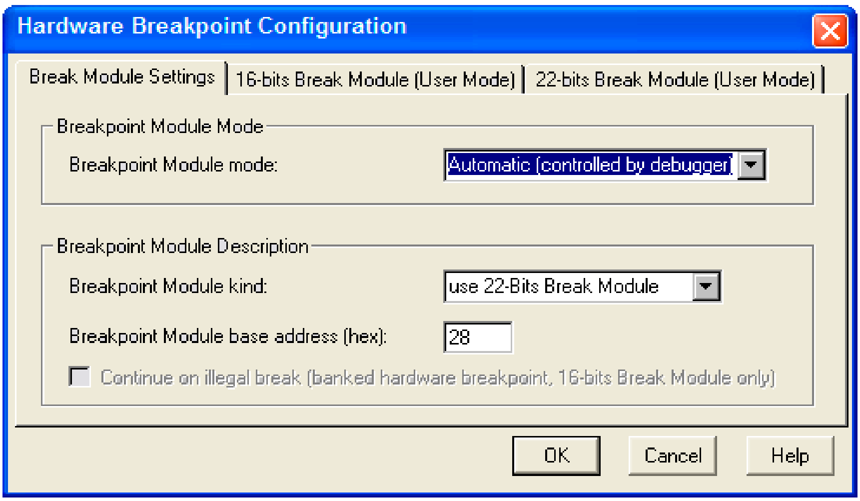
On some HC12 and HCS12 derivatives, you can use an on-chip hardware breakpoint module to set hardware breakpoints and watchpoint. To invoke this module, you must first set up the debugger to use the module.

During the first connection, the hardware breakpoints module settings resolve according to the specified derivative. If you change the derivative later, it is your responsibility to correctly set up the hardware breakpoint mechanism for the project using the Hardware Breakpoint Configuration dialog.

Hardware Breakpoint Configuration dialog

Choose the **HC12MultilinkCyclonePro > Set Hardware BP** menu command. The *Hardware Breakpoint Configuration* dialog *Break Module Settings* index tab appears, as shown in Figure 21.1.

Figure 21.1 Hardware Breakpoint Configuration dialog

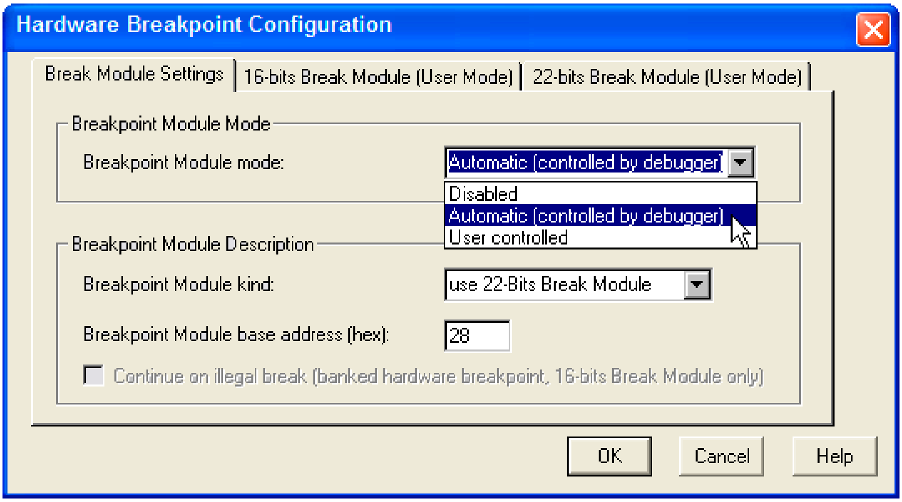


Breakpoint Module Mode

The *Mode* combo box allows you to select one of three different modes: *Disabled*, *Automatic (controlled by debugger)* and *User controlled* (see Figure 21.2).

This dialog allows you to set up the hardware breakpoint module of your HC12 or HCS12 derivative.

Figure 21.2 Hardware Breakpoint Configuration Breakpoint Module mode



NOTE This feature is available only if the HC(S)12(X) derivative is connected to the device via a P&E hardware interface and also has an embedded hardware breakpoint module. Check your MCU documentation.

Disabled mode

In Disabled mode, it is not possible to set a breakpoint in Flash or in EEPROM. It is also not possible to set any watchpoint, even if the application is loaded in RAM.

NOTE Some actions, like stepping over or stepping out, use one internal breakpoint and therefore cannot be used when debugging in non-volatile memory if the hardware breakpoint module is disabled.

Automatic (controlled by debugger) mode

This is the default mode for the debugger.

If you select the *Automatic (controlled by debugger)* mode, you have the option to set up to two breakpoints or one watchpoint in *Non-Volatile Memory*, as shown in Figure 21.3.

Figure 21.3 Module base address edit box

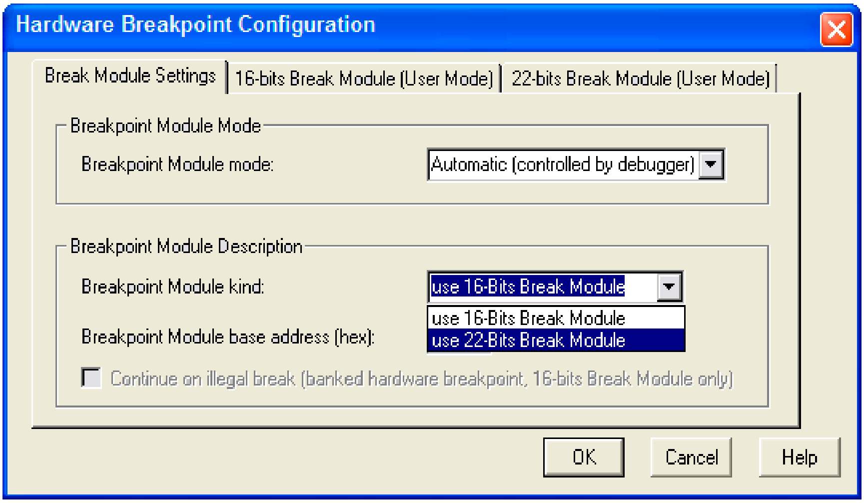


Table 21.1 describes the available options.

Table 21.1 Description of Settings

Setting	Description
Breakpoint Module kind	Select hardware breakpoint module supported by connected derivative: • Select Use 16-Bits Break Module for an *HC12* derivative • Select Use 22-Bits Break Module for an *HCS12* derivative.
Breakpoint Module base address (hex)	Use to set address of hardware breakpoint module in Module base address edit box. Base address is typically 0x20 for M68HC912B32, M68HC912D60 and M68HC912DG128, and 0x28 for HCS12 derivatives.

Table 21.1 Description of Settings

Setting	Description
Continue on illegal break	Allows you to debug in banked memory model when using 16 bits-break module. 16-bits break module does not allow you to set a breakpoint in bank. To address this problem, when the debugger stops on a hardware breakpoint it compares the address to an internal breakpoint list. If the low 16-bit portion of the address compares to the low 16-bit portion of the address of a set breakpoint, the breakpoint is located in an alternate bank. Debugger automatically restarts target.

When you finish making these settings, the debugger considers any breakpoint set in *Non Volatile Memory* as a *Hardware Breakpoint*.

If your application is loaded in RAM, breakpoints are software breakpoints. In this case the *Hardware Breakpoint* module allows you to debug using breakpoints and one watchpoint (only one watchpoint is available).

NOTE In Automatic mode, the HC12 or HCS12 hardware breakpoint modules allow only two breakpoints (or one watchpoint) at a time. If you are debugging your code in Flash, you cannot set more than two breakpoints or one watchpoint. Some actions, like stepping over or stepping out, use one internal breakpoint and therefore reduce the number of available hardware breakpoints to one. The MC68HC812A4 does not have a Hardware Breakpoint module.

User Controlled mode

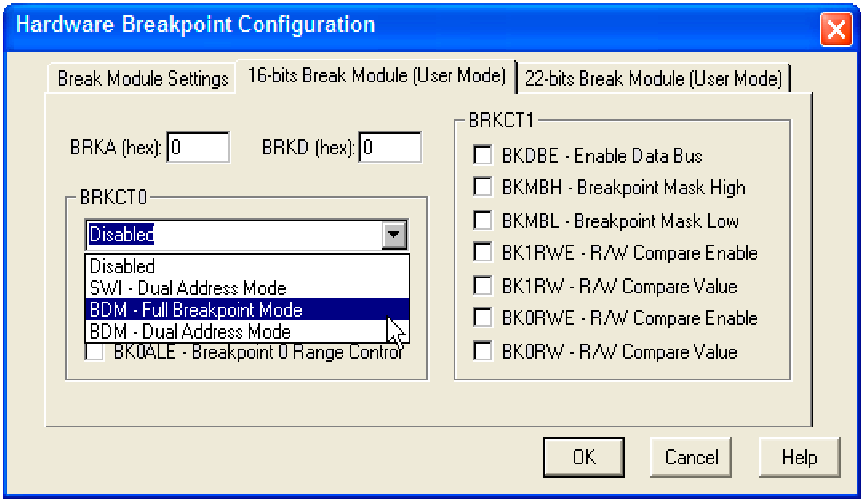
This mode allows you to fully set up the breakpoint module according to documentation.

Depending on the breakpoint module kind selected using the *Breakpoint Module Description* box in the Break Module Setup index tab, select either the *16-bits Break Module (User Mode)* or *22-bits Break Module (User Mode)*. The controls are grayed in the *User Mode* index tab if the correct Mode and correct breakpoint module kind are not selected.

16-Bits Break Module (User Mode)

The *16-bits Break Module (User Mode)* index tab allows you to set up the hardware breakpoint module of the connected *HC12* derivative when the *Breakpoint Module mode* is set to *User controlled* and the *Breakpoint Module Kind* is set to use *16-Bits Break Module*.

Figure 21.4 16-bits Break Module (User Mode) index tab



You can modify the following registers:

- `BRKCT0`: Breakpoint Control Register 0
- `BRKCT1`: Breakpoint Control Register 1
- `BRKA`: Breakpoint Address Register
- `BRKD`: Breakpoint Data Register

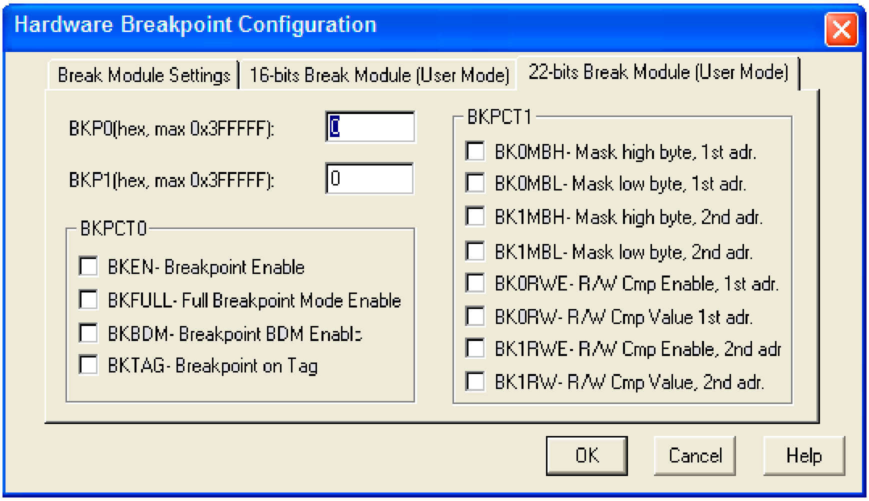
For more information about those registers, refer to your MCU reference manual section *Breakpoints* of the *Background Debug Mode* (*Development Support* part of the manual).

CAUTION When you set a hardware breakpoint or watchpoint in *User controlled* mode, the ILLEGAL_BP message appears in the status bar when the breakpoint or watchpoint is reached.
If the control point set is a breakpoint, you need to perform a single step before running again, otherwise the target endlessly breaks on the same address bus access.

22-bits Break Module (User Mode)

The *22-bits Break Module (User Mode)* index tab allows you to set up the hardware breakpoint module of the connected HCS12 derivative when the *Breakpoint Module mode* is set to *User controlled* and the *Breakpoint Module Kind* is set to use *22-Bits Break Module.*

Figure 21.5 22-bits Break Module (User Mode) index tab



You can modify the following registers:

- `BKPCT0`: Breakpoint Control Register 0
- `BKPCT1`: Breakpoint Control Register 1
- `BKP0`: Breakpoint Address Register
- `BKP1`: Breakpoint Data Register

For more information about these registers, refer to your MCU reference manual.

CAUTION When a hardware breakpoint or watchpoint is set in *User controlled* mode, the `ILLEGAL_BP` message appears in the status bar when the breakpoint or watchpoint is reached.
If the control point set is a breakpoint, you must perform a single step before running again, otherwise the target endlessly breaks on the same address bus access.

Book IV - Commands and Environment Variables

Book IV Contents

Each section of the Debugger manual includes information to help you become more familiar with the Debugger, to use all its functions and help you understand how to use the environment. This book defines the HC12, HCS12 and HC(S)12(X) commands used by the debugger engine and those specific to individual debugger connections. It also defines the HC12, HCS12, and HC(S)12(X) environment variables used by the debugger engine and those specific to individual debugger connections.

This book is divided into the following chapters:

22

Debugger Engine Commands

Commands Overview

The debugger supports scripting with the use of commands and command files. When you script the debugger, you can automate repetitive, time-consuming, or complex tasks.

You do not need to use or have knowledge of commands to run the Simulator/Debugger. However these commands are useful for editing debugger command files, for example, after a recording session, to generate your own command files, or to set up your applications and targets.

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
```

- **component** is the name of the component named in the component window title, such as Data or Register.
- **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, if you have one **Memory** component window open, and you open a second Memory component window, the first window becomes **Memory:1** and the second is **Memory:2**. The debugger automatically associates a number with a component when there are several components of the same type open.

Command Example

```
in>Memory:2 < SMEM 0x8000,8
```

The **<** symbol directs a command to a specific component (in this example: **Memory:2**). Some commands are valid for several or all components and if you do not direct the command to a specific component, the command affects all components. Directing the command to specific components prevents mismatches caused by differing parameter requirements of different components.

Command Syntax

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

Syntax Example

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

Available Command Lists

Commands described on the following pages are sorted into five groups, according to their specific actions or targets. However, these groups have no relevance in the use of these commands. It is possible to build powerful programs by combining Kernel commands with Base commands, common commands and component-specific commands. The following sections detail all commands in their respective groups.

Kernel Commands

Kernel commands are commands that can be used to build command programs. You can only use Kernel commands in a debugger command file, since the Command Line component can only accept one command at a time. Table 22.1 contains all available Kernel commands.

Table 22.1 List of Kernel Commands

Command, Syntax	Short Description
`A`	Affects a **value**
`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

Table 22.1 List of Kernel Commands (*continued*)

Command, Syntax	Short Description
`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
`ICD12EXEC HELP`	Output a list and the descriptions of all P&E available debugger commands to the Command Window.
`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
`TESTBOX`	Displays a message box with a string
`UNDEF symbol	*`	Undefines a user defined symbol
`UNTIL condition`	Condition of a REPEAT loop
`WAIT [time] [;s]`	Command file execution pause
`WHILE condition`	WHILE loop instruction

Base Commands

Use Base commands to monitor the Simulator/Debugger target execution. These commands handle target input/output files, target execution control, direct memory editing, breakpoint management and CPU register setup. Base commands can be executed independent of open components. Table 22.2 contains all available Base commands.

Table 22.2 List of 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
`MS range list`	Sets memory bytes
`NOCR`	Closes the record file
`NOLF`	Closes the log file
`P [address]`	Single assembly steps into program
`RESTART`	Restarts the loaded application

Table 22.2 List of Base Commands (*continued*)

Command, Syntax	Short Description
RD [list|*]	Displays the content of registers
RS register[=]value{,register [=]value}	Sets a register
S	Stops execution of the loaded application
STEPINTO	Steps to the next source instruction of loaded application
STEPOUT	Executes program out of a function call
STEPOVER	Steps over the next source instruction of the loaded application
STOP	Stops execution of the loaded application
SAVEBP on|off	Saves breakpoints
T [address][,count]	Traces program instructions at the specified address
WB range list	Writes bytes
WL range list	Writes longwords
WW range list	Writes words

Environment Commands

Use Simulator/Debugger environment commands to monitor the debugger environment, specific component window layouts, and framework applications and targets. Table 22.3 contains all available Environment commands.

Table 22.3 List of Environment Commands

Command, Syntax	Short Description
ACTIVATE component	Activates a component window
AUTOSIZE on|off	Autosizes windows in the main window layout
BCKCOLOR color	Sets the background color

Table 22.3 List of Environment Commands (*continued*)

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

Component Commands

Use Component commands to monitor component behaviors. Since these commands are common to more than one component, direct the commands to specific components using the < character in the command line. Table 22.4 lists all available Component commands.

Table 22.4 List of Component Commands

Command, Syntax	Short Description
`CMDFILE`	Specify a command file state and full name
`COM_START ["<path to Hiwave>\HIWAVE.EXE" ]`	Creates a new Hiwave
`COM_EXE (<debugger command>)`	Executes command in the created Hiwave Instance.
`COM_EXIT (COM_EXIT)`	Destroys the created Hiwave Instance.Q
`EXIT`	Terminate the application
`HELP`	Displays a list of available commands
`RESET`	Resets statistics
`SMEM range`	Shows a memory range
`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 22.5 shows all available component-specific commands.

Table 22.5 List of Component-Specific Commands

Command, Syntax	Short Description
`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
`COPYMEM <Source addr range> dest-addr`	Copies memory
`CYCLE on|off`	Switches cycles and milliseconds in SofTrace 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>]`	Selects 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

Table 22.5 List of Component-Specific Commands (*continued*)

Command, Syntax	Short Description
LS [symbol | *][;C|S]	Displays the list of symbols
NB [base]	Sets the base of arithmetic operations
OUTPUT fileName	Redirects the coverage component results
PTRARRAY on|off	Switches on /off the pointer as array display
RECORD on|off	Switches on/off the frame recorder
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 color attributes of 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
ZOOM address in|out	Zooms in/out a variable

Command Syntax Terms

The following lists includes all relevant syntax terms:

- `address`
  - A number matching a memory address. This number must be in ANSI format (i.e., **$** or **0x** for hexadecimal value, **0** for octal).
  - Example: `255, 0377, 0xFF, $FF`

NOTE `address` can also be an expression if `constant address` is not specifically 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, or numerical constants.

  – Example: `DEFINE IO_PORT = 0x210`

    `WB IO_PORT 0xFF`

- `range`
  – A composition of two 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. This example is equivalent to the previous example.

- `fileName`
  – A DOS file name and path that identifies a file and its location. The command interpreter does not assume any file name extension. Use backslash (\) or slash (/) as a directory delimiter.

  – The parser is case insensitive. If no path is specified, it looks for (or edits) the file in the current project directory: that is, when no path is specified, the default directory is the project directory.

  – Example:

    `d:/demo/myfile.txt`

  – Example:

    `layout.hwl`

  – Example:

    `d:/work/project.hwc`

- `component`
  – The name of a debugger component. Choose **Component > Open** to display a list of all debugger components. The parser is case insensitive.

  – Example:

`Memory`

– Example:

`SoUrCe`

Module Names

Correct module names appear in the Module component window. Make sure that you use the correct module name for any command you implement:

- If the `.abs` is in **HIWARE** format, some debug information is in the object file (`.o`), and module names have an `.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 ELF format assigns all debugging information in the `.abs` file and does not use object files.

Debugger Commands

This section describes the commands available when you use the Simulator/Debugger.

A

The **A** command assigns an expression to an existing variable. The quoted expression must be used for string and enum expressions.

Usage

`A variable = value` or `A variable = "value"`

Components

Debugger engine

Example

```
in>a counter=8
```

The variable **counter** is now equal to **8**.

```
in>A day1 = "monday_8U"    (Monday_8U is defined in an
Enum)
```

The variable **day1** is now equal to **monday_8U**.

```
in>A value = "3.3"
```

The variable **value** is now equal to **3.3**

ACTIVATE

ACTIVATE activates a component window as if you clicked on its title bar. The window appears in the foreground and the title bar is highlighted. If the window has active icons, the title bar is activated and appears in the foreground.

Usage

```
ACTIVATE component
```

Components

Debugger engine

Example

```
in>ACTIVATE Memory
```

Activates the Memory Component and brings the window to the foreground.

ADDXPR

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

Usage

```
ADDXPR "expression"
```

where the parameter `expression` is an expression to add and evaluate in the data component.

Components

Data component

Example

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

This adds the expression `counter +10` in the data component.

ATTRIBUTES

This command can affect several components, as described in the next sections. Ensure that you direct this command properly to prevent unexpected changes.

In the Command Component

This 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 1,000,000.

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 Equivalent Operations sections of the following component descriptions.

Usage

```
ATTRIBUTES list
```

where `list=command{,command})`

```
command=CACHESIZE value
```

Example

```
command < ATTRIBUTES 2000
```

In the Procedure Component

This 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

This command allows you to set the display and state options for the Assembly component window.

- `ADR` displays or hides the address of a disassembled instruction.
  - `ON | OFF` switches 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.
- `CODE` displays or hides the machine code of the disassembled instruction.
  - `ON | OFF` switches the machine code on or off.
- `ABSADR` shows or hides the absolute address of a disassembled instruction, such as `branch to`.
  - `ON | OFF` switches the absolute address on or off
- `TOPPC` scrolls the Assembly component to display the code location given as an argument on the first line of Assembly component window.
- `SYMB` displays or hides the symbolic names of objects.
  - `ON | OFF` switches 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 descriptions for more details 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 **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
```

This displays addresses, hexadecimal codes, and symbolic names in the Assembly component window, and highlights assembly instructions at addresses 0x800,16.

