Intel® IXP2400/IXP2800 Network Processor

Development Tools User's Guide

July 2003

Order Number: 278733-007

Revision History

Revision Date	Revision	Description
07/03	007	SDK 3.1 Pre-Release 3.
06/03	006	SDK 3.1 Pre-Release 2.
01/03	005	Fifth release of documentation IXP2400/IXP2800 for IXA SDK 3.0 Pre-Release 6.
10/02	004	Fourth release of documentation for IXP2400/IXP2800 for IXA SDK 3.0 Pre-Release 5
08/02	003	Third release of documentation for iXP2400/IXP2800 for IXA SDK 3.0 Pre-Release 4
05/02	002	Second release of documentation for IXP2400/IXP2800 for IXA SDK 3.0
1/25/02	001	First IXP2400/IXP2800 only release for Pre-Release II

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

The IXP2400/IXP2800 Network Processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

This document and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's Web site at http://www.intel.com.

Copyright © Intel Corporation, 2003

Intel is a registered trademark and XScale is a trademark of Intel Corporation or its subsidiaries in the United States and other countries

*Other names and brands may be claimed as the property of others.

Contents

Figures

Tables

Introduction 1

1.1 About this Document

This manual is a reference for network processor development tools and is organized as follows:

- Chapter 2, "Developer Workbench" - Describes the Workbench and its graphical user interface (GUI).
- Chapter 3, "Assembler" - Describes how to run the Assembler.
- Chapter 4, "Microengine C Compiler" - Describes how to run the Microengine C Compiler.
- Chapter 5, "Linker" - Describes how to run the Linker.
- Chapter 6, "Foreign Model Simulation Extensions" - Provides information on interfacing the Network Processor to foreign models.
- Chapter 7, "Transactor" - Describes the Transactor and its commands.
- Chapter 8, "Simulator APIs" - Desribes the Simulation APIs.
- Appendix A, "Transactor States" - Describes the Transactor internal states.
- Appendix B, "Developer Workbench Shortcuts" - Contains a listing and description of commonly used shortcuts.
- Appendix C, "XScale Core Memory Bus Functional Model" - Describes the XScale Bus Functional Model.
- Appendix D, "PCI Bus Functional Model" - Describes the PCI Bus Functional Model.
- Appendix E, "SPI4 Bus Functional Model" - Describes the SPI4 Bus Functional Model.

1.2 Intended Audience

The intended audience for this book is Developers and Systems Programmers.

1.3 Related Documents

Further information on the network processors is available in the following documents:

Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual—Contains detailed programming information for designers.

Intel® IXP2850 Network Processor Programmer's Reference Manual—Contains detailed programming information for designers.

Intel® IXP2800 Network Processor Datasheet—Contains summary information on the IXP2800 Network Processor including a functional description, signal descriptions, electrical specifications, and mechanical specifications.

Intel® IXP2850 Network Processor Datasheet—Contains summary information on the IXP2800 Network Processor including a functional description, signal descriptions, electrical specifications, and mechanical specifications.

Intel® IXP2400 Network Processor Datasheet—Contains summary information on the 2400 Network Processor including a functional description, signal descriptions, electrical specifications, and mechanical specifications.

Intel® Intel® *IXP2400/IXP2800 Microengine C Compiler LIBC Library Reference Manual* - Contains a modified subset of standard C Library functions supported on the IXP2400 and IXP2800 Network Processors

Intel® IXP2400 Network Processor Hardware Reference Manual -Contains detailed hardware technical information about the IXP2400 Network Processor for designers.

Intel® IXP2800 Network Processor Hardware Reference Manual - Contains detailed hardware technical information about the IXP2800 Network Processor for designers.

Intel® IXP2850 Network Processor Hardware Reference Manual - Contains detailed hardware technical information about the IXP2850 Network Processor for designers.

Developer Workbench 2

2.1 Overview

The Developer Workbench is an integrated development environment for assembling, compiling, linking, and debugging microcode that runs on the IXP2400, IXP2800, and IXP2850 Network Processor Microengines. The Workbench is a Microsoft* Win32* application that runs on Windows 2000* platforms.

Features:

Important Workbench features include:

- Source level debugging.
- Debug-only project creation mode.
- Execution history.
- Statistics.
- Media Bus device and network traffic simulation for the Network Processors
- Command line interface to the Network Processor simulators (Transactors).
- Customizable graphical user interface (GUI) components.

Debugging Support:

The Workbench supports debugging in four different configurations:

- **Local simulation with no foreign model**, in which the Workbench and the Network Processor simulator (Transactor) both run on the same Microsoft Windows* platform.
- **Local simulation with local foreign models**, in which the Workbench, the Transactor, and one or more foreign model Dynamic-Link Libraries all run on the same Windows platform.
- **Local simulation with a remote foreign model**, in which the Workbench and the Transactor both run on the same Windows platform and communicate over the network with a foreign model running on a remote system.
- **Hardware**, in which the Workbench runs on a Windows host and communicates over a network or a serial port with a subsystem containing actual Network Processors. (Not currently available for IXP2800 Network Processors).

Getting Help:

You can get help about the Developer Workbench and the Network Processors in several ways:

- On the **Help** menu, click **Help Topics**. This opens the Developer Workbench online help tool.
- In the **Project Workspace** window (see Figure 1), click the **InfoView** tab. This give you access to documentation installed along with the Software Development Kit (SDK). See Section 2.4.3, “About InfoView.”
- On the Web, go to http://developer.intel.com/design/network/products/npfamily/index.htm to get more information about Intel products.

Developer Workbench Revision Information:

To determine the revision:

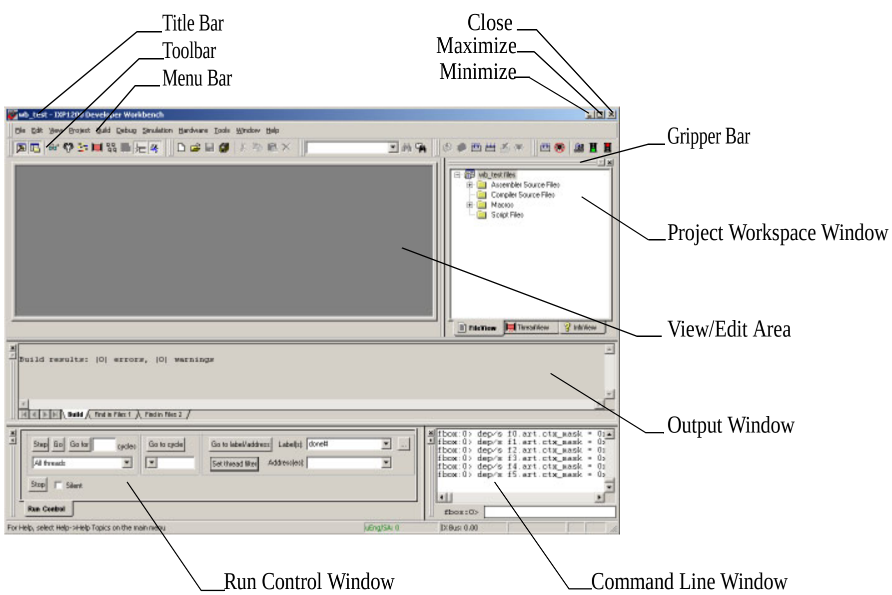
- On the **Help** menu, click **About Developer Workbench**.
  The **About Developer Workbench** information box appears displaying the revision of your Developer Workbench.

2.2 About the Graphical User Interface (GUI)

The Workbench GUI (Figure 1) conforms to the standard Windows look and feel. You can do the following:

- **Dock and undock** (float) windows, menu bars, and toolbars (see Section 2.2.1).
- **Hide and show** windows and toolbars (see Section 2.2.2).
- **Customize** toolbars and menu bars (see Section 2.2.3).
- **Save and restore** GUI customizations (see Section 2.2.3.6).

Figure 1. The Developer Workbench GUI

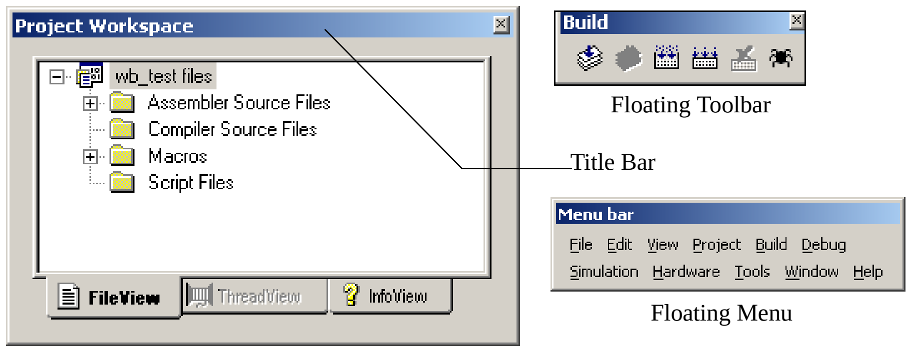


2.2.1 About Windows, Toolbars, and Menus

Dockable windows contain controls and data. You can attach them to a location on the Workbench main window or you can float them over the main window. All toolbars and menu bars are dockable. (See Figure 2.)

To float, or undock, a window or toolbar, double-click its gripper bar (see Figure 1). To restore it to its previously docked location, double-click its title bar. You can also drag the window to a new docking location.

Figure 2. Floating Window, Tool Bar, and Menu Bar



2.2.2 Hiding and Showing Windows and Toolbars

Form the **View** menu, you can toggle the visibility of the following windows in the Workbench's GUI:

Toolbar — If you are viewing a source file, or the edit/view area is empty, selecting **Toolbar** on the **View** menu displays the **Toolbars** dialog box. Here you can select to view or clear to hide any of the available toolbars. You can also select **Show Tooltips**, **Large Buttons**, and **Cool Look**.

Workbook Mode — This control puts the tabs at the bottom of the view/edit area (see Figure 1). Without the tabs you must use other methods to select different windows, such as going to the Window menu and selecting the window; cascading the windows using the button and selecting with the mouse pointer; pressing CTRL+F6 to switch from one window to the next. Removing the tabs gives you more workspace in the windows.

Project Workspace — See Section 2.4.

Output Window — This window displays the results of Find in Files, assembly and compile results, build results and other messages. See Figure 1.

Click the button to show or hide this window.

Debug Windows — Command Line - see Section 2.12.6.

Data Watch - see Section 2.12.12.1.

Memory Watch - see Section 2.12.13.2.

Packet Simulation - see Section 2.10

History - see Section 2.12.17.6.

Thread Status - see Section 2.12.18.

Queue Status - see Section 2.12.17.

Run Control - see Section 2.12.9.

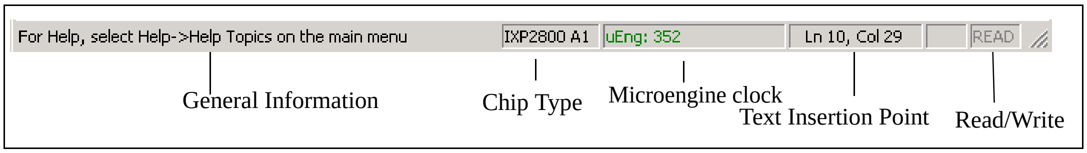
To toggle the visibility of a dockable window, select or clear the window's name on the **View** menu.

If a window is visible, you can hide it by clicking the ☒ button in either the upper-right or upper-left corner of the window.

If a toolbar is floating, you can hide it by clicking the ☒ button in the upper right corner.

Note: You can float and dock the GUI's default menu bar but you cannot hide it. If you create a customized menu bar, you can display or hide in it using the same method used for windows and toolbars.

Status Bar The status bar appears at the bottom on the Workbench GUI.



General Information	Information and tips appear here as you work.
Chip Type	Identifies the network processor and revision (stepping).
Microengine Clock	The present cycle count of the Microengine clock (simulation debug mode only). In hardware debug mode, it shows **stopped** or **running** to indicate microengine state.
Text Insertion Point	The location of the text insertion point (cursor) by line and column.
Read-only/Write	The Read/Write status of the selected file. If READ is dimmed, the status is Read/Write.

2.2.3 Customizing Toolbars and Menus

You can add and remove buttons from toolbars and create your own toolbars.

2.2.3.1 Creating Toolbars

To create a toolbar:

1. On the **Tools** menu, click **Customize**.
   The **Customize** dialog box appears.
2. Click the **Toolbars** tab.
3. Click **New**.
   The **New Toolbar** dialog box appears.
4. Type a name for the new toolbar and click **OK**.

The toolbar name is added to the **Toolbars** list and the new toolbar appears in a floating state. If you want the toolbar to be docked, drag it to the desired location.

To populate the toolbar with buttons, go to Section 2.2.3.4.

2.2.3.2 Renaming Toolbars

You can rename toolbars that you have created.

To rename a toolbar:

1. On the **Tools** menu, click **Customize**.
   The **Customize** dialog box appears.
2. Click the **Toolbars** tab.
3. Select the desired toolbar in the **Toolbars** list.
4. Edit the name in the **Toolbar Name** box at the bottom.
5. Click **OK**.

Note: You cannot rename the GUI's default toolbars (Menu bar, File, Debug, Build, Edit, View).

2.2.3.3 Deleting Toolbars

To delete a toolbar you have created:

1. On the **Tools** menu, click **Customize**.
   The **Customize** dialog box appears.
2. Click the **Toolbars** tab.
3. Select the toolbar to delete in the **Toolbars** list.
4. Click **Delete**.

Note: You cannot delete the GUI's default toolbars (Menu bar, File, Debug, Build, Edit, View).

2.2.3.4 Adding and Removing Toolbar Buttons and Controls

To customize the buttons on the toolbars:

1. On the **Tools** menu, click **Customize**.
   The **Customize** dialog box appears.
2. Click the **Commands** tab.
3. From the **Categories** list, select a command category.
   A set of toolbar buttons for that category appears in the **Buttons** area.
   To get a description of the command associated with a button, click the button. The description appears in the **Description** area at the bottom of the dialog box.
4. To place a button in a toolbar, drag the button to a location on a toolbar.
5. To remove a button from a toolbar, drag the button into the dialog box.
6. Click **OK** when done.

2.2.3.5 Customizing Menus

You can change the appearance of the main menu or you can put menus on toolbars.

Main Menu Appearance:

To change the order of the main menu items:

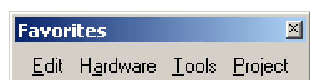
1. On the **Tools** menu, click **Customize**.
   The **Customize** dialog box appears.
2. Drag any main menu item to the new position on the main menu bar. For example, drag File and drop it after Help.
3. To remove a menu from the main menu bar, drag it into the work area below.
4. To add a menu to the main menu bar:
   a. In the **Customize** dialog box, click the **Commands** tab.
   b. Click **Menu** in the **Commands** box.
      All the menus appear in the **Buttons** box.
   c. Select a menu and drag it to the main menu bar.
      That menu then becomes a new menu on the main menu bar.

Menus on Toolbars:

To put a menu on a toolbar:

1. In the **Customize** dialog box, click the **Commands** tab.
2. Click **Menu** in the **Categories** box.
   All the menus appear in the **Buttons** box.
3. Drag any menu to any toolbar.

Note: You can put your most used or favorite menus on a floating toolbar by creating a new toolbar (see example at right) and dragging the menus to that toolbar.



2.2.3.6 Returning to Default Toolbar Settings

To set toolbars to their default configurations:

1. On the **Tools** menu, click **Customize**.
   The **Customize** dialog box appears.
2. Click the **Toolbars** tab.
3. Select the desired toolbar and click **Reset**.

Only the Workbench default toolbars can be reset.

2.2.4 GUI Toolbar Configurations

Build Versus Debug:

The Workbench maintains two sets of toolbar and docking configurations, one for debug mode and one for build, or non-debug mode. The GUI configuration that you establish while in build mode applies only when you are in build mode. Similarly, the debug mode GUI configuration applies only for debug mode.

Save and Restore:

Menu bar and toolbar configurations are saved when you exit the Workbench. These configurations persist from one Workbench session to the next.

2.3 Workbench Projects

Projects may be created in two ways: **standard** and **debug-only**.

A **standard** project consists of one or more network processor(s), microcode source files, debug script files, and Assembler, Compiler, and Linker settings used to build the microcode image files. This project configuration information is maintained in a Developer Workbench project file (.dwp).

A debug-only project is one in which the user specifies an externally built uof file for each chip in the project. If a project is created as "debug-only" the user does ***not*** specify assembler and compiler source files, manage build settings, or perform uof file builds using the Workbench GUI.

When you start the workbench you can:

- Create a new project (see Section 2.3.1).
- Open an existing project (see Section 2.3.2).
- Save a project (see Section 2.3.3).
- Close a project (see Section 2.3.4).
- Specify a default folder for creating and opening projects (see Section 2.3.5).

2.3.1 Creating a New Project

The processor type, that you select when you create a new project, determines which Transactor is used for simulation. The Workbench will display only the GUI components that are relevant to the selected processor type. The processor family cannot be changed once a project is created; i.e. you cannot change your project from an IXP2400 processor to an IXP2800 processor, or vice versa.

The processor types supported by the workbench are:

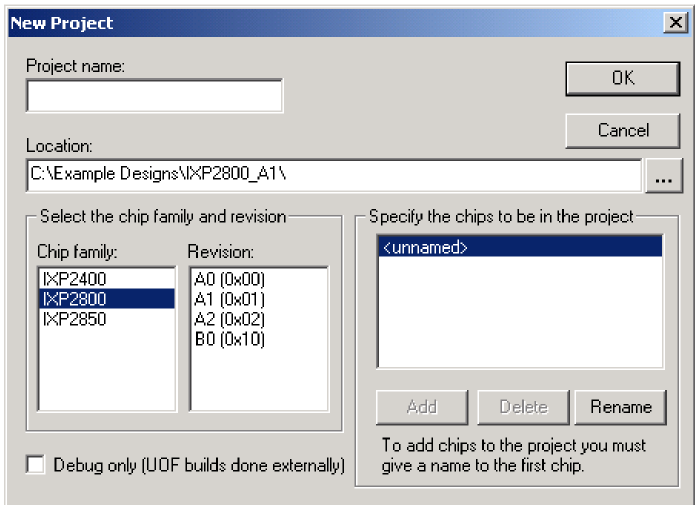
- IXP2800 A0
- IXP2800 A1
- IXP2800 A2
- IXP2800 B0
- IXP2850 A0

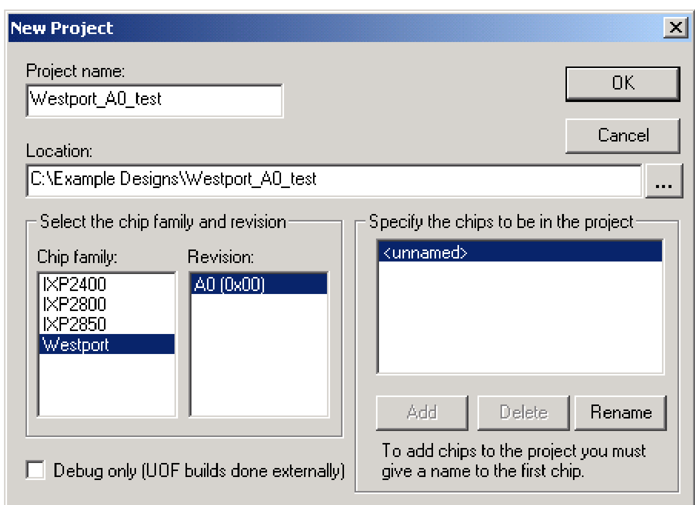
- IXP2850 A1
- IXP2850 A2
- IXP2850 B0
- IXP2400 A1
- IXP2400 B0

To create a new project:

1. On the **File** menu, click **New Project**.

   The **New Project** dialog box appears.





2. Type the name of the new project in the **Project name** box.

3. Specify a folder where you want to store the project in the **Location** box.

   If the folder doesn't exist, the Workbench creates it. You can browse to select the folder by clicking the ... button.

4. Select the chip family in the **Chip Family** box.
5. Select the revision number (stepping) for the chip in the **Revision** box.
6. Specify the number of chips to be in the project in the **Specify the chips to be in the project** box. Do the following:

   — If you have only one chip in your project, it can be <unnamed>.

   — To specify more than one chip, they must all have unique names. You cannot add a second chip until you have named the first chip. When you finish creating a project, you cannot change the number of chips in it.

     — To rename a chip, select the chip in the list and click **Rename**. The **Chip Name** dialog box appears. Type the chip's name and click **OK**.

     — To add a chip to the configuration, click **Add**. The **Chip Name** dialog box appears. Type the chip's name and click **OK**.

     — To delete a chip from the configuration, select the chip in the list and click **Delete**.

7. If this project is to be "debug-only", click the **Debug only** check box.

   This specification tells the Workbench that the source files and list files to be used in this project will be built externally. Since the uof file contains the absolute paths of the source and list files, you must make sure to specify the correct locations for those files. If the Workbench cannot find the proper files, debugging will not work as expected.

Caution: Once a project is created as **Debug only**, it cannot be converted to a standard, Workbench buildable project. Neither can an existing standard project be converted to **Debug only**.

8. When you are finished, click **OK** to create the project.

The project name you typed, by default, becomes a folder containing two files—*project_name*.dwp and *project_name*.dwo (optionally you can specify any name for this folder). From this point on, all the project files and information defaults to this folder or one of its subfolders. For example, a project named CrossBar has a project file named Crossbar.dwp.