In the Register Component

The `ATTRIBUTES` command allows you to set the display and state options of the Register component window.

- `FORMAT` sets the display format of register values.
- `VSCROLLPOS` 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.
- `HSCROLLPOS` 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 deselects 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 appear in binary format in the Register component window.

`in>Register < ATTRIBUTES VSCROLLPOS 3`

Scrolls three positions down. The third line of registers appears on the top of the register component.

`in>Register < ATTRIBUTES VSCROLLPOS 0`

Returns to the default display. The first line of registers appears on the top of the register component.

`in>DEFINE vpos = 5`

`in>Register < ATTRIBUTES HSCROLLPOS vpos`

Scrolls five positions right. The second column of registers appears on the left of the register component.

`in>Register < ATTRIBUTES HSCROLLPOS 0`

Returns to the default display. The first column of registers appears on the left of the register component.

`in>Register < ATTRIBUTES COMPLEMENT One`

Sets the first complement display option. All registers appear in reverse bit.

In the Source Component

The `ATTRIBUTES` command allows you to set the display and state options of the Source component window.

- `SMEM` (show memory range) and `SPC` (show PC address) load the corresponding module's source text, scroll to the corresponding text range location or text address location and highlight the corresponding statements.
- `SMOD` (show module) loads the corresponding module's source text. If the module is not found, a message appears in the Component Windows Object Information Bar.

- `SPROC` (show procedure) loads the corresponding module's source text, scrolls to the corresponding procedure and highlights the statement in the procedure chain.
- `numberAssociatedToProcedure` is the level of the procedure in the procedure chain.
- `MARKS` (`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

The `ATTRIBUTES` command allows you to set the display and state options of the Data component window.

- `FORMAT` selects the format for the list of variables. The format is one of the following:
  - binary

  - octal
  - hexadecimal
  - signed decimal
  - unsigned decimal
  - symbolic
- `MODE` selects the display mode of variables.
  - `Automatic` mode (default), updates variables when the target stops. Variables from the currently executed module or procedure appear in the data component.
  - In `Locked and Frozen` mode, the same variables from a specific module always appear in the data component.
  - In `Locked` mode, values from variables displayed in the data component update when the target stops.
  - In `Frozen` mode, values from variables displayed in the data component do not update when the target stops.
  - In `Periodical` mode, variables update at regular time intervals while the target runs. The default update rate is 1 second, but you can modify this rate up to 100 ms using the associated dialog box or `UPDATERATE`.
- `UPDATERATE` sets the variables update rate (see also UPDATERATE command).
- `SPROC` (show procedure) and `SMOD` (show module) display local or global variables of the corresponding procedure or module.
- `SCOPE` selects and displays global, local, or user-defined variables.
- `COMPLEMENT` sets the display complement format of Data values: `one` sets the first complement (each bit is reversed), `none` deselects the first complement.
- `NAMEWIDTH` 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.

Usage

`ATTRIBUTES list`

where `list=command{,command})`

`command=FORMAT(bin|oct|hex|signed|unsigned|symb)| SCOPE (global|local|user|external) | MODE (automatic|periodical| locked|frozen) | SPROC level | SMOD module | UPDATERATE rate | COMPLEMENT(none|one)|`

```
NAMEWIDTH width
```

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 do not refreshed when the application is running.

In the Memory Component

The `ATTRIBUTES` command allows you to set the display and state options of the Memory component window.

- `WORD` 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).
- `ADR ON` or `OFF` displays or hides the address in front of the memory dump lines.
- `ASC ON` or `OFF` displays or hides the ASCII dump at the end of the memory dump lines.
- `ADDRESS` 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 to display the code location given as an argument, and select the corresponding memory area (`SPC` selects an address, `SMEM` selects a range of memory and `SMOD` selects the module name where the global variable is in the window).
- `FORMAT` selects the format for the list of variables. The format is one of the following: binary, octal, hexadecimal, signed decimal, unsigned decimal or symbolic.

- `COMPLEMENT` sets the display complement format of memory values: `one` sets the first complement (each bit is reversed), `none` deselects the first complement.
- `MODE` selects the display mode of memory words.
  - In `Automatic` mode (default), memory words update when the target stops. Memory words from the currently executed module or procedure appear in the Memory component.
  - In `Frozen` mode, value from memory words displayed in the Memory component do not updated when the target stops.
  - In `Periodical` mode, memory words update at regular time intervals while the target runs. 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.
- `UPDATERATE` 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
```

This removes the ASCII dump and addresses from the Memory component window.

In the Inspector Component

The `ATTRIBUTES` command allows you to set the display and state of the Inspector component window.

Usage

`ATTRIBUTES list`

where `list=command{,command})`

`command= COLUMNWIDTH columnname columnfield columnsize |`

`EXPAND [name {subname}] deep |`

`COLLAPSE name {subname}|`

`SELECT name {subname} |`

`SPLIT pos |`

`MAXELEM (ON | OFF) [number] |`

`FORMAT (Hex|Int)`

- `COLUMNWIDTH` 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, 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.

- `EXPAND` computes and displays all subitems of a specified item up to a given depth. Specify an item by stating the complete path, starting at one of the root items like `"Symbol Table"` or `"Object Pool"`. Names with spaces must be surrounded by quotes. To expand all subitems of TargetObject in the Object Pool up to four levels, use the following command:

```
inspect < ATTRIBUTES EXPAND "Object Pool" TargetObject
4
```

NOTE Because the name Object Pool contains a space, it must be surrounded by quotes.

NOTE The symbol table, stack, or other items may have recursive information, so the information tree may grow with the depth. Specifying large expand values may use a large amount of memory.

- `COLLAPSE` folds one item. You must specify the item name. The following command folds `TargetObject`:

```
inspect < ATTRIBUTES COLLAPSE "Object Pool"
TargetObject
```

- `SELECT` shows the information of the specified item on the right pane. The following command shows all `Objects` attached to `TargetObject`:

```
inspect < ATTRIBUTES SELECT "Object Pool" TargetObject
```

- `SPLIT` 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 shows only the right pane; a value of 100 shows only 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
```

- `MAXELEM` sets the number of subitems to display. After the following command, the Inspector prompts for 1000 subitems:

```
inspect < ATTRIBUTES MAXELEM ON 1000
```

- `FORMAT` specifies whether to display integral values, such as addresses, as hexadecimal or decimal. The following command specifies 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.

In the MCURegisters Component

The `ATTRIBUTES` command allows you to set the display and state options of the MCURegisters component window.

Usage

`ATTRIBUTES` list

where `list=command{,command})`

`command=FORMAT(bin|hex|oct|dec|udec)|`

`EXPAND name|`

`COLLAPSE name|`

`MODE (automatic|periodical) |`

`UPDATERATE rate`

The `FORMAT` command selects the format for all the registers. The format can be binary, hexadecimal, octal, signed decimal, unsigned decimal

The `EXPAND` command unfolds the node with given name in tree view.

The `COLLAPSE` command folds the node with given name in tree view.

NOTE Refer to EXPAND and COLLAPSE command descriptions for more detail about these commands.

The `MODE` command selects the display mode of variables.

- In `Automatic mode` (default), registers are updated when the target is stopped.

- In `Periodical mode`, registers 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 registers update rate (see also `UPDATERATE` command).

Equivalent Operations

- `ATTRIBUTES FORMAT` ~ Select menu **MCURegisters** > **Format**
- `ATTRIBUTES MODE` ~ Select menu **MCURegisters** > **Mode**
- `ATTRIBUTES EXPAND` ~ Select menu **MCURegisters** >**Tree** > **Expand**
- `ATTRIBUTES COLLAPSE` ~ Select menu **MCURegisters** >**Tree**> **Collapse**
- `ATTRIBUTES UPDATERATE` ~ Select menu **MCURegisters** > **Mode** > **Periodical**

Example

```
MCURegisters < ATTRIBUTES MODE PERIODICAL
```

In MCURegisters the registers update is set in periodical mode. Registers are updated at regular time.

AT

The `AT` command temporarily suspends a command file from executing until after a specified delay in milliseconds. The delay is measured from the time the command file starts. In the event that command files are chained (one calling another), the delay is measured from the time the first command file starts.

NOTE This command can only be executed from a command file. The time specified is relative to the start of command file execution.

Usage

`AT` **time**

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

Components

Debugger engine

Example

```
AT 10 OPEN Command
```

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

AUTOSIZE

`AUTOSIZE` enables/disables windows autosizing. When on, the size of component windows are automatically adapted to the Simulator/Debugger main window when it is resized.

Usage

```
AUTOSIZE on|off
```

Components

Debugger engine

Example

```
in>AUTOSIZE off
```

Windows autosizing is disabled.

BASE

In the Profiler component, the `BASE` command sets the profiler base to `code` (total code) or `module` (each module code).

Usage

```
BASE code|module
```

Components

Profiler component

Example

```
in>BASE code
```

BC

BC clears a breakpoint at the specified address. Specifying * clears all breakpoints.

You can also point to the breakpoint in the Assembly or Source component window, right click and choose **Delete Breakpoint** in the context menu, or open the ControlPoints Window, select the breakpoint from the list and click **Delete**.

NOTE Correct module names appear in the Module component window. Make sure that you use the correct module name in your command: if the .abs is in **HIWARE** format, some debug information is in the object file (.o), and module names have an .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. Specify this address in ANSI C or standard Assembler format. You can also replace address by an expression as shown in the example below.

Specifying * deletes all breakpoints.

Components

Debugger engine

Example 1

```
in>BC 0x8000
```

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

Example 2

```
in>BC &FIBO.C:Fibonacci
```

In this example, an expression replaces the address. FIBO.C is the module name and Fibonacci is the function from which the breakpoint is cleared.

BCKCOLOR

`BCKCOLOR` sets the background color.

The background color defined with the `BCKCOLOR` command is valid for all component windows. Using the same color for the font and background makes text in the component windows invisible. 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 Using WHITE as a parameter sets all component windows to their default background colors.

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
```

This sets the background color of all currently open component windows to `Lightcyan`. To return to the original display, enter `BCKCOLOR WHITE`.

BD

In the Command Line component, the `BD` command displays the list of all breakpoints currently set with addresses and types (temporary, permanent).

Usage

BD

Components

Debugger engine

Example

```
in>BD
Fibonacci 0x805c T
Fibonacci 0x8072 P
Fibonacci 0x8074 T
main 0x8099 T
```

This sets one permanent and two temporary breakpoints in the function **Fibonacci**, and one temporary breakpoint in the **main** function.

NOTE From the list, it is not possible to know if a breakpoint is disabled or not.

BS

`BS` sets a temporary (`T`) or a permanent (`P`) breakpoint at the specified address. If `P` or `T` is unspecified, the default is a permanent (`P`) breakpoint.

Equivalent Operation

Point at a statement in the Assembly or Source component window, right click and choose **Set Breakpoint** in the context menu, or open the Controlpoints Configuration Window and choose **Show Breakpoint**, then select the breakpoint and set its properties.

NOTE The Module component window displays the correct module names. Make sure that the module name in your command is correct:
If `.abs` is in **HIWARE** format, some debug information is in the object file

(.o), and module names have the .o extension. In **ELF** format, module name extensions are .c, .cpp or .dbg (.dbg for program sources in assembler), since the .abs file contains all debugging information 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. Specify this address in ANSI C format. You can replace `address` with 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` indicates enabled, and `D` indicates disabled. If state is unspecified, default state is `E`.

`condition` is an expression matching the Condition field in the Controlpoints Configuration window for a conditional breakpoint.

`command` is any Debugger command (at this level, the commands `G`, `GO` and `STOP` are not allowed). It matches the Command field in the Controlpoints Configuration window for associated commands. For the Command function, the states are `E` (enabled) or `C` (continue).

`current` is an expression matching the Current field (Counter) in the Controlpoints Configuration window, for counting breakpoints.

`interval` is an expression matching the Interval field (Counter) in the Controlpoints Configuration window, for counting breakpoints.

`codeSize` is an expression, 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, 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 sets a temporary breakpoint at the address 0x8000.

```
in>BS $8000
```

This 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.

```
in>BS &main + 22 P E ; cdSz = 66 srSz = 134
```

This sets a breakpoint at the address of the main procedure + 22, where the code size of the main procedure is 66 bytes and its source size is 134 characters.

```
in>BS Fibo.c:main{3}
```

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

```
in>BS &counter + 5; cond ="fib1>fib2";cmd="bckcolor red"
```

This 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
```

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

CALL

Executes a command in the specified command file.

NOTE If path is unspecified, the destination directory is the current project directory.

Usage

```
CALL FileName [;C][;NL]
```

Components

Debugger engine

Example

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

Loads the config command file.

CD

The `CD` command changes the current working directory to the directory specified in the path. Entering the command with no parameter displays the current directory.

The directory specified in the `CD` command must be a valid directory and 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 path is unspecified, the default directory is the project directory. Using the `CD` command can affect all commands that refer to files with unspecified paths.

Usage

```
CD [path]
```

`path`: The pathname of a directory that becomes the current working directory (case insensitive).

Components

Debugger engine

Example

```
in>cd..
C:\Freescale\demo
in>cd
C:\Freescale\demo
in>cd /Freescale/prog
C:\Freescale\prog
```

The new project directory is `C:\Freescale\prog`

CF

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

NOTE If path is unspecified, 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 you use the `;C` option in the calling file, the command interpreter quits the calling file and executes the called file. The commands following the `CF ... ;C` command never execute.
- When you omit the `;C` option, calling file execution resumes after the execution of the commands in the called file.

`;NL`: This option prevents the commands in the called file from being logged in the Command Line window even if the `CMDFILE` type is set to `ON` (see LOG). This option does not log the commands to a log file opened with an LF command.

Components

Debugger engine

Examples

`in>CF commands.txt`

Executes the `COMMANDS.TXT` file containing debugger commands like those described in this chapter.

Example without “;C” Option

If a `command1.txt` file contains:

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

and a `command2.txt` file contains:

```
bckcolor red
```

Execution:

```
in>cf command1.txt
executing command1.txt

!bckcolor green
!cf command2.txt
executing command2.txt

1!bckcolor red
1!
1!
done command2.txt

!bckcolor white
!
done command1.txt
```

Example with ";C" Option

If a `command1.txt` file contains:

```
bckcolor green
cf command2.txt ;C
bckcolor white
```

and a `command2.txt` file contains:

```
bckcolor red
```

Execution:

```
in>cf command1.txt
executing command1.txt

!bckcolor green
!cf command2.txt ;C
executing command2.txt

1!bckcolor red
1!
1!
done command2.txt

done command1.txt
```

CLOCK

In the SoftTrace component, the `CLOCK` command sets the clock speed.

Usage

```
CLOCK frequency
```

Where `frequency` is a decimal number, which is the CPU frequency in Hertz.

Components

SoftTrace component

Example

```
in>CLOCK 4000000
```

CLOSE

Use the `CLOSE` command to close a component.

Component names are: Assembly, Command, Coverage, Data, Inspect, Memory, Module, Procedure, Profiler, Recorder, Register, Source, Stimulation.

Usage

```
CLOSE component | *
```

where `*` means all components.

Components

Debugger engine

Example

```
in>CLOSE Memory
```

This closes (unloads) the Memory component window.

COLLAPSE

In the MCURegisters component, the `COLLAPSE` command is used to fold node in the register tree view.

Usage

```
COLLAPSE name
```

where `name` is the name of a node in the register tree view. If the name belongs to the register item, then item's parent node is folded. If the name belongs to group of registers, then module or board item, item's own node is folded. If the name is empty then all nodes are folded.

Components

MCURegisters component

Example

```
MCURegisters < collapse mc9s08dv60
```

This command folds the root node in the register tree view. This node contains the name of the board so all the module items are folded.

COM_START

Description

Creates a new Hiwave Instance. Only one new Hiwave Instance can be created in each ComMaster component.

Syntax

```
in> COM_START ["<path to Hiwave>\HIWAVE.EXE"]
```

If parameter is omitted, the Hiwave instance registered as a COM server is created.

To register Hiwave as a COM server, start it with the key -RegServer

Examples

```
in> COM_START "C:\Freescale\prog\hiwave.exe"
```

```
in> ComMaster:2< COM_START
```

COM interface analog (in Perl)

In Perl:

```
system ("C:\\Freescale\\prog\\hiwave.exe -RegServer");
$g_hwInst = Win32::OLE->new("Metrowerks.Hiwave");
```

COM_EXE

Description

Executes a debugger command within the created Hiwave instance.

Syntax

```
COM_EXE "<debugger command>"
```

Examples

```
in> COM_EXE "open data"
in> ComMaster:2< COM_EXE "bs main"
main 0x410 P
```

COM interface analog (in Perl)

```
$g_hwInst->ExecuteCmd ("open data");
$result = $g_hwInst->ExecuteCmdRes ("bs main");
$erCode= Win32::OLE->LastError();
```

COM_EXIT

Description

Finishes debug session and destroys the previously created Hiwave Instance

Syntax

```
COM_EXIT
```

Examples

```
in>  COM_EXIT
in>  ComMaster:2< COM_EXIT
```

COM interface analog (in Perl)

```
$g_hwInst->ExecuteCmd ("exit");
```

COPYMEM

Use the `COPYMEM` command to copy a memory range to a destination range defined by the beginning address. This command works on defined memory only. The debugger compares the source range and destination range to ensure they do not overlap.

Usage

```
COPYMEM <Source address range> dest-address
```

Components

Memory

Example

```
in>copymem 0x3FC2A0..0x3FC2B0 0x3FC300
```

This copies the memory located in the range `0x3FC2A0` to `0X3FC2B0` to the memory at `0x3FC300` to `0x3FC310`. This Memory range appears in red in the Memory Component.

CMDFILE

The `CMDFILE` command allows you to define all target-specific commands in a command file.

Usage

```
CMDFILE <Command File Kind> ON|OFF ["<Command File Full
Name>"]
```

Components

Simulator/target engine

Example

```
in>cmdfile postload on "c:\temp\myposloadfile.cmd"
```

This executes the `myposloadfile` command file after loading the absolute file.

CR

The `CR` command instructs the debugger to write records of commands to an external file. Writing continues until a close record file (NOCR) command executes.

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

NOTE If path is unspecified, the destination directory is the current project directory.

Usage

```
CR [fileName][;A]
```

`fileName` specifies the name of the record file. Use the `CR` command without this parameter to open a standard **Open File** dialog.

`;A` opens a file `fileName` in append mode, and appends new records at the end of an existing record file. Omitting this option when `fileName` is an existing file clears the file before writing new records.

Components

Debugger engine

Example

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

This opens the `myrecord.txt` file in Append mode for a recording session.

CYCLE

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

Usage

```
CYCLE on|off
```

Components

Softtrace component.

Example

```
in>CYCLE on
```

DASM

The `DASM` command displays the assembler code lines of an application, starting at the address given in the parameter. Without the parameter, the `DASM` command displays the assembler code following the last address of the previous display.

Stop this command by pressing the **Esc** key.

Equivalent Operation

Right click in the Assembly component window, select **Address** and enter the address to start disassembly in the **Show PC** dialog.

Usage

`DASM [address|range][;OBJ]`

`address`: A constant expression representing the `address` at which disassembly begins.

`range`: An address range constant that specifies addresses to be disassembled. When you omit `range`, a maximum of sixteen instructions are disassembled.

When you omit `address` and `range`, disassembly begins at the address of the instruction that follows the last instruction that was disassembled by the most recent `DASM` command. If this is the first `DASM` command of a session, disassembly begins at the current address in the program counter.

`;OBJ`: Displays assembler code in hexadecimal.

Components

Debugger engine

Example

```
in>dasm 0xf04b
00F04B LDHX    #0x0450
00F04E TXS
00F04F CLRH
00F050 CLRX
00F051 STX     0x80
00F053 INC     0x80
00F055 LDX     0x80
00F057 JSR     0xF000
00F05A STX     0x82
00F05C STA     0x81
00F05E LDA     #0x17
00F060 CMP     0x80
00F062 BEQ     *-20         /abs = F050
00F064 BRA     *-19         /abs = F053
00F066 DECX
00F067 DECX
```

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

Disassembled instructions appear in the Command Line component window. Therefore, you must open the Command Line component before executing this command to see the dumped code.

DB

The DB command displays the hexadecimal and ASCII values of the bytes in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first byte displayed in the line, followed by the number of specified hexadecimal byte values. The corresponding ASCII characters, separated by spaces, follow the hexadecimal byte values. Between the eighth and ninth values, a hyphen (-) replaces the space as the separator. Each non-displayable character is represented by a period (.).

Cancel this command by clicking the **Esc** key.

Usage

`DB [address|range]`

When you omit `address` and `range`, the first longword displayed is taken from the address following the last longword displayed by the previous `DB`, `DW`, or `DL` command, or from address `0x0000` (for the first `DB`, DW, or 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
þ_Eý_C'.5í1ì1ipƒ
```

This displays memory bytes in the Command Line component window, with matching ASCII characters. Open the Command Line component before executing this command to see the dumped code.

```
in>DB &TCR
```

```
0012: 5A Z
```

This displays the byte that is at the address of the TCR I/O register. The `DEFAULT.REG` file contains the I/O register definitions.

DDEPROTOCOL

Use the `DDEPROTOCOL` command 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 tells you whether the DDE protocol is active (on or off) and if display is active (Show or Hide)

Components

Debugger engine

Example

```
in>DDEPROTOCOL ON
in>DDEPROTOCOL SHOW
in>DDEPROTOCOL STATUS
DDEPROTOCOL ON - DISPLAYING ON
```

This example activates and displays the DDE protocol, and gives the status in the command line component.

NOTE For more information on Debugger/Simulator DDE implementation, refer to Debugger DDE Capabilities.

DEFINE

The DEFINE command creates a symbol and associates the value of an expression with it. Arithmetic expressions are evaluated when the command is interpreted. Use the symbol to represent the expression until the symbol is redefined, or until the UNDEF command undefine the symbol. 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. You can overwrite an application variable or I/O register with a DEFINE command.

NOTE Use this command to assign meaningful names to expressions, for used in other commands. This increases the command file readability and avoids re-evaluation of complex expressions.

Usage

```
DEFINE symbol [=] expression
```

Components

Debugger engine

Example

```
in>DEFINE addr $1000
in>DEFINE limit = addr + 15
```

These commands first define `addr` as a constant equivalent to `$1000`, and then define `limit` with the value (`$1000 + 15`).

You can redefine a symbol defined in the loaded application by using the `DEFINE` command on the command line. The symbol defined in the application is not accessible until an `UNDEF` on that symbol name is detected in the command file.

Example

In this example, we define a symbol named `testCase` 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.

DETAILS

The `DETAILS` command opens a profiler split view in the Source or Assembly component.

Usage

```
DETAILS assembly|source
```

Components

Profiler component

Example

```
in>DETAILS source
```

DL

The `DL` command displays the hexadecimal values of the longwords in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first longword displayed in the line, followed by the number of specified hexadecimal longword values.

When you specify a size in the range, this size represents the number of longwords to display in the command line window.

Stop this command by pressing the **Esc** key.

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

Usage

```
DL [address|range]
```

Omitting `range` displays the first longword 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
```

Displays the content of the memory range starting at 0x8000 and ending at 0x8007 as longword (4-byte) values.

```
in>DL 0x8000,2
8000: FE8045FD 80432710
```

Displays the content of two longwords starting at 0x8000 as longword (4-byte) values.

Memory longwords appear in the Command Line component window.

DUMP

The `DUMP` command writes names of all visible items in the Data component and MCURegisters component to the command line component. In MCURegisters component if the visible item is a register item its name and value are written.

Usage

DUMP

Components

Data and MCURegisters component

Example 1

```
in> Data:1 < DUMP
```

Example 2

```
in> MCURegisters < DUMP
```

DW

The `DW` command displays the hexadecimal values of the words in a specified range of memory. The command displays one or more lines, depending on the address or range specified. Each line shows the address of the first word displayed in the line, followed by the number of specified hexadecimal word values.

When you specify a size in the range, this size represents the number of words to display in the command line window.

Stop this command by pressing the **Esc** key.

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

Usage

```
DW [address | range]
```

When `address` is an address constant expression, this command displays the address of the first word.

When you omit `address` and `range`, this command displays the first word 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, or DL command of a session).

Components

Debugger engine

Example

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

```
8000: FE80 45FD 8043 2710
```

Displays the content of four words starting at `0x8000` as word (2-byte) values.

Memory words appear in the Command Line component window.

E

The `E` command evaluates an expression and displays the result in the Command Line component window. When the expression is the only parameter entered (no option specified) the value of the expression appears in the default number base. The result appears as a signed number in decimal format and as an unsigned number in all other formats.

Usage

```
E expression[;O|D|X|C|B]
```

where:

`;O` displays the value of expression as an octal (base 8) number.

`;D` displays the value of expression as a decimal (base 10) number.

`;X` displays the value of expression as an hexadecimal (base 16) number.

`;C` displays the value of expression as an ASCII character. Displays the remainder resulting from dividing the number by 256. Displays all values in the current font. Displays control characters (<32) as decimal.

`;B` displays the value of expression as a binary number.

Components

Debugger engine

Example

```
in>define a=0x12
in>define b=0x10
in>e a+b
in>=34
```

This evaluates the addition operation of the two previously defined variables `a` and `b` and displays the result in the Command Line window. You can redirect the output to a file by using the `LF` command (see LF and LOG commands).

ELSE

The `ELSE` keyword is associated with the IF command.

Usage

```
ELSE
```

Components

Debugger engine

Example

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

ELSEIF

The `ELSEIF` keyword is associated with the IF command.

Usage

```
ELSEIF condition
```

where `condition` is same as defined in C language.

Components

Debugger engine

Example

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

ENDFOCUS

The ENDFOCUS command resets the current focus. It is associated with the FOCUS command. Commands following the ENDFOCUS command 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
```

This example first redirects the ATTRIBUTES command to the Assembly component using the FOCUS Assembly command, and displays the code next to assembly instructions. Then the ENDFOCUS command releases the Assembly component and the FOCUS Source command redirects the second ATTRIBUTES command to the Source component. Marks appear in the Source window.

ENDFOR

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` equals 5.

ENDIF

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

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` equals 5

EXECUTE

In the Stimulation component, the `EXECUTE` command executes a file containing stimulation commands. Refer to the **I/O Stimulation** documentation.

Usage

```
EXECUTE fileName
```

Components

Stimulation component

Example

```
in>EXECUTE stimu.txt
```

EXIT

In the Command line component, the `EXIT` command closes the Debugger application.

Usage

```
EXIT
```

Components

Debugger engine

Example

```
in>EXIT
```

This closes the Debugger application.

EXPAND

In the MCURegisters component, the `EXPAND` command is used to unfold node in the register tree view.

Usage

```
EXPAND name
```

where `name` is the name of a node in the register tree view. If the name belongs to the register item, then item's parent node is unfolded. If the name belongs to group of registers, module or board item, then item's own node is unfolded. If the name is empty then all nodes are unfolded.

Components

MCURegisters component

Example

```
MCURegisters < expand mc9s08dv60
```

This command unfolds the root node in the register tree view. This node contains the name of the board so all the module items are visible.

FILL

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 context menu and by selecting the **Fill** or **Memory > Fill** menu entry.

Example

```
in>FILL 0x8000..0x8008 0xFF
```

This fills the memory range `0x8000..0x8008` with the value 0xFF.

FILTER

In the Memory component, with the `FILTER` command, you select what you want to display. You can also specify a range to be logged in your file.

Usage

```
FILTER Options [<range>]
```

```
Options = modules|functions|lines
```

`modules`: displays modules only

`functions`: displays modules and functions

`lines`: displays modules, functions, and code lines.

`Range`: a number between 0 and 100.

Components

Coverage component

Example

```
in>coverage < FILTER functions 25..75
```

FIND

In the Source component, use the `FIND` command to search for a specified pattern in the source file currently loaded, and highlights the pattern if found. Search forward (default), backward (`;B`), match case sensitive (`;MC`) or match whole word sensitive (`;WW`). The operation begins at the currently highlighted statement or from the beginning of the file (if nothing is highlighted). If the item is found, the Source window scrolls to the position of the item and the highlights the item in grey.

Equivalent Operation

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

Usage

```
FIND "string" [;B] [;MC] [;WW]
```

Where `string` is the pattern to match. You must enclose `string` in quotes. See the example below.

`;B` searches backwards, default is forwards.

`;MC` matches case sensitive.

`;WW` matches on the whole word.

Components

Source component

Example

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

Searches for the `"this"` string (considered as a whole word) in the Source component window, and performs the search backward.

FINDPROC

If a valid procedure name is given as parameter, the source file where the procedure is defined opens in the Source Component, displays the procedure's definition and highlights the procedure's title.

Equivalent Operation

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

Usage

```
FINDPROC procedureName
```

Components

Source component

Example

```
in>findproc Fibonacci
```

Displays the **Fibonacci** procedure and highlights the title.

FOCUS

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 see 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
```

This example first redirects the ATTRIBUTES command to the Assembly component using the FOCUS Assembly command, and displays the code next to assembly instructions. Then the ENDFOCUS command releases the Assembly component and the FOCUS Source command redirects the second ATTRIBUTES command to the Source component. Marks appear in the Source window.

FOLD

In the Source component, the FOLD command hides the source text at the program block level. Folded program text appears as if the program block is empty. When you unfold the folded block, 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

`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

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. Optionally, you may specify a **variable** (either a symbol or a program variable), which is assigned to all values of the range that are met during execution of the for loop. If you use a variable, you must define it with a DEFINE command before executing the `FOR` 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 have no impact on the number of iterations.

Stop this command by pressing the **Esc** key.

Usage

```
FOR[variable =]range [“,” step]
```

Where `variable` is the name of a defined variable.

`range`: This is an address range constant that specifies addresses to be disassembled.

`step`: constant number matching the step increment of the loop.

Components

Debugger engine

Example

```
DEFINE loop = 0
FOR loop = 1..6,1
T
ENDFOR
```

This performs the `T` (Trace) command six times.

FPRINTF

`FPRINTF` is a standard ANSI-C command, that writes a 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

In the **SoftTrace component**, the `FRAMES` command sets the maximum number of frame records.

Usage

```
FRAMES number
```

Where `number` is a decimal number equal to the maximum number of recorded frames. This number must not exceed 1000000.

Components

SoftTrace component

Example

```
FRAMES 10000
```

G

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 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 starts at 0x8000. **RUNNING** appears in the status bar. The application runs until a breakpoint is reached or you stop the execution.

GO

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 starts at address 0x8000. **RUNNING** appears in the status bar. The application runs until a breakpoint is reached or you stop execution.

GOTO

The `GOTO` command diverts execution of the command file to the command line that follows the Label. You must define the Label 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

The `GOTOIF` command diverts execution of the command file to the command line that follows the label if the condition is true. Otherwise, the command is ignored. The `GOTOIF` command fails if the condition is true and the label is not found.

Usage

```
GOTOIF condition Label
```

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

Components

Debugger engine

Example

```
DEFINE jump = 0
...
DEFINE jump = jump + 1
...
GOTOIF  jump == 10 MyLabel
T
...
MyLabel:  // comments
```

The program pointer jumps to `MyLabel` only if `jump` equals 10. Otherwise, the next instruction (`T` (Trace) command) executes.

GRAPHICS

In the Profiler component, `GRAPHICS` switches the percentages display in the graph bar `on/off`.

Usage

```
GRAPHICS on|off
```

Components

Profiler component

Example

```
in>GRAPHICS off
```

HELP

In the Command line component, the `HELP` command displays all available commands.

Subcommands from the `ATTRIBUTES` command are not listed.

Component-specific commands for closed components are not listed.

Usage

```
HELP
```

Components

Debugger engine

Example

```
in>HELP
HI-WAVE Engine:
  VER
  LF
  NOLF
  CR
  NOCR
   ....
```

ICD12EXEC HELP

The ICD12EXEC HELP command outputs a list and descriptions of all P&E available debugger commands to the Command Window.

Usage

```
icd12exec help
```

Components

Debugger engine

Example

```
in>icd12exec help
```

IF

The conditional commands (IF, ELSEIF, ELSE and ENDIF) allow you to execute different sections depending on the result of the corresponding condition. You may next the conditional commands. 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
```

Evaluates the `jump = = 10` condition and, depending on the test result, executes the `T Trace` instruction, or evaluates the `ELSEIF jump = = 100` test.

INSPECTOROUTPUT

The Inspector dumps the content of the specified item and all computed sub-items to the command window. Uncomputed sub-items are not printed. To compute all information, use the `ATTRIBUTES EXPAND` command.

Usage

```
INSPECTOROUTPUT [name {subname}]
```

The `name` specifies any of the root items. The `subname` specifies a recursive path to sub-items.

If a name contains a space, you must surround the name with quotes (`" "`).

Components

Inspector component

Example

```
in>loadio swap
in>Inspect<ATTRIBUTES EXPAND 3
in>INSPECTOROUTPUT "Object Pool" Swap
      Swap
      *  Name       Value   Address   Init...
      -  IO_Reg_1   0x0     0x1000    0x0 ...
      -  IO_Reg_2   0x0     0x1001    0x0 ...
```

INSPECTORUPDATE

The Inspector displays various information. Some types of information update automatically. To make sure that displayed values correspond to the current situation, the `INSPECTORUPDATE` command updates all information.

Usage

```
INSPECTORUPDATE
```

Components

Inspector component

Example

```
in>INSPECTORUPDATE
```

LF

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

Logging continues until a close log file (NOLF) command executes. When you use the `LF` command with no filename, the Open File Dialog appears to allow you to specify a filename.

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

If you specify a path in the file name, this path must be a valid path. When you specify a relative path, 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, so that `LF` appends logged lines at the end of an existing log file.

If you omit the `;A` option and `fileName` is an existing file, `LF` clears the file before logging begins.

Components

Debugger engine

Example

```
in>lf /mcuez/demo/logfile.txt ;A
```

Opens the `logfile.txt` file as a Log File in "append" mode.

NOTE If the path is unspecified, the destination directory is the current project directory.

LOAD

The `LOAD` command loads a framework application (`.abs` file) for a debugging session. When no application name is specified, the **LoadObjectFile** dialog opens.

If no target is installed, the following error message appears:

```
Error: no target is installed
```

If no target is connected, the following error message appears:

```
Error: no target is connected
```

Usage

```
LOAD[applicationName]
```

or

```
LOAD[applicationName] [CODEONLY|SYMBOLSONLY]
[NOPROGRESSBAR] [NOBPT] [NOXPR] [NOPRELOADCMD]
[NOPOSTLOADCMD] [VERIFYFIRST|VERIFYALL|VERIFYONLY]
[AUTOERASEANDFLASH] [NORUNAFTERLOAD|RUNANDSTOPAFTERLOAD
= functionName|RUNAFTERLOAD] [DELAY] [ADD_SYMBOLS]
```

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.
- `RUNAFTERLOAD` runs application after loading
- `RUNANDSTOPAFTERLOAD` runs application after loading and set temporary breakpoint at the specified function
- `functionName` is the name of the function to set temporary breakpoint at
- `NORUNAFTERLOAD` doesn't run application after loading (default)
- `ADD_SYMBOLS` appends the symbol information to the existing symbol table instead of replacing it

NOTE By default, the LOAD command is `code+symbols` with no verification.

NOTE If you use the `ADD_SYMBOLS` parameter, the debugger plays `PRELOAD` and `POSTLOAD` files for the first loaded application only.

Components

Debugger engine

Example

```
LOAD FIBO.ABS
```

Loads the `FIBO.ABS` application.

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

LOADCODE

This command loads code into the target system. Use this command when no debugging is needed. If no target is installed, the following error message appears:

```
Error: no target is installed
```

If no target is connected, the following error message appears:

```
Error: no target is connected
```

Usage

```
LOADCODE [applicationName]
```

Components

Debugger engine

Example

```
LOADCODE FIBO.ABS
```

Loads `FIBO.ABS` application code.

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

LOADSYMBOLS

This command is similar to the `LOAD` command but only loads debugging information into the debugger. Use this command 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 appears:

```
Error: no target is installed
```

If no target is connected, the following error message appears:

```
Error: no target is connected
```

Usage

```
LOADSYMBOLS [applicationName]
```

Components

Debugger engine

Example

```
LOADSYMBOLS FIBO.ABS
```

Loads debugging information of the `FIBO.ABS` application. If no path is specified, the destination directory is the current project directory.

LOG

The `LOG` command enables or disables information logging in the Command Line component window (and to logfile, when 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 appear in red in Command Line component. Protocol messages are not errors.
- **about NOTICES**: Notices appear 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
```

Does not record error messages in the Log File. Records commands entered in the Command Line component window.

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 “**-**”.

Example 1

Executing the following command file:

```
define truth = 1
IF truth
  bckcolor blue
  at 2000 bckcolor white
else
  bckcolor yellow
  at 1000 bckcolor white
ENDIF
```

Generates the following log file:

```
!define truth = 1
!IF truth
!  bckcolor blue
!  at 2000 bckcolor white
!else
!-  bckcolor yellow
!-  at 1000 bckcolor white
!ENDIF
```

When commands executed from a command file are logged, all executed commands that are in the `FOR` loop are logged the number of times they have been executed. That is, a command file executed with the `CF` or `CALL` command without the `NL` option and with the `CMDFILE` flag of the `LOG` command set to `TRUE`.

Example 2

Executing the following file:

```
define i = 1
FOR i = 1..3
   ls
ENDFOR
```

Generates the following log file:

```
!define i = 1
!FOR i = 1..3
!   ls
i             0x1  (1)
!ENDFOR
!   ls
i             0x2  (2)
!ENDFOR
!   ls
i             0x3  (3)
!ENDFOR
```

When commands executed from a command file are logged, all executed commands that are in the `WHILE` loop are logged as many times as they are executed. That is, a command file executed with the `CF` or `CALL` command without the `NL` option and with the `CMDFILE` flag of the `LOG` command set to `TRUE`.

Example 3

Executing the following file:

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

Generates the following log file:

```
!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 as many times as they are executed. That is, a command file executed with the `CF` or `CALL` command without the `NL` option and with the `CMDFILE` flag of the `LOG` command set to `TRUE`.

Example 4

Executing the following file:

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

Generates the following log file:

```
repeat
until condition
!define i = 1
!REPEAT
!   define i = i + 1
! ls
i             0x2 (2)
!UNTIL i == 4
!   define i = i + 1
! ls
i             0x3 (3)
!UNTIL i == 4
!   define i = i + 1
! ls
i             0x4 (4)
!UNTIL i == 4
```

LS

In the Command Line window, the `LS` command lists the values of symbols defined in the symbol table and by the user. There is no limit to the number of symbols that can be listed. Memory size determines the symbol table size. Use the DEFINE command to define symbols, and the UNDEF command to delete symbols.

The `LS` command lists symbols split into two parts: Applications Symbols and User Symbols.

Usage

```
LS [symbol | *][;C|S]
```

Where `symbol` is a restricted regular expression that specifies the symbol whose values are to be listed.

`*` specifies to list all symbols.

`;C` specifies to list symbols in canonical format, which consists of a `DEFINE` command for each symbol.

`;S` specifies to list symbol table statistics following the list of symbols.

Components

Debugger engine

Example

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

Performing `LS` on a single symbol (e.g., `in > ls counter`) that is an application variable as well as a user symbol, displays the application variable.

Example with `j` being an application symbol as well as a user symbol:

```
in>ls j
Application Symbol:
j             0x83  (131)
```

MEM

The `MEM` command displays a representation of the current system memory map and lower and upper boundaries of the internal module that contains the MCU registers.

Usage

MEM

Components

Debugger engine

Example

```
in>mem

Type     Addresses    Comment

-------------------------------------------------------

IO          0..   3F   PRU or TOP    TOP board resource or
the PRU

NONE       40..   4F   NONE

RAM        50..  64F   RAM

NONE      650..  7FF   NONE

EEPROM    800..  A7F   EEPROM

NONE      A80..3DFF   NONE

ROM      3E00..FDFF   ROM

IO       FE00..FE1F   PRU or TOP    TOP board resource or
the PRU

NONE     FE20..FFDB   NONE

ROM      FFDC..FFFE   ROM

COP      FFFF..FFFF   special ram for cop

RT MEM      0.. 3FF   (enabled)

-------------------------------------------------------
```

MS

The `MS` command sets a specified block of memory to a specified list of byte values. Specifying a `range` that is wider than the list of byte values repeats the list of byte values as many times as necessary to fill the memory block.

When the `range` is not an integer multiple of the length of `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`: an address range constant that defines the block of memory to be set to the values of the bytes in the list.

`list`: a list of byte values to be stored in the block of memory.

Components

Debugger engine

Example

```
in>MS 0x1000..0x100F 0xFF
```

Fills the memory range between addresses `0x1000` and `0x100F` with the `0xFF` value.

NB

The `NB` command changes or displays the default number `base` for the constant values in expressions. The initial default number base is 10 (decimal). Use `NB` to change to base 16 (hexadecimal), base 8 (octal), base 2 (binary), or reset to base 10. Always specify the base 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 22.6.

Usage

```
NB [base]
```

Where `base` is the new number base (2, 8, 10 or 16).

Components

Debugger engine

Table 22.6 ANSI C Constant Notation

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

Table Example

```
0x2F00, /* Hexadecimal Constant */
043,    /* Octal Constant */
255     /* Decimal Constant */
```

In the same way, the debugger supports the **Assembler** notation for constant. That means that independent of the current number base you can specify hexadecimal, octal or binary constants using the **Assembler** prefixes shown in Table 22.7.

Table 22.7 Assembler Notation for Constant

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

Table Example

```
$2F00,  /* Hexadecimal Constant */
@43,    /* Octal Constant */
%10011  /* Binary Constant */
```

When the default number base is 16, constants starting with a letter A, B, C, D, E or F must be prefixed either by `0x` or by `$`, as shown in Table 22.8. Otherwise, the command line interpreter cannot detect if you are specifying a number or a symbol.

Table 22.8 Base is 16: Constants Starting with Letter A, B, C, D, E or F

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

Table Example

```
in>NB 16
```

The number base is hexadecimal.

NOCR

The `NOCR` command closes the current record file. Open the record file with the CR command.

Usage

```
NOCR
```

Components

Debugger engine

Example

```
in>NOCR
```

Closes the current record file.

NOLF

The `NOLF` command closes the current Log File. Open the log file with the LF command.

Usage

```
NOLF
```

Components

Debugger engine

Example

```
in>NOLF
```

Closes the current Log File.

OPEN

Use the `OPEN` command 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

Specify `I` to activate the icons in the component window; specify `MAX` to maximize the component window.

Component names are: Assembly, Command, Coverage, Data, Inspect, Memory, Module, Procedure, Profiler, Recorder, Register, Source, Stimulation.

Components

Debugger engine

Example

```
in>OPEN Terminal 0 78 60 22
```

Opens the Terminal component and window at the specified positions and with specified width and height.

OUTPUT

With `OUTPUT`, you can redirect the Coverage component results to an output file indicated by the file name and his path.

Usage

```
OUTPUT FileName
```

Where `FileName` is `file name (path + name)`.

Components

Coverage component

Example