Note: Creating a new project automatically closes the active project, if one is open, and asks you if you want to save any changes if there are any.

2.3.1.1 Debug-only Projects

If you select the **Debug-only** option when you create the project, there are some Workbench features that will be unavailable when you open the project.

- There are no source files available since the Workbench does not do the builds, and there is no way to add source files to a **Debug-only** project. The conventional options on the **Project** menu are replaced with the option **Specify Debug-only UOF files**.
- The Project Workspace **Fileview** tab does not display the tree elements **Assembler Source Files, Compiler Source Files,** or **Macros** since the Workbench does not associate source files with the **Debug-only** project.

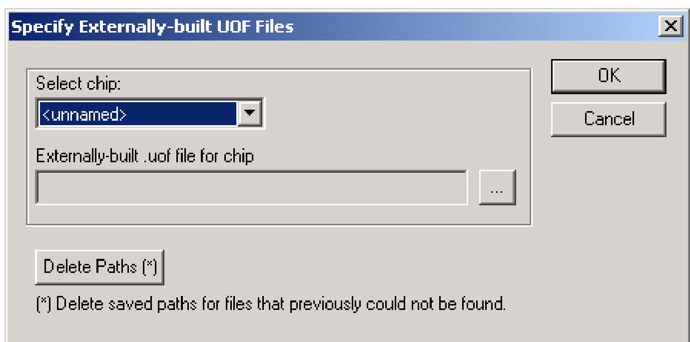
Select **Specify Debug-only UOF Files** from the **Project** menu. When you select the Debug-only option, the dialog box shown in Figure 3 is used to specify the uof file for each chip in the project.

If you try to start debugging without specifying a uof file, or if the uof or any list file identified in the uof file is not readable, errors will occur and debugging will not take place. If a list file cannot be found in the location specified in the uof file, the user is prompted to browse to the correct location for the list file. This can occur if the list file has been moved from where it was when the uof file was created or if the build was done on a different system from the one where the Workbench is being run.

Similarly, if the user executes a **Go To Source** command but the source file cannot be found in the location specified in the uof file, the user is prompted to browse to the correct location for the list file.

The user also has the option to delete all file paths that were saved by the Workbench, as previously described. This may be required if the user moves the list and source files to different locations. To delete saved file paths, click the **Delete Paths** button.

Figure 3. Specify Debug-only UOF Files Dialog Box



2.3.2 Opening a Project

To open an existing standard project:

1. On the **File** menu, click **Open Project**.
   The **Open Project** dialog box appears.
2. Browse to the folder that contains the project file (*.dwp) for the project you want to open.
3. Double-click the project filename or select the project filename and click **Open**.
   Once open, the processor type is displayed in the status bar, as shown below:

You can also select a project from the most recently used list of projects, if it is one of the most recent four projects opened.

1. On the **File** menu, click **Recent Projects**.
2. Click the project file from the list.

Note: Opening a project automatically closes the currently open project, if any, after asking you if you want to save changes if there are any.

2.3.3 Saving a Project

To save a modified project:

- On the **File** menu, click **Save Project**.

This saves all project configuration information, such as debug settings to the project file. If your project hasn't been modified, the **Save Project** selection is unavailable. Also, on the **File** menu, click **Save All** to save all files and the current project.

The project is saved in the folder that you specified when you created it. If you opened an existing project, it is saved in the folder from which that you opened it.

Note: You do not have the option of saving the project in a different folder.

2.3.4 Closing a Project

To close a project:

- On the **File** menu, click **Close Project**.

  If there are any modified but unsaved files in the opened project, you are asked if you want to save these changes.

  — Click **Yes** to save the file and close it, or

  — Click **No** to close it without saving any changes, or

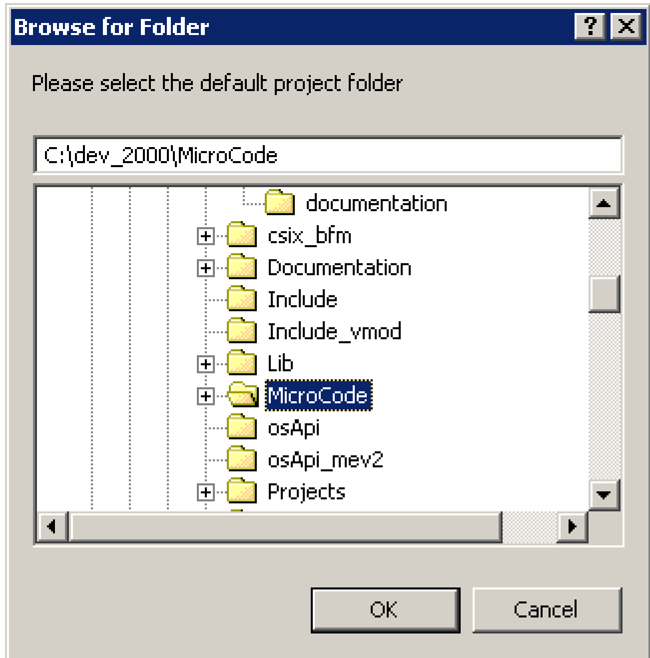
  — Click **Cancel** to abort closing the project.

An open project is automatically closed whenever you open another project or create a new project.

2.3.5 Specifying a Default Project Folder

You can specify a default folder for creating new projects and opening projects. When you select **Default Project Folder** from the **File** menu, the **Browse for folder** dialog box appears. The default project folder is used as the initial folder in the following cases:

- If the user selects **New Project** from the **File** menu.
- If the user selects **Open Project** from the **File** menu.
- If no project is open and the user creates a new file then selects **Save As** from the **File** menu.
- If no project is open and the user selects **Open** from the **File** menu.



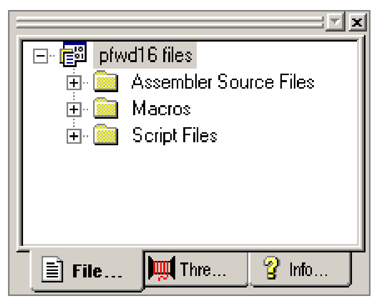
2.4 About the Project Workspace

The project workspace is a dockable window where you access and modify project files. It consists of three tabbed windows:

FileView

ThreadView

InfoView



To select a window, click its tab.

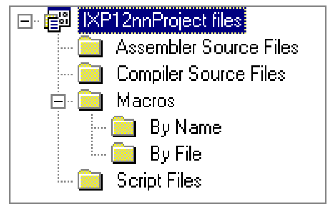
- When you start the Workbench, only **InfoView** is visible.
- When a project is open, **FileView** and **ThreadView** become visible, but access to **ThreadView** is unavailable.
- When you start debugging, access to **ThreadView** is enabled.
- When you stop debugging, access to **ThreadView** is disabled.

To toggle the visibility of the **Project Workspace**:

- On the **View** menu, select or clear **Project Workspace**, or
Click the  button on the **View** toolbar.

2.4.1 About FileView

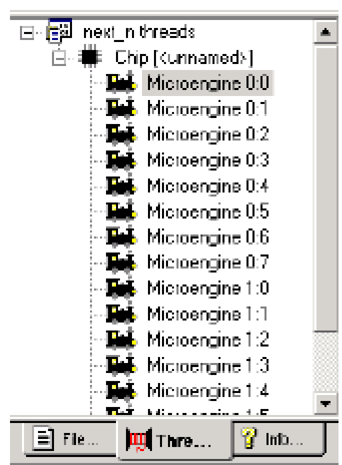
FileView contains a tree listing your project files. The top-level item in the tree is labeled <project-name> files. There are four second-level folders:



- **Assembler Source Files**, which expands to an alphabetical list of all project Assembler source files.
- **Compiler Source Files**, which expands to an alphabetical list of all project Compiler source files.
- **Macros**, which expands to list the macros that are defined in the project's source files. This folder expands to:
  — **Macros by name**, and
  — **Macros by file**.
- **Script Files**, which expands to an alphabetical list of all debugging script files.

2.4.2 About ThreadView

ThreadView contains a tree listing all Microengines that are loaded with microcode. **ThreadView** provides access to all enabled threads for each chip and is only available while debugging.



The top-level item in the tree is labeled <project-name> threads. There is a second-level item for each chip in the project. Each chip item expands to list the Microengines in the chip. Microengines are implemented in two clusters, 0 and 1, with a maximum of 16 Microengines in each cluster. For the IXP2800, there are eight Microengines per cluster, with addresses 0 - 7 and 16 - 23. For the IXP2400, there are four Microengines per cluster, with addresses 0 - 3 and 16 - 19. The Workbench displays each Microengine name as **Microengine c:n** where **c** represents the cluster number (0 or 1) and **n** is the number within the cluster.

Each Microengine item expands to list the four or eight threads in a Microengine, but only if the threads are active in the microcode. If a Microengine is not loaded with code, no "+" sign appears to the left of the icon and therefore cannot be expanded to show the threads.

By default, a chip's threads are named **Thread 0** through **Thread *n***.

The last thread by default varies depending on which network processor you choose:

- IXP2800 Network Processor - Thread 127
- IXP2400 Network Processor - Thread 63

2.4.2.1 Expanding and Collapsing Thread Trees

You can expand the entire tree for a chip as follows:

1. Right-click the chip name.
2. Click **Expand All** from the shortcut menu.

Note that in the tree to the right, Microengines 0:2, 0:3, and 0:4 cannot be expanded because they contain no microcode.

To collapse a chip's tree, double-click the chip name.

2.4.2.2 Renaming a Thread

You can rename a thread (to indicate its function or for any other reason). To do this:

1. Right-click the thread name in **ThreadView**.
2. Click **Rename Thread** from the shortcut menu.
   The **Rename Thread** dialog box appears.
3. Type the new name for the thread.
4. Click **OK**.

2.4.3 About InfoView

InfoView provides access to documentation as part of the Software Developer's Kit (SDK).

To view a document, double-click its name or icon. This invokes Adobe Acrobat Reader*, which then displays the document. A copy of Acrobat Reader is provided on the distribution CD-ROM.

2.5 Working with Files

The Workbench allows you to:

- Create files (see Section 2.5.1).
- Open files (see Section 2.5.2).
- Close files (see Section 2.5.3).
- Save files (see Section 2.5.4).
- Save copies of files (see Section 2.5.5).
- Save all files at once (see Section 2.5.6).
- Print files (see Section 2.5.8).
- Insert files into a project (see Section 2.5.9).
- Remove files from a project (see Section 2.5.9).
- Edit a file (see Section 2.5.10).
- Bookmarks, error/tags (see Section 2.5.11).

See also:

- Working with File Windows (see Section 2.5.7).
- About Find in Files (see Section 2.5.12).
- About Fonts and Syntax colors (see Section 2.5.13).
- About Macros (see Section 2.5.14).

2.5.1 Creating New Files

To create a new file:

5. On the **File** menu, click **New**, or

   Click the  button on the **File** toolbar.
   The **New** dialog box appears.
6. Select which type of file you want to create from the list.
7. Click **OK**.

This creates a new document window. The name of the window in the title bar reflects the type of file you have created.

2.5.2 Opening Files

To open an file for viewing or editing, do one of the following:

- On the **File** menu, click **Open**, and select a file from the **Open** dialog box, or

  Click the  button on the **File** toolbar, or
  If the file is in your project, double-click the file name in **FileView**.

In the open dialog box you can filter your choices using the **Files of type:** list to select a file extension. This limits your choices to only files with that extension. If you select **All files (*.*)**, your choices are unlimited. You can select any unformatted text file to view or edit.

You can open any of the last four files that you have opened. To do this:

1. On the **File** menu, click **Recent Files**.
2. Select from the list of files that appears to the right.

2.5.3 Closing Files

To close an open file:

- On the **File** menu, click **Close**, or

  On the **Window** menu, click **Close** to close the active file and its document window, or

  On the **Windows** menu, click **Close All** to close all open files and their document windows.

Note: All files that have been edited but not saved are automatically saved when you perform any operation which uses file data, such as assembling, building, updating dependencies, and finding in files.

2.5.4 Saving Files

To save a file:

1. On the **File** menu, click **Save**, or

   Click the button on the **File** toolbar.

   If you have just created the new file, the **Save As** dialog box appears. If you are saving an existing file, the **Save** dialog box appears.
2. Type the name of the new file.
3. Click **OK**.

This saves your work to a file when you are finished editing. It also displays the new file name in the title bar of the window. By convention, microcode source files have the file type .uc, C Compiler source files have the file type .c, and script files have the file type .ind.

If you are saving an existing file, you do not need to type a new name.

To save a file under a new name:

1. On the **File** menu, click **Save As**.

   The **Save As** dialog appears. The current name of the file appears in the **File Name** box.
2. Type a new name in the **File Name** box and click **Save**.

Note that the old file remains in the folder but will not have edits that you have made. The new name appears in the title bar.

2.5.5 Saving Copies of Files

You can save a copy of a file that you are viewing or editing.

To do this:

1. On the **File** menu, click **Save As**.

   The **Save As** dialog box appears.
2. Browse to the folder where you want to save the file.
3. Type the new name of the file in the **File name** box.
4. Click **Save**.

The **Save as type** list is used only if you don’t include the extension in the **File name** box. If you select **All files (*.*)**, you must include the extension in the name.

2.5.6 Saving All Files at Once

You can save all modified files in your project at once.

To do this:

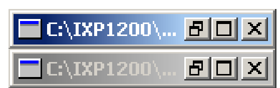
- On the **File** menu, click **Save All**, or

  Click the  button on the **File** toolbar.

2.5.7 Working With File Windows

When you select a file (text, Assembler source, Compiler source, source header, or script) for viewing or editing, it appears in a file window in the upper-left part of the Workbench. The **Windows** menu deals mostly with the text file windows in the Workbench.

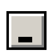
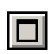
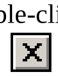
New Window	Creates a new window containing a copy of the file in the active window. The **Title Bar** displays filename.ext:2.
Close	Closes the active window.
Close All	Closes all the open windows.
Cascade	Cascades all windows that are not minimized in the viewing area.
Tile	Tiles all windows that are not minimized in the viewing area.
Arrange Icons	Tiles the window icons (if minimized) at the bottom of the viewing area.



Open Windows Selection:

At the bottom of the **Windows** menu is a list of the first nine windows that you opened. Click any one of these windows to make it the active window. If you opened more than nine windows, click **More Windows**. From the **Select Window** dialog box, click the window that you want to make active and then click **OK**.

Other Window Controls:

Minimize	Click the  button on the window that you want to minimize.
Maximize	Click the  button on the window that you want to maximize. You can also double-click the title bar to do this.
Close	Click the  button on the window that you want to close.
Restore	Click the  button on the minimized window that you want to restore to its previous view.

2.5.8 Printing Files

2.5.8.1 Setting Up the Printer

1. On the **File** menu, click **Printer Setup**.
   The **Print Setup** dialog box appears.
2. Select the printer properties for your printer. They will vary depending on the printer you select in the **Name** box.
3. Click **OK** when done.

Setting the printer properties does not print the file. To do this see Section 2.5.8.2.

2.5.8.2 Printing the File

You can print text files to a hardcopy printer or to a file.

To do this:

1. Make sure that the file you want to print is in the active window.
2. On the **File** menu, click **Print**, or

   Click the  button on the **File** toolbar. (This button is not on the default **File** menu. To put this button there, see Section 2.2.3.4.)
   The **Print** dialog box appears.
3. Select the printer (or printer driver) from the **Name** list.
4. Click **Properties** to customize your particular printer. Each printer has its own printer settings.
5. If you want to print to a file (*.prn), select **Print to file** and select a folder and file name after you click **Print**.
6. Select the pages you want to print in the **Print range** area.
7. Select the number of copies in the **Copies** area.
8. Click **Print**.

2.5.9 Inserting Into and Removing Files from a Project

2.5.9.1 Inserting Files Into a Project

You can insert Assembler source files, Compiler source files, and script files into a project.

To do this:

1. On the **Project** menu, click **Insert Assembler Source File**, or
   Insert Compiler Source File, or
   Insert Script Files, whichever is appropriate.
   The corresponding dialog box appears.

2. Browse to the desired folder and select one or more files to be inserted.
3. Click **Insert**.

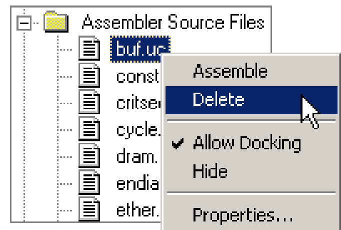
The newly inserted files are added to the list of files displayed in FileView in the corresponding folder.

2.5.9.2 Removing Files From a Project

To remove a file from your project:

1. In the **Project Workspace**, click the **File View** tab.
2. Right-click the file that you want to delete.
3. Click **Delete** on the shortcut menu, or

   Select the file and then press the DELETE key.



Note: The file is removed from the project but it is not deleted from the disk.

2.5.10 Editing Files

The Workbench editor is similar to standard text editors. See Table 5 on page 256 for a list of Edit controls.

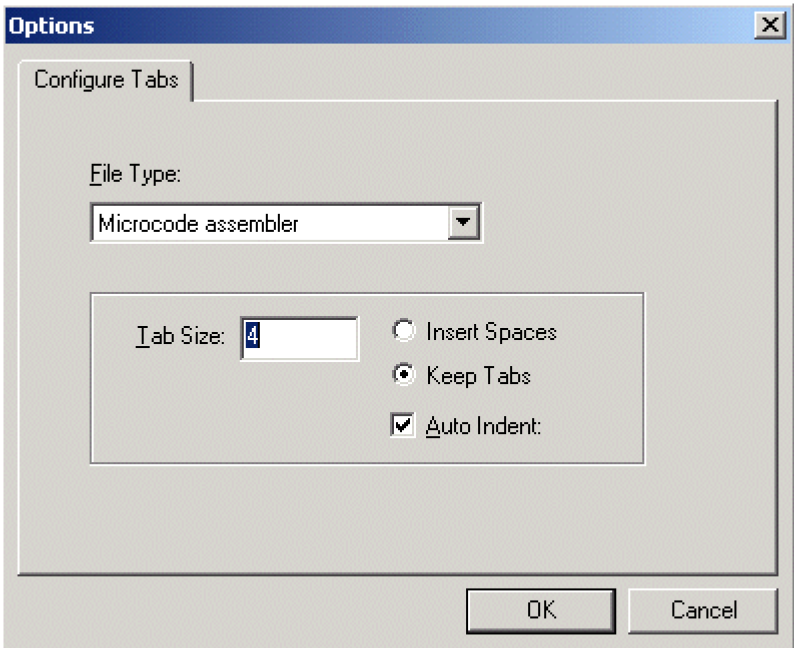
If a file has been modified, an asterisk appears after its name in the Workbench title bar.

2.5.10.1 Tab Configuration

To configure tab settings, select **Options** from the **Tools** menu. The dialog box shown in Figure 4 appears. The user selects the file type – Microcode assembler, Microengine C or Default – for which the tab settings will have effect. For the selected file type, the user specifies:

- The tab size, which determines the number of space characters that equal one tab character.
- Whether or not the editor converts tab characters to spaces.
- Whether or not to automatically indent a new line to the same column as the first non-whitespace character in the previous line

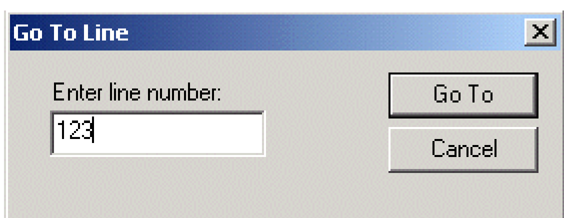
Figure 4. Configure Tabs Dialog Box



2.5.10.2 Go To Line

The Workbench allows for navigating directly to a specified line within an opened document or thread window. If the user selects **Go To** from the **Edit** menu, the dialog shown in Figure 5 appears. The user enters the desired line number and clicks **Go To**. The insertion cursor in the document or thread window that currently has focus is moved to the beginning of the specified line and the window is scrolled so that the specified line is visible.

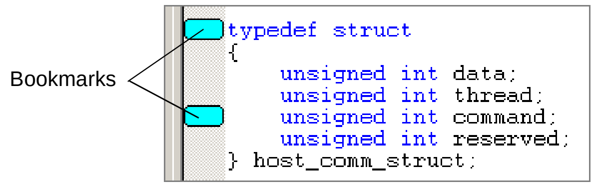
Figure 5. Go To Line Dialog Box



2.5.11 Bookmarks and Errors/Tags

You can mark your place in a file using bookmarks. Table 6 lists the tools to manipulate bookmarks in your files.

You can find errors in your files using the Error/Tag tools listed in the tables below.

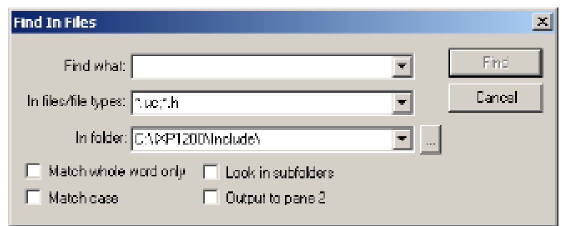


See Table 9 for a list of Bookmark and Error/Tag controls.

2.5.12 About Find In Files

The Workbench supports the ability to search multiple files for the occurrence of a specified text string. To perform this search:

1. On the **Edit** menu, click **Find In Files**, or
   Click the button on the **Edit** toolbar.
   The **Find In Files** dialog box appears.

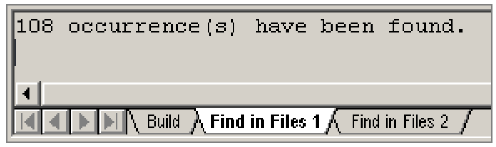


2. Type the text string you want to search for, or select from the list of previously searched-for strings from the **Find what** list.
3. Type the file types to be searched, or select from a predefined list of file types in the **In files/file types** list.
   This box acts as a filter on the names of files to be searched. For example, you can specify "foo*.txt" to search only files with names that begin with "foo" and have an file extension of "txt".
4. Type the name of the folder to be searched in the **In folder** box, or select from the list of previously searched folders in the list. You can also browse for the folder by clicking the **...** button to the right of the **In folder** box.
5. You can also select from the options:

Match whole word only	Search for whole word matches only. The characters (){}[]"'<>,.?/ \;$#@!~+=-|:*&^%, plus space, tab, carriage return and line feed are considered delimiters of whole words.
Match case	Search only for strings that match the case of the characters in your string.
Look in subfolders	Search all subfolders beneath the specified folder.
Output to pane 2	Display the search results in the second output pane, labeled **Find In Files 2**.

6. When you have selected all the options, click **Find**.

The results of the search are displayed in the **Find In Files 1** (or 2) tab of the **Output** window. For each occurrence of the search string that is found, the file name, line number, and line of text are displayed.

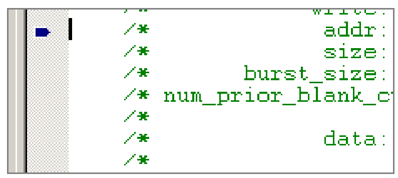


Do any of the following to display an occurrence of the search string:

- Double-click the occurrence.
- Click the occurrence and then press ENTER.
- Press F4 (the default key binding for the GoToNextTag command) to go to the next occurrence. If no occurrence is currently selected, then the first occurrence becomes selected. If the last occurrence is currently selected, then no occurrence is selected, or
- Press SHIFT+F4 to go to the previous occurrence. If no occurrence is currently selected, then the last occurrence becomes selected. If the first occurrence is currently selected, then no occurrence is selected.

In all cases, the window containing the file is automatically put on top of the document windows. If the file isn't already open, it is automatically opened.

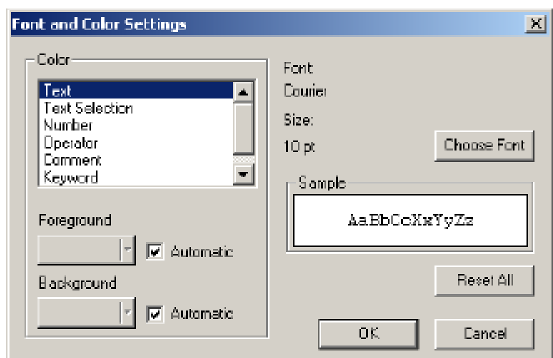
The line containing the occurrence is marked with a blue arrow.



2.5.13 About Fonts and Syntax Coloring

Source files, that is, those with file extensions of .uc, .c or .h, appear with syntax coloring of keywords and comments. Keywords are words that are reserved by the Assembler and Compiler are used in specific context. For example, 'alu_shf' is reserved because it is an Assembler instruction.

Comments comprise ';' followed by text on a line in Assembler language. By default, keywords are colored blue and comments are colored green.



To change color defaults:

1. Open a source file.
2. On the **Tools** menu, click **Font and Color Settings**.

   The **Font and Color Settings** dialog box appears.
3. In the **Color** list box, select the item for which you want to specify a color.

At the **Foreground** and **Background** controls, the colors already selected for the item you selected in Step 3 are displayed.

- Select **Automatic** to use the Window's default colors.

- Clear **Automatic** to enable the color selection controls. Then select a color for the item you selected.

Continue this procedure for any other items that you want to change.

- To change fonts, click **Choose Font** to select a different font for display.
- To go back to original settings, click **Reset All**.

  Your customized settings are saved in the UcSyntaxColoring.ini file located in the folder with the Workbench executable.

2.5.14 About Macros

The **FileView** tab in the **Project Workspace** has a Macro folder that contains the macros that are defined in the project's source files.

The macros are:

- Listed alphabetically, in the **By Name** folder, and
- Grouped according to the file that they are defined in, in the **By File** folder.

The Workbench:

- Creates these folders when you open a project.
- Updates them when:
  - — An edited source file is saved,
  - — A source file is inserted into or deleted from the project, or
  - — You update dependencies, by selecting **Update Dependencies** from the Project menu.

To go to the location in the source file where a macro is defined, double-click the macro name.

If an opened source file contains a macro reference and you want to go to the file and location where that macro is defined:

1. Right-click the macro reference.
2. Click **Go To Macro Definition** on the shortcut menu.

2.6 The Assembler

The Workbench contains an Assembler for your *.uc source files. The following topics on the Assembler will help you understand:

- How root files and dependencies are determined (see Section 2.6.1).
- How to make and change Assembler build settings (see Section 2.6.2).
- How to invoke the Assembler (see Section 2.6.3).
- How to handle assembly errors (see Section 2.6.4).

For information on:

- Creating new files, see Section 2.5.1.

- Saving files, see Section 2.5.4.
- Opening files, see Section 2.5.2.
- Editing files, see Section 2.5.10.
- Closing a file, see Section 2.5.3.
- Searching for text in a source file, see Section 2.5.10 and Section 2.5.12.
- Fonts and syntax colors in a source file, see Section 2.5.13.

For details, refer to the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*.

2.6.1 Root Files and Dependencies

The executable image for a Microengine is generated by a single invocation of the Assembler that produces an output '.list' file. You can place all the code for a Microengine into a single source file, or you can modularize it into multiple source files. However, the Assembler allows you to specify only a single filename. Therefore, to use multiple source files, you must designate a primary, or root, file as the one that gets specified to the Assembler. You include the other files from within the root file or from within already included files, by nesting or chaining them. The included files are considered to be descendants of the root file. In the **FileView** tab of the **Project Workspace**, root files are distinguished by having an R to the left of it.

You can designate the same output file to be loaded into more than one Microengine. You can also include the same source file under more than one root file, making the file a descendant of multiple root files.

In order for the Workbench to build list and image files, you must assign a .list file to at least one Microengine. You set root files as part of setting Assembler options. On the **Project** menu, click **Update Dependencies** to have the Workbench update the dependencies for all source files in the project. If a file is included by a source file but is not itself a source file in the project, the Workbench automatically inserts that source file into the project. The Workbench automatically performs a dependency update when a project is opened. When you insert a microcode file into a project, the Workbench checks that file for dependencies.

2.6.2 Selecting Assembler Build Settings

To make or change Assembler settings:

1. On the **Build** menu, click **Settings**.

   The **Build Settings** dialog box appears.
2. Click the **General** tab to specify additional include directories and the processor revision (stepping) range (see Section 2.6.2.1).
3. Click the **Assembler** tab to specify parameters for creating .list files and other Assembler options.

Note: Compiler settings on the **General** tab are covered in Section 2.7.2.

2.6.2.1 General Build Settings

The following settings, on the **General** tab, apply to the compiler as well as the assembler.

Specifying Preprocessor Definitions:

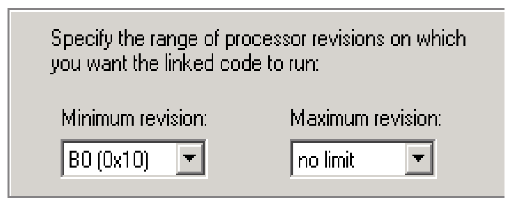
Use the **Preprocessor definitions** edit box to enter preprocessor definitions that will be applied to all microengine list file assembles and compiles in the project. After entering preprocessor definitions on the **General** page, when you open the **Assembler** or **Compiler** pages you will see that the General definitions appear in the command line just prior to any microengine-specific settings. This means that an engine-specific preprocessor definition will override a general setting.

Specifying Processor Revision Range:

The network processors are available in different versions with different features. You can specify a range of revisions (steppings) for which you want your microcode assembled. Section 3.1.5 covers this topic in more detail.

Do the following:

1. On the **Build** menu, click **Settings**.
   The **Build Settings** dialog box appears.
2. Click the **General** tab.
3. Select the range of processor revisions on which you want the linked code to run.



Note: Select from the **Minimum revision** and the **Maximum revision** lists. If you select **no limit** as the maximum revision number then you are specifying that your microcode is written to run on all future revisions of the processor.

2.6.2.2 Specifying Additional Include Paths

The Assembler needs to know which folders contain the files referenced in `#include` statements in Assembler source files.

To do this:

1. On the **Build** menu**,** click **Settings**.
2. Click the **General** tab.

To specify additional Assembler include directories, the following controls are provided:

- A button to specify a new path.
  Type the path name in the space provided or use the browse button **...** to search for it. You must double-click the include path listed in order to display the browse button.

- A  button to delete an included path from the list.
- A  button to move an included path up the list.
- A  button to move an included path down the list.

Absolute versus Relative Paths:

Regardless of whether the path information is entered in an absolute or relative format, it is automatically converted to a relative format. This allows the project to be moved to other locations on a system or to other systems without rendering the path information invalid in most instances as long as files are maintained in the same relative locations. This path information is passed to the Assembler so it may locate the files referenced in #include statements in the source code. It is also used by the dependency checker for locating assembly source files in the project.

2.6.2.3 Specifying Assembler Options

To specify Assembler options, do the following:

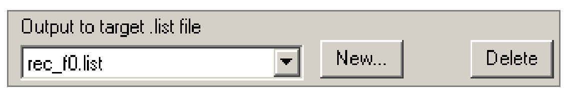
1. On the **Build** menu**,** click **Settings**.

   The **Build Settings** dialog box appears.
2. Click the **Assembler** tab.

The .list File:

The **Output to target .list file** list allows you to select a .list file from the set of .list files that are currently defined in the project. All other controls on the page are updated according to which .list file you select in the list.

Insert file:



1. On the **Assembler** tab, click **New**.

   The **Insert New List File into Project** dialog box appears.
2. Select a path for the .list file.
3. Type a filename.

   You cannot insert a .list file that has already been inserted into the project.
4. Click **Insert List File**.

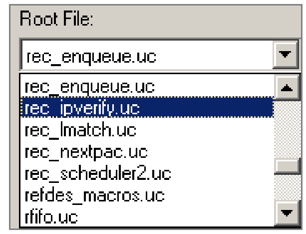
   This closes the dialog box and adds the new filename to the list. The file's path appears in the read-only **Path of target .list file** box. The rest of the boxes assume default values.

Delete File:

To delete a .list file from a project, click **Delete**. This removes the .list file currently selected in the list box from the project. All references to the file on the **Linker** page are removed. The actual .list file, if it exists on disk, is not altered or deleted.

Root files:

The **Root File** list provides a read-only list of all of the .uc and .h files in the project. Select a file to designate it as the root file for the .list file.



If no root file is selected, "- no root file -" (default) is displayed. If a root file is not selected and you attempt to select another page or close the dialog box with the **OK** button, a warning message appears.

Warning Level:

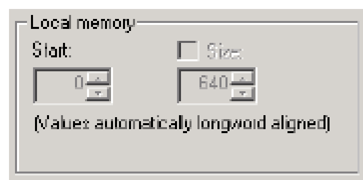
Use the arrow buttons to raise or lower the **Warning level** numbers corresponding to the warning level that you want to specify. For more information on warning levels, refer to **Chapter 3 Assembler**.

Warnings as Errors:

Select **Warnings as errors** to indicate to the Assembler to treat all warnings as errors.

Local Memory:

The **Local Memory** settings allow the user to specify the region of local memory that is available to the assembler for allocating local memory variables.



The **Start** value is a longword-aligned byte address which specifies the start of the region. If **Size** is unchecked, then the region begins at the start address and extends to the end of local memory. If **Size** is checked, then the region begins at the start address and extends for the number of bytes specified in the **Size** field.

Produce Debug Information:

Select **Produce debug info** to add debug information to the output file. If you do not select this option, you will not be able to open a thread window in debug mode.

Note: The **Produce debug info** switch must be set for the necessary debug information to be present in the uof file. Unchecking the **Produce debug info** check box causes the size of the uof file to be smaller at the expense of the project not being debuggable (in any fashion) through the Workbench

Require Register Declarations:

Select **Require register declarations** to force the programmer to explicitly declare registers in the Assembler source code. Undeclared registers will cause an error. The default for IXP2nnn network processors is enabled.

Automatically Fix A/B Bank Conflicts:

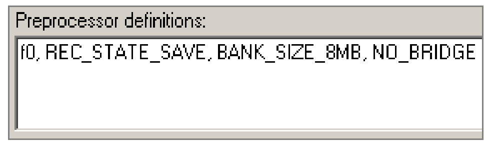
Select **Automatically fix A/B bank conflicts** to have the Assembler try to resolve A/B bank conflicts among registers.

Automatically spill GPRs:

Select **Automatically spill GPRs** to instruct the Assembler to spill GPR Contents to local memory in the event that there are too many registers to fit in the available number of GPRs.

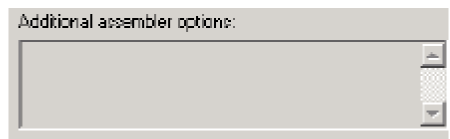
Preprocessor Definitions:

Preprocessor definitions are symbols used in `#ifdef` and `#ifndef` statements to conditionally assemble sections of source files. Multiple definitions are separated by spaces. Optionally a replacement value may be assigned to a definition by append an "=" and a value; no spaces can occur between the symbol name and the "=" or between the "=" and the value. Default is blank.



Additional Assembler Options:

This control allows you to enter text that is used to edit the command line, replacing the **Edit/Override** check box used in previous releases of the Workbench. Text added in this control will appear in the command line just prior to the list file.



Save Build Settings:

The **Build Settings** dialog box works with a copy of the build settings in the project. When you click **OK**, the Workbench validates your data and does not allow the dialog box to close if there are any errors. Validation is independent of which page is active at the time.

You have the option to fix the errors or click **Cancel** if you choose not to save any changes you have made. When the data in Build Settings passes validation, the data in the project is updated.

2.6.3 Invoking the Assembler

To assemble a microcode source file:

1. On the **File** menu, click **Open** to open the file or double-click on the file in **FileView**.

   If the file is already open, activate its document window by clicking on the file window.

2. On the **Build** menu, click **Assemble,** or

   Press CTRL+F7, or

   Click the  button.

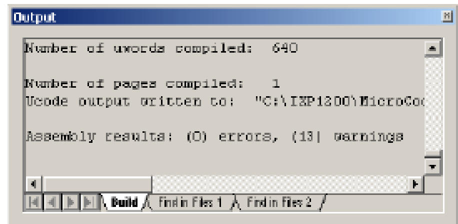
Root Files:

If the file is a project source file, the Workbench assembles all list files for which that file is a root or for which that file is a descendant of a root.

If the file is a project source file, but is not a root or a descendant of a root, or if the file is not in the project, the Workbench assembles it using default assembler settings and produces a list file of the same name with the '.list' file type.

Results:

The results of an assembly appear in the **Build** tab of the **Output** window, which appears automatically.



You can control the amount of detail provide in the results. On the **Build** menu, click **Verbose Output** to toggle between getting detailed results and summary results.

Assembly is also done as part of a build operation.

Note: You can toggle the visibility of the **Output** window by clicking the  button on the **View** toolbar.

2.6.4 Assembly Errors

Assembly errors appear in the **Build** tab of the **Output** window. Do any of the following to display the line of source code that caused an error:

- Double-click the error description, or

  Press F4, or

  Click the  button.

If no error is selected, the first error becomes selected. If the last error is selected, then no error is selected.

To go to the source line for the previous error:

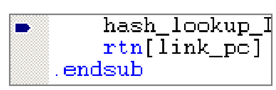
- Press SHIFT+F4, or

  Click the  button.

If no error is selected, then the last error becomes selected. If the first error is selected, then no error is selected.

In all cases, the window containing the source file is put on top of the document windows. If the source file isn't open, Workbench opens it.

A blue arrow in the left margin marks lines containing errors. Only one error at a time is marked.



Note: The default **Debug** toolbar does not contain these buttons. To add them, go to Section 2.2.3.4

2.7 The Microengine C Compiler

The Workbench contains a C Compiler to compile C source code into microcode for the Microengines. The Microengine C Compiler is a general purpose Compiler but the C language used for the Microengines is limited. Refer to the *Microengine C Compiler Language Support* Reference Manual for information on the functions and intrinsics designed for use with the network processors.

The C Compiler can compile a source file (.c, .h) or an object file (.obj).

For information on:

- Creating new C source files, see Section 2.5.1.
- Saving C source files, see Section 2.5.4.
- Opening C source files, see Section 2.5.2.
- Editing C source files, see Section 2.5.10.
- Closing a C source file, see Section 2.5.3.
- Searching for text in a C source file, see Section 2.5.10 and Section 2.5.12.
- Fonts and syntax colors in a C source file, see Section 2.5.13.

This section details:

- Adding C source files to your project (see Section 2.7.1).
- Selecting the target .list file (see Section 2.7.2.2).
- Selecting the target .obj file to compile (see Section 2.7.2.6).
- Deleting a target .list file (see Section 2.7.2.7).
- Selecting C source files to compile (see Section 2.7.2.3).
- Selecting C object files to compile (see Section 2.7.2.4).
- Removing C source files from project (see Section 2.7.2.5).
- Selecting compile parameters (see).
  - "Optimize:"
  - "Inlining:"
  - "Endian Mode:"
  - "Warning level:"
  - "Number of contexts:"
  - "Produce debug information:"
  - "Produce assembly code file:"
  - "Preprocessor definitions:"
  - "Additional compiler options:"

2.7.1 Adding C Source Files to Your Project

After creating and saving C source files, you need to add them to your project. To do this:

1. On the **Project** menu, click **Insert Compiler Source Files**.
   The **Insert Compiler Source Files into Project** dialog box appears.
2. From the **Look in** list, browse to the folder containing your C source file(s).
3. Select the file(s) that you want to insert into your project.
4. Click **Insert**.

In the Project Workspace window, to the left of the Compiler Source Files folder, a '+' appears (if the folder was previously empty) indicating the folder now contains files. Click the '+' to expand the folder and display the files. You should see the files that you have just added to your project.

2.7.2 Selecting Compiler Build Settings

Before building your project, you must select your C Compiler options.

Note: General build settings are detailed in Section 2.6.2.1.

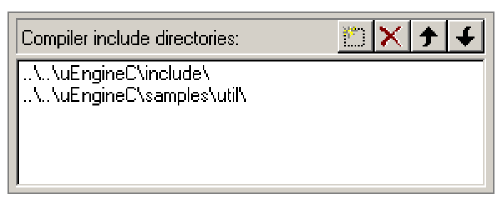
2.7.2.1 Selecting Additional Compiler Include Paths

The C Compiler needs to know which areas of the file system to search for locating files referenced in #include statements in C source code files. This control displays a list of paths with a GUI for typing in or editing of directory paths, or browsing to directories to be added to the list. The GUI also provides the means for deleting or changing the search order of the paths.

Regardless of whether the path information is entered in an absolute or relative format, it is automatically converted to a relative format. This allows the project to be moved to other locations on a system or to other systems without rendering the path information invalid in most instances, as long as the relative location of the paths is maintained. This path information is passed to the Assembler so it may locate the files referenced in #include statements in the source code. It is also used by the dependency checker for locating C source files in the project.

To specify additional Compiler include directories:

1. On the **Build** menu, click **Settings**.
2. In the **Build Settings** dialog box, click the **General** tab (if not already selected).



The following controls are provided:

- A button to specify a new path.
  Type the path name in the space provided or use the ... button to search for it. You must double-click the include path listed in order to display the browse button.
- A button to delete an included path from the list.
- A button to move an included path up the list.

- A  button to move an included path down the list.

2.7.2.2 Selecting the target .list File

When you compile your C source file, the result can become a .list file. You must select the name of the .list file.

To do this:

1. On the **Build Settings** dialog box, click the **Compiler** tab.
2. To change the settings for a previously created .list file, select the name of the .list file from the **Output target .list and .obj files** list.
3. To create a new .list file, click **New .list file**.

   The **Insert New List File into Project** dialog box appears.

   a. In the **Look in** list, browse to the folder where you want to store the .list file.

   b. Type the file name in the **File name** box.

   c. Click **Insert List File**.

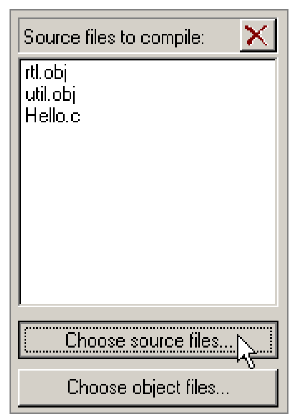
The path of the target .list file box is a read-only text field displaying the absolute path of the target .list file. If this path is not correct, click New again and select a new path.