```
in>coverage < OUTPUT c:\Freescale\myfile.txt
```

Redirects the Coverage output results to the file `myfile.txt` from the directory `C:\Freescale`.

P

The `P` command executes a CPU instruction, either at a specified address or at the current instruction indicated 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 executes, 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 executes, 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`: an address constant expression, the address at which execution begins.

If you omit `address`, execution begins with the instruction indicated by the current value of the program counter.

Components

Debugger engine

Example

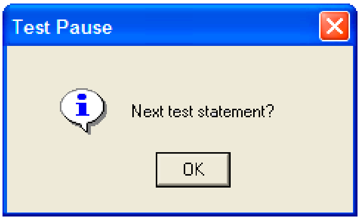
```
in>p

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


00F04E 94                 TXS

STEPPED
```

Contents of registers are displayed and the current instruction is disassembled.

PAUSETEST

Displays a modal message box shown in Figure 22.1 for testing purpose.

Figure 22.1 Test Pause Message Box



Usage

`PAUSETEST`

Components

Debugger engine

Example

```
in> pausetest
```

PRINTF

The `PRINTF` is the standard ANSI C command: Prints formatted output to the standard output stream.

Usage

`PRINTF ("[Text ]%format specification" , value)`

Components

Debugger engine

Example

```
in>PRINTF("The value of the counter is: %d", counter)
```

The output is: `The value of the counter is: 2`

PTRARRAY

Use the `PTRARRAY` command to display a pointer as an array.

Usage

```
PTRARRY on|off [nb]
```

Where:

- `on` displays pointers as arrays.
- `off` displays pointers as usual (`*pointer`).
- `nb` is the number of elements to display in the array when unfolding a pointer displayed as array.

Components

Data component

Example

```
in>Ptrarray on 5
```

Display content of pointers as array of five items.

RD

The `RD` command displays the content of specified registers. The register display includes both the name and hexadecimal representation. If the specified register is not a CPU register, then the debugger looks for an I/O register in a register file called `MCUIxxxx.REG` (where `xxxx` is a number related to the MCU).

NOTE This command is processor/derivative specific. Banked registers are not displayed unless the processor supports banking.

Usage

```
RD { <list> | CPU | * }
```

where `list` is a list of registers to be displayed. Separate registers to be displayed by a space. Specifying `RD CPU` displays all CPU registers. This command displays an error message of `No CPU loaded` if no CPU is found.

When you specify `*`, the `RD` command lists the content of the currently loaded register file. If no register file is loaded, the RD command displays a `No register file loaded` error message.

Specifying the `RD` command with no parameter processes the previous `RD` command again. The first `RD` command of a session displays all CPU registers.

If you omit `list`, the `RD` command uses the `list` and any other parameters from the previous **RD** command.

Components

Debugger engine

Example 1

```
in>rd a hx
A=0x14
HX=0x2
```

Example 2

```
in>rd cpu
A=0x0 HX=0x450 SR=0x70 PC=0xF04E SP=0xFF
```

RECORD

In the **SoftTrace component**, the `RECORD` command switches frame recording `on` or `off` while the target is running.

Usage

```
RECORD on|off
```

Components

SoftTrace component

Example

```
in>RECORD on
```

REPEAT

Use the `REPEAT` command to execute a sequence of commands until a specified condition is true. You may nest the `REPEAT` command.

Click the **Esc** key to stop this command.

Usage

```
REPEAT
```

Components

Debugger engine

Example

```
DEFINE var = 0
...
REPEAT
  DEFINE var = var + 1
  ...
UNTIL var == 2
```

The `REPEAT-UNTIL` loop is identical to the ANSI C loop. The operation `DEFINE var = var + 1` is done twice, then `var = = 2` and the loop ends.

RESET

In the **Profiler and Coverage component**, the `RESET` command resets all recorded frames (statistics).

In the **SoftTrace component**, the `RESET` command resets statistics and recorded frames.

NOTE Make sure that you redirect the `RESET` command to the correct component. Since targets have their own `RESET` command, using `RESET` without redirecting it to the correct component resets the target.

Usage

```
RESET
```

Components

Profiler and Coverage

Example

```
in>Profiler < RESET
```

RESTART

Resets execution to the first line of the current application and executes the application from this point.

Usage

```
RESTART
```

Components

Engine component

Example

```
in>RESTART
```

The `RESTART` command initializes the cycle counter to zero.

RETURN

The `RETURN` command terminates the current command processing level (returns from a CALL command). If executed within a command file, control returns to the caller of the command file (i.e. the first instance that did not chain execution).

Usage

```
RETURN
```

Components

Debugger engine

Example

In file `d:\demo\cmd1.txt`:

```
...
CALL d:\demo\cmd2.txt
T
...
                    In file d:\demo\cmd2.txt
...
...
RETURN // returns to the caller
```

The command file `cmd1.txt` calls a second command file `cmd2.txt`. You must use the `RETURN` instruction to return to the caller file. Then the T Trace instruction is executed.

RS

The `RS` command assigns new values to specified registers. Follow the `RS` mnemonic by the register name and new value(s).

Use an equal sign (=) to separate the register name from the value to be assigned to the register; otherwise they must be separated by a space. Set the contents of any number of registers using a single **RS** command. If the specified register is not a CPU register, then the debugger searches for the register as an I/O register in a register file 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 change. String `register` is any of the CPU register names, or name of a register in the register file.

`value`: is an integer constant expression (in ANSI format representation).

Components

Debugger engine

Example

```
in>rs a=0xff hx=0x7fff
```

S

The S command stops execution of the emulation processor. Use the Go (G) command to start the emulator.

NOTE The S command ends as soon as the PC is changed.

Usage

S

Alias

STOP

Components

Debugger engine

Example

```
in>s
STOPPING
HALTED
```

Current application debugging is stopped/halted.

SAVE

The SAVE command saves a specified block of memory to a specified file in Freescale S-record format. Reload the memory block 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 the file specified by `fileName` exists, then omitting this option clears the file before saving the S-records.

Components

Debugger engine

Example

```
in>SAVE 0x1000..0x2000 DUMP.SX ;A
```

Appends the memory range `0x1000..0x2000` to the `DUMP.SX` file.

SAVEBP

The `SAVEBP` command saves all currently loaded `.ABS` file breakpoints into the matching breakpoints file. The matching file has the name of the loaded `.ABS` file but with the `.BPT` extension (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.

Setting `on` stores all breakpoints defined in the current application into the matching `.BPT` file.

Setting `off` prevents the breakpoints defined in the current application from being stored in the matching `.BPT` file.

Use this command only in `.BPT` files. It is similar to the **Save & Restore on load** checkbox in the Controlpoints Configuration Window, which is used to store currently defined breakpoints (`SAVEBP on`) when the user quits the Simulator/Debugger or loads another `.ABS` file.

NOTE For more information about this syntax, refer to BS command and to the Control Points chapter.

Usage

```
SAVEBP on|off
```

Components

Debugger engine

Example

content of the `FIBO.BPT` file

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

SET

Sets a new current target for the debugger by loading the `targetName` component.

Usage

```
SET targetName
```

where `targetName` is `name` without extension of the target to set.

Components

Debugger engine

Example

```
in>SET Sim
```

The debugger’s current target is Simulator.

SETCOLORS

Use the `SETCOLORS` command to change the colors for a specific channel from the Monitor component.

Usage

```
SETCOLORS ( "Name" )  ( Background)  ( Cursor )  ( Grid
)  ( Line )  ( Text )
```

`Name` is the name of the channel to modify.

`Background` is the new color for the channel background (the format is `0x00bbggrr`).

`Cursor` is the new color for the channel cursor (the format is `0x00bbggrr`).

`Grid` is the new color for the channel grid (the format is `0x00bbggrr`).

`Line` is the new color for the channel lines (the format is `0x00bbggrr`).

`Text` is the new color for the channel text (the format is `0x00bbggrr`).

Components

Monitor component

Example

```
in>SETCOLORS "Leds.Port_Register bit 0" 0x00123456
0x00234567 0x00345678 0x00456789 0x00567891
```

This changes the color attributes from the channel `Leds.Port_Register` bit 0 to these new values.

SLAY

Use the `SLAY` command to save the layout of all window components in the main application window to a specified file.

NOTE Layout files usually have a `.HWL` extension. However, you can specify any file extension.

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

Usage

```
SLAY fileName
```

Components

Debugger engine

Example

```
in>slay /hiwave/demo/mylayout.hwl
```

This saves the current debugger layout to `mylayout.hwl` file in the `/hiwave/demo` directory.

SLINE

The `SLINE` command makes a line of the source file visible. If the line is not currently visible, the source scrolls 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 Use a line number between 1 and the number of lines in the source file, or an error message appears.

Usage

```
SLINE line number
```

Components

Source component

Example

```
in>sline 15
```

SMEM

In the **Source component**, the `SMEM` command loads the corresponding module's source text, scrolls to the corresponding text location (the code address) and highlights the statements that correspond to this code address range.

In the **Assembly component**, the `SMEM` command scrolls the Assembly component, shows the location (the assembler address) and select/highlights the memory lines of the address range given as the parameter.

In the **Memory component**, the `SMEM` command scrolls the memory dump component, shows the locations (the memory address) of the address range given as the parameter.

Usage

```
SMEM range
```

Components

Source, Assembly and Memory components

Example

```
in>Memory < SMEM 0x8000,8
```

This scrolls the Memory component window and highlights the specified memory addresses.

SMOD

In the **Source component**, the `SMOD` command loads/displays the corresponding module's source text. If the module is not found, a message is displayed in Command Line window.

In the **Data component**, the `SMOD` command loads the corresponding module's global variables.

In the **Memory component**, the `SMOD` command scrolls the memory dump component and highlights the first global variable of the module.

NOTE The Module component window displays the correct module names. Make sure that you use the correct module name in your command. If the `.abs` is in **HIWARE** format, some debug information is in the object file (`.o`), and module names have an `.o` extension (e.g., `fibo.o`). In **ELF** format, module name extensions are `.c`, `.cpp` or `.dbg` (`.dbg` or program sources in assembler) (e.g., `fibo.c`), since the `.abs` file contains all debugging information and object files are not used. 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. Do not include the path name in the module name. You must specify the module extension (i.e. `.DBG` for assembly sources or `.C` for C sources, etc.).

The debugger searches for the module name in the directories associated with the `GENPATH` environment variable, and displays an error message:

- 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
```

Displays global variables found in the `fibo.c` module in the Data:1 component window.

SPC

In the **Source component**, the `SPC` command loads the corresponding module's source text, scrolls to the corresponding text location (the code address) and highlights the statement that corresponds to this code address.

In the **Assembler component**, the `SPC` command scrolls the Assembly component, shows the location (the assembler address) and select/highlights the assembler instruction of the address given as parameter.

In the **Memory component**, the `SPC` command scrolls the memory dump component, and 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
```

This scrolls the Assembly component window to the address `0x8000` and highlights the associated instruction.

SPROC

In the **Data component**, the `SPROC` command shows local variables of the corresponding procedure stack level.

In the **Source component**, the `SPROC` command loads the corresponding module's source text, scrolls to the corresponding procedure and highlights the statement of this procedure in the procedure chain.

`level = 0` is the current procedure level. `level = 1` is the caller stack level and so on.

NOTE This command is relevant when debugging C source code.

NOTE Giving a procedure level greater than 0 as parameter to the `SPROC` command selects the statement corresponding to the call of the lower procedure.

Usage

```
SPROC level
```

Components

Data and Source components

Example

```
in>Source < SPROC 1
```

This command displays the source code associated with the caller function in the Source component window.

SREC

The `SREC` command initiates the loading of Freescale S-Records from a specified file.

NOTE If the path is unspecified, 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
```

Loads the `DUMP.SX` file into memory.

STEPINTO

The `STEPINTO` command single-steps through instructions in the program, and enters each function call that is encountered.

NOTE This command works while the application is paused in break mode (program is waiting for user input after completing a debugging command).

Usage

```
STEPINTO
```

Components

Debugger engine

Example

```
in>STEPINTO
STEP INTO
TRACED
```

`TRACED` in the status line indicates that the application is stopped by an assembly step function.

STEPOUT

The `STEPOUT` command executes the remaining lines of a function in which the current execution point lies. The next statement displayed is the statement following the procedure call. All of the code is executed between the current and final execution points. Using this command, you can quickly finish executing the current function after determining that a bug is not present in the function.

NOTE This command works while the application is paused in break mode (program is waiting for user input after completing a debugging command).

Usage

```
STEPOUT
```

Components

Debugger engine

Example

```
in>STEPOUT
STEP OUT
STARTED
RUNNING
STOPPED
```

`STOPPED` in the status line indicates that the application is stopped by a step-out function.

STEPOVER

The `STEPOVER` command executes the procedure as a unit, and then steps to the next statement in the current procedure. Therefore, the next statement displayed is the next statement in the current procedure regardless of whether the current statement is a call to another procedure.

NOTE This command works while the application is paused in break mode (program is waiting for user input after completing a debugging command).

Usage

```
STEPOVER
```

Components

Debugger engine

Example

```
in>STEPOVER
STEP OVER
STARTED
RUNNING
STOPPED
```

`STEPPED OVER` (or `STOPPED`) in the status line indicates that the application is stopped by a step over function.

STOP

The `STOP` command stops execution of the emulation processor. Use the `Go` (G) command to start the emulator.

NOTE The `STOP` command ends as soon as the PC changes.

Usage

```
STOP
```

Alias

```
S
```

Components

Debugger engine

Example

```
in>STOP
```

```
STOPPING
```

```
HALTED
```

Current application debugging is stopped.

T

The `T` command executes one or more instructions at a specified address, or at the current address (the address in the program counter). The `T` command traces into subroutine calls and software interrupts. For example, if the current instruction is a Branch to Subroutine instruction (`BSR`), the `T` command traces the `BSR`, 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.

Stop this command by typing the **Esc** key.

Usage

```
T [address][,count]
```

`address`: is an address constant expression, the address where execution begins. If you omit `address`, 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 you omit `count`, only one instruction is traced.

Components

Debugger engine.

Example

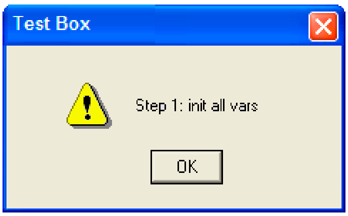
```
in>T 0xF030
TRACED
A=0x0 HX=0x7F02 SR=0x62 PC=0xF032 SP=0x44D


00F032 B787              STA     0x87
```

Displays contents of registers and disassembles the current instruction.

TESTBOX

Displays a modal message box, shown in Figure 22.2, with a given string.

Figure 22.2 Test Box Message Box



Usage

```
TESTBOX "<String>"
```

Components

Debugger engine

Example

```
in>TESTBOX "Step 1: init all vars"
```

TUPDATE

Switches the time update feature on or off.

Usage

```
TUPDATE on|off
```

Components

Profiler and Coverage components

Example

```
in>TUPDATE on
```

UNDEF

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 | *
```

Specifying the `*` argument undefines all symbols previously defined using the `DEFINE` command.

Components

Debugger engine

Example

```
DEFINE test = 1
...
UNDEF test
```

When the test variable is no longer needed in a command program, you can undefine it and remove it from the list of symbols. After `UNDEF test`, the `test` variable must be redefined before you can use it.

NOTE See also examples of the DEFINE command.

Examples

You can change the value of an existing symbol by reapplying the `DEFINE` command, which replaces and loses the previous value. It is not put on a stack. Then when you apply `UNDEF` to the symbol, it no longer exists, even if the value of the symbol has been replaced several times:

```
in>DEFINE apple 0
in>LS
apple          0x0 (0)    // apple is equal to 0
in>DEFINE apple = apple + 1
in>LS
apple          0x1 (1)    // apple is equal to 1
in>DEFINE apple = apple + 1
in>LS
apple          0x2 (2)    // apple is equal to 2
in>UNDEF apple
in>LS
 // apple no longer exists
```

In the next example, we assume that the `FIBO.ABS` sample is loaded. At the beginning, no user symbol is defined:

```
in>UNDEF *
in>LS
User Symbols:    // there is no user symbol
Application Symbols:    // symbols of the loaded
```

```
application
fiboCount    0x800 (2048)
counter      0x802 (2050)
_startupData 0x84D (2125)
Fibonacci    0x867 (2151)
main         0x896 (2198)
Init         0x810 (2064)
_Startup     0x83D (2109)
in>DEFINE counter = 1
in>LS
User Symbols: // there is one user symbol: counter
counter      0x1 (1)
Application Symbols: // symbols of the loaded application
fiboCount    0x800 (2048)
counter      0x802 (2050)
_startupData 0x84D (2125)
Fibonacci    0x867 (2151)
main         0x896 (2198)
Init         0x810 (2064)
_Startup     0x83D (2109)
in>undef counter
in>LS
User Symbols: // there is no user symbol
Application Symbols: // symbols of the loaded application
fiboCount    0x800 (2048)
counter      0x802 (2050)
_startupData 0x84D (2125)
Fibonacci    0x867 (2151)
main         0x896 (2198)
Init         0x810 (2064)
_Startup     0x83D (2109)
```

UNFOLD

In the Source component, use the `UNFOLD` command to display the contents of folded source text blocks; for example, source text that has been collapsed at program block level. All text unfolds once or (`*`) completely, until no more folded parts are found.

Usage

```
UNFOLD [*]
```

Where `*` means unfolding completely, otherwise unfolding only one level.

Components

Source component

Example

```
in>UNFOLD *
```

UNTIL

The `UNTIL` keyword is associated with the REPEAT command.

Usage

```
UNTIL condition
```

Where `condition` is defined as a 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` equals 3.

UPDATERATE

In the **Data component**, **Memory component** and **MCURegisters component** use the `UPDATERATE` command to set the data refresh update rate. This command has an effect only if the Data, Memory or MCURegisters 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 tenths of second (0.1 to 60 seconds).

Components

Data, Memory and MCURegisters component

Example

```
in>Memory < updaterate 30
```

This commands sets the Memory component updaterate to 3 seconds.

VER

The `VER` command displays the version number of the Debugger engine and components currently loaded in the Command line window.

Usage

```
VER
```

Components

Debugger engine

Example

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

Displays Debugger engine and components versions in the Command Line component window.

WAIT

The `WAIT` command pauses command file execution for a time in tenths of second or pauses until the target halts when the option `;s` is set.

Specifying no parameter pauses command file execution for 50 tenths of a second (5 seconds).

Specifying `time` only halts execution of the command file for the specified time.

Specifying `;s` only halts execution of the command file until the target halts. If the target is already halted, command file execution is not halted.

When you specify both `time` and `;s`:

- If the target is running, command file execution halts 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 command file ignores the time delay and runs.
- If the target is halted, command file execution does not halt (command file ignores the time delay).

NOTE The `Wait` instruction ends as soon as the PC changes.

Usage

```
WAIT [time] [;s]
```

Components

Debugger engine

Example

```
WAIT 100
T
...
```

Pauses for 10 seconds before executing the `T` Trace instruction.

WB

The WB command sets a specified block of memory to a specified list of byte values. When the range is wider than the list of byte values, the list of byte values repeats 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`: an address range constant that defines the block of memory equal to the values of the bytes in the list.

`list`: a list of byte values to store in the block of memory.

Alias

```
MS
```

Components

Debugger engine

Example

```
in>WB 0x0205..0x0220 0xFF
```

This command fills up the memory range `0x0205..0x0220` with the `0xFF` byte value.

WHILE

The `WHILE` command allows you to execute a sequence of commands as long as a certain condition is true. You may nest the `WHILE` command.

Stop this command by pressing the **Esc** key.

Usage

```
WHILE condition
```

Where `condition` is defined as in C-language definition.

Components

Debugger engine

Example

```
DEFINE jump = 0
...
WHILE jump < 20
  DEFINE jump = jump + 1
ENDWHILE
T
...
```

While `jump < 100`, the `jump` variable increments by the instruction `DEFINE jump = jump + 1`. Then the loop ends and the `T` Trace instruction executes.

WL

The `WL` command sets a specified block of memory to a specified list of longword values. When the range is wider than the list of longword values, the list of longword values repeats 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 you specify a size in the range, this size represents the number of longwords to modify.

Usage

```
WL range list
```

`range`: an address range constant that defines the block of memory to set to the longword values in the list.

`list`: a list of longword values to store in the block of memory.

Components

Debugger engine

Example

```
in>WL 0x2000 0x0FFFFF0F
```

This command fills up memory starting at address `0x2000` with the `0x0FFFFF0F` longword value, and modifies the addresses `0x2000` to `0x2003`.

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

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

WW

The `WW` command sets a specified block of memory to a specified list of word values. When the range is wider than the list of word values, the list of word values repeats as many time as necessary to fill the memory block. When the range is not an integer or a multiple of length of the list, the last copy of the list is truncated accordingly.

Usage

```
WW range list
```

`range`: an address range constant that defines the block of memory to set to the word values in the list.

`list`: a list of word values to store in the block of memory.

Components

Debugger engine

Example

```
in>WW 0x2000..0x200F 0xAF00
```

This command fills up the memory range `0x2000..0x200F` with the `0xAF00` word value.

ZOOM

In the Data component, use the `ZOOM` command to display the member fields of structures by ‘diving’ into the structure. This is in contrast to the UNFOLD command, where member fields are not expanded in place. The display of the member fields replaces the previous view. Use `ZOOM out` to return to the nesting level indicated by the given identifier.

NOTE You do not need addresses to zoom out. Simply type `ZOOM out`.

NOTE This command is relevant when debugging C source code.

Usage

```
ZOOM address in|out
```

Where `address` is the address of the structure or pointer variable to zoom in or zoom out, respectively.

Components

Data component

Example

```
in>ZOOM 0x1FE0 in
```

Zooms in the variable structure located at address `0x1FE0`.

```
in>zoom &_startupData
```

Zooms in the `_startupData` structure (`&_startupData` is the address of the `_startupData` structure).

Signal Commands

Use the following commands to specify files and file parameters in the Signal component.

SETSIGNALFILE Command

`SETSIGNALFILE` specifies the signal file to use.

Syntax

```
SETSIGNALFILE <value (0-15)> <"file name">
```

Remarks

The `SETSIGNALFILE X` command creates a virtual `SignalGeneratorX` module having a `SignalPin`.

The `file name` can include the path of the file. If you specify no path, the Signal component first searches in the current project folder, then in the `prog\FCSsignals` folder of the debugger installation path.

Example

Create three generators:

```
setsignalfile 0 "sinus_11bit_0_5v_1Hz.txt"
setsignalfile 1 "saw_11bit_0_5v_1Hz.txt"
setsignalfile 2 "square_1_5v_1Hz.txt"
```

Then create virtual pin connections with the Pinconn Commands `CONNECT` command:

```
connect "SignalGenerator0.SignalPin","Atd0.PAD0"
connect "SignalGenerator1.SignalPin","Atd0.PAD1"
connect "SignalGenerator2.SignalPin","Atd0.PAD2"
```

TIP You can place commands to create signal generators in a command file such as a `Postload` command file.

CLOSESIGNALFILE Command

`CLOSESIGNALFILE` closes the current signal file and generator.

Syntax

```
CLOSESIGNALFILE <value (0-15)>
```

Example

```
CLOSESIGNALFILE 1
```

Remarks

A message box displays the line error in case of signal file scripting error.

The Signal component is compatible with cycle time duration modification (bus speed change via PLL) and True Time feature, and automatically reprograms level duration (when duration in seconds is provided or no duration information is provided).

Currently, all header parameters are mandatory, also EOF, in the same order as described in EBNF above, **without spacing between words**.

Pinconn Commands

CONNECT

Connects output pin to input.

Syntax

```
CONNECT "<OutputPin>", "<InputPin>"
```

Example

```
CONNECT "Pim.PORTHPin0", "Pim.PORTPPin3"
```

DISCONNECT

Removes connection between pins.

Syntax

```
DISCONNECT "<OutputPin>", "<InputPin>"
```

Example

```
DISCONNECT "Pim.PORTHPin0", "Pim.PORTPPin3"
```

CONNECT_STATE

Displays the list of active connections.

Syntax

```
CONNECT_STATE
```

NOTE There is no limitation of connections.

NOTE The **Inspect** component provides this list of simulated pins for a derivative FCS, under the **Object Pool** key.

Command Line Options

This section lists the DOS command line options.

NOTE Options are not case sensitive.

-T=<time>: Test mode

The debugger terminates after a specified time (in seconds). The default value is 300 seconds.

Example

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

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

-Target=<targetname>

This option sets the specified connection.

Example

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

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

-W: Wait mode

Debugger waits even when an `<exeName>` is specified.

`-w(W)` command line option for HIWAVE indicates that, in case executable file is specified in command line string, it would be loaded on target but not started. Without `-w` option HIWAVE loads executable on target and run it.

```
C:\Program Files\Freescale\CodeWarrior for
Microcontrollers V6.x\Prog\hiwave.exe c:\Program
Files\Freescale\CodeWarrior for Microcontrollers
V6.x\demo\hc12\sim\fibo.abs -w -Target=sim
```

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

-Instance=%currentTargetName

This option defines a build instance name. The debugger uses the defined build instance.

Example

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

Starting the debugger again brings the existing instance of the debugger to the foreground.

-Prod= <fileName>

This option specifies the startup project directory and/or project file.

Example

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

-Nodefaults

Instructs the debugger not to load the default layout (see section 4 of the Project file Activation).

Example

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

-Cmd = <Command>

This option specifies a command to be executed at startup:

```
-cmd = '"' {characters}.
```

Example

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

-C <cmdFile>

This option specifies a command file to be executed at startup.

Example

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

-ENVpath: "-Env" <Environment Variable> "=" <Variable Setting>

This option sets an environment variable. You may use this environment variable to overwrite system environment variables.

Example

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

23

Connection-Specific Commands

This chapter describes the unique connection-specific commands available for the HC(S)12(X) debugger.

Abatron BDI Connection Commands

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

The *Abatron BDI* Connection-specific commands are:

- BDI
- PROTOCOL
- RESET

Enter these commands in the Abatron BDI Connection Command Files or in the **Command Line** component of the debugger.

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

Table 23.1 Command Description Parameters

Topic	Description
Description	Provides a detailed description of the command and how to use it.
Syntax	Specifies the syntax of the command.
Example	Small example of how to use the command.

BDI

Description

The BDI command executes any ABATRON direct command. See the *ABATRON User Manual* for your CPU for complete descriptions of ABATRON direct commands. Direct commands are commonly used to download to non-volatile memory areas (see also the Flash Programming section).

Syntax

```
BDI <ABATRON_direct_command>
```

where `ABATRON_direct_command` has the following syntax:

```
<Object>.<Action> [<parName>=<parameterValue>]...
```

Example

```
BDI FLASH.ERASE addr=8000 size=8000 sram=0800
```

PROTOCOL

Description

Use this command to switch the Show Protocol functionality on or off, and report all the messages sent to and received from the debugger in the **Command Line** window of the debugger.

You can switch the `Show Protocol` facility on or off using the corresponding check box in the Communication Device Specification dialog box.

The state of the Show Protocol is stored in the `[BDIK]` section of the project file using variable `SHOWPROT`.

Syntax

```
PROTOCOL ON|OFF
```

Example

```
PROTOCOL ON
```

NOTE Use `Show Protocol` to assist when debugging communication problems.

RESET

Description

Use this command to reset the target board from the **Command Line** component of the debugger. This command executes the Reset Command File, and the BDI interface automatically processes the initialization list (startup `init` list) stored in the interface.

Syntax

```
RESET
```

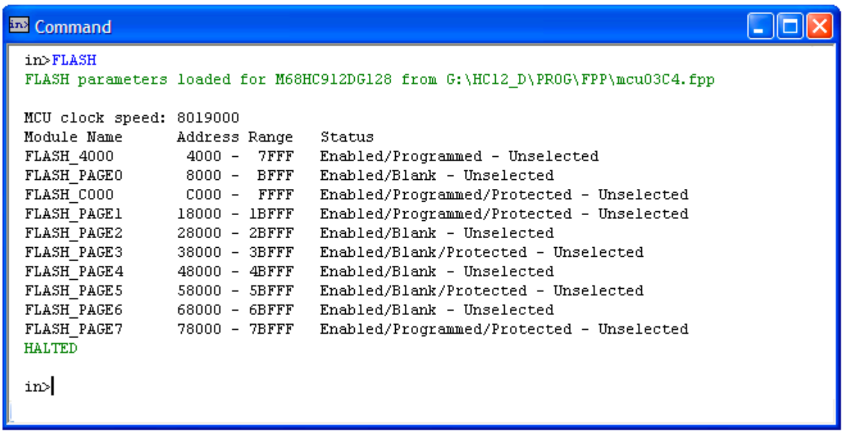
Example

```
RESET
```

NVMC Commands

Issue the following Flash Commands through the debugger Command component window, as shown in the figure below.

Figure 23.1 NVMC Commands In Command Window



FLASH

Description

Displays Flash modules, loads the `.fpp` file, or performs Flash operations. The `FLASH` command displays names, locations, and states of all available modules, provided that a parameter (`.fpp`) file is already loaded. If no parameter file is loaded, this command loads either the `.fpp` file for the current MCUID or the last-used `.fpp` file.

Syntax

```
FLASH [(SELECT|UNSELECT|ERASE|ENABLE|DISABLE|PROTECT|
UNPROTECT|AEFSKIPERASING) [<blockNo>]]
```

```
|[ARM|DISARM|SAVECONTEXT|LOADCONTEXT|MEMMAP|MEMUNMAP|RE
LEASE|OVLBACKUP|OVLRESTORE|PROTOCOLON|PROTOCOLOFF|SKIPS
TATUSON|SKIPSTATUSOFF|NOUNSECURE|UNSECURE]
```

```
|[NVMFREQUENCY <frequency in Hz>]
```

```
|[NVMIF2RELOCATE <address>]
```

```
|[NVMIF2WORKSPACE <address> <address>]
```

```
|[INIT <fileName> | AUTOID]
```

Usage

`FLASH INIT` *`<fileName>`*`|AUTOID` loads the parameter file according to `fileName` (you can specify the path). If this command includes `AUTOID`, the MCUID determines the parameter file (provided **autocheck** is checked in the NVMC dialog box).

`FLASH RELEASE` releases the current `.FPP` file loaded by the Flash Programmer, disabling the Flash Programmer address mapping. No non-volatile memory is handled.

`FLASH MEMMAP` maps the Flash Programmer address filtering to route the code for block programming.

`FLASH MEMUNMAP` unmaps the Flash Programmer address filtering. Programming is disabled as long as `FLASH MEMMAP` is not executed.

`FLASH ENABLE` enables the specified modules. If no modules are specified, enables all available blocks. This command ignores modules that cannot be enabled.

`FLASH DISABLE` disables the specified modules. If no modules are specified, all disables all available blocks. This command ignores modules that cannot be disabled.

`FLASH ERASE` erases the specified modules. If no modules are specified, erases all available blocks.

`FLASH AEFSKIPERASING` specifies non-volatile memory blocks to protect from mass erasing at application automated programming. Place this command in a `Startup` command file. If no modules are specified, no blocks are erased.

NOTE This command is compatible and replicated in the **NVM Programming Selection** dialog.

`FLASH UNPROTECT` unprotects the specified modules. If no modules are specified, unprotects all available blocks. This command ignores modules that cannot be unprotected.

`FLASH PROTECT` protects the specified modules. If no modules are specified, protects all available blocks. This command ignores modules that cannot be protected.

`FLASH SELECT` selects the specified modules for Flash programming. If no modules are specified, selects all available blocks for Flash programming.

`FLASH UNSELECT` deselects the specified modules. If no modules are specified, deselects all available blocks. The deselected state protects against accidental Flash programming.

`FLASH ARM` prepares the NMVC utility for loading; as does a normal `LOAD` command. The system executes the `VPPON.CMD` file specified in the Command Files user interface. This command is required before loading Flash.

`FLASH DISARM` ends a load process. The system executes the `VPPOFF.CMD` file specified in the Command Files user interface.

`FLASH SAVECONTEX` backs up current SRAM content into a buffer.

`FLASH LOADCONTEX` restores current buffer content into the MCU SRAM.

`FLASH OVLBACKUP` backups application code overlap with programming runtime/algorithm (RAM preset for debugging). Execute this command before the application/file loading.

`FLASH OVLRESTORE` restores/installs (writes in RAM) the application code overlap with programming runtime/algorithm. Execute this command after the last `FLASH` command.

`FLASH PROTOCOLON` displays the Flash Programmer debug protocol.

`FLASH PROTOCOLOFF` stops displaying the Flash Programmer debug protocol.

`FLASH SKIPSTATUSON` skips the Flash Programmer device Non-Volatile Memory blocks diagnostic. Use this command to speed up project application loading and programming from the IDE **debug** run. When used, the Flash Programmer does NOT verify whether blocks are programmed or erased.

`FLASH SKIPSTATUSOFF` removes the `SKIPSTATUSON` mode and therefore diagnostics are performed again.

`FLASH NOUNSECURE` asserts that the security byte location is **not** programmed to set the device to unsecure mode.

WARNING! Unless the user application programs the security byte location to an unsecured state, then after the next hardware reset, the debugger will be unable to communicate via BDM with the device, and debugging will fail.

`FLASH UNSECURE` asserts that the security byte location is programmed to set the device to unsecure mode.

CAUTION The user application **cannot** overwrite the security byte and linked byte cells (2-byte word programming or 8-byte row programming are usually required by specifications).

`FLASH NVMFREQUENCY <frequency in Hz>` specifies the non-volatile memory programming frequency in Hertz, typically the device bus speed after reset. When used, the Flash Programmer does not try to evaluate this speed and the debugger gain 2-3 seconds at application loading time. A value of 0 reengages the speed detection.

`FLASH NVMIF2RELOCATE <address>` tells the Flash programmer to load the Flash driver in RAM to a non-default location (default is the start of on-chip RAM). This provides more flexibility for **EB386 Example 1 Layout device** RAM memory relocation. The `data to program` buffer follows the same address translation. This command is a Legacy command; `FLASH NVMIF2WORKSPACE` is more user friendly and performs a secured relocation. A value of 0 resets the location.

`FLASH NVMIF2WORKSPACE <address> <address>` tells the Flash programmer to load the Flash driver in RAM to a non-default location (default is at the start of on-chip RAM). The command also resizes the workspace, by passing a range as a parameter. The command is more powerful than `FLASH NVMIF2RELOCATE`, although you must set up the range correctly to match the targeted part. `FLASH NVMIF2RELOCATE 0` resets any setup made with the `FLASH NVMIF2WORKSPACE` or `FLASH NVMIF2RELOCATE` commands. Ideally, execute this command from a `Startup.cmd` file. For example:

```
FLASH NVMIF2WORKSPACE 0x3800 0x3FFF
```

The command implies that on-chip RAM is available at relocation position and range before loading any Flash driver. This command provides more flexibility for **EB386 Example 1 Layout device** RAM memory relocation.

[<blockNo>]

Description

`blockNo` is a list of Flash block or module numbers.

Syntax

```
blockNo = {number["-"number][","]}
```

Examples

```
FLASH ERASE 2,7
```

This erases memory blocks 2 and 7.

```
FLASH ERASE 2,4-6 8
```

This erases memory blocks 2, 4, 5, 6, and 8.

```
FLASH ERASE
```

This erases all available memory blocks.

While Flash modules are armed, execution of user code is not possible. If you enter a command such as run or step, a message box prompts you to disarm the modules or cancel the command. If you click the **OK** button, the system disarms all Flash modules, then executes your command. If you click the **CANCEL** button, the system cancels the command and leaves the Flash modules armed.

Listing 23.1 Flash Programming Example from Command Line in Component Window

```
in>Flash

FLASH parameters loaded for M68HC912DG128 from
J:\HC12_EA\PROG\FPP\mcu03C4.fpp

MCU clock speed: 8025000
Module Name       Address Range   Status
FLASH_4000        4000 -  7FFF   Enabled/Blank - Unselected
FLASH_PAGE0       8000 -  BFFF   Enabled/Blank - Unselected
FLASH_C000        C000 -  FFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE1      18000 - 1BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE2      28000 - 2BFFF   Enabled/Blank - Unselected
```

```
FLASH_PAGE3      38000 - 3BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE4      48000 - 4BFFF   Enabled/Blank - Unselected
FLASH_PAGE5      58000 - 5BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE6      68000 - 6BFFF   Enabled/Blank - Unselected
FLASH_PAGE7      78000 - 7BFFF   Enabled/Blank/Protected - Unselected
HALTED
```

As used in Listing 23.1, the FLASH command loads the applet that corresponds to the CPU derivative (MCUID) and displays the state of all modules.

To program an application into module number 7 (FLASH_PAGE5), you must unprotect the module, as in Listing 23.2.

Listing 23.2 Unprotect Module

```
in>Flash unprotect 7

MCU clock speed: 8025000
Module Name       Address Range    Status
FLASH_4000         4000 -  7FFF    Enabled/Blank - Unselected
FLASH_PAGE0        8000 -  BFFF    Enabled/Blank - Unselected
FLASH_C000         C000 -  FFFF    Enabled/Blank/Protected - Unselected
FLASH_PAGE1       18000 - 1BFFF    Enabled/Blank/Protected - Unselected
FLASH_PAGE2       28000 - 2BFFF    Enabled/Blank - Unselected
FLASH_PAGE3       38000 - 3BFFF    Enabled/Blank/Protected - Unselected
FLASH_PAGE4       48000 - 4BFFF    Enabled/Blank - Unselected
FLASH_PAGE5      58000 - 5BFFF   Enabled/Blank/Unprotected - Unselected
FLASH_PAGE6       68000 - 6BFFF    Enabled/Blank - Unselected
FLASH_PAGE7       78000 - 7BFFF    Enabled/Blank/Protected - Unselected
```

The updated display resulting from this code shows that FLASH_PAGE5 is unprotected. To select FLASH_PAGE5 for programming, enter:

```
in>Flash select 7
```

To arm for programming:

```
in>Flash arm
```

Now load your application:

```
in>load a:\my_page5.sx
RUNNING
```

To stop loading and disarm:

```
in>Flash disarm
FLASH disarmed.
Halted
```

Use the FLASH command to display the final state of the modules (Listing 23.3).

Listing 23.3 Display Module Final State

```
in>Flash
MCU clock speed: 8025000
Module Name    Address Range   Status
FLASH_4000      4000 -  7FFF   Enabled/Blank - Unselected
FLASH_PAGE0     8000 -  BFFF   Enabled/Blank - Unselected
FLASH_C000      C000 -  FFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE1    18000 - 1BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE2    28000 - 2BFFF   Enabled/Blank - Unselected
FLASH_PAGE3    38000 - 3BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE4    48000 - 4BFFF   Enabled/Blank - Unselected
FLASH_PAGE5   58000 - 5BFFF   Enabled/Programmed/Unprotected - Selected
FLASH_PAGE6    68000 - 6BFFF   Enabled/Blank - Unselected
FLASH_PAGE7    78000 - 7BFFF   Enabled/Blank/Protected - Unselected
HALTED
```

The FLASH_PAGE5 module is programmed. Now, protect and unselect the module (Listing 23.4).

Listing 23.4 Protect and Unselect Module

```
in>Flash protect 7

MCU clock speed: 8025000
Module Name      Address Range   Status
FLASH_4000        4000 -  7FFF   Enabled/Blank - Unselected
FLASH_PAGE0       8000 -  BFFF   Enabled/Blank - Unselected
FLASH_C000        C000 -  FFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE1      18000 - 1BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE2      28000 - 2BFFF   Enabled/Blank - Unselected
FLASH_PAGE3      38000 - 3BFFF   Enabled/Blank/Protected - Unselected
FLASH_PAGE4      48000 - 4BFFF   Enabled/Blank - Unselected
FLASH_PAGE5     58000 - 5BFFF   Enabled/Programmed/Protected - Selected
FLASH_PAGE6      68000 - 6BFFF   Enabled/Blank - Unselected
FLASH_PAGE7      78000 - 7BFFF   Enabled/Blank/Protected - Unselected

in>Flash unselect 7
```

DMM Commands

You can make all DMM GUI settings using debugger command line commands within the Command component window or from a command file.

Debugging Memory Map Manager Commands

The following list of commands allows you to fully script the debugging device memory mapping. It is recommended to limit the use of these commands to special debugging purposes, as the default mapping is typically sufficient, and script setups can be complex and may lead to debugger errors.

NOTE The Debugging Memory Map Manager and associated command set are available with hardware connections only.

List of Commands

```
DMM
DMM ADD <parameters>
DMM DEL <module handle>
DMM SAVE <mcuid>
DMM DELETEALLMODULES
DMM RELEASECACHES
DMM CACHINGON|CACHINGOFF
DMM WRITEREADBACKON|WRITEREADBACKOFF
DMM HCS12MERHANDLINGON|HCS12MERHANDLINGOFF
DMM OPENGUI [mcuid]
DMM SETAHEADREADSIZE <front size when halted> <back size
when halted> <front size when running> <back size when
running>
```

DMM

Description

Displays the current DMM Memory Types, Overlap Priorities and memory ranges in the Command window.

Syntax

```
DMM
```

DMM ADD

Description

Inserts a new memory range in the DMM, as if added via the DMM dialog/user interface.

Syntax

```
DMM ADD <comment> <address> <size> <handle> <type> <cache
locking> <priority> <mapping> <access while running>
<access kind> <access size>
```

Arguments

`<comment>`: a string for Comment field; use "£" for " " (space).

`<address>`: the start address of the memory range

`<size>`: the size of the memory range

`<handle>`: a long value allowing the DMM to handle the memory range (duplicated handled is not allowed).

WARNING! User-defined handles must have a value greater than or equal to 100.

`<type>`: a value corresponding to a memory type handle, as given/listed in the DMM command.

`<cache locking>`: a 0 or 1 value, 0 forces a memory range refresh after each debugger halt.

`<priority>`: a value corresponding to an overlap priority handle/value, as given/listed with the DMM command.

`<mapping>`: a 0 or 1 value; 1 enables memory range mapping.

`<access while running>`: a 0 or 1 value; 1 enables memory range access while running.

`<access kind>`: 0 for R/W; 1 for write only; 2 for read only; 3 for none.

DMM DEL

Description

Deletes one specific DMM memory range module by handle reference.

Syntax

```
DEL <module handle>
```

Arguments

`<module handle>`: a memory range module handle as given/listed with the DMM command.

DMM SAVE

Description

Saves the DMM current setup in current `project.ini` file, under `DMM_MCUIDxxxx_MODULEn=...` keys.

Syntax

```
DMM SAVE <mcuid>
```

Arguments

`<mcuid>`: a part/device MCUID value in range `$0-$FFFF`.

DMM DELETEALLMODULES

Description

Removes all current DMM memory range modules. Use to start a scripted DMM setup.

Syntax

```
DMM DELETEALLMODULES
```

DMM RELEASECACHES

Description

Flushes all currently cached data once for each memory range module, even if cache locking is active (i.e. no refresh on halting is active).

Syntax

```
DMM RELEASECACHES
```

DMM CACHINGON

Description

Engages data caching (default DMM setup). Refresh on halting is inactive for memory range modules defined with this option.

Syntax

```
DMM CACHINGON
```

DMM CACHINGOFF

Description

Disables data caching. The debugger flushes all caches even for memory range modules defined without this option. Each time the debugger halts, the memory data for all memory range modules is retrieved from the target hardware.

Syntax

```
DMM CACHINGOFF
```

DMM WRITEREADBACKON

Description

Upgrades only the cached data in the matching memory location when the debugger writes data to a memory location. For example, if the debugger performs a `WB 0x80 0x01` command, the debugger reads back only the byte at address 0x80 and upgrades its internal cache. This is the default behavior of the debugger.

Syntax