2.7.2.3 Selecting C Source Files to Compile

The C Compiler in the Workbench can compile one or more C source files into one .list file. You must select the source files that you want to compile. To do this:

1. In the **Build Settings** dialog box, click the **Compiler** tab.
2. Click **Choose source files**.

   The **Compiler Sources** dialog box appears. The files displayed here are all the *.c files in your project, that is all the files in the **Compiler Source Files** folder in the **Project Workspace** window.



3. Click the file(s) that you want to compile.

   Clicking the file once selects the file and clicking a selected file deselects it.

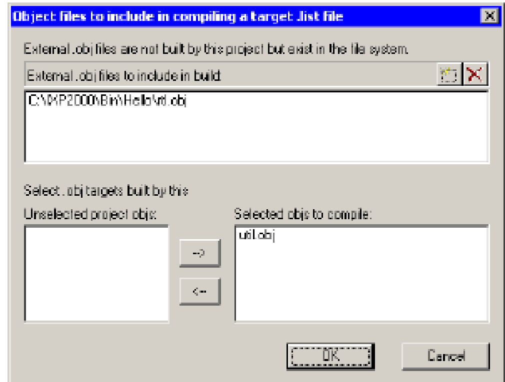
4. Click the  button to move the selected files from the left window to the right window.

   You can select any file(s) in the right window and move them back to the left window by clicking the  button.

5. Click **OK** when done.

In the **Source files to compile** box is a list of C source files that you selected to compile.

2.7.2.4 Selecting C Object Files to Compile



The C Compiler in the Workbench can compile one or more C object files into one .list file. You must select the object files that you want to compile. To do this:

1. On the **C Compiler** tab, click **Choose object files**.

   The **Object file to include...** dialog box appears.
2. Enter the absolute path of any external object files you want to include in the build in the **External .obj files...** box.
3. Use the arrows to select or deselect any of your project object files you want to compile.

2.7.2.5 Removing C Source Files to Compile

To remove any file:

1. Click the desired file in the **Source files to compile** list.
2. Click the  button.

This removes the file from the compilation but not from the project.

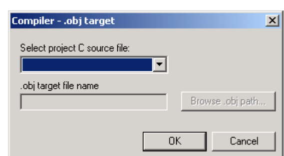
2.7.2.6 Selecting the Target .obj File

You can compile your C source file to create an .obj file rather than a .list file.

To do this:

1. In the **Build Settings** dialog box, click the **Compiler** tab.
2. Click **New .obj file**.

   The **Compiler - .obj target** dialog box appears.



   a. In the **Select project C source file** list, select the name of the .c file you want to compile.

      In the **.obj target file** name box, the source file you selected above appears with an .obj extension. You can change the name of this file if you like. By default, the .obj file that you are creating goes into the current project folder. If you want to place this file into another folder:

      1: Click the **Browse .obj path** button.

      2: Select a new folder.

   b. Click **OK** when done.

2.7.2.7 Deleting a Target .list or .obj File

To delete a target .list or .obj file from the project:

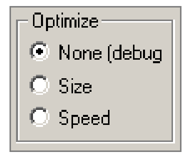
1. Select the file from the list in the **Output to target .list and .obj files** box.
2. Click **Delete**.

Note: This removes the file from the project but does not delete it from the disk.

2.7.2.8 Selecting Compile Options

In the **Compiler Options** box, select:

Optimize:

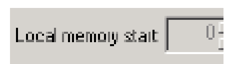


None (debug) — Turns off optimizations for better code troubleshooting.

Size (default) — Compiled for smallest memory footprint. Speed may be sacrificed.

Speed — Compiled for fastest instruction execution. Size may be sacrificed.

Local memory start:

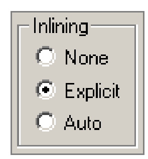


0 (default) — Determines the region in local memory where the Compiler can allocate variables. The region starts at the address you specify and extends to the end of local memory.

Spill sequence:

Determines the algorithm used by the Compiler for spilling register contents to memory

Inlining:



None — No inlining is done, including functions explicitly tagged in the source code with the inline specifier.

Explicit (default) — Only functions tagged with the inline specifier are inlined. Any function that could be inlined by the Compiler but not having this tag is not inlined.

Auto — All functions with the inline tag and all other functions thought by the Compiler to be inlinable are inlined.

Endian Mode:

Little Endian — Compile in little-endian mode.

Big Endian (default) — Compile in big-endian mode.

Neighbor mode:

Neighbor Writing to a neighbor register will write to the neighbor register in the adjacent Microengine.

Self Writing to a neighbor register will write to the neighbor register in the same Microengine as the one executing the instruction.

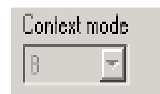
Warning level:

0 Print only errors.

1, 2, or 3 (default) Print only errors and warnings.

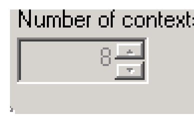
4 Print errors, warnings, and remarks.

Context mode:



4 or 8 Specify whether the Microengine is configured to have four or eight contexts in use.

Number of contexts:



1, 2, 3, 4, 5, 6, 7, or 8 Select the number of contexts that you want to be active in the Microengine. All others are killed.

Produce debug information:

Select (default) Produces debug information in the .list file. This information is needed for many of the debugging features of the Workbench.

Clear No debug information is compiled into the .list file.

Note: The **Produce debug info** switch must be set for the necessary debug information to be present in the uof file. Unchecking the **Produce debug info** check box causes the size of the uof file to be smaller at the expense of the project not being debuggable (in any fashion) through the Workbench

Produce assembly code file:

Select Produces an assembly code file (*.uc).

Clear (default) Does not produce an assembly code file.

Allow mixed C and assembler source files:

Select Allows the user to select both C and MicroCode project files instead of only C files. The compiler is invoked with the “-uc” switch to indicate that the compiler should automatically invoke the assembler after generating the microcode. The -FA”path\filename.ext” switch is used to specify the name of the generated assembler source file. The filename chosen by the Workbench is always the same path and filename as the LIST file, but with a file extension of “.ucg” instead of “.list”.

Clear (default) Allows only C files to be selected.

Preprocessor definitions:

This is a text edit box where you type symbols used in #ifdef and #ifndef statements to conditionally compile sections of Assembler sources. Multiple definitions are separated by spaces. Optionally a replacement value may be assigned by appending an "=" and a value. There can be not spaces between the symbol name and the "=" or between the "=" and the value. The default is blank.

Additional compiler options:

Here you can enter additional command line options that can not be implemented by normal GUI controls. See Chapter 4, "Microengine C Compiler" for complete list of options.

2.7.2.9 Saving Build Settings

The **Build Settings** dialog box works with a copy of the build settings in the project. When you click **OK**, the Workbench validates your data and does not allow the dialog box to close if there are any errors. Validation is independent of which page is active at the time. You have the option to fix the errors or click **Cancel** if you choose not to save any changes you have made. When the data in **Build Settings** passes validation, the data in the project is updated.

2.7.3 Invoking the Compiler

To compile a C source file:

1. On the **File** menu, click **Open**, or

   You can also double-click the file in **FileView**. If the file is already open, activate its document window by clicking on the file window.

2. On the **Build** menu, click **Compile**, or

   Press CNTRL+SHIFT+F7, or

   Click the  button on the **Build** toolbar.

Results:

The results of an assembly appear in the **Build** tab of the **Output** window, which automatically appears.

You can control the amount of detail provide in the results. On the **Build** menu, select **Verbose Output** to display detailed results, or clear it to display summary results.

Compilation is also done as part of a build operation.

2.7.4 Compilation Errors

Compiler errors appear in the **Build** tab of the **Output** window. To locate the error in the source file:

- Double-click the error description in the **Output** window, or

  Click the error description, then press ENTER.

You can press F4 or click the button to go to the next error. If no error is selected in the **Output** window, the first error becomes selected. If the last error is selected, then no error is selected, or

You can press SHIFT+F4 or click the button to go to the previous error. If no error is selected in the **Output** window, the last error becomes selected. If the first error is selected, then no error is selected.

In all cases, the window containing the source file is put on top of the document windows and becomes the active document. If the source file isn't already open, it opens.

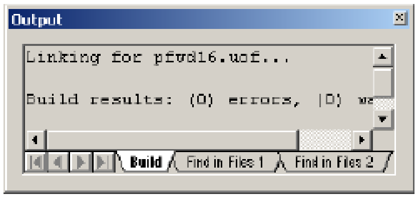
A blue arrow in the left margin marks lines containing errors. Only one error at a time is marked.

2.8 The Linker

The Linker:

The Linker takes the Assembler or Compiler output (.list files) on a per-Microengine basis and generates an image file for all the Microengines specified. To invoke the Linker and build a project:

Results:



The results of the build appear in the **Build** tab of the **Output** window, which appears automatically. You can control the amount of detail provide in the results. On the **Build** menu, click **Verbose Output** to toggle between getting detailed results and summary results.

Rebuild:

To perform a full unconditional build of your configuration:

- On the **Build** menu, click **Rebuild**, or

  Press ALT+F7, or

  Click the button on the **Build** toolbar.

2.8.1 Customizing Linker Settings

To customize your build configuration:

1. On the **Build** menu, click **Settings**.

   The **Build Settings** dialog box appears.
2. Click the **Linker** tab to view the Linker settings.

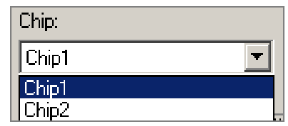
3. Customize the Linker settings (see Section 2.8.1).
4. Click **OK**.

The **Linker** page provides an interface for selecting options for the Linker and directing the packaging of one or more Microengine specific *.list files into a *.uof file. Each chip has several Microengines that can each be loaded with execution code according to the *.list file selected for that Microengine.

You can also specify the assembly options by clicking the **Assembler** tab and the **C Compiler** tab in the **Build Settings** dialog box.

Chip

The **Chip** box contains a list of all the Network Processor chips in your project. Select the chip for which you want to change Linker settings. The other controls on the page are updated based on the selected chip.



Output to target .uof file

The **Output to target .uof file** box displays the .uof file that the Linker produces.

Note: The Developer Workbench does not support multiple .uof files.

To change the output *.uof file to the project for the selected chip:

1. Click the ... button.

   The **Select Name and Location for the Linker Output File** dialog box appears.
2. In the **Look in** box, browse to the folder where you want to put the output file.
3. Type a new name in the **File name** box.

   You do not have to type the .uof extension—the Workbench adds it for you. Typing it does no harm.
4. Click **Select**.

Microengine .list file selection

The project has one or more .list file(s) generated using the Assembler or Compiler. On the **Linker** page you can control which .list file is linked into the .uof file and for which Microengine.

To do this:

1. Click the list box to the right of **Microengine 0:0**. A scrollable list of list files is displayed.
2. Select either:
   a. <none>, or
   b. Any .list file from list.
3. Do the same for remaining Microengines.

This method allows you to select any combination of .list files or no .list file for any or all the Microengines to be linked to the .uof file.

If you specify <none>, no microcode gets loaded into that Microengine. If you select <none> for all the Microengines, you get an error.

Produce debug information

Select **Produce debug info** to add debug information to the output file. If you do not select this option, you will not be able to open a thread window in debug mode.

Note: The **Produce debug info** switch must be set for the necessary debug information to be present in the uof file. Unchecking the **Produce debug info** check box causes the size of the uof file to be smaller at the expense of the project not being debuggable (in any fashion) through the Workbench

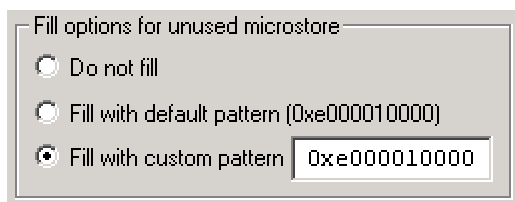
Generate memory-map file

Select **Generate memory-map file** to have the Workbench pass the **-map** option switch to the linker to generate a .map file. The file contains the symbols and their addresses. The edit control allows the user to specify the filename or browse to it.

Hex ".c" files

Select **Generate hex '.c' file** to request the Linker to create a *.c file, with the same name as the corresponding *.uof file. This file contains a microcode listing in a form that can be included in a processor core application. This is usually done when deploying microcode into a final product.

Unused microstore



Unused microstore can be initialized by using the controls in the **Fill options for unused microstore** area.

To leave the unused microstore unchanged, select **Do not fill.**

To fill the unused microstore:

1. Select **Fill with default pattern (0xe000010000)**.

   0xe000010000 for the IXP2400 and IXP2800/IXP2850 network processors.

or

2. Click **Fill with custom pattern** and type a 10 character hex pattern to be used. Make sure the number begins with "0x."

Reserved memory segment for variables

The reserved memory segment for variables provides the Linker with information needed for allocating memory to be used for variable data storage.

Scratch offset

The Scratch offset is a parameter sent to the Linker. The Linker uses scratch memory starting at the base address, allocating as much memory as needed up to the Scratch offset size for variables.

Scratch segment size (bytes)

The Scratch segment size is a parameter sent to the Linker. The Linker reserves as much scratch memory as necessary for variables up to the segment size.

SRAM offset

The SRAM offset is a parameter sent to the Linker. The Linker uses scratch memory starting at the base address, allocating as much memory as needed up to the SRAM segment size for variables.

SRAM segment size (bytes)

The SRAM segment size is a parameter sent to the Linker. The Linker reserves as much SRAM as necessary for variables up to the segment size.

DRAM offset

The DRAM offset is a parameter sent to the Linker. The Linker uses DRAM memory starting at the base address, allocating as much memory as needed up to the DRAM segment size for variables.

DRAM segment size (bytes)

The DRAM segment size is a parameter sent to the Linker. The Linker reserves as much DRAM as necessary for variables up to the segment size.

Header file generation

Selecting Generate a header file causes the Linker to produce a C language *.h file with the same filename as the linked *.uof file. The defined symbols are set to values based on how the Linker allocated memory for the reserved memory variables. The base address symbols should have the same values as the ones defined in the GUI, but the size symbols have the actual sizes used by the Linker.

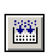
Saving settings

Linker settings are saved when you save the project.

The **Build Settings** dialog box works with a copy of the build settings in the project. When you click **OK**, the Workbench validates your data and does not allow the dialog box to close if there are any errors. Validation is independent of which page on **Build Settings** is active at the time. You have the option to fix the errors or click **Cancel** if you choose not to save any changes you have made. When the data in **Build Settings** passes validation, the data in the project is updated.

Building a project

- On the **Build** menu, click **Build**, or

  Click the  button on the **Build** toolbar, or

  Press F7.

Stopping

There is no way to stop a build in progress. You must wait until it finishes or encounters an error.

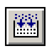
Out-of-date files

To perform a link, the Workbench requires that all .list files be up to date. If any microcode or compiler source file is newer than the list file generated from it, or if Assembler or Compiler settings have been changed since the last build, the Workbench automatically assembles or compiles a new .list file.

Rebuilding a project

To force the assembling or compiling of all sources, regardless of whether the list files are up to date:

- On the **Build** menu, click **Rebuild**, or

  Click the  button on the **Build** toolbar, or

  Press Alt + F7.

2.9 Configuring the Simulation Environment

To configure the simulation environment, select **System Configuration** from the **Simulation** menu. You can set or change configuration values in the following property pages depending on the Chip Family you have selected:

- **Clock frequencies** - see Section 2.9.1
- **Memory** - see Section 2.9.2
- **MSF Devices** - see Section 2.9.4
- **MSF Connections** - see Section 2.9.4
- **CBUS Connections** - see Section 2.9.4

The contents of each dialog depends on the processor type defined for the project. This configuration data is passed to the Transactor and device models through the command line interface when you start debugging.

2.9.1 Clock Frequencies

Depending on the processor type defined for the project the **Clock Frequencies** property page will display different default values. The values that you specify on this property page are passed to the Transactor using the `set_clocks()` console function.

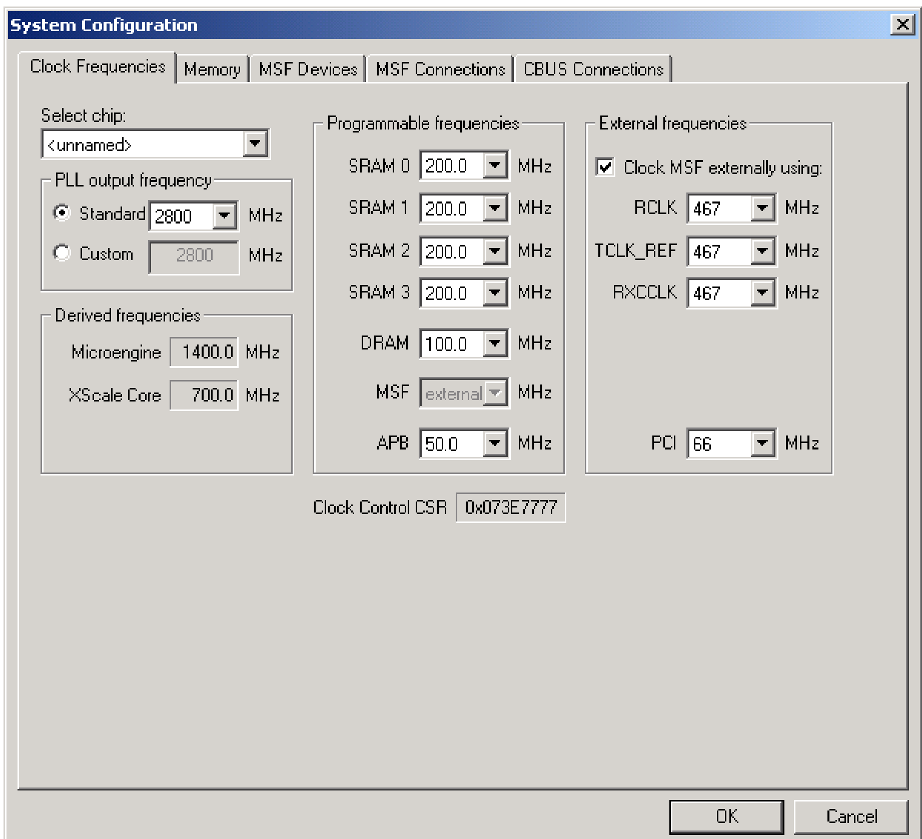
2.9.1.1 IXP2800 Clock Frequencies

- Clock frequencies are set independently for each chip in the project.
- In the **PLL output frequency** group box, select the PLL output frequency: Supported frequencies are 1600, 2000, and 2800.

- The **Derived frequencies** group box is for information only and displays the frequencies for the Microengines and the Intel® XScale™ core. These frequencies are derived from the **PLL output frequency** using fixed divisors and they cannot be modified.
- The **Programmable frequencies** group box contains selectable values for the four SRAM channels, DRAM, MSF and APB. These frequencies are programmable as a fixed set of ratios of the **PLL output frequency**. These ratios correspond to fields in the **Clock Control CSR**, which is displayed for reference only.
- In the **External frequencies** group box, you can optionally specify that the MSF unit be externally clocked. The PCI unit is always externally clocked. Select either 33 or 66 MHz from the PCI combo box.

The values shown in the following figure are the default values. The complete description of clock frequencies, ratios, can be found in the *Intel® IXP2800 Network Processor Hardware Reference Manual*. Clock and MSF CSRs can be found in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual.*

Figure 6. Clock Frequencies for the IXP2800

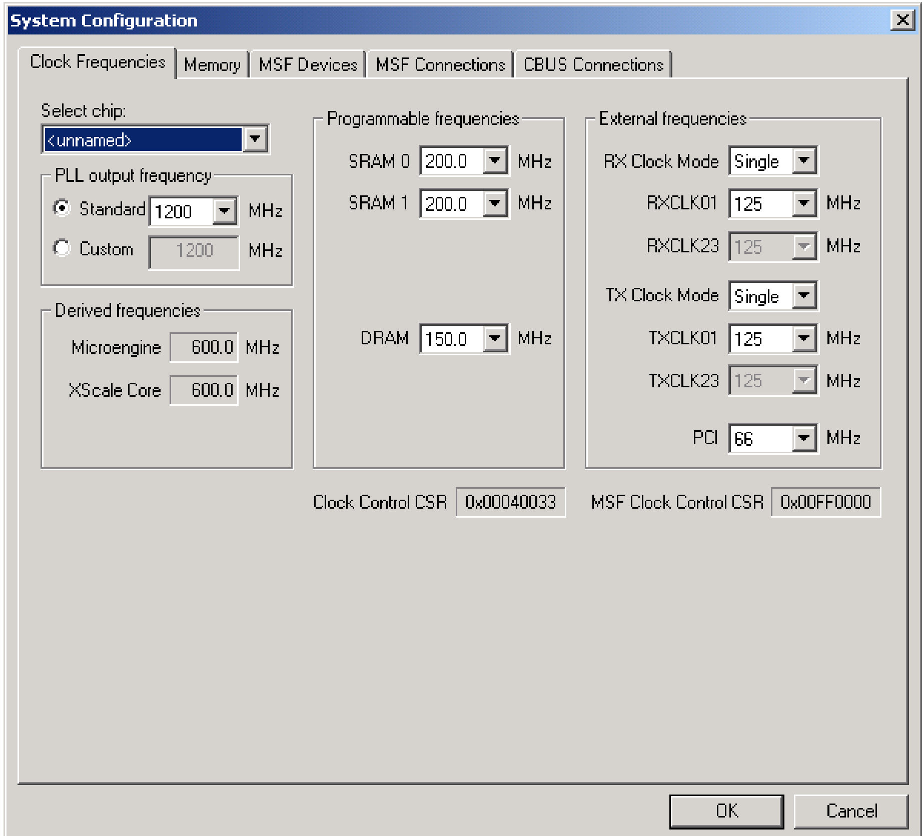


2.9.1.2 IXP2400 Clock Frequencies

- Clock frequencies are set independently for each chip in the project.
- In the **PLL output frequency** combo box, you may currently select the PLL output frequency of 1200 MHz.
- The **Derived frequencies** group box is for information only and displays the frequencies for the Microengines and Intel® XScale™ core. These frequencies are derived from the **PLL output frequency** using fixed divisors and they cannot be modified.
- The **Programmable frequencies** group box contains selectable values for the two SRAM channels and DRAM. These frequencies are programmable as a fixed set of ratios of the **PLL output frequency**. These ratios correspond to fields in the **Clock Control CSR**, which is displayed for reference only.
- In the **External frequencies** group box, the MSF unit is externally clocked. Select either **Single** or **Dual** mode clocking for receive and transmit interfaces. If you select **Single** mode, only one clock value can be specified; the other control is disabled. Ratios correspond to fields in the **MSF Clock Control CSR**, which is displayed for reference only.
- The **PCI** unit is always externally clocked. Select either 33 or 66 MHz from the scroll values.