```
DMM WRITEREADBACKON
```

DMM WRITEREADBACKOFF

Description

Clears the cached data of the entire DMM range, including this location, when the debugger writes data to a memory location. For example, if the debugger performs a `WB 0x80 0x01` command, the debugger reads back the entire block of memory around the location. You can use this legacy implementation to perform a live update on IO registers belonging to the same IO module, although a Memory window **Refresh** operation is more relevant and keeps the default debugger setup.

Syntax

```
DMM WRITEREADBACKOFF
```

DMM HCS12MERHANDLINGON

Description

Enables the handling of Memory Expansion Registers (MER) for HCS12 devices, i.e., `INITRM`, `INITRG` and `INITEE`. The DMM automatically remaps memory range module addresses according to the real value of these registers when halting.

NOTE The debugger does not poll the MER registers while running. The debugger performs remapping only on factory-defined memory range modules, not user-defined memory range modules.

Syntax

```
DMM HCS12MERHANDLINGON
```

DMM HCS12MERHANDLINGOFF

Description

Completely disables `DMM HCS12MERHANDLINGON`.

Syntax

```
DMM HCS12MERHANDLINGOFF
```

DMM OPENGUI

Description

Opens the DMM Graphical User Interface.

Syntax

```
DMM OPENGUI [mcuid]
```

Arguments

`<mcuid>`: (optional) a part/device MCUID value in range `$0-$FFFF`.

DMM SETAHEADREADSIZE

Description

Provides special debugger memory cache tuning in case of slow connection with hardware.

Syntax

```
DMM SEATAHEADREADSIZE <front size when halted> <back size
when halted> <front size when running> <back size when
running>
```

Arguments

`<front size when halted>`: amount of bytes to read ahead of exact start of read block address, when the hardware is halted.

`<back size when halted>`: amount of bytes to read after the exact block of read addresses, when the hardware is halted.

`<front size when running>`: amount of bytes to read ahead of exact start of read block address, when the hardware is running.

`<back size when running>`: amount of bytes to read after the exact block of read addresses, when the hardware is running.

Full Chip Simulator Commands

Use simulator environment commands to monitor the debugger environment, specific component window layouts, and framework applications and targets. Table 23.2 contains all available Environment commands.

Table 23.2 Full Chip Simulator Commands

Command, Syntax	Short Description
SETCPU ProcessorName	Sets a new CPU simulator
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

Component-specific commands are associated with specific components supported by the Full Chip Simulator. Table 23.3 contains all available Component Specific commands.

Table 23.3 List of Component-Specific Commands

Command, Syntax	Short Description
ADCPORT (address | ident) (address | ident) (address | ident)	Sets the ports addresses used by the ADC_DAC component.
ADCPORT (<Name>)	Creates a new channel <Name> for the Monitor component.
CPORT (address | ident) (address | ident) (address | ident) (address | ident) (address | ident)	Sets the five port addresses and the control port address of the IO_Ports component
DELCHANNEL (<Name>)	Deletes a specific channel for the Monitor component.
ITPORT (address | ident) (address | ident)	Sets the line and column port addresses of the IT_Keyboard component
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
PBPORT (address | ident)	Sets the port address of the Push_Buttons component
PORT address	Sets the LED components port address
REGBASE <Address><;R>	Sets or resets the base I/O address.
SEGPORT (address | ident) (address | ident)	Sets the display selection port and the segment selection port addresses of the 7-Segments display component.

Table 23.3 List of Component-Specific Commands (*continued*)

Command, Syntax	Short Description
SETCONTROL (<Name>) (Ticks) (Pixels)	Changes the number of `ticks` and `pixels` for the `<Name>` channel from the Monitor component
WPORT (address | ident) (address | ident	Sets the ports addresses of the Wagon component

ADCPORT

Use the `ADCPORT` command to set the port addresses used by the ADC_DAC component.

Syntax

```
ADCPORT ( address | ident ) ( address | ident )
( address | ident )
```

Arguments

`Address:` locates the port address value of the component. The default format is hexadecimal (many formats are allowed).

`Ident:` a known identifier. Its content defines the port address.

Components

ADC_DAC component.

Example

```
in>ADCPORT 0x100 0x200 0x300
```

Defines the ports of the ADC_DAC component at the addresses 0x100, 0x200 and 0x300.

ADDCHANNEL

Use the `ADDCHANNEL` command to create a new channel for the Monitor component.

Syntax

```
ADDCHANNEL ( <Name> )
```

Arguments

`<Name>`**:** the name for the new channel.

Components

Monitor component.

Example

```
in>ADDCHANNEL "Leds.Port_Register bit 0"
```

Creates a new channel, `Leds.Port_Register` bit 0, in the Monitor component.

CPORT

Use the `CPORT` command to set the five coupler-port addresses and the control port address of the coupler component.

Syntax

```
CPORT ( address | ident ) ( address | ident ) ( address
| ident )
```

Arguments

`Address`**:** locates the port address value of the component. The default format is hexadecimal (many formats are allowed).

`Ident`: a known identifier. Its content defines the port address.

Components

Programmable Parallel Couplers component.

Example

```
in>CPORT 0x100 0x200 0x300
```

This defines the ports of the programmable parallel couplers at addresses 0x100, 0x200 and 0x300.

DELCHANNEL

Use the `DELCHANNEL` command to delete a specific channel for the Monitor component.

Syntax

```
DELCHANNEL ( <Name> )
```

Arguments

`Name`: the name of the channel to delete.

Components

Monitor component.

Example

```
in>DELCHANNEL "Leds.Port_Register bit 0"
```

Deletes the channel `Leds.Port_Register` bit 0 in the Monitor component.

ITPORT

Use the `ITPORT` command to set the line and column port addresses of the IT_Keyboard component.

Syntax

```
ITPORT ( address | ident ) ( address | ident ) ( address
| ident )
```

Arguments

`Address:` locates the port address value of the component. The default format is hexadecimal (various formats are allowed).

`Ident`: a known identifier. Its content defines 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

Use the `ITVECT` command to set the interrupt vector port address of the IT_Keyboard component.

Syntax

```
ITVECT ( address | ident )
```

Arguments

`Address:` locates the port address value of the component. The default format is hexadecimal (various formats are allowed).

`Ident`: a known identifier. Its content defines the port address.

Components

IT_Keyboard component.

Example

```
in>ITVECT 0x400
```

Defines the interrupt vector port address of the IT_Keyboard at address 0x400.

KPORT

Use the `KPORT` command to set the line and column ports addresses of the Keyboard component.

Syntax

```
KPORT ( address | ident ) ( address | ident ) ( address
| ident )
```

Arguments

`Address`: locates the port address value of the component. The default format is hexadecimal (many formats are allowed).

`Ident`: a known identifier. Its content defines the port address.

Components

Keyboard component.

Example

`in>KPORT 0x100 0x200 0x300`

Defines the ports of the Keyboard at addresses 0x100, 0x200 and 0x300.

LCDPORT

Description

Use the `LCDPORT` command to set the data port and the control port address of the LCD component.

Syntax

`LCDPORT ( address | ident ) ( address | ident ) ( address | ident )`

Arguments

`Address`: locates the port address value of the component. The default format is hexadecimal (many formats are allowed).

`Ident`: a known identifier. Its contents define the port address.

Components

LCD component.

Example

`in>LCDPORT 0x100 0x200`

Defines the ports of the LCD at addresses 0x100, 0x200 and 0x300.

LINKADDR

Use the `LINKADDR` command to set the components internal ports addresses used with the Programmable Couplers as memory buffers.

Syntax

`LINKADDR ( address | ident ) ( address | ident ) ( address | ident )`

Arguments

`Address:` locates the port address value of the component. The default format is hexadecimal (many formats are allowed).

`Ident`: a known identifier. Its content defines the port address.

Components

Couplers, Keyboard, 7-segments display,

Example

`in>LINKADDR 0x100 0x200 0x300 0x400 0x500`

All components working with the Programmable Couplers have

- PortA set to 0x100
- PortB set to 0x200
- PortC set to 0x300
- PortD set to 0x400
- PortE set to 0x500

PBPORT

Use the `PBPORT` command to set the port address of the Push_Buttons component.

Syntax

`PBPORT ( address | ident )`

Arguments

`Address:` locates the port address value of the component. The default format is hexadecimal (various formats are allowed).

`Ident:` a known identifier. Its content defines the port address.

Components

Push_Buttons component.

Example

`in>PBPORT 0x100 0x200`

Defines the ports of the Push_Buttons at addresses `0x100` and `0x200`.

PORT

In the **LED components**, the PORT command sets the port LED location.

Syntax

```
PORT address
```

Components

LED component.

Example

```
in> PORT 0x210
```

REGBASE

This command allows you to change the base address of the I/O registers or to set (Reset) this address to 0.

Syntax

```
Regbase <Address><;R>
```

Arguments

Address: an address to define the base address of the I/O registers,

R : sets this address to 0 (Reset).

Components

Debugger engine.

Example

```
in>regbase 0x500
```

0x 500 is now the base address of the I/O registers.

RESETCYCLES

This command sets the Simulator CPU cycles counter to the user-defined value. If not specified, the value is 0. The Debugger status and Register component displays the cycles counter. This command does not affect the context.

Syntax

```
RESETCYCLES <Value>
```

Arguments

`Value`: 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

Marks the given range of memory (RAM + ROM) as uninitialized (undefined).

Syntax

```
RESETMEM range
```

Components

Simulator component.

Example

`in>RESETMEM`

This initializes all configured memory to undefined.

`in>RESETMEM 0x100...0x110`

This resets the memory located between `0x100` and `0x110` (if configured) to undefined.

`in>RESETMEM 0x003F`

This resets the memory location `0x003F` (if configured) to undefined.

NOTE In the *Auto on Access* memory configuration the full memory is defined as RAM, so `RESETMEM` has the same effect as `RESETRAM`.

RESETRAM

This command marks all RAM as uninitialized (undefined).

NOTE In the memory configuration *Auto on Access*, the full memory is defined as RAM, so `RESETMEM` has the same effect as `RESETRAM`.

Syntax

`RESETRAM`

Components

Simulator component.

Example

`in>RESETRAM`

After the `RESETRAM` command, the content of RAM is undefined.

RESETSTAT

This command resets the statistics (read and write counters) to zero

Syntax

```
RESETSTAT
```

Components

Simulator component.

Example

```
in>RESETSTAT
```

The `RESETSTAT` command initializes all counters to zero.

SEGPORT

Use the `SEGPORT` command to set the display selection port and segment selection port addresses of the 7-Segments display component.

Syntax

```
SEGPORT display selection port ( address | ident )
segment selection ( address | ident )
```

Arguments

`Address:` locates the port address value of the component. The default format is hexadecimal (many formats are allowed).

`Ident`: a known identifier. Its content defines the port address.

Components

7-Segments display

Example

```
in>SEGPORT 0x100 0x200
```

The ports of the 7-Segments display are now defined at addresses 0x100 and 0x200.

SETCONTROL

Use the `SETCONTROL` command to modify the number of ticks and pixels for a Monitor component specific channel. This changes the horizontal scale of this channel.

Syntax

```
SETCONTROL ( <Name> )  ( <Ticks> )  ( <Pixels> )
```

Arguments

`<Name>`: the name of the channel to modify.

`<Ticks>`: the new number of ticks for this channel.

`<Pixels>`: the new number of pixels for this channel.

Components

Monitor component.

Example

```
in>SETCONTROL "Leds.Port_Register bit 0" 100 1
```

This defines the horizontal scale from the channel `Leds.Port_Register` bit 0 with the value 100 for the `Ticks` value and 1 for `pixels` value.

SETCPU

Loads CPU awareness for the debugger.

Syntax

```
SETCPU ProcessorName
```

Arguments

`ProcessorName:` a supported processor (`HC08`, `HC11`, `HC12`, `HC16`, `M68K`, `XA`, and `PPC`).

Components

Simulator component.

Example

```
in>SETCPU HC12
```

Loads the `HC12.sim` simulator

SHOWCYCLES

The `SHOWCYCLES` command returns the number of CPU cycles already done since the beginning of the simulation in the Command Line component (performs `RESETCYCLES` internally), or since the last RESETCYCLES command. Also displays the number of cycles executed in the status bar (CPU cycles counter).

Syntax

```
SHOWCYCLES
```

Components

Debugger engine.

Example

```
in>SHOWCYCLES
133801
in>RESETCYCLES
in>SHOWCYCLES
0
```

This command displays the number of CPU cycles executed since the last `RESETCYCLES` command in the Command Line component.

WPORT

Use the `WPORT` command to set the port addresses of the Wagon component.

Syntax

```
WPORT   ( address | ident )  ( address | ident )
```

Arguments

`Address`: locates the port address value of the component (various formats are allowed), the default format is hexadecimal.

`Ident`: a known identifier. Its content defines the port address.

Components

Wagon

Example

```
in>WPORT 0x100 0x200
```

Ports of the Wagon are now defined at addresses `0x100` and `0x200`.

Full Chip Simulation Connection Commands

This section describes the Full Chip Simulation connection-specific commands that are used when the Full Chip Simulation (FCS) connection is set.

The Full Chip Simulation connection-specific commands are:

- ADCx_SETPAD
- BGND_CYCLES
- HALT_ON_TRAP
- HCS12_SUPPORT
- MESSAGE_HIDE_ID
- MESSAGE_HIDE_RESET
- MESSAGE_SHOW_ID
- PSMODE
- SELECTCORE
- STACK_AREA_CHECK
- STACK_POINTER_INFO
- WARNING_SETUP

Enter these commands in any command files that will be executed by the debugger or in the **Command Line** component of the debugger.

ADCx_SETPAD

Description

Sets PAD pin to a given voltage in floating point format, on ADC module `x`

Sets the value of an input channel of an Analog-to-Digital Converter to the specified voltage. The module name is an integral part of the command name. The voltage is given as a float constant value in volts.

The allowed range is from 0.0 to 5.12 Volts.

Syntax

```
<moduleName>_SETPAD <channel> <voltage>
```

Example

```
ADC1_SETPAD  4  2.5
```

This sets the input of channel 4 of the ADC1 module to 2.5 volts.

```
ATD2_SETPAD  2  1.5
```

This sets the input of channel 2 of the ATD2 module to 1.5 volts

BGND_CYCLES

Description

This command allows you to adapt the simulator's clock cycles for the BGND instruction, by specifying the number of cycles it takes to execute a BGND instruction. The default value is five CPU cycles.

Syntax

```
BGND_CYCLES <number>
```

Example

```
BGND_CYCLES  10
```

A BGND instruction now takes 10 CPU cycles to execute.

HALT_ON_TRAP

Description

Stops on call to exception handler.

This command allows you to specify the address of an interrupt handler (start address of an ISR) using either the ISR name or an expression. During simulation of the exception processing, this address is compared with the fetched vector. If they match, the simulation stops instead of calling the exception handler.

Syntax

```
HALT_ON_TRAP (OFF | <interrupt_function> | <expression>)
```

Example

Source code of exception handler:

```
interrupt MyISR(void) {
    ...
}
```

Command:

```
HALT_ON_TRAP  MyISR
```

Instead of calling the function `MyISR` because of an exception, the simulator stops.

HCS12_SUPPORT

Description

Enables HCS12-specific core emulation modes.

NOTE Typically, use this command only to override automatic debugger settings done when selecting the derivative (by GUI or project wizard). The `HCS12X_MAP4000` option may still be relevant, as it is not covered by wizard project setup, and is not automatically preset (always set by default to FLASH).

Syntax

```
HCS12_SUPPORT ( ? | ON | OFF | HC12 | HCS12 | HCS12X |
HCS12XE | STATUS ) [ HCS12X_FLASH=<num> ]
```

```
[ XGATE_RAM=<num> ] [ HCS12X_MAP4000=( FLASH | RAM |
EXTERNAL )]
```

Arguments

`OFF` : HC12 simulator is in legacy CPU12 mode.

`ON` or `HCS12` : HC12 simulator is in HCS12 core mode.

`HCS12X` : enables the HCS12X family instruction set

`HCS12XE` : enables the instruction set of the HCS12XE family (superset of HCS12X).

`STATUS` : returns the current state of emulation.

`XGATE_FLASH=` : sets the size of the device Flash.

`XGATE_RAM=` : sets the size of the XGATE RAM.

`HCS12X_MAP4000=` : Supports alternative mapping of memory range `0x4000...0x7FFF` and defines the mapping of the memory range `0x4000` to `0x7FFF` to one of these memory types:

- `FLASH` : maps non-banked FLASH (default)
- `RAM` : maps non-banked RAM
- `EXTERNAL` : maps external space.

NOTE The `HCS12X_MAP4000` option is designed for the HCS12XE family, for the MMCCTL1 register, ROMHM and RAMHM flags.

CAUTION Match the setup of the debugger with the `HCS12X_MAP4000` option with the HC12 Compiler Code Generation `Define mapping for memory space 0x4000..0x7FFF` option in the compiler options settings dialog (i.e., `-MapRAM`, `-MapFlash`, or `-MapExternal` command line options).

Example

```
in>HCS12_SUPPORT HCS12XE HCS12X_MAP4000=RAM

in>HCS12_SUPPORT status

HC12 simulator is currently in HCS12XE mode

HCS12X_FLASH size is 0x80000

XGATE_RAM size is 0x8000

0x4000..0x7FFF maps to RAM
```

MESSAGE_HIDE_ID

Description

Hides a message with a specific ID.

Components

Debugger engine.

Syntax

```
MESSAGE_HIDE_ID  <message number(ID)>
```

Example

```
in>MESSAGE_HIDE_ID 1
in>warning_setup status
WARNING_SETUP STATUS: CLMSG
Hidden message ID: 1
```

MESSAGE_HIDE_RESET

Description

Resets all hidden messages to display them again.

Components

Debugger engine.

Syntax

```
MESSAGE_HIDE_RESET
```

Example

```
in>MESSAGE_HIDE_RESET
```

All previously hidden messages are displayed again now.

MESSAGE_SHOW_ID

Description

Shows a message with specific ID.

Components

Debugger engine.

Syntax

```
MESSAGE_SHOW_ID  <message number(ID)>
```

Example

```
in>MESSAGE_SHOW_ID 1
```

PSMODE

Description

This command changes the power save mode.

Syntax

```
PSMODE (STOP | WAIT | WAKEUP)
```

Arguments

`STOP` : places the CPU in its lowest power consumption mode; halts all internal CPU processing.

`WAIT` : places the CPU in low power consumption; halts all internal CPU processing, except the internal clock, the programmable timer, SPI and SCI remain active (for more detail see the appropriate microcontroller manual). This option consumes more power than the `STOP` option.

`WAKEUP` : turns off the low power consumption mode; the processor resumes normal processing.

Example

```
in>PSMODE STOP  /* The processor is completely stopped */
in>PSMODE WAKEUP /* The processor is out of power save
```

```
mode */
```

SELECTCORE

Description

Select CPU12 or XGATE as a current core.

This is the same as selecting **HCS12X FCS > Select Core** menu item.

NOTE Only for HCS12X and HCS12XE derivatives.

Syntax

```
SELECTCORE ( ? | CPU12 | XGATE )
```

Example

```
in>SELECTCORE XGATE
```

Selects XGATE as a current core.

STACK_AREA_CHECK

Description

Controls special checks if SP remains in the stack area.

Syntax

```
STACK_AREA_CHECK ( ? | AUTO | OFF | ON low..high | STATUS)
```

Arguments

`?` : displays the help

`AUTO` : enables stack checking. Reads the range information from the ELF file. This option works only if an ELF file is actually loaded. It does not work with a HIWARE format object file or with an SRECORD. An area specified with `ON` overrules the area set by the `AUTO` setting.

`OFF` : disables the stack checking (default). This command only has an effect after enabling it with the `ON` or `AUTO` command.

`ON <low..high range>` : enables stack checking and sets the low and high range. If the SP goes out of the `low..high range` area, the simulation stops with a stack overflow status.

`STATUS` : prints a message which tells whether stack checking is currently enabled, and if so which area is used.

Example

```
STACK_AREA_CHECK ON 0x1234..0x1245
```

STACK_POINTER_INFO

Description

Prints minimum and maximum value of the SP register.

Syntax

```
STACK_POINTER_INFO (? | RESET | INFO)
```

Arguments

`?` : displays the help

`RESET` : resets the collection stack pointer values. Future `INFO` calls only report the SP area from now on.

`INFO` : prints the area the SP was pointing to since the last RESET.

Example

```
STACK_POINTER_INFO RESET
```

WARNING_SETUP

Description

The `WARNING_SETUP` command sets the level of debugger warning to inform you about the usage of an unsimulated register.

Components

Debugger Engine

Syntax

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

Arguments

TIP For HALT, CLMSG and MSGBOX options, executing the command more than once toggles the setup.

STATUS: Displays the current warning setup status.

HALT: FCS stops/halts the debugger when a warning message occurs.

CLMSG: Displays warning messages in the Command window.

MSGBOX: A message box pops up on warning. Clicking Cancel stops the FCS. Clicking OK resumes the FCS.

NONE: Clears all warning messages.

Example

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

Example

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

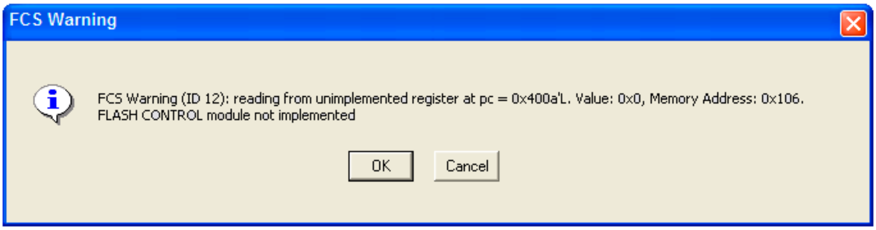
Example