The values shown in the following figure are the default values. The complete description of clock frequencies, ratios, can be found in the *Intel® IXP2400 Network Processor Hardware Reference Manual*. The Clock and MSF CSRs can be found in the *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual.*

Figure 7. Clock Frequencies for the IXP2400



2.9.2 Memory

The **Memory** tab on the **System Configuration** property sheet supports the configuration of simulator memory. There are some variations on the screen depending on which network processor is being configured. For both the IXP2400 and the IXP2800 the following simulator conditions apply:

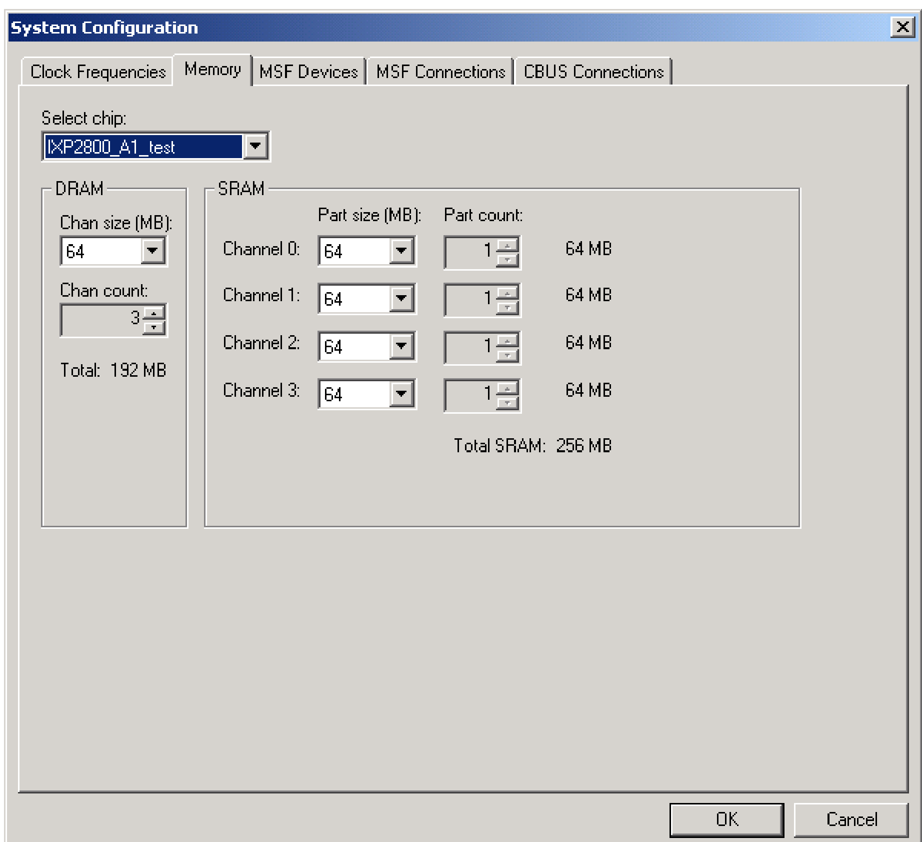
- No SRAM channel can exceed 64 MB, so the **Part count** option of **2** becomes unavailable if the **Part size** is 64.
- The simulator must have a populated SRAM channel. Zero memory cannot be configured, therefore there is no option for a zero **Part size** or **Part count**.

2.9.2.1 IXP2800 Memory Options

There are four channels available for configuring the IXP2800 SRAM memory.

- The DRAM **Chan size** value multiplied by the **Chan count** value yields the total DRAM memory available in the simulator. Between 1 and 3 DRAM channels are possible. The DRAM size is limited to 2046 MB, so the available channel count value options become limited at the higher channel sizes.
- Each of the four SRAM channels may be configured independently.

Figure 8. IXP2800 Memory Options

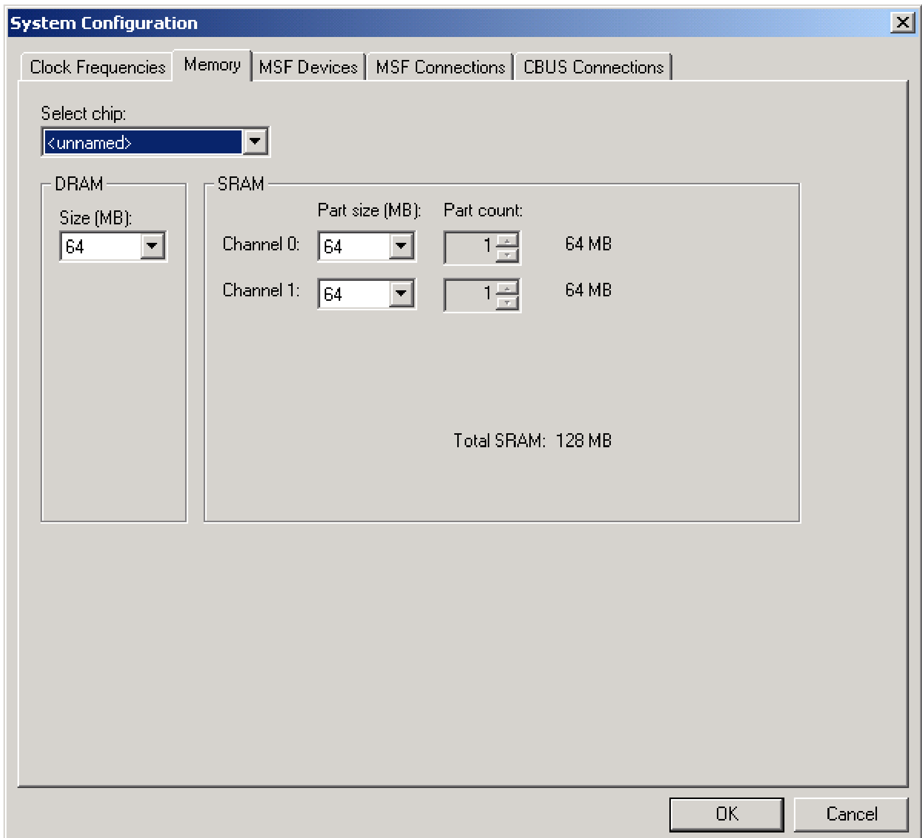


2.9.2.2 IXP2400 Memory Options

There are two channels available for configuring the IXP2400 SRAM memory.

- The DRAM size is currently limited to 64 MB by the simulator, so there is only one DRAM size option available. **Channel count** is unavailable for the DRAM.
- Each of the two SRAM channels may be configured independently.

Figure 9. IXP2400 Memory Options



2.9.3 MSF Device Configuration

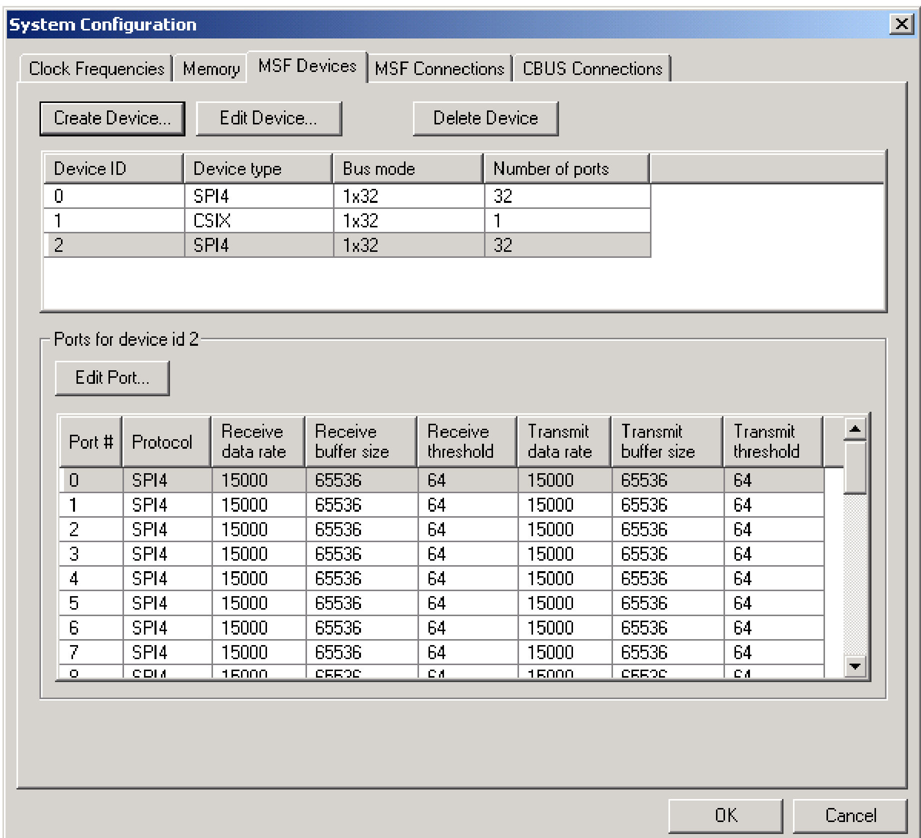
The **MSF Device Configuration** tab on the **System Configuration** property page supports the configuration of media and switch fabric interfaces.

You have the following options on this tab:

- **Create Device...**
- **Edit Device...**
- **Delete Device**
- **Edit Port...**

If no device is currently configured, only the **Create Device...** button will be available. The **Delete Device** and **Edit Port...** buttons become active when devices are configured for selection.

Figure 10. MSF Devices



Device Creation

To create a device:

1. Click **Create Device...**

   The **Create Media Bus Device** dialog box appears. (see Figure 11.)

2. Select the device type from those available on the **Select device type...** scrolling list, for example, **SPI4** or **CSIX** for the IXP2800.

   Supported device types for IXP2800/IXP2850 A0, A1, A2 and B0: **SPI4** and **CSIX.**

   Supported device types for IXP2400 A1: **SPHY, x32MPHY4, x32MPHY16,** and **CSIX**.

   Supported device types for IXP2400 B0: **SPHY, x32MPHY4, x16MPHY32, x32MPHY32,** and **CS1X.** The IXP2400 B0 architecture also supports connecting two devices with different restrictions.

   The **Device parameters** and **Default port parameters** areas will display default values once you select the device type.

Figure 11. The Create Media Bus Device Dialog Box for SPI-4

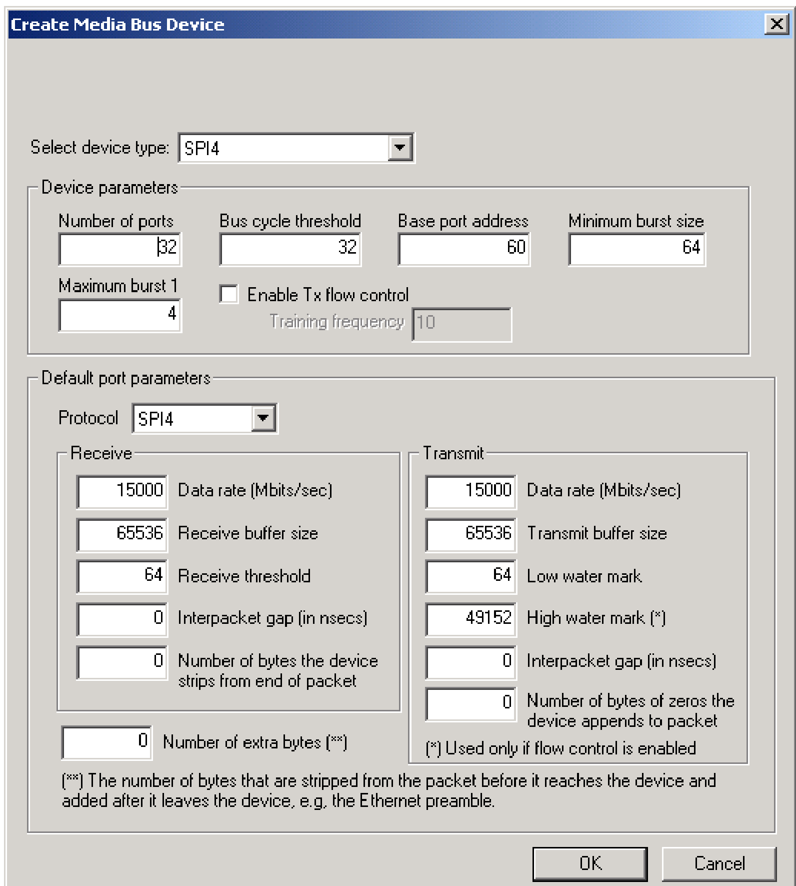


Figure 12. The Create Media Bus Device Dialog Box for CSIX

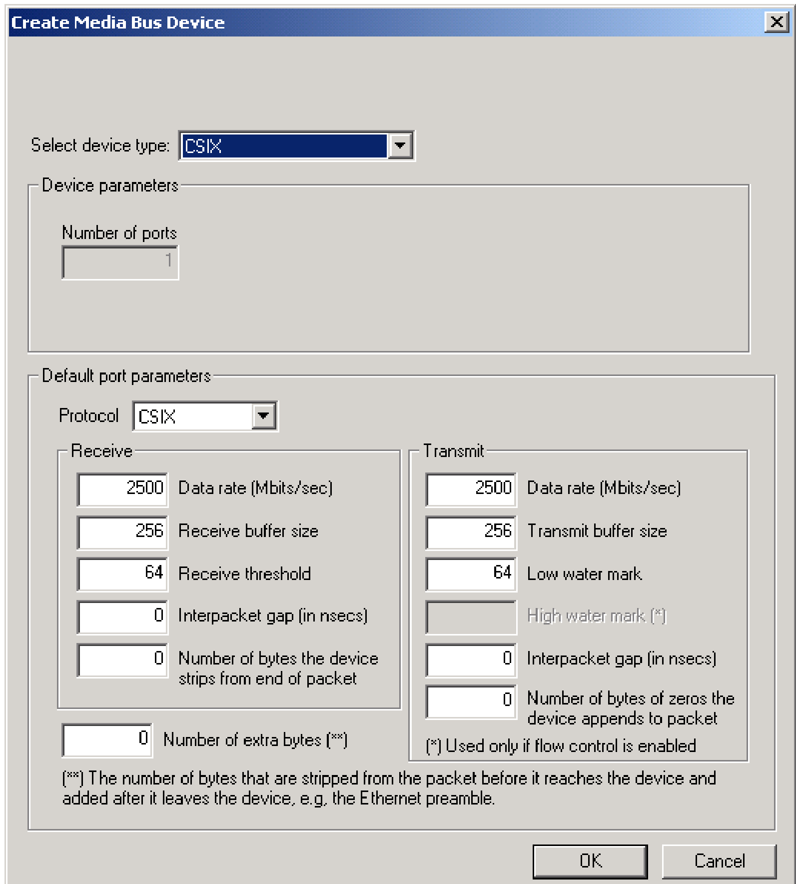
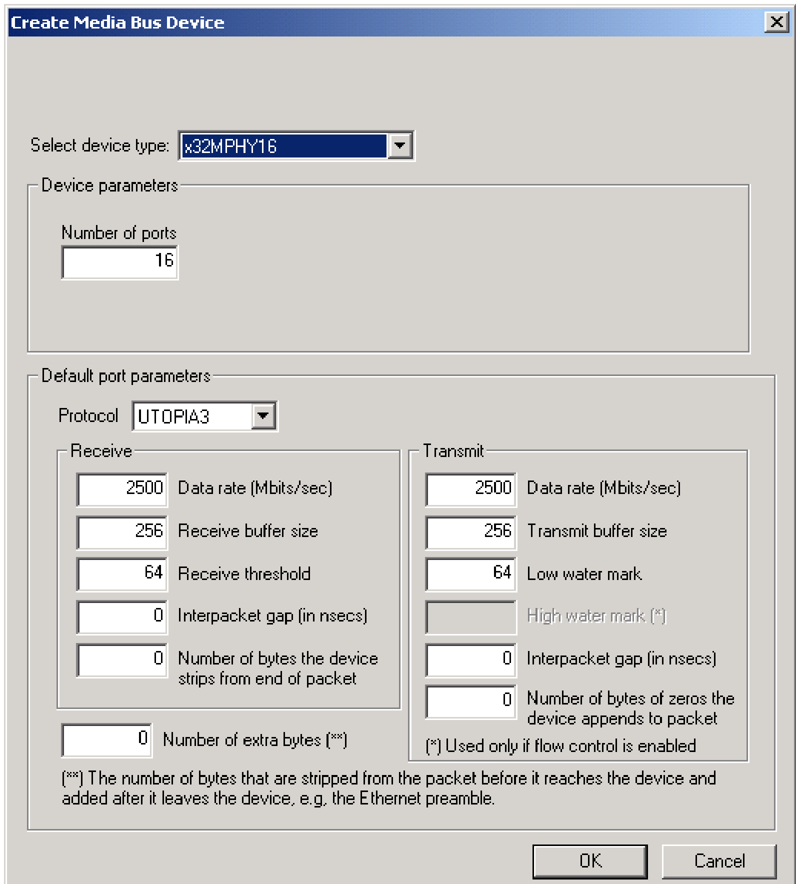


Figure 13. The Create Media Bus Device Dialog Box for x32MPHY16



Note: If you select **SPI4** as the device type, the **Create Media Bus Device** dialog box will display the defaults for the controls shown in Figure 11 in the **Device parameters** section: **Number of ports, Bus cycle threshold, Base port address, Minimum burst size, Maximum burst 1, Enable Tx flow control, and Training frequency**.

If you select a device type other than **SPI4** (CSIX, SPHY, etc.), the **Device parameters** will be re-positioned on the screen as appropriate to the device type.

Bus cycle threshold — This value determines the default maximum # of MSF cycles (one cycle is equal to 2 bytes) per burst. A long burst will be broken into small bursts with non-payload cycles inserted

Base port address — The base address of a SPI4 MAC device. The SPI4 supports 8-bit address, so for a device with 10-port, the base port address could be between 0 and 245. The SPI4 device assigns consecutive address to ports of the same device. So, for a device of 10-port with base port address of 220. The port address space for this device will be 220 to 229.

Minimum burst size — This value determines the minimum number of bytes that the SPI4 device will transfer in one transaction, unless EOP is reached.

Maximum burst 1 — This value is the maximum number of 16 byte blocks that the Tx FIFO can accept when the FIFO Status channel indicates a “Starving” condition.

Enable Tx flow control — This check box enable Tx flow control training between the SPI4 device and the network processor.

Training frequency — This value indicates the maximum interval (in TS clock cycles, with the TS clock being one-fourth the Tx clock Frequency) between scheduling of flow control training sequences.

The **Default port parameters** section is further divided into **Receive** and **Transmit** areas. You may edit these characteristics, which are:

Data rate (Mbits/sec) — Specifies the rate at which data is taken from the network and inserted into the port’s receive (Rx) buffer and the rate at which data is taken from the port’s transmit (Tx) buffer and put onto the network.

Receive buffer size — Specifies the number of bytes in the receive buffer. The receive buffer holds the data received from the network until the Network Processor reads it from the port.

Receive threshold — Specifies number of bytes that must be in the port’s receive buffer in order for the port to signal the Network Processor that it can select the port and request data from it.

Transmit buffer size — Specifies the number of bytes in the transmit buffer. The transmit buffer holds the data transmitted by the Network Processor until it is transmitted onto the network.

Low water mark — See **High water mark**, below.

High water mark — If flow control is enabled, the high water mark is used to determine if the device is “Hungry” or “Satisfied”. If the number of bytes in the Tx buffer is between the low and high water marks, then the device tells the network processor that it is Hungry. If the number of bytes is above the high water mark, then the device tells the network processor that it is Satisfied.

Interpacket gap (nsec) — Specifies the amount of time between packets when receiving packets from and transmitting packets to the network.

Number of bytes the device strips from end of packet

Specifies the number of bytes that the device must strip from the end of each received packet before the packet is passed to the Network Processor. For example, for POS IP packets, the trailing checksum bytes are normally stripped.

Number of bytes of zeros the device appends to packet

Specifies the number of bytes of zeroes the device appends to the packet before it is transmitted by the Network Processor.

Number of extra bytes — Specifies the number of bytes that are stripped from the beginning of the packet before it reaches the device and appended to the beginning of the packet after it leaves the device, for example the Ethernet preamble.

Note that no bytes are actually stripped or appended to the packet data. Instead, the number of extra bytes are added into the calculation of data rate at the network.

Device Removal:

To remove a device from the project:

1. On the **Simulation** menu, click **System Configuration**, then click the **MSF Devices** tab.

The **MSF Devices** property sheet appears.

2. Select a device that you want to remove. Previously created devices appear in the list box under the **Create** button. You can select one by clicking anywhere in the row listing.
3. Click **Delete Device.**

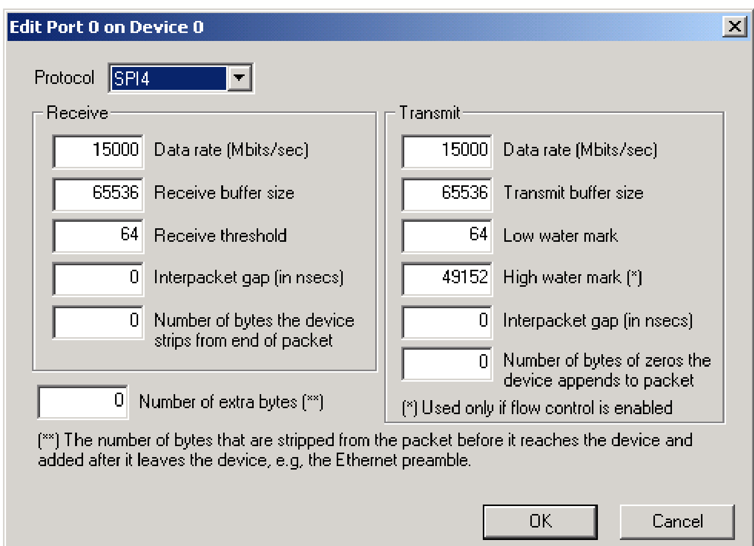
Port Characteristics Edit:

To edit an individual port's characteristics:

1. On the **Simulation** menu, click **System Configuration**, then click the **MSF Devices** tab.
   The **MSF Devices** property sheet appears.
2. In the **Port** section of the property sheet, select the port that you want to modify, click the **Edit Port ...** button, and the Edit Port dialog opens (see Figure 14).
3. When you have finished editing the port, click **OK**.

Any changes that you have made now appear in the corresponding column of the edited port.

Figure 14. Port Characteristics Edit Port Dialog Box



2.9.4 MSF Connections

After you have configured the packet simulation with devices and ports and created or imported data streams, you need to specify the connections to the media/switch fabric for each chip in your project.

1. On the **Simulation** menu, click **System Configuration**, then click the **MSF Connections** tab.

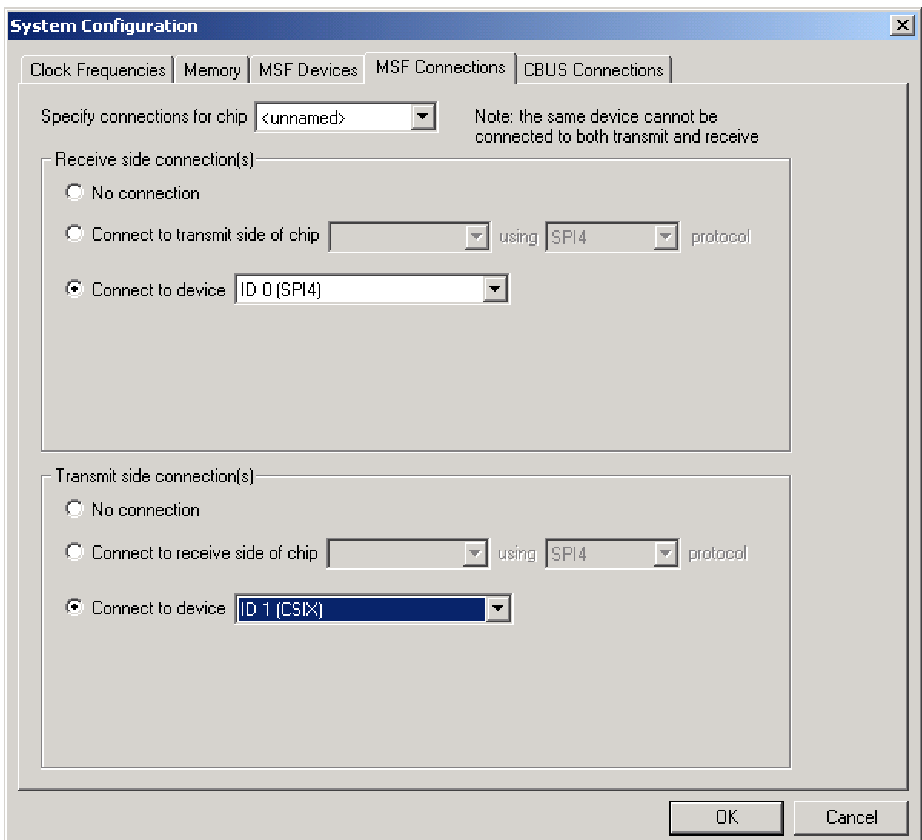
   The **MSF Connections** property page appears. The page is divided into two sections: **Receive side connection** and **Transmit side connection.**

2. Select the chip to which you want to make connections in the **Specify connections for chip** combo box.

3. When you have finished click **OK**.

Figure 15 displays the property page for the IXP2400 (A1), IXP2800 and IXP2850. Figure 16 displays the property page for the IXP2400 (B0).

Figure 15. MSF Connections Property Page - IXP2400 (A1), IXP2800 and IXP2850



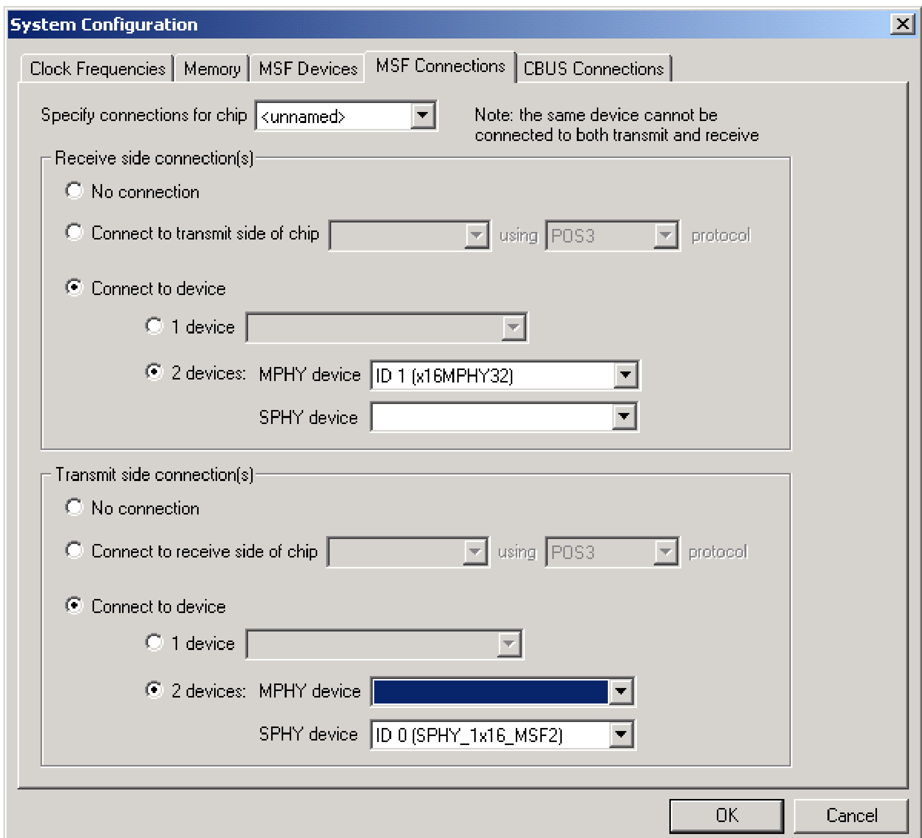
If **No Connection** is selected, then the simulation runs without anything connected to that side of the MSF.

In multi-chip projects, the receive side can be connected to the transmit side of another chip in the project by selecting **Connect to transmit side of chip**. The user must select which chip and what protocol to use for the connection. For IXP28xx chips, the only supported protocol is SPI4. For IXP2400 A1, the only supported protocol is CSIX. For IXP2400 B0, the protocols are POS3 and CSIX. Similarly, the transmit side can be connected to the receive side of another chip in the project by selecting **Connect to receive side of chip**.

To connect a device to either side, the user selects **Connect to device** then selects the desired device in the combo box. Note that because a device can only be connected once, if it is selected for a connection then the Workbench removes it as a choice for all other connections.

For the IXP2400 B0, the Workbench displays the property page shown in Figure 16. It is different because the IXP2400 B0 supports connecting two devices. When two devices are connected, the 32-bit bus is considered to be split into two 16-bit busses – a lower and an upper bus. Only an x16MPHY32 device can be connected to the lower bus and only an SPHY device with 1x16 or 2x8 bus mode can be connected to the upper bus.

Figure 16. MSF Connections Property Page - IXP2400 (B0)



2.9.5 CBUS Connections

After you have configured the packet simulation with devices and ports and created or imported data streams, you need to specify the connections to the media/switch fabric for each chip in your project.

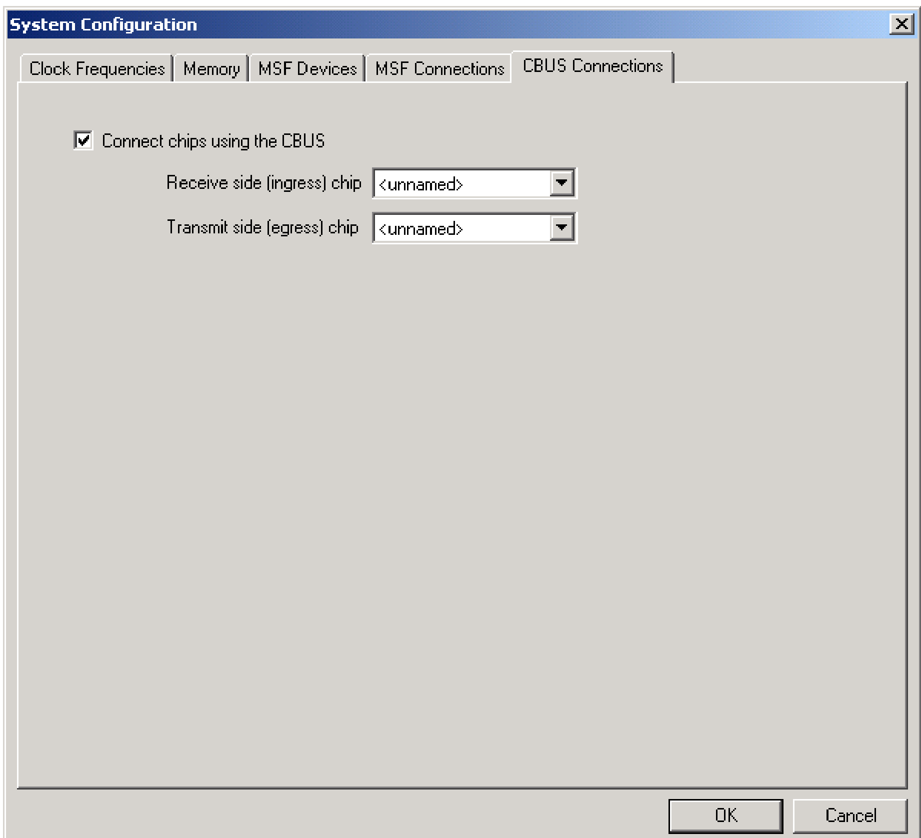
1. On the **Simulation** menu, click **System Configuration**, then click the **CBUS Connections** tab.

   The CBUS **Connections** property page appears. The page contains a check box to enable connections using the CBUS. When you select **Connect using the CBUS** the receive (ingress) and transmit (egress) pull down boxes are active.

2. Select the chip to which you want to make connections in the pull down boxes.
3. When you have finished click **OK**.

Figure 17 displays the CBUS Connections property page for the IXP2400, IXP2800, and IXP2850.

Figure 17. CBUS Connections Property Page



2.10 Packet Simulation

The Workbench provides packet simulation of Media bus devices as well as simulation of network traffic. To simulate devices and network traffic you need to:

1. Configure the devices on the media bus using the **System Configuration** menu (or busses if you have multiple Network Processor chips).

   This involves specifying how many devices are on the bus as well as the characteristics of each device.

2. Create one or more data streams (see Section 2.11).

   These streams can consist of, but are not limited to, Ethernet frames or ATM cells.

3. Assign one or more data streams or a network traffic DLL to each device port that you want to receive network traffic.

4. Specify the options under which you want the traffic simulation to operate using the **Packet Simulation Options ...** menu.

   The **Packet Simulation Options** dialog box has four tabs:

   General - (see Section 2.10.1).

   Port Logging - (see Section 2.10.2).

   Stop Control - (see Section 2.10.3).

   Traffic Assignment - (see Section 2.10.4).

   Manage NTS Plug-ins - (see Section 2.10.5).

While you are running your simulation, you can observe the connections you have assigned to the devices that you have created.

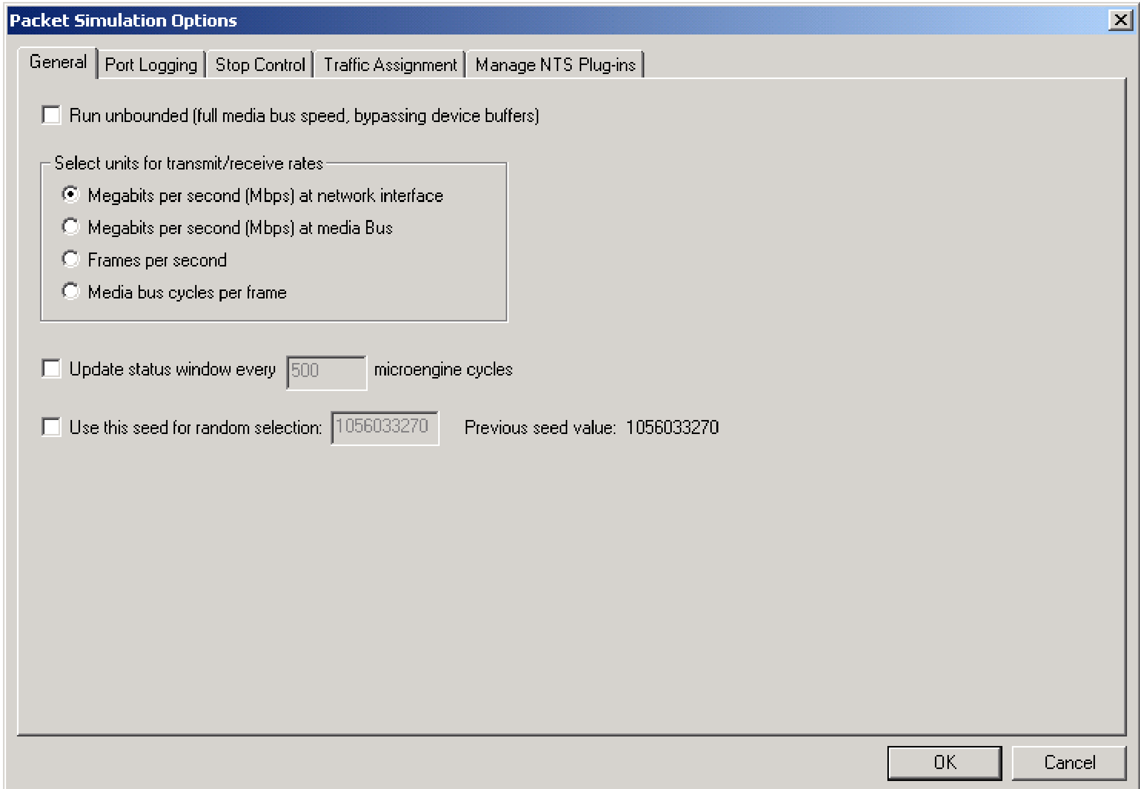
Packets won't be received until you execute the command line function `ps_start_packet_receive()`. This can be done in several ways:

— Go to the **Command Line** window and type the command *ps_start_packet_receive();*, or

— Add the command `ps_start_packet_receive();` to one of your startup scripts at the point where you want packet reception to begin, or

Create a command script (see Section 2.12.7) with the command *ps_start_packet_receive();*, add the command script's button to the toolbar, then click the button when you want to start receiving packets.

Figure 18. Packet Simulation Options Property Sheet- General Tab



2.10.1 General Options

In the **Packet Simulation Options** property sheet (see Figure 18), Under the **General** tab:

- **Run unbounded (infinite wire speed)**.

  Enable **Run unbounded (infinite wire speed)** to have data always ready to be received by the Network Processor and to have the ports always ready to receive data from the Network Processor.

  This makes the simulation act as if data is coming from and going to the network at full media bus speed, bypassing the receive and transmit buffer and ignoring the data rate and interpacket gap values set for the port.

  If this option is cleared, data is received from and transmitted to the network at the specified data rate with an interpacket gap.

- The **Select units for transmit/receive rates** section of the dialog box displays receive and transmit rate data.

  — **Megabits per seconds (Mbps) at the network interface**—this calculation includes extra bits and bits that would have been processed in an interpacket gap.

  — **Megabits per seconds (Mbps) at media Bus**.

  — **Frames per second (fps)**.

— **Media bus cycles per frame**—this represents the average number of cycles between frames.

For ATM streams, a cell is considered to be a frame. For each port, the Workbench starts counting cycles for the receive rate calculation when the port asserts start-of-packet (SOP) for the first packet received by the Network Processor after the user starts debugging or resets statistics. For the transmit rate calculation, cycle counting starts when the port gets start-of-packet (SOP) for the first packet transmitted by the Network Processor after the user starts debugging or resets statistics.

- By selecting **Update status window every *xxx* microengine cycles**, the Workbench updates the status at the specified interval while the simulation is running. By default, the Workbench updates the data displayed in the window only when simulation stops.
- **Use this seed for random selection ... Previous value xxxxxxxxxx** . You can force the random selection generator to use the number used on the last simulation, or you can type a new number. The number displayed is the number last used. To keep the same seed value, click the box.

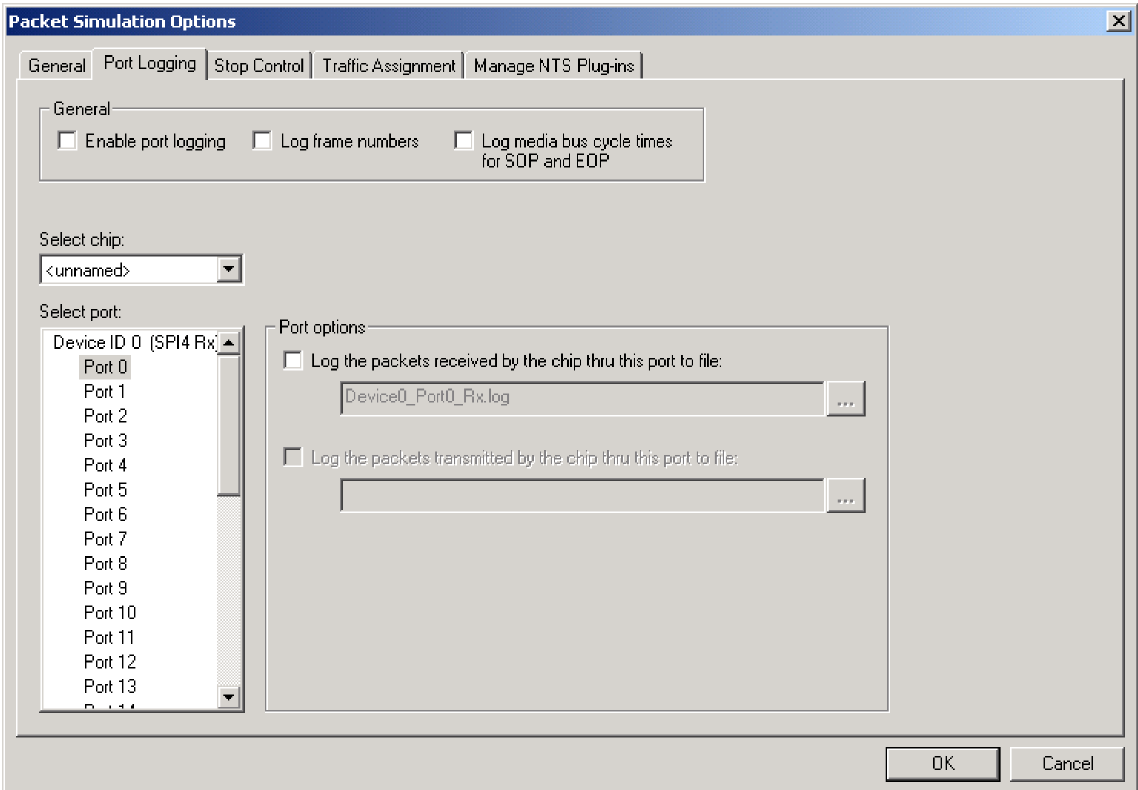
When you have completed specifying all your options, click **OK** to apply your choices and dismiss the dialog box or select another tab.