```
in>warning_setup none
in>warning_setup clmsg
in>warning_setup status
WARNING_SETUP STATUS: CLMSG
```

Example

```
in>warning_setup none
in>warning_setup msgbox
in>warning_setup status
WARNING_SETUP STATUS: MSGBOX
```

Figure 23.2 FCS Warning Message Box



Example

```
in>warning_setup none
in>warning_setup status
WARNING_SETUP STATUS: No warning messages
```

NOTE With `HALT`, `CLMSG` and `MSGBOX` options, executing the command several times toggles the setup on and off.

On-Chip Hardware Breakpoint Module Commands

The following sections describe the Hardware Breakpoint Settings Command Line command used by the Target Interface. This command is:

- HWBPM

You can enter this command in the Target Interface associated command files or in the Command Line component of the debugger.

HWBPM

Description

The command `HWBPM` allows you to set up the debugger to work with the on-chip hardware breakpoints dialog.

- Use `HWBPM` with no parameters to get the current breakpoints settings.
- Use `HWBPM MODE` to specify which module to use, and debugger response when using the on-chip hardware breakpoint module and the on-chip module

address. This command has the same effect as the *Break Modules Settings* index tab in the Hardware Breakpoint Configuration dialog.

- The HWBPM SET16BITS command has the same effect as the *16-bits Break Module (User Mode)* index tab in the Hardware Breakpoint Configuration dialog. Parameters set using are only relevant when the *User controlled* mode is active and the 16-bits break module is used.
- The HWBPM SET22BITS command has the same effect as using the *22-bits Break Module (User Mode)* index tab in the Hardware Breakpoint Configuration dialog. Parameters set using this command are only relevant when the *User controlled* mode is active and the 22-bits break module is used.

NOTE The hardware breakpoints settings are stored in the ["targetName"] section of the PROJECT file using variable HWBPMn.

When using the 22-bits module, use the HWBPM REMAP_22BITS commands to perform page remapping, and to set breakpoints in non-banked memory areas when using this on-chip break module. When selecting a derivative, the debugger uses this command to set up the corresponding remapping needed for the specified derivative.

- Use HWBPM REMAP_22BITS DISPLAY to display all the currently set remapping for the selected derivative.
- Use HWBPM REMAP_22BITS RANGE to specify that the prefix <mask> must be used to set a hardware breakpoint in range <start address> <end address>
- Use HWBPM REMAP_22BITS MCUID_DEFAULT to retrieve the derivative default setting (in case it has been modified using HWBPM REMAP_22BITS RANGE or HWBPM REMAP_22BITS DELETE)
- Use HWBPM REMAP_22BITS DELETE <range number> to delete a specific range. Display the range number using HWBPM REMAP22BITS DISPLAY.

NOTE The remapping range is stored in the ["targetName"] section of the PROJECT file using variable HWBPD_MCUIDnnn_BKPT_REMAPn.

Syntax

```
HWBPM
```

```
HWBPM MODE <MODE> BPM16BITS|BPM22BITS <module adr.>
```

```
    [SKIP_OFF|SKIP_ON]
with MODE = DISABLED|AUTOMATIC|USER

HWBPM SET16BITS <BRKCT0 value> <BRKCT1 value> <BRKA value>
    <BRKD value>
HWBPM SET22BITS <BKPCT0 value> <BKPCT1 value> <BKP0 value>
    <BKP1 value>
HWBPM REMAP_22BITS RANGE <start address> <end address> <mask>
HWBPM REMAP_22BITS DISPLAY
HWBPM REMAP_22BITS MCUID_DEFAULT
HWBPM REMAP_22BITS DELETE <range number>
```

Example

Retrieve the Hardware Breakpoints mechanism settings by typing `HWBPM` without any parameters in the Command Line component:

```
in>HWBPM
Hardware Breakpoints Module Settings:
Module kind:    22BITS
Module mode:    Automatic
Module address: 0x28
Skip illegal BP (16bits only): off
HWBPM 16 bits: BRKCT0: 0x0 BRKCT1: 0x0 BRKA: 0x0 BRKD:
0x0
HWBPM 22 bits: BKPCT0: 0x0 BKPCT1: 0x0 BKP0: 0x0 BKP1:
0x0
```

Modify the current Module mode to User controlled and the on-chip hardware breakpoint module to 16-bits (relevant only if present on the hardware):

```
in>HWBPM MODE USER BPM16BITS 0x20 SKIP_OFF
in>HWBPM
Hardware Breakpoints Module Settings:
Module kind:    16BITS
Module mode:    User Defined
Module address: 0x20
Skip illegal BP (16bits only): off
HWBPM 16 bits: BRKCT0: 0x0 BRKCT1: 0x0 BRKA: 0x0 BRKD:
0x0
HWBPM 22 bits: BKPCT0: 0x0 BKPCT1: 0x0 BKP0: 0x0 BKP1:
0x0
```

Enter values in the on-chip breakpoint module registers:

```
in>HWBPM SET16BITS 0xa4 0x0 0xc004 0x0
in>HWBPM
Hardware Breakpoints Module Settings:
Module kind:    16BITS
Module mode:    User Defined
Module address: 0x20
Skip illegal BP (16bits only): off
HWBPM 16 bits: BRKCT0: 0xa4 BRKCT1: 0x0 BRKA: 0xc004
BRKD: 0x0
HWBPM 22 bits: BKPCT0: 0x0 BKPCT1: 0x0 BKP0: 0x0 BKP1:
0x0
```

Display the currently set remapping:

```
in>HWBPM REMAP_22BITS DISPLAY
HWBPM Remappings for 0x3CA:
Range0:    0x4000..0x7FFF mask: 0x3e
Range1:    0xC000..0xFFFF mask: 0x3f
```

Add a new remapping:

```
in>HWBPM REMAP_22BITS RANGE 0x8000 0xbfff 0x47
in>HWBPM REMAP_22BITS DISPLAY
HWBPM Remappings for 0x3CA:
Range0:    0x4000..0x7FFF mask: 0x3e
Range1:    0xC000..0xFFFF mask: 0x3f
Range2:    0x8000..0xBFFF mask: 0x47
```

Delete a remapping:

```
in>HWBPM REMAP_22BITS DELETE 1
in>HWBPM REMAP_22BITS DISPLAY
HWBPM Remappings for 0x3CA:
Range0:    0x4000..0x7FFF mask: 0x3e
Range1:    0x8000..0xBFFF mask: 0x47
```

Retrieve the default remapping for the currently set derivative:

```
in>HWBPM REMAP_22BITS MCUID_DEFAULT
in>HWBPM REMAP_22BITS DISPLAY
HWBPM Remappings for 0x3CA:
Range0:    0x4000..0x7FFF mask: 0x3e
Range1:    0xC000..0xFFFF mask: 0x3f
```

Unsecure Commands

The following sections describe the HCS12 Unsecure Command Line command used by the Target Interface. This command is:

- CHIPSECURE

Enter this command in the Target Interface associated command files or in the Command Line component of the debugger.

CHIPSECURE

Description

The CHIPSECURE SETUP command allows you to set up the debugger unsecure mechanism.

The CHIPSECURE UNSECURE command allows you to unsecure the connected derivative. This is the same as selecting **HC12MultilinkCyclonePro > Unsecure** and using the same settings.

Using CHIPSECURE UNSECURE executes the Unsecure command file and performs the secured derivative check process. To find out if the derivative is unsecured, the debugger reads **<addr. reg to check>**, masks it with **<mask>** and compares it to **<compare value>**.

Arguments

`<addr. reg to check>` : address of the security register (0xFF0F default)

`<mask>` : comparison mask for the security register (0x03 default)

`<compare value>` : comparison value for the security register (0x02 default)

Syntax

```
CHIPSECURE UNSECURE
```

```
CHIPSECURE SETUP <addr. reg to check> <mask> <compare
value>
```

Example

The following command sets up the CHIPSECURE for most HCS12 derivatives:

```
in>CHIPSECURE SETUP 0xFF0F 0x3 0x2
```

```
in>
```

XGATE-Specific Hardware Connection Commands

This section describes a set of commands that are used when debugging with a hardware connection (i.e. not for Simulation) on a device with an XGATE core.

The specific commands are:

- HCS12X_MAP4000

- SELECTCORE
- STEPBOTHCORES
- XDBG*
- XGATECODERANGE
- XGATECODERANGESRESET

Enter these commands in any command files to be executed by the debugger or in the debugger **Command Line** component.

HCS12X_MAP4000

Description

Use this command to indicate to the debugger, for the S12X series, where the `$4000-$7FFF` memory range is mapped. By default, it is mapped to FLASH.

NOTE Place this command in a Startup command file.

Maps the S12X `$4000-$7FFF` range to RAM, FLASH or EXTERNAL memory.

Syntax

```
HCS12X_MAP4000 FLASH|RAM|EXTERNAL
```

Example

```
in>HCS12X_MAP4000 RAM
$4000-$7FFF memory range mapped to RAM.
```

NOTE The `HCS12X_MAP4000` command is designed for the HCS12XE family, for the MMCCTL1 register, and the ROMHM and RAMHM flags.

CAUTION Match the setup of the debugger with the `HCS12X_MAP4000` option with the HC12 Compiler Code Generation `Define mapping for memory space 0x4000..0x7FFF` option in the compiler options settings dialog (i.e., `-MapRAM`, `-MapFlash`, or `-MapExternal` command line options).

SELECTCORE

Description

Select CPU12 or XGATE as a current core

This is the same as selecting Select Core menu item in target specific menu.

Syntax

```
SELECTCORE ( ? | CPU12 | XGATE )
```

Example

```
in>SELECTCORE XGATE
```

Selects XGATE as a current core.

STEPBOTHCORES

Description

Single steps XGATE and HCS12X core at the same time. Disabled by default.

WARNING! This is a simulation and does not match with any real-time instruction cycling.

Syntax

```
STEPBOTHCORES <ON|OFF>
```

Example

```
STEPBOTHCORES ON
```

XDBG*

Description

XGDBG bit debugger setup when starting and stopping the debugger. XDBG* commands define how the debugger sets the XGATE core when halting and starting the main (e.g. HCS12X) core.

Syntax

```
XGDBGDONTSETONSTOP
XGDBGAUTOONSTOP
XGDBGCLEARONRUN
XGDBGDONTCLEARONRUN
XGDBGAUTOONRUN
```

Arguments

XGDBGDONTSETONSTOP: Does not set XGDBG bit on stop.

XGDBGAUTOONSTOP: Sets XGDBG bit automatically on stop if XGFRZ bit is set (default mode).

XGDBGCLEARONRUN: Clears XGDBG bit on run.

XGDBGDONTCLEARONRUN: Does not clear XGDBG bit on run.

XGDBGAUTOONRUN: Clears XGDBG bit automatically on run if XGFRZ bit is set (default mode).

Example

```
XGDBGDONTSETONSTOP
```

XGATECODERANGE

Description

Defines the XGATE code memory area in RAM. If using this command you must properly insert breakpoints in XGATE code.

TIP You can extend address values with quotes to specify address spaces: `'L` for logical, `'X` for XGATE and `'G` for global.

Syntax

```
XGATECODERANGE <first address> <last address>
```

Example

```
XGATECODERANGE 0x800'X 0xFFFF'X
```

XGATECODERANGESRESET

Description

Removes all `XGATE` code memory ranges inserted with the `XGATECODERANGE` command.

Syntax

```
XGATECODERANGESRESET
```

Example

```
XGATECODERANGESRESET
```

Other Hardware Connection Commands

This section describes the other hardware connection commands (i.e. not for Simulation) that might be provided for a connection.

HWBREAKONLY

Description

Forces the debugger to use only hardware breakpoints without attempting to try to use BGND software breakpoint patching.

TIP Use this command when implementing and debugging flash programming algorithms executed from Flash. This avoids the Flash access error flag caused by a BGND instruction write attempt in the array.

Syntax

```
HWBREAKONLY OFF | ON | STATE
```

TIP For some connections, this command might be associated to a GUI checkbox in the connection setup dialog.

Example

```
HWBREAKONLY ON
```

ISRDISABLEDSTEP

Description

This command forces the debugger to disable maskable interrupts while stepping by setting the CCR `I` bit each time some assembly or single steps occur. The debugger cares about setting back the flag to its initial state, based on the results of the stepped instruction (that might also affect the `I` flag).

NOTE The debugger corrects the `I` flag stacking, according to the initial flag value.

TIP For some connections, this command might be associated to a GUI checkbox in the connection setup dialog.

Syntax

```
ISRDISABLEDSTEP  OFF|ON|STATE
```

Example

```
ISRDISABLEDSTEP ON
```

24

Debugger Engine Environment Variables

This chapter describes the environment variables that the Debugger uses. Other tools, such as the Linker, also use some of these environment variables. For more information about other tools, see their respective manuals.

Debugger Environment

Use environment variables to set the various debugger parameters. The syntax is always the same:

```
Parameter = KeyName "=" ParamDef.
```

NOTE Do not use blanks in the definition of an environment variable.

For example:

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/lib;/
home/me/my_project
```

You can define the debugger parameters in several ways:

- Use system environment variables supported by your operating system.
- Put 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.

- Put definitions in a file given by the value of the system environment variable `ENVIRONMENT`.

NOTE Set the default directory using the `DEFAULTDIR` system environment variable (see 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, the debugger assumes a default value.

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 operating system or the launching program determines the current directory of a tool. For MS Windows-based operating systems, the current directory definition is more complex.

- If you launch the tool using a File Manager/Explorer, the current directory is the location of the executable launched.
- If you launch the tool using an Icon on the Desktop, the current directory is the one specified and associated with the Icon.
- If you launch the tool by dragging a file on the icon of the executable under Windows NT® 4.0 operating system or Windows 2000® operating system, the desktop is the current directory.
- If you launch the tool from another tool with its own current directory specified (for example, WinEdit), the current directory is the one specified by the launching tool.
- For the Debugger tools, the current directory is the directory containing the local project file. Changing the current project file also changes the current directory, if the other project file is in a different directory. Note that browsing for a C file does not change the current directory.

To overwrite this behavior, use the `DEFAULTDIR` environment variable (see DEFAULTDIR: Default Current Directory).

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 no `MCUTOOLS.INI` file exists in this directory, the tool looks for the file in the MS Windows installation directory (for example, `C:\WINDOWS`).

Example

```
C:\WINDOWS\MCUTOOLS.INI
D:\INSTALL\PROG\MCUTOOLS.INI
```

If you start a tool in the `D:\INSTALL\PROG\DIRECTORY`, the tool uses the project file located in the same directory as the tool (`D:\INSTALL\PROG\MCUTOOLS.INI`).

If you start the tool outside the `D:\INSTALL\PROG` directory, the tool uses the project file located in the Windows directory (`C:\WINDOWS\MCUTOOLS.INI`).

NOTE For more information about `MCUTOOLS.INI` entries, see the compiler manual.

Local Configuration File (usually project.ini)

The debugger does not change the read-only `default.env` file. The configuration file contains and stores all configuration properties. Different applications can use the same configuration file.

The shell only uses the configuration file, named `project.ini`, located in the current directory. It is suggested that you use this name for the Debugger configuration file. The debugger can use the editor configuration written and maintained by the shell only when the shell and the compiler use the same file. Apart from this, the Debugger can use any file name for the project file. The configuration file has the same format as Windows `.ini` files. The Debugger stores its own entries with the same section name as in the global `mcutools.ini` file.

The current directory is always the directory containing the configuration file. If you load a configuration file from a different directory, then the current directory also changes. Changing the current directory reloads the `default.env` file. Loading or storing a configuration file reloads the options in the environment variable `COMPOPTIONS` and adds these options to the project options. Beware of this behavior when different `default.env` files exist in different directories, as they may contain incompatible options in `COMPOPTIONS`.

Loading a project using the first `default.env` adds its `COMPOPTIONS` to the configuration file. If you store this configuration in a different directory, where a `default.env` file exists with incompatible options, the Debugger attempts to add the options to the `default.env` file 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 you 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 you wish to use in the future.

Load the configuration at startup using one of three ways:

- use the command line option `prod`
- the `project.ini` file in the current directory
- or **Open Project** entry from the file menu.

If you use the `prod` option, then the current directory is the directory containing the project file. Specifying a directory with `prod` loads the `project.ini` file in this directory.

Default Layout Configuration (PROJECT.INI)

The `PROJECT.INI` file located in the project directory defines the default layout activated when starting the Debugger, as shown in Listing 24.1. The `[DEFAULTS] section` contains all default layout-related parameters.

Listing 24.1 Example Content of PROJECT.INI

```
[HI-WAVE]
Window0=Source      0   0  60  30
Window1=Assembly   60   0  40  30
Window2=Procedure   0  30  50  15
Window3=Terminal    0  45  50  15
Window4=Register   50  30  50  30
Window5=Memory     50  60  50  30
Window6=Data        0  60  50  15
Window7=Data        0  75  50  15
Target=Sim
```

`Target`: Specifies the target used when starting the Debugger (loads the file `<target>` with a `.tgt` extension), for example, `Target=Sim` for HC(S)12(X) Freescale Full Chip Simulator, or `Target=Motosil`, `Target=Bdi`.

`Window<n>`: Specifies coordinates of the windows that must be open when the Debugger is started. The syntax for a window is:

`Window<n>=<component> <XPos> <YPos> <width> <height>`

where `n` is the index of the window. This index increments for each window and determines the sequence in which windows open. This index determines which windows appear on top when windows overlap. Values for the index must be in the range 0–99.

`component` specifies the type of component to open, for example, **Source** or **Assembly**.

`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: 50% from the main window width, 60% from the main window height. Its width is 50% from the main window width and its height is 30% from the main window height.

Other Parameters

You can 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 You can specify the layout path if the layout is not in the project directory.

NOTE If you define `Layout` in `PROJECT.INI`, the `Layout` parameter overwrites any `Window<n>` definition, describing the default windows layout.

You can load a previously saved project from a file by inserting the following line in your `PROJECT.INI` file:

`Project=<ProjectName>`

where `ProjectName` is the name of the file describing the project to be loaded, for example, `Project=Proj1.hwc`

NOTE You can specify the project path if the project is not in the project directory. Use this option for compatibility with the old `.hwp` format (`Project=oldProject.hwp`). The file opens as a new project file.

See File Menu for more information about Projects.

NOTE If you define both `Layout` and `Project` in `PROJECT.INI`, the `Project` parameter overwrites the `Layout` parameter, which also contains layout information.

`MainFrame=<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>,<nbr.>`

Use this variable to save and load the Debugger main window states: positions, size, maximized, minimized, icons used when open, etc. This entry is used for internal purposes only.

You can specify the toolbar, status bar, heading line, title bar and small border in the default section:

- Show or hide the toolbar using the following parameters and syntax:

  `Toolbar = (0 | 1)`

  Specify 1 to show the toolbar, otherwise it is hidden.
- Show or hide the status bar using the following parameters and syntax:

  `Statusbar = (0 | 1)`

  Specify 1 to show the status bar, otherwise it is hidden.
- Show or hide the title bars using the following parameters and syntax:

  `Hidetitle = (0 | 1)`

  Specify 1 to hide the title bars, otherwise they show.
- Show or hide the heading lines using the following parameters and syntax:

  `Hideheadlines = (0 | 1)`

  Specify 1 to hide the heading lines. otherwise they show.
- Reduce the border using the following parameters and syntax:

  `Smallborder = (0 | 1)`

  Specify 1 for thin borders, otherwise they appear 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

Activating a project file (`PROJECT.INI`) initiates the following actions (from first action to last):

- Closes the old Project file
- Unloads the Target Component
- Adds the environment variable (Path) from the Project file.

Select HI-WAVE section from which to retrieve the value:

- If you can retrieve a `Windows0` or `Target` entry from the `[HI-WAVE]` section then, use `[HI-WAVE]`
- If you can retrieve a `Windows0` or `Target` entry from the `[DEFAULTS]` section then use `[DEFAULTS]`
- Otherwise, use `[HI-WAVE]`

The debugger loads the environment variables from the `default.env` file.

If an entry `Layout=lll` exists, the debugger loads and executes the layout file `lll.hwl`.

The debugger sets the target (if entry `Target=ttt` exists, load target `ttt`).

If an entry `Project=ppp` exists, the debugger executes the `ppp` command file.

The debugger loads the configuration file (`*.hwc`)
(`entry configuration=*.hwc`).