2.10.2 Port Logging

In the **Packet Simulation Options** property sheet, click the **Port Logging** tab in order to specify whether and how the logging of received and transmitted packets is to occur (see Figure 19).

Logging is done on a per-port basis with receive and transmit logs being written to separate files. Only complete packets are logged. This means that if you enable logging during simulation, logging for a port will start when the next SOP occurs. Similarly, if you disable logging, any packet that is in the process of being received or transmitted will not get logged. Packet simulation log files are closed automatically when simulation stops. The log file is opened and appended to when simulation resumes. Existing log files are automatically cleared when the first packet is logged after debugging is started.

Figure 19. Packet Simulation Options Dialog Box - Port Logging



If there is more than one chip in your system configuration, select the chip for which you want to specify port logging:

There are three general options available:

- Select or clear **Enable port Logging** to toggle whether packet logging occurs or not. This is a global setting which determines whether the individual port logging settings are in effect.
- Select or clear **Log frame numbers** to toggle whether or not frame numbers are logged along with the packet data. If enabled, the frame number appears as the first item on a line. Frame numbering starts when debugging is started, with the first packet received on a port and first packet transmitted to a port being number 1. The numbers continue to increment regardless of whether logging is enabled or not. So if you enable logging, disable it, then enable it again you will see a gap in the logged frame numbers.
- Select or clear **Log media bus cycle times for SOP and EOP** to toggle whether or not to log the cycle times at which the first byte and last byte of a packet are received or transmitted. If enabled, the cycles times appear before the packet data on the line but after the frame number, if **Log frame numbers** is also selected.

  If logging of both frame numbers and cycle times are enabled, the logged data looks like:

  25 4387 4395 01010101010202020202...

To specify port-specific options, select a port from the list of devices and ports. The options are:

- If you want to get a log of packets received by the Network Processor to a port, select the port and then select **Log packets received by the chip from this port to file:** and type a file path in the box. You can browse to a file by clicking the ... button to the right of the box. The packet data is written to the file in hexadecimal format with one packet per line.
- If you want to get a log of packets transmitted by the Network Processor to a port, select, select the port and then select **Log packets transmitted by the chip to this port to file:** and type a file path in the box. You can browse to a file by clicking the ... button to the right of the box. The packet data is written to the file in hexadecimal format with one packet per line.

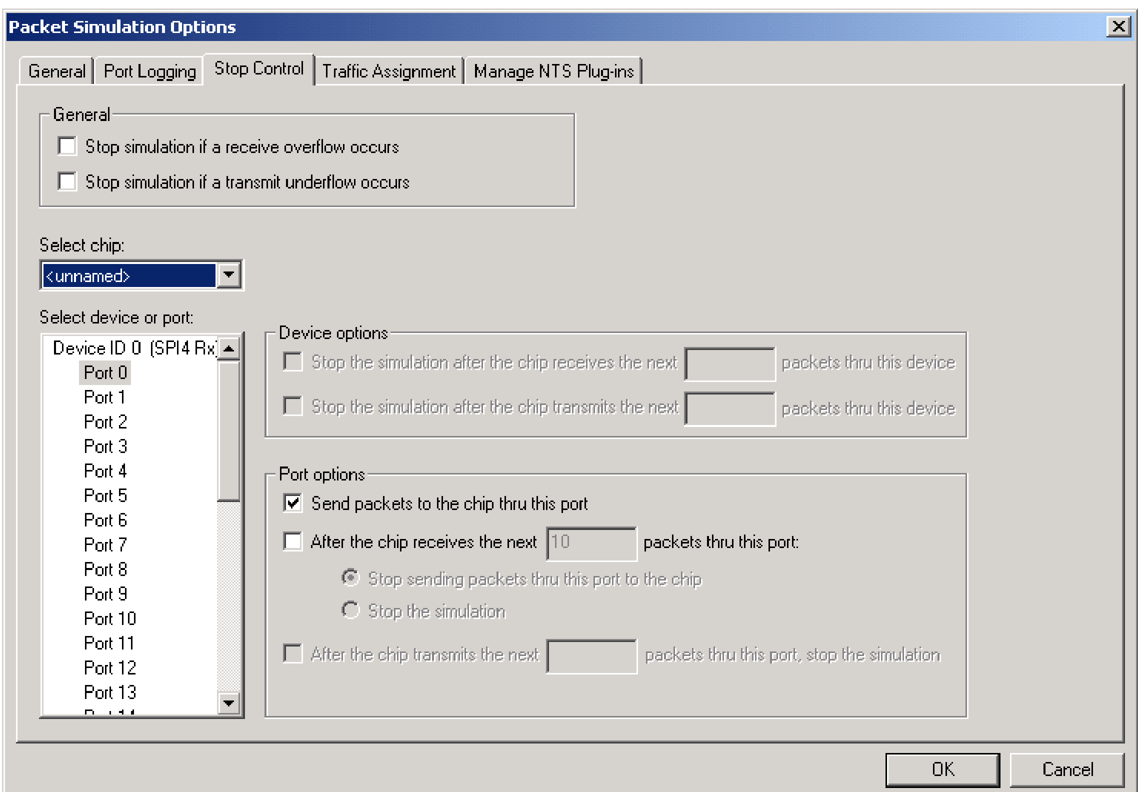
When you have completed specifying all your options, click **OK** to apply your choices and dismiss the dialog box or select another tab.

2.10.3 Stop Control

In the **Packet Simulation Options** property sheet, click the **Stop Control** tab in order to specify whether and when you want simulation or packet reception to stop (see Figure 20).

If there is more than one chip in your system configuration, select the chip for which you want to specify stop control.

Figure 20. Packet Simulation Options - Stop Control Tab



There are two general options. The options are:

- Enable **Stop simulation if a receive overflow occurs** to control whether or not the Workbench stops the simulation when a receive overflow occurs.
- Enable **Stop simulation if a transmit underflow occurs** to control whether or not the Workbench stops the simulation when a transmit underflow occurs.

To specify device-specific options, select a device from the list of devices and ports. The options are:

- If you want to stop the simulation after a specific number of packets are received by the Network Processor from the selected chip, select **Stop the simulation after the chip receives the next *nnn* packets from this device** and type the number of packets in the box.

  When the specified number of packets are received, the simulation stops and a message box is displayed. If you continue the simulation from that point, it will again stop after the next nnn packets are received.
- If you want to stop the simulation after a specific number of packets are transmitted by the Network Processor to the selected device, select **Stop the simulation after the chip transmits the next *nnn* packets to the device** and type the number of packets in the box.

To specify port-specific options, select a port from the list of devices and ports. The options are:

- If you want to enable the port to receive packets from the network, select **Send packets to the chip from this port**. Ports are always enabled to accept packets transmitted by the Network Processor.
- If you want to take action after a specified number of packets are received by the Network Processor from the port, select **After the chip receives the next *nnn* packets from this port:**, type the number of packets in the box, then click **Stop sending packets from this port to the chip** or click **Stop the simulation**.
- If you want to take action after a specified number of packets are transmitted from the Network Processor to the port, select **After the chip transmits the next *nnn* packets to this port:**, click **Stop the simulation** and type the number of packets in the box.

When you have completed specifying all your options, click **OK** to apply your choices and dismiss the dialog box or select another tab.

2.10.4 Traffic Assignment

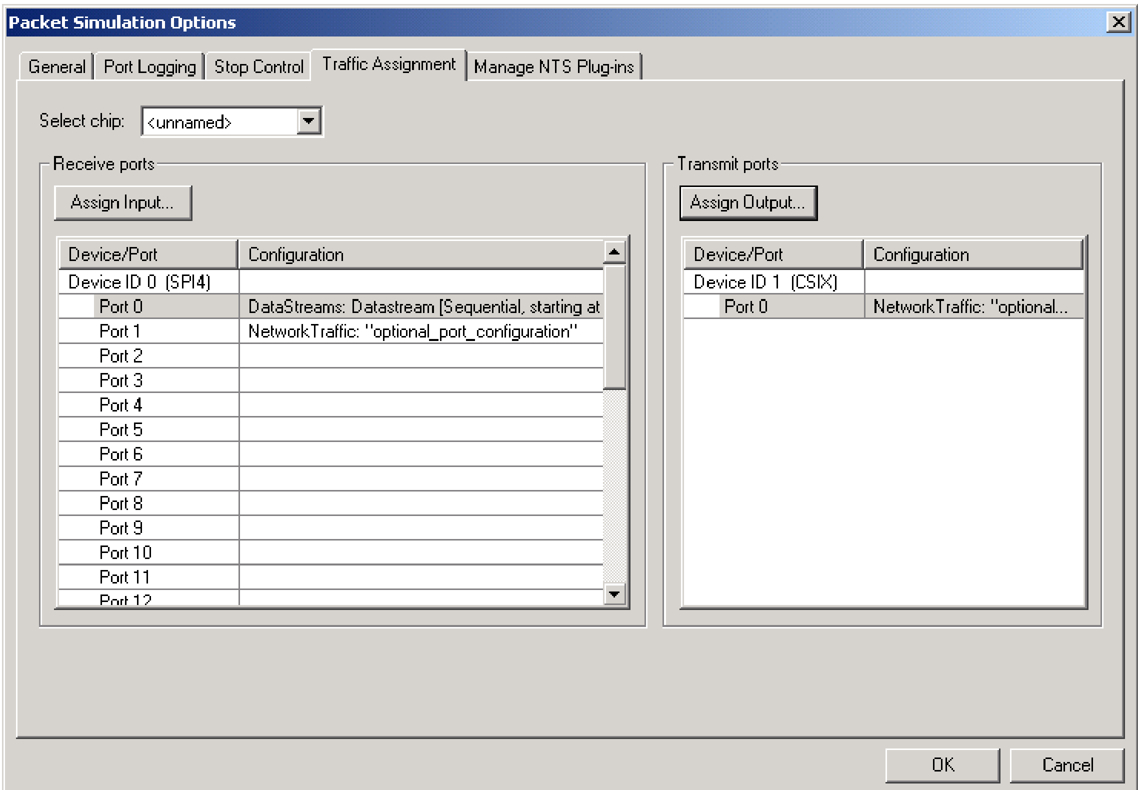
To assign traffic (formerly called Port Connections) select **Packet Simulation Options** and then select the **Traffic Assignment** tab. The property page shown in Figure 21 appears.

Receive side port connections:

If you connect a device to the receive side of a chip, you can specify the data to be received by each port in that device. A port is connected to the network processor on one side and to the 'network' on the other. Data comes into the port from the network and is placed into the port's receive buffer. To effectively simulate a port's operation, network traffic must also be simulated. Input from the network can either be simulated by the Workbench using data from streams or you can provide a Network Traffic DLL that supplies the input data.

In the receive ports list, the Workbench displays the input that is assigned to each port configured for the connected device. It shows either the name of the DLL that is to supply input data to the port or the data streams assigned to supply input data. In the latter case, the method by which packets are selected from the streams is shown in brackets after the stream names. If no input is assigned, the area is blank.

Figure 21. The Traffic Assignment Property Page

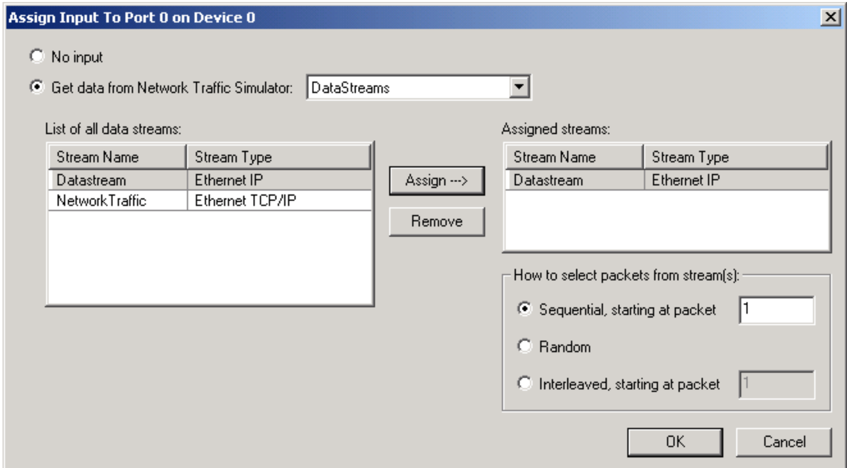


To assign the port input:

1. Select the port in the **Receive Ports** area of the **Traffic Assignment** property page.
2. Click the **Assign Input...** button.

   The **Assign Input to Port** dialog box appears (see Figure 22).

Figure 22. Assign Input to Port



Select one of the radio buttons to set how you want to supply input data for the port's receive buffer:

No input If you don't want the port to receive any data.

Get data from Network Traffic Simulator

If you want the Workbench to get data from datastreams or your network traffic simulator DLL.

If Datastreams is selected:

- All data streams associated with the project along with their type are displayed in the **List of all data streams** list box.
- The streams that are assigned to the port are displayed in the list box labeled Assigned streams.

If a user-defined Network Traffic Simulator is selected, then an optional configuration string may be supplied.

To assign a datastream to the port:

1. Select the stream in the **List of all data streams** list box.
2. Click **Assign**.
   The selected stream is added at the end of the assigned streams list. If a stream is already assigned, it is not assigned again.

To deassign a stream:

1. Select the stream in the **Assigned streams** list.
2. Click **Remove**.

In the **How to select...** area at the bottom right of the dialog box are controls that allow you to specify the method by which packets are selected from the assigned streams to be received by the port. When multiple streams are assigned, the Workbench treats them as one continuous sequence of packets.

Click **Sequential, Starting at packet**

If you want the packets to be selected sequentially from the stream(s). You can specify which packet you want to be the first packet received by the port. After the port receives the last packet in the last assigned stream, the Workbench wraps to the first packet in the first assigned stream.

Click **Random**

If you want packets to be selected at random from the assigned streams. For ATM streams, the PDUs are selected at random but the ordering of cells within the PDU is always maintained. For example, assume a stream has five PDUs. If PDU#2 is selected, its first cell will then be placed into the receive buffer. The next time PDU#2 is selected, its second cell is placed in the buffer, and so on, until all cells in the PDU are selected. Then the first cell is selected again

Click **Interleaved, starting at packet**

If you want interleaved cell selection for ATM streams. PDUs are selected sequentially but only one cell in a PDU is selected at a time. For example, assume an ATM stream has three PDUs, with PDU#1 having one cell, PDU#2 having three cells and PDU#3 having two cells.

The packet selection sequence will be:

PDU#1 Cell#1

PDU#2 Cell#1

PDU#3 Cell#1

PDU#1 Cell#1

PDU#2 Cell#2

PDU#3 Cell#2

PDU#1 Cell#1

PDU#2 Cell#3

PDU#3 Cell#1

For non-ATM streams, the sequential and interleaved choices are identical.

When you have completed assigning streams and specifying the packet selection method, click **OK** to apply your choices and return to the **Traffic Assignment** dialog box.

Transmit side port connections:

If you connect a device to the transmit side of a chip, you can specify what is to be done with the data sent to the network by each port in that device. Output to the network can be thrown away or you can provide a Network Traffic Simulator DLL that receives the transmitted data.

In the transmit ports list, the Workbench shows the name of the DLL that is assigned to receive output from the port. If no DLL is assigned, the area is blank.

To assign port output:

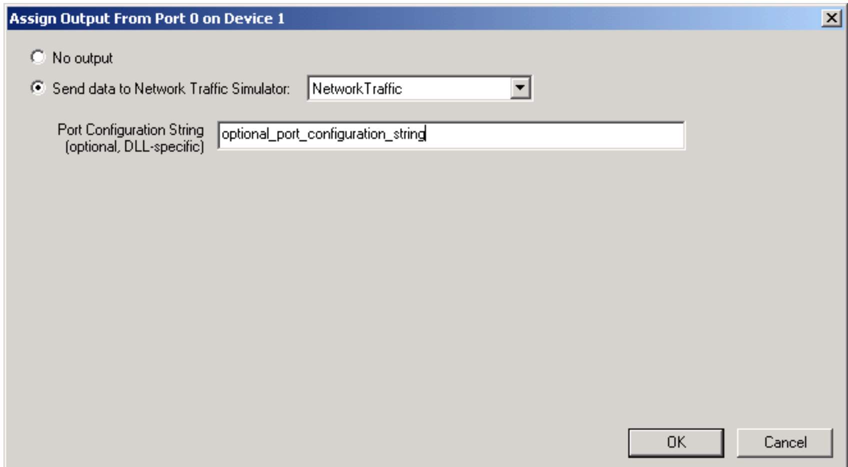
1. Click the port in the **Transmit Ports** area of the **Traffic Assignment** property page.
2. Click the **Assign Output...** button.

   The **Assign Output from Port** dialog box appears (see Figure 23).

Select how you want to process data taken out of the port's transmit buffer:

- **No output** - If you don't want to process it.
- **Send data to Network Traffic Simulator** - If you want the data passed to your network traffic simulator.

Figure 23. Assign Output from Port



2.10.5 Manage NTS Plug-ins

The **Manage NTS Plug-ins** tab is for managing Network Traffic Simulation plug-ins (see Figure 24). You can create, edit, or delete a Network Traffic Simulator plug-in. A Network Traffic Simulator consists of a unique name and a DLL file name for sending and/or receiving port traffic.

On the **Simulation** menu, click **Packet Simulation**, then click the **Manage NTS Plug-ins** tab and the **Manage NTS Plug-ins** property page appears. This page contains the:

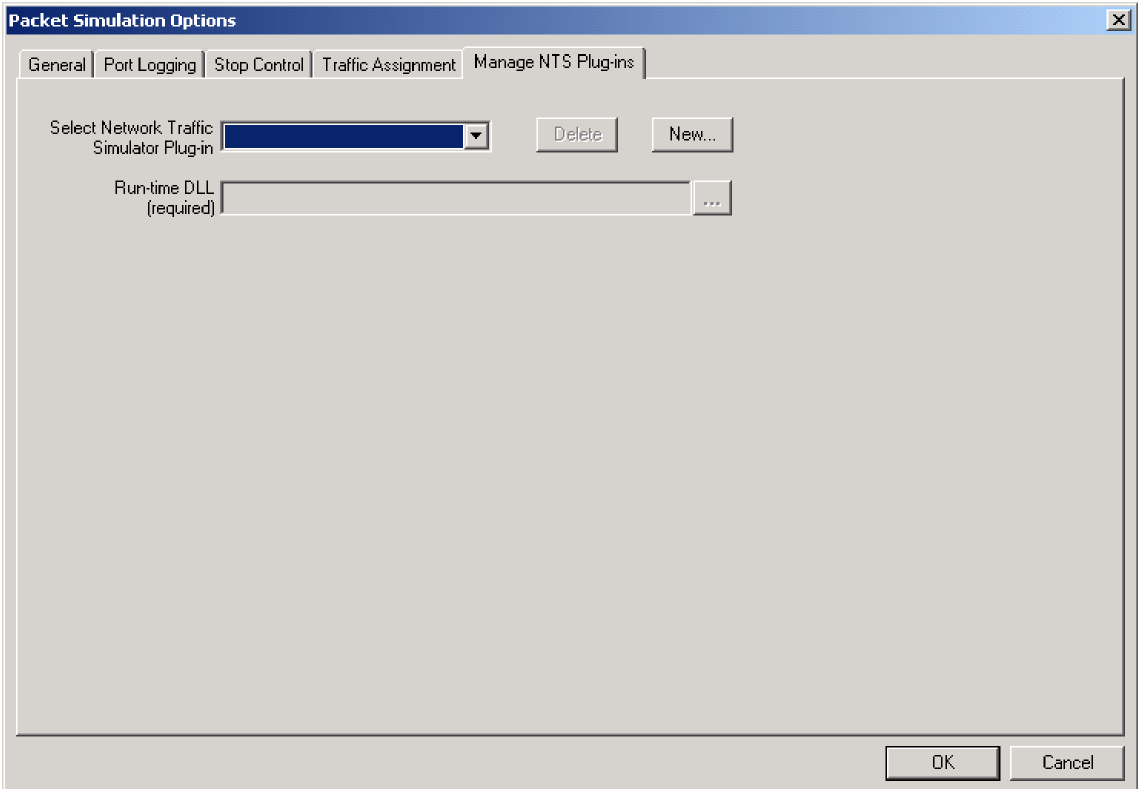
— **Select Network Traffic Simulator Plug-in**

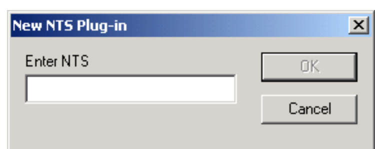
A pulldown box that lists all know plug-ins, a **Delete** button for deleting an existing plug-in, and an associated **New...** button that allows the user to add new plug-ins.

— **Run time DLL (required)**

The filename of the NTS Run-time DLL.

Figure 24. Manage NTS Plug-ins Property Page





1. To add a new Network Traffic Simulator Plug-in click the **New**... button and the **Manage NTS Plug-in** pop up appears. Specify the name and click **OK**. The name then appears in the **Select Network Traffic DLL** pulldown box.
2. To specify the **Run Time DLL**, enter the path. or click the  button to Browse for the filename.

The Network Traffic Simulator has now been plugged into the Workbench project and is available for selection on the Traffic Assignment page.