Environment Variable Paths

Most environment variables contain path lists indicating where to search for files. A path list is a list of directory names separated by semicolons following the syntax below:

```
PathList = DirSpec {";" DirSpec}.
```

```
DirSpec  = ["*"] DirectoryName.
```

Example:

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/hiwave/
lib;/home/me/my_project
```

If an asterisk ("*") precedes a directory name, the programs recursively search the directory tree for a file, not just the given directory. Directories are searched in the order they appear in the path list.

Example:

```
GENPATH=.\;*S;O
```

NOTE Some DOS environment variables (like GENPATH and LIBPATH) are used.

We strongly recommend working with WinEdit and setting the environment by means of a `DEFAULT.ENV` file in your project directory. You can set this 'project directory' 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

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
```

Results 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
```

Search Order for Source Files

This section describes the search order (from first to last) used by the debugger.

In the Debugger for C Source Files (*.c, *.cpp)

1. Path coded in the absolute file (`.abs`)
2. Project file directory (where the `.pjt` or `.ini` file is located)
3. Paths defined in the GENPATH environment variable (from left to right)
4. Abs File directory

In the Debugger for Assembly Source Files (*.dbg)

1. Path coded in the absolute file (`.abs`)
2. Project file directory (where `.pjt` or `.ini` file is located)
3. Paths defined in the GENPATH environment variable (from left to right)
4. Abs File directory

In the Debugger for Object Files (HILOADER)

1. Path coded in the absolute file (.abs)
2. Abs File directory
3. Project file directory (where `.pjt` or `.ini` file is located)
4. Path defined in the OBJPATH environment variable
5. Paths defined in the GENPATH environment variable (from left to right)

Debugger Files

The Debugger comes with several program, application, configuration files and examples. Table 24.1 lists these files and file extensions.

Table 24.1 Debugger Files and File Extensions

File Extension	Description
`*.ABS`	Absolute framework application file (e.g., `fibo.abs`)
`*.ASM`	Assembler specific file (e.g., `macrodem.asm`)
`*.BBL`	Burner Batch Language file (e.g, `fibo.bbl`)
`*.BPT`	Debugger Breakpoint file (e.g., `fibo.bpt`)
`*.C *.CPP`	C and C++ source files
`*.CHM`	Compiled HTML help file
`*.CMD`	Command File Script (e.g., `Reset.cmd`)
`*.CNF`	Specific CPU configuration file
`*.CNT`	Help Contents File (e.g., `cxa.cnt`)
`*.CPU`	Central Processor Unit Awareness file
`*.DBG`	Debug listing files (e.g., `Fibo.dbg`)
`DEFAULT.ENV`	Debugger Default Environment file
`*.DLL`	A `.DLL` file contains one or more functions compiled, linked, and stored separately from the processes that use them. The operating system maps the DLLs into the process's address space when the process is starting up or while it is running. The process then executes functions in the DLL. The DLL of the Debugger is provided for supported library and extended functions.
`*.H`	Header file
`HIWAVE.EXE`	The Debugger for Windows executable program.
`*.HWL`	Debugger Layout file (e.g., `default.hwl`)
`*.HWC`	Debugger Configuration file (e.g., `project.hwc`)
`*.EXE`	Other Windows executable program (e.g., `LINKER.EXE`)
`*.FPP`	CPU-specific Flash Programming Parameters files (e.g., `mcu0e36.fpp`)
`*.HLP`	Application Help file (e.g., `Hiwave.hlp`)
`*.IO`	I/O simulation file (e.g., `sample11.io`)

Table 24.1 Debugger Files and File Extensions (*continued*)

File Extension	Description
*.ISU	Uninstall Application File
*.PJT	Debugger configuration Settings File (e.g., Project.pjt)
*.INI	Debugger configuration Settings File (e.g., Project.ini)
*.LST	Assembler Listing File (e.g., fibo.lst)
*.MCP	Freescale CodeWarrior IDE project file
*.MAK	Make file (e.g., demo.mak)
*.MAP	Mapping file (e.g., macrodem.map)
*.MEM	Memory Configuration file (e.g., 000p4v01.mem)
*.MON	Firmware loading, file that allows loading a specified target (e.g., Firm0508.mon)
*.O	Object file code (e.g., Fibo.o)
*.PDF	Portable Document Format file
*.PRM	Linker parameter file (e.g., fibo.prm)
Project.Ini	Debugger Project Initialization File
*.REC	Recorder File
*.REG	Register Entries files (e.g., mcu081e.reg)
*.SIM	CPU Awareness file (e.g., st7.sim)
*.SX	S-Record file (e.g., fibo.sx)
*.TXT	General Text Information file.
*:TGT	Target File for the Debugger (e.g., xtend-g3.tgt)
*.WND	Debugger Window Component File (e.g., recorder.wnd)
*.XPR	Debugger User Expression file (e.g., Fibo.xpr)

Environment Variables

This section describes each of the environment variables available for the Debugger. The options are listed in alphabetical order and each is divided into several sections.

ABSPATH: Absolute Path

Tools

SmartLinker, Debugger

Synonym

None

Syntax

`ABSPATH=" {<path>}.`

Arguments

`<path>`: Paths separated by semicolons, without spaces.

Description

When you define this environment variable, the SmartLinker stores the absolute files it produces in the first directory specified. If you do not set ABSPATH, the SmartLinker stores the generated absolute files in the directory in which the parameter file is located.

Example

```
ABSPATH=\sources\bin;..\..\headers;\usr\local\bin
```

DEFAULTDIR: Default Current Directory

Tools

Compiler, Assembler, Linker, Decoder, Librarian, Maker, Burner, Debugger.

Synonym

None.

Syntax

`"DEFAULTDIR=" <directory>.`

Arguments

`<directory>`: Directory specified as default current directory.

Default

None.

Description

Use this environment variable to specify the default directory for all tools. All tools indicated above take the directory specified as their current directory instead of the one defined by the operating system or launching tool.

NOTE This is an environment variable at the system level (global environment variable). It CANNOT be specified in a default environment file (`DEFAULT.ENV/.hidefaults`).

Example

```
DEFAULTDIR=C:\INSTALL\PROJECT
```

See also

The Current Directory

Global Initialization File (MCUTOOLS.INI - PC Only)

ENVIRONMENT=: 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

You must specify this variable at the system level. Normally the application looks in the The Current Directory for an environment file named `default.env`. Use the `ENVIRONMENT` variable to specify a different file name.

NOTE This is a system level (global) environment variable. It CANNOT be specified in a default environment file (`DEFAULT.ENV/.hidefaults`).

Example

```
ENVIRONMENT=\Freescale\prog\global.env
```

GENPATH: #include "File" Path

Tools

Compiler, Linker, Decoder, Burner, Debugger.

Synonym

`HIPATH`

Syntax

```
"GENPATH=" {<path>}.
```

Arguments

<path>: Paths separated by semicolons, without spaces.

Default

Current directory

Description

If you include a header file with double quotes, the Debugger searches in the current directory, then in the directories given by GENPATH and finally in the directories given by `LIBRARYPATH` (see LIBRARYPATH: ‘include <File>’ Path).

NOTE If a directory specification in this environment variable starts with an asterisk (*), the debugger searches the whole directory tree recursively. All subdirectories and their subdirectories are searched. Within one level in the tree, search order is random.

Example

```
GENPATH=\sources\include;..\..\headers;
\usr\local\lib
```

See also

Environment variable `LIBPATH`

LIBRARYPATH: ‘include <File>’ Path

Tools

Compiler, ELF tools (Burner, Linker, Decoder)

Synonym

`LIBPATH`

Syntax

```
"LIBRARYPATH=" {<path>}.
```

Arguments

`<path>`: Paths separated by semicolons, without spaces.

Default

Current directory

Description

If you include a header file with double quotes, the Compiler searches in the current directory, then in the directories given by `GENPATH` (see GENPATH: #include “File”

Path) and finally in directories given by `LIBRARYPATH` (see LIBRARYPATH: 'include <File>' Path).

NOTE If a directory specification in the environment variable starts with an asterisk (*), the Compiler searches the whole directory tree, including subdirectories and their subdirectories, recursively. Within one level in the tree, search order is random.

Example

```
LIBRARYPATH=\sources\include;..\..\headers;\usr\local\
lib
```

See also

Environment variable GENPATH: #include "File" Path

Environment variable USELIBPATH: Using LIBPATH Environment Variable

OBJPATH: Object File Path

Tools

Compiler, Linker, Decoder, Burner, Debugger.

Synonym

None.

Syntax

```
"OBJPATH=" <path>.
```

Default

Current directory

Arguments

`<path>`: Path without spaces.

Description

If a tool looks for an object file (for example, the Linker), it first checks for an object file specified by this environment variable, then in `GENPATH` (see GENPATH: #include "File" Path) and finally in `HIPATH`.

Example

```
OBJPATH=\sources\obj
```

TMP: Temporary directory

Tools

Compiler, Assembler, Linker, Librarian, Debugger.

Synonym

None.

Syntax

```
"TMP=" <directory>
```

Arguments

`<directory>`: Directory to use for temporary files.

Default

None.

Description

If a temporary file must be created, normally you use the ANSI function `tmpnam()`. 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 that says “Cannot create temporary file”.

NOTE This is a system level (global) environment variable. It CANNOT be specified in a default environment file (`DEFAULT.ENV`/`.hidefaults`).

Example

```
TMP=C:\TEMP
```

See also

The Current Directory

USELIBPATH: Using LIBPATH Environment Variable

Tools

Compiler, Linker, Debugger.

Synonym

None.

Syntax

```
"USELIBPATH=" ("OFF" | "ON" | "NO" | "YES")
```

Arguments

`"ON"`, `"YES"`: Uses the `LIBRARYPATH` environment variable (see LIBRARYPATH: ‘include <File>’ Path) to look for system header files `<*.h>`.

`"NO"`, `"OFF"`: Does not use the `LIBRARYPATH` environment variable.

Default

ON

Description

This environment variable allows flexible use of the `LIBRARYPATH` environment variable, because `LIBRARYPATH` may be used by other software (for example, version management PVCS).

Example

```
USELIBPATH=ON
```

See also

Environment variable LIBRARYPATH: ‘include <File>’ Path

25

Connection-Specific Environment Variables

Some of the environment variables that can be used in the debugging process are imported with the connection software and are specific to that connection. This chapter lists and describes those variables.

Abatron BDI Connection Environment Variables

This section describes the environment variables used by the *Abatron BDI* Connection. The *Abatron BDI* Connection-specific environment variables are:

- BDICONF
- COMDEV
- COMPRESS
- SHOWPROT
- SKIPILLEGALBREAK
- VERIFY

These variables are stored in the `[BDIK]` section from the project file.

Listing 25.1 Example of the [BDIK] section from the project file

```
[BDIK]
CMDFILE0=CMDFILE STARTUP ON "startup.cmd"
CMDFILE1=CMDFILE RESET ON "reset.cmd"
CMDFILE2=CMDFILE PRELOAD ON "preload.cmd"
CMDFILE3=CMDFILE POSTLOAD ON "postload.cmd"
COMDEV=COM1 57600
SHOWPROT=0
BDICONF=C:\tmp\B10c12.exe
SKIPILLEGALBREAK=0
VERIFY=1
COMPRESS=1
```

The remainder of this section describes each of the variables available for the *Abatron BDI* Connection. The variables are listed in alphabetical order and are divided into several topics.

BDICONF

Description

This variable defines the communication device between the computer and the BDI. It is set according to the **BDI Box Configuration Tool Path** edit box of the **Setup Dialog Box**. You can set up the **BDI Box Configuration Tool Path** edit box with the path and application name of the configuration tool from ABATRON. The application tool is automatically browsed when selecting the **Abatron BDI > Configure BDI Box** menu entry and browsing for the application. Otherwise, click the **Browse** button to look for the tool.

Syntax

`BDICONF=`*`ConfigurationToolFileNameandPath`*

Arguments

`ConfigurationToolFileNameandPath`: the ABATRON configuration tool file name and path.

Default

No default value exists. The string "`Enter here the path to the ABATRON configuration tool.`" appears in the edit box.

Example

```
BDICONF=C:\tmp\B10c12.exe
```

COMDEV

Description

This variable defines the communication device between the computer and the BDI. It is set according to the **Communication Device** edit box of the Communication Device Specification Dialog Box.

Syntax

`COMDEV=COMn baudrate`

where `n` is the COM port number like 1, 2, 3, etc. and where `baudrate` is 9600, 19200, 38400, 57600, 115200, according to the setup done in the ABATRON configuration application.

For the communication via an Ethernet:

`COMDEV=NETWORK ip_address port`

where `ip_address` is the IP address of the BDI box or bdiNet in the form `xxx.xxx.xxx.xxx` and `port` is the bdiNet port, usually "1" for BDI1000 and BDI2000.

Default

The default value is `COM1 57600`.

Example

`COMDEV=COM1 57600`

COMPRESS

Description

This variable sets the BDI download mode with data compression. By default, data compression is enabled for asynchronous communication channels. With older computers, it is possible that download speed is faster without data compression. It is set according to the **Use Data Compression** check box of the Setup Dialog Box.

Syntax

`COMPRESS=1|0`

Default

The default value is **`1`**.

Example

`COMPRESS=1`

SHOWPROT

Description

If the **Show Protocol** is used, all the commands and responses sent and received are reported in the **Command Line** component of the debugger.

If the variable is set to 1, **Show Protocol** is activated.

This variable is set according to the **Show Protocol** check box of the Communication Device Specification Dialog Box.

Syntax

```
SHOWPROT=1|0
```

Default

The default value is **0**.

Example

```
SHOWPROT=1
```

NOTE The Show Protocol is a useful debugging feature if there is a communication problem.

SKIPILLEGALBREAK

Description

This variable is set according to the **Continue on illegal break (banked hardware breakpoint)** option check box of the Setup Dialog Box.

The **Continue on illegal break (banked hardware breakpoint)** option check box is only available for the HC12/CPU12 derivative. Check this check box to override the 2-byte address size on-chip break module which does not handle the PPAGE. Note that internally, the hardware breakpoint halts the target (in Flash memory), compared with the breakpoint that you set, then relaunched, if not using matching.

Syntax

```
SKIPPILLEGALBREAK=1|0
```

Default

The default value is **0**.

Example

```
SKIPILLEGALBREAK=1
```

VERIFY

Description

This variable sets the BDI download mode with data verification. By default, use **Verify only first** option. If necessary, you can set a different option to improve transfer speed or security. Set this variable using to the **Data Transfer Verification** radio buttons of the Setup Dialog Box.

Syntax

```
VERIFY=0|1|2|3
```

with `0` for no verification at all (fastest mode), `1` for first byte verification only, `2` for all data read back verification, and `3` for only verification (no write).

Default

The default value is **1**.

Example

```
VERIFY=1
```

Banked Memory Location-Associated Environment Variables

The following sections describe the Banked Memory Location environment variables used by the Abatron BDI connection. These variables are:

BANKWINDOWn

This variable is stored in the `["targetName"]` section from the project file.

Listing 25.2 Example of the `[BDIK]` target section from a project file

```
[BDIK]
BANKWINDOW0=BANKWINDOW PPAGE ON 0x8000..0xBFFF 0x30 64
BANKWINDOW1=BANKWINDOW DPAGE OFF 0x7000..0x7FFF 0x34 256
BANKWINDOW2=BANKWINDOW EPAGE OFF 0x400..0x7FF 0x36 256
```

The following sections describe the variable available for this connection.

BANKWINDOWn

Description

The `BANKWINDOWn` variable specifies a command file definition using `BANKWINDOW` *`Command Line`* command. Usually three or four of those entries are present in the project file, depending on the connection.

Those variables are used to store the Banked Memory Location definition (range, address, number of pages) and status (enable/disable) specified either with the `BANKWINDOW` *`Command Line`* command or the PPage tab in the Banked Memory Window.

Syntax

```
BANKWINDOWn=<one BANKWINDOW Command Line command>
```

Default

All available banked memory area are disabled by default.

The default PPAGE memory banked area is 0x8000 to 0xBFFF, 8 pages allowed, with PPAGE register at address 0x35.

The default DPAGE memory banked area is 0x7000 to 0x7FFF, 256 pages allowed, with PPAGE register at address 0x34.

The default EPAGE memory banked area is 0x400 to 0x7FF, 256 pages allowed, with PPAGE register at address 0x36.

The default settings for the *VARIOUS* page is that the bank window dialog is displayed automatically when connecting when settings are not done.

Example

```
BANKWINDOW0=BANKWINDOW PPAGE OFF 0x8000..0xBFFF 0x30 64
BANKWINDOW1=BANKWINDOW DPAGE OFF 0x7000..0x7FFF 0x34 256
BANKWINDOW2=BANKWINDOW EPAGE OFF 0x400..0x7FF 0x36 256
BANKWINDOW3=BANKWINDOW VARIOUS DLGATCONNECT
```

Unsecure Environment Variable

The following section describes the HC12 Unsecure environment variable used by the Target Interface. This variable is:

- CHIPSECURE

This variable is stored in the ["targetName"_GDI_SETTINGS] section.

Listing 25.3 Example of [ICD12] target section

```
[ICD12_GDI_SETTINGS]
CHIPSECURE=CHIPSECURE SETUP 0xFF0F 0x3 0x2
```

CHIPSECURE

Description

The `CHIPSECURE` variable specifies the HCS12 Unsecure mechanism setup using a CHIPSECURE Command Line command.

Syntax

```
CHIPSECURE=<CHIPSECURE SETUP Command Line command>
```

Example

```
CHIPSECURE=CHIPSECURE SETUP 0xFF0F 0x3 0x2
```

On-Chip Hardware Breakpoint Module Environment Variables

This section describes the Hardware Breakpoint Settings environment variables used by the Target Interface. These variables are:

- HWBPD_MCUIDnnn_BKPT_REMAPn
- HWBPMn

These variables are stored in the ["targetName"_GDI_SETTINGS] section.

Listing 25.4 Example of the [ICD12] target section from a project file

```
[ICD12_GDI_SETTINGS]
HWBPM0=HWBPM MODE AUTOMATIC BPM16BITS 0x28 SKIP_OFF
HWBPM1=HWBPM SET16BITS 0x0 0x0 0x0 0x0
HWBPM2=HWBPM SET22BITS 0x0 0x0 0x0 0x0
HWBPD_MCUID3C6_BKPT_REMAP0=HWBPM REMAP_22BITS RANGE 0x4000 0x7FFF 0x3E
HWBPD_MCUID3C6_BKPT_REMAP1=HWBPM REMAP_22BITS RANGE 0xC000 0xFFFF 0x3F
HWBPD_MCUID3C7_BKPT_REMAP0=HWBPM REMAP_22BITS RANGE 0x4000 0x7FFF 0x3E
HWBPD_MCUID3C7_BKPT_REMAP1=HWBPM REMAP_22BITS RANGE 0xC000 0xFFFF 0x3F
HWBPD_MCUID3CA_BKPT_REMAP0=HWBPM REMAP_22BITS RANGE 0x4000 0x7FFF 0x3E
HWBPD_MCUID3CA_BKPT_REMAP1=HWBPM REMAP_22BITS RANGE 0xC000 0xFFFF 0x3F
```

The following sections describe each variable available for the Target Interface. The variables are listed in alphabetical order.

HWBPD_MCUIDnnn_BKPT_REMAPn

Description

The `HWBPD_MCUIDnnn_BKPT_REMAPn` variable specifies a command file definition using HWBPM `REMAP22BITS` Command Line command.

The variable name depends on the derivative MCU-ID and on the remapping range number.

Those variables are used to store the current Hardware Breakpoints Module remapping settings specified with the HWBPM `REMAP22BITS` Command Line command.

Syntax

```
HWBPD_MCUIDnnn_BKPT_REMAPn=<one HWBPM REMAP22BITS Command
     Line command>
```

Default

Defaults settings are retrieved according to the derivative from a common `ini` file.

Example

```
HWBPD_MCUID3C6_BKPT_REMAP0=
HWBPM REMAP_22BITS RANGE 0x4000 0x7FFF 0x3E
```

HWBPMn

Description

The HWBPMn variable specifies the configuration of the Hardware Breakpoints module using HWBPM Command Line command. Three entries appear in the project file.

Those variables are used to store the current Hardware Breakpoints Module settings specified either with the HWBPM Command Line command or through the Hardware Breakpoint Configuration dialog.

Syntax

```
HWBPMn=<one HWBPM Command Line command>
```

Default

Defaults settings are retrieved according to the derivative from a common `.ini` file.

Example

```
HWBPM0=HWBPM MODE AUTOMATIC BPM16BITS 0x28 SKIP_OFF
HWBPM1=HWBPM SET16BITS 0x0 0x0 0x0 0x0
HWBPM2=HWBPM SET22BITS 0x0 0x0 0x0 0x0
```

Book V - Debugger Legacy

Book V Contents

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

Book V: HC(S)12(X) Debugger Legacy

This book is divided into the following chapters

26

HC(S)12 (X) Full-Chip Simulator Components No Longer Supported

List of HC(S)12(X) FCS Components No Longer Supported

The following legacy components are no longer supported and excluded from the product:

- MicroC
- Softtrace
- Segments_display
- Wagon
- Adc_dac
- Push_buttons
- Monitor
- IT_keyboard
- Lcd
- IO_ports
- Phone
- Template
- IO_led
- Led
- WinLift

27

Debugger DDE Capabilities

The DDE is a form of interprocess communication that uses shared memory to exchange data between applications. Applications can use DDE for one-time data transfers and for ongoing exchanges in applications that send updates to one another as new data becomes available.

NOTE **The DDE capabilities of the Debugger are deprecated.** Future versions of the Debugger will have no DDE capabilities. Use the Component Object Model (COM) Interface (Debugger COM Capabilities) instead .

DDE Implementation

The Debugger integrates a DDE server and DDE client implementation in the KERNEL. The DDE application name of the IDF server is **HI-WAVE**.

The Debugger DDE support allows you to execute almost any command available from within the debugger (from Command line). There are also special DDE items for more commonly performed tasks.

This section describes topics and DDE items available to CodeWright clients. In addition to the required System topic, CurrentBuffer and the names of all CodeWright non-system buffers (documents) are available as topics.

Driving Debugger through DDE

The DDE implementation in the Debugger allows you to drive it easily by using the DDE command. To do this, use a program that can send a DDE message (a DDE client application) like `DDECLient.exe` from Microsoft®.

The service name of the Debugger DDE Server is **HI-WAVE** and the Topic name for the Debugger DDE Server is **Command**.

The following example is done with `DDECLient.exe` from Microsoft.

1. Run the Debugger and in the **Service** field in the DDEClient type: HI-WAVE
2. In the **Topic** field type **Command**
3. Push the **Connect** button of the DDEClient. The following message appears in DDECLient: **Connected to HI-WAVE|Command**.

4. In the **Exec** field of DDEClient type a Debugger command, for example, **open recorder**, and click the **Exec** button. The command executes by way of DDE and you'll see a new recorder component in the Debugger.

NOTE You can disconnect the DDE in the Debugger. You can start the Debugger 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.

Index

B

D

G

H

I

N

O

P

S

T