2.10.5.1 Network Traffic Simulation DLLs

To simulate supplying data to a port or taking data from the port, you can provide a dynamic-link library (DLL) called a Network Traffic Simulator Run-time DLL. You assign this DLL to the input and/or output side of a media bus device port (see Section 2.10.5).

A network traffic simulation (NTS) Run-time DLL must provide the following functions:

```
Initialize
Close
Reset
```

If the DLL is assigned to supply data to a port, then it must also have the following functions:

```
InitializeRxPort
GetNextByte
GetInterpacketTime
GetReturnStatus
CloseRxPort
```

If the DLL is assigned to take data from a port, then it must also have the following functions:

```
InitializeTxPort
SendNextByte
CloseTxPort
```

If the DLL supports a port configuration string, then it must also have the following functions:

```
ConfigureRxPort
ConfigureTxPort
```

When you **Start Debugging**, if an NTS DLL is assigned to a port and the NTS DLL is not already loaded due to a previous assignment to another port, the PacketSim DLL loads the NTS DLL and calls its `Initialize()` function. This is the only time that this function is called. In the `Initialize()` function the NTS DLL can register console functions with the Transactor. You can then call these functions from a script in order to configure your traffic simulation.

When the device model is connected to a chip model when debug is started, the NTS DLL is called to initialize each port. The PacketSim DLL iterates through all the ports on every media bus device. If the NTS DLL is connected to the receive side of the port, the function `InitializeRxPort()` is called. If the NTS DLL is connected to the transmit side of the port, the function `InitializeTxPort()` is called. The `ConfigureRxPort` and `ConfigureTxPort` functions are called, if provided, to send the port configuration string into the NTS DLL.

Note: See the file `PortConfigData.h` for the contents of the port configuration data structures that are passed to these initialization functions.

As the simulation progresses, the PacketSim DLL calls function `GetNextByte()` whenever a byte of data is required on a receive port. In addition to the byte of data, the NTS DLL must also return a flag indicating whether the byte is the last byte (EOP) in the frame/cell. When the PacketSim DLL receives the last byte, it will call the `GetReturnStatus()` function if you have provided one. It also calls the `GetInterpacketTime()` function in order to determine how long it waits before asking for the first byte of the next frame/cell. This allows the DLL to randomize the arrival of frames/cells.

On the transmit side, the PacketSim DLL calls the function `SendNextByte()` when it takes a byte of data out of the transmit buffer to be sent over the network. An EOP flag is asserted along with the last byte in the frame/cell.

When the user presses **Stop Debugging** in the Workbench, or if the `sim_reset` command is executed directly, the functions `CloseRxPort()` and `CloseTxPort()` are called for each connected receive and transmit port, respectively. The `Reset()` function is also called.

When the user closes the project or exits the Workbench, the function `Close()` is called just before the NTS DLL is freed.

A Visual C++ project which is an example of a network traffic simulation DLL can be found at:

`...\IXA_SDK_3.1\me_tools\Samples\NetworkTraffic.`

2.11 Data Streams

Data streams are used to simulate network traffic. To create and edit data streams:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears (see Figure 25).
2. Click the **Create Stream ...** button.
   The **Create Stream** pop-up appears (see Figure 26).

You can create and edit the following data streams (see the following):

POS IP	(see Section 2.11.1)
ATM AAL5	(see Section 2.11.2).
Custom Ethernet IP	(see Section 2.11.3).
Ethernet IP	(see Section 2.11.4).
Ethernet TCP/IP	(see Section 2.11.5).
PPP TCP/IP	(see Section 2.11.6

Figure 25. Define Network Traffic - Data Stream Dialog Box

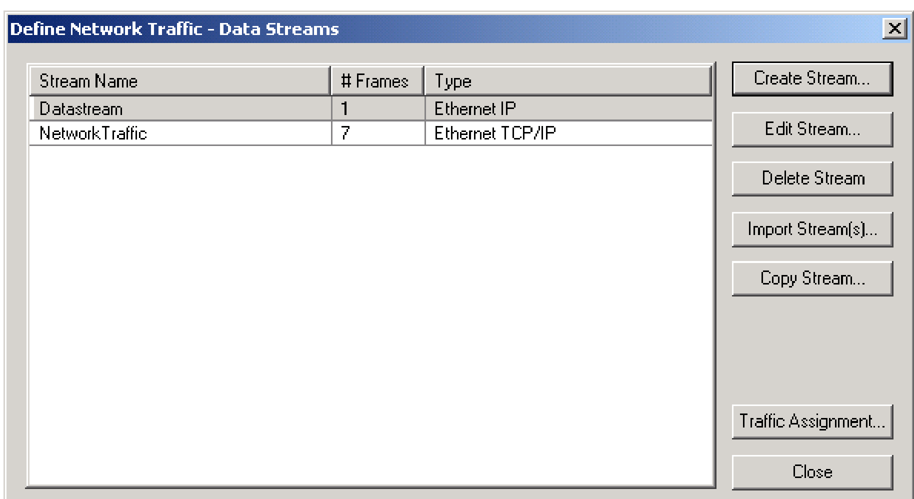
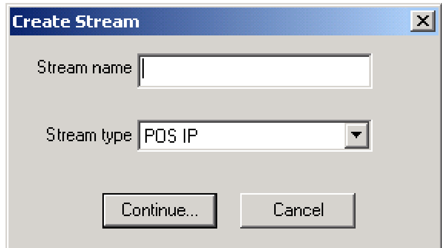


Figure 26. Create Stream Pop-up



Data Stream Deletion:

To delete a data stream:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears (see Figure 25).
2. Click the data stream that you want to delete.
3. Click **Delete Stream**.

Note: The Workbench only deletes the data stream from the project, not from the folder. If you delete in error, click Import Stream(s) to retrieve it.

Data Stream Import:

To import a data stream from a previously saved file:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Click **Import Stream**(s).
   The **Import Stream** dialog box appears.
3. Browse to the desired folder and select one or more stream files (.strm).
4. Click **OK** to import the selected files.

Data Stream Copy:

Copying a data stream then editing the copy gives you a quick way to create a new data stream that is similar to an existing data stream.

To copy an existing data stream:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Click the data stream that you want to copy.
3. Click **Copy Stream**.
   The **Stream Name** dialog box appears.
4. Type the name of the new data stream.
5. Click **OK**.

The **Specify file...** dialog box appears.

6. Select the folder where you want to save the stream.
7. Click **OK**.

   The stream appears (or reappears if you did not change the name) in the **Data Streams** dialog box. It has all the characteristics of the stream copied.

Media Bus Device Port Assignment:

To assign data streams to configured Media Bus device ports:

1. On the **Simulation** menu, click **Data Streams...**.

   The **Data Streams** dialog box appears.
2. Click the data stream that you want to assign.
3. Click **Port Connections...**

   The **Connections** tab of the **Packet Simulation Configuration** dialog box appears.

   See Section 2.10 for more detailed information.

2.11.1 Creating and Editing a POS IP Data Stream

Create:

1. On the **Simulation** menu, click **Data Streams...**.

   The **Data Streams** dialog box appears.
2. Click **Create Stream**.

   The **Create Stream** dialog box appears.
3. Type the name of the stream in the **Stream name** box.
4. Select **POS IP** from the **Stream type** list.
5. Click **Continue**.

   The **Custom Header Size** dialog box appears.
6. Type the number of bytes to be in the custom header.
7. Click **OK**.

   The **POS IP** dialog box appears.
8. Type a new name in the **Stream name** box if you want to change it.

To create one or more frames:

1. Click **Create Frame(s)**.
2. Click **Custom Header**.
3. Type the data for the custom header in the **Custom header** box.
4. Click **IP Header** (go to Section 2.11.9).
5. Click **Data Payload** (go to Section 2.11.11).
6. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.12).
7. Type the number of frames you want to create in the **Number of new...** box.

8. Click **Create**.

   The number of frames you specified are created and added to the data stream. The dialog box remains active so you can change settings and create additional frames.
9. When you are finished creating frames click **Close**.
10. When you are finished creating the data stream, click **OK**.
    The **Save** dialog box appears.
11. Type in the file name if you want to change it.
12. Browse to the folder where you want to save the file.
13. Click **Save**.
14. In the **Data Streams** dialog box, click **OK.**

Edit:

To edit a POS IP data stream:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Select stream that you want to edit.
3. Click **Edit Stream**.
   The **POS IP Data Stream** dialog box appears.
4. Click **Create Frame(s)** to create a new frame.
5. Click **Edit Frame(s)** to edit the:
   — PPP Header
   — IP Header (go to Section 2.11.9)
   — **Data Payload** (go to Section 2.11.11)
6. Click **Delete Frame** to delete the selected frame.
7. Click the **Up** and **Down** arrows to change the order of the frames.
8. Click **OK** when done.

2.11.2 Creating and Editing an ATM Data Stream

An ATM Protocol Data Unit (PDU) comprises the unsegmented components of ATM data:

- The ATM Header
- An AAL5 trailer
- An optional LLC/SNAP header
- An IP packet payload

Creation:

To create an ATM stream:

1. On the **Simulation** menu, click **Data Streams...**.

The **Data Streams** dialog box appears.

2. Click **Create Stream...**.

   The **Create Steam** dialog box appears.

3. Type the name of the ATM stream in the **Stream name** box.
4. Select **ATM** from the **Stream type** list.
5. Click **Continue**.

   The **ATM Stream** dialog box appears.

To create a PDU:

1. Click **Create PDU(s)**.

   The **Create AAL5 PDU** dialog box appears.

2. Type the values you want for the **ATM header**.

   This header is prepended to each segmented ATM cell. If you select **Automatic** for **PTI**, then the box will contain zero for all cells except for the last one, in which the box will contain a one.

3. For **RFC1483 options**, you can select **LLC/SNAP**, which prepends a header to the packet data, or **VCMUX**, which does not prepend a header. The optional header plus the packet data constitute a CS-DSU information box. Currently, an AAL5 trailer is always appended to the CS-DSU information box.
4. If you want to encapsulate a single IP packet, click the **Single Packet** option.
   - Click **IP Header** to specify the fields of the IP header (see Section 2.11.9),
   - Click **Data Payload** to specify the data payload for the IP packet (see Section 2.11.11).
5. If you want to encapsulate a pool of packets, click the **Multiple packets from pool** option. Select a packet pool from the list of available pools.
6. To import a previously created packet pool, click **Import Pool**.
7. To create a new pool, click **Create Pool**.

   The **IP Packet Pool** dialog box appears. Go to Section 2.11.7 to create the IP packet pool.

8. Click **Create Packet(s)** to create a PDU(s) for each packet in the selected pool.
9. Click **IP Header** to specify IP Header information (see Section 2.11.9).

   The created PDUs are added to the ATM data stream. The dialog box remains active so you can change settings and create additional PDUs.

10. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.12).
11. When you are finished creating PDUs, click **Close**.
12. When you are finished creating the data stream, click **OK**.

Edit:

To edit an ATM stream:

1. On the **Simulation** menu, click **Data Streams...**.

   The **Data Streams** dialog box appears.

2. Select an ATM stream that you want to edit.

3. Click **Edit Stream**.
   The **ATM Stream** dialog box appears.
4. Here you can:
   — **Edit PDUs** (similar to creating–see previous section).
   — Delete a PDU by selecting the PDU and clicking **Delete PDU**.
   — Change the order of the PDUs using the Up or Down arrows.
5. Click **OK** when done.

2.11.3 Creating and Editing a Custom Ethernet TCP/IP Data Stream

Create:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Click **Create Stream**.
   The **Create Stream** dialog box appears.
3. Type the name of the stream in the **Stream name** box.
4. Select **Custom Ethernet TCP/IP** from the **Stream type** list.
5. Click **Continue**.
   The **Custom Header Size** dialog box appears.
6. Type the number of bytes to be in the custom header.
7. Click **OK**.
   The **Custom Ethernet TCP/IP Data Stream** dialog box appears.
8. Type a new name in the **Stream name** box if you want to change it.

To create one or more frames:

1. Click **Create Frame(s)**.
2. Click **Custom Header**.
3. Type the data for the custom header in the **Custom header** box.
4. Click **Ethernet Header** (go to Section 2.11.8).
5. Click **IP Header** (go to Section 2.11.9).
6. Click **Data Payload** (go to Section 2.11.11).
7. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.12).
8. Type the number of frames you want to create in the **Number of new...** box.
9. Click **Create**.

   The number of frames you specified are created and added to the data stream. The dialog box remains active so you can change settings and create additional frames.
10. When you are finished creating frames click **Close**.

11. When you are finished creating the data stream, click **OK**.
    The **Save** dialog box appears.
12. Type in the file name if you want to change it.
13. Browse to the folder where you want to save the file.
14. Click **Save**.
15. In the **Data Streams** dialog box, click **OK.**

Edit:

To edit a Custom Ethernet TCP/IP data stream:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Select stream that you want to edit.
3. Click **Edit Stream**.
   The **Custom Ethernet TCP/IP Data Stream** dialog box appears.
4. Click **Create Frame(s)** to create a new frame.
5. Click **Edit Frame(s)** to edit the:
   — Custom Header
   — Ethernet Header (go to Section 2.11.8)
   — IP Header (go to Section 2.11.9)
   — Data Payload (go to Section 2.11.11)
6. Click **Delete Frame** to delete the selected frame.
7. Click the **Up** and **Down** arrows to change the order of the frames.
8. Click **OK** when done.

2.11.4 Creating and Editing an Ethernet IP Data Stream

Create:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Click **Create Stream**.
   The **Create Stream** dialog box appears.
3. Type the name of the stream in the **Stream name** box.
4. Select **Ethernet IP** from the **Stream type** list.
5. Click **Continue**.
   The **Ethernet IP Data Stream** dialog box appears.
6. Type a new name in the **Stream name** box if you want to change it.

To create one or more frames:

1. Click **Create Frame(s)**.

2. Click **Ethernet Header** (go to Section 2.11.8).
3. Click **IP Header** (go to Section 2.11.9).
4. Click **Data Payload** (go to Section 2.11.11).
5. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.12).
6. Type the number of frames you want to create in the **Number of new...** box.
7. Click **Create**.

   The number of frames you specified are created and added to the data stream. The dialog box remains active so you can change settings and create additional frames.
8. When you are finished creating frames click **Close**.
9. When you are finished creating the data stream, click **OK**.
   The **Save** dialog box appears.
10. Type in the file name if you want to change it.
11. Browse to the folder where you want to save the file.
12. Click **Save**.
13. In the **Data Streams** dialog box, click **OK.**

Edit:

To edit an Ethernet IP data stream:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Select stream that you want to edit.
3. Click **Edit Stream**.
   The **Ethernet IP Data Stream** dialog box appears.
4. Click **Create Frame(s)** to create a new frame.
5. Click **Edit Frame(s)** to edit the:
   — **Ethernet Header** (go to Section 2.11.8)
   — **IP Header** (see Section 2.11.9)
   — **Data Payload** (go to Section 2.11.11)
6. Click **Delete Frame** to delete the selected frame.
7. Click the Up and Down arrows to change the order of the frames.
8. Click **OK** when done.

Note: The Workbench puts a four byte CRC value at the end of each packet which is included in the byte count.

2.11.5 Creating and Editing an Ethernet TCP/IP Data Stream

Create:

1. On the **Simulation** menu, click **Data Streams...**.

The **Data Streams** dialog box appears.

2. Click **Create Stream**.
   The **Create Stream** dialog box appears.
3. Type the name of the stream in the **Stream name** box.
4. Select **Ethernet TCP/IP** from the **Stream type** list.
5. Click **Continue**.
   The **Ethernet TCP/IP Data Stream** dialog box appears.
6. Type a new name in the **Stream name** box if you want to change it.

To create one or more frames:

1. Click **Create Frame(s)**.
2. Click **Ethernet Header** (go to Section 2.11.8).
3. Click **IP Header** (go to Section 2.11.9).
4. Click **TCP Header** (go to Section 2.11.10).
5. Click **Data Payload** (go to Section 2.11.11).
6. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.12).
7. Type the number of frames you want to create in the **Number of new...** box.
8. Click **Create**.
   The number of frames you specified are created and added to the data stream. The dialog box remains active so you can change settings and create additional frames.
9. When you are finished creating frames click **Close**.
10. When you are finished creating the data stream, click **OK**.
    The **Save** dialog box appears.
11. Type in the file name if you want to change it.
12. Browse to the folder where you want to save the file.
13. Click **Save**.
14. In the **Data Streams** dialog box, click **OK**.

Edit:

To edit an Ethernet TCP/IP data stream:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Select stream that you want to edit.
3. Click **Edit Stream**.
   The **Ethernet TCP/IP data Stream** dialog box appears.
4. Click **Create Frame(s)** to create a new frame.
5. Click **Edit Frame(s)** to edit the:
   — **Ethernet Header** (go to Section 2.11.8)

— **TCP Header** (go to Section 2.11.10)

— **IP Header** (go to Section 2.11.9)

— **Data Payload** (go to Section 2.11.11)

6. Click **Delete Frame** to delete the selected frame.
7. Click the Up and Down arrows to change the order of the frames.
8. Click **OK** when done.

2.11.6 Creating and Editing a PPP TCP/IP Data Stream

Create:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Click **Create Stream**.
   The **Create Stream** dialog box appears.
3. Type the name of the stream in the **Stream name** box.
4. Select **PPP TCP/IP** from the **Stream type** list.
5. Click **Continue**.
   The **PPP TCP/IP Data Stream** dialog box appears.
6. Type a new name in the **Stream name** box if you want to change it.

To create one or more frames:

1. Click **Create Frame(s)**.
2. Click **PPP Header**. Type the **Protocol** value and click either **8 bit protocol** or **16 bit protocol**. Enable **Include Address** and **Include Control** fields
3. Click **IP Header** (go to Section 2.11.9).
4. Click **TCP Header** (go to Section 2.11.10).
5. Click **Data Payload** (go to Section 2.11.11).
6. Click **PPP Trailer**
7. Specify the frame size in the **Frame size (in bytes)** area (see Section 2.11.12).
8. Type the number of frames you want to create in the **Number of new...** box.
9. Click **Create**.
   The number of frames you specified are created and added to the data stream. The dialog box remains active so you can change settings and create additional frames.
10. When you are finished creating frames click **Close**.
11. When you are finished creating the data stream, click **OK**.
    The **Save** dialog box appears.
12. Type in the file name if you want to change it.
13. Browse to the folder where you want to save the file.
14. Click **Save**.

15. In the **Data Streams** dialog box, click **OK**.

Edit:

To edit a PPP TCP/IP data stream:

1. On the **Simulation** menu, click **Data Streams...**.
   The **Data Streams** dialog box appears.
2. Select stream that you want to edit.
3. Click **Edit Stream**.
   The **PPP TCP/IP Data Stream** dialog box appears.
4. Click **Create Frame(s)** to create a new frame.
5. Click **Edit Frame(s)** to edit the:
   — PPP Header
   — TCP Header (go to Section 2.11.10)
   — IP Header (go to Section 2.11.9)
   — Data Payload (go to Section 2.11.11)
   — PPP Trailer
6. Click **Delete Frame** to delete the selected frame.
7. Click the Up and Down arrows to change the order of the frames.
8. Click **OK** when done.

2.11.7 Creating an IP Packet Pool

In the ATM data stream, you can create IP Packet Pools. Do the following:

1. Create an ATM data stream.
2. Create a PDU.
3. In the **Create AAL5 PDU** dialog box, click **Multiple packets from pool** in the lower-left corner.
4. Click **Create Pool**.
   The **IP Packet Pool** dialog box appears.
5. Click **Create Packet(s)**.
   The **IP Packet dialog** box appears.
6. Click **IP Header** (see Section 2.11.9).
7. Click **Payload** (see Section 2.11.11).
   You can specify a fixed packet size or, if you are creating multiple packets, a size which is randomly selected from within a specified range or which is incremented within a specified range.
8. Specify the number of packets you want to create
9. Click **Create**.

The created packets are added to the pool. The dialog box remains active so you can change settings and create additional packets.

10. When you are finished creating packets, click **Close**.

To delete a packet:

1. Click the packet you want to delete.
2. Click **Delete Packet**.

To edit a packet:

1. Click the packet you want to edit.
2. Click **Edit Packet**.

To change the order of the packets:

1. Click a packet.
2. Click the up or down arrow buttons to move the frame up or down in the list.

When you are finished creating the packet pool:

1. Click **OK**.
   The **Specify File...** dialog box appears.
2. Browse to the folder where you want to store the file.
3. Type the name of the file in the **File Name** box.
4. Click **Save**.

The packet pool that you just created appears in the **Select packet pool...** box.

5. Click on the name of the pool.
6. Click **Create**.
7. If done, click **Close**.
   The **ATM Stream dialog** box appears.
8. Click **OK**.

2.11.8 Specifying an Ethernet Header

1. Create an Ethernet IP data stream (see Section 2.11.4) or an Ethernet TCP/IP data stream (see Section 2.11.5).
2. In the **Ethernet IP Data Stream** dialog box, click **Create Frames**.
3. Click **Ethernet Header**.
4. Enter values directly into the boxes.
   If you are creating multiple frames and want each frame to have a different value for the destination or source MAC address, click **Advanced** next to the appropriate address box.
   The **Specify how you want...** dialog box appears. It displays options for generating MAC addresses.
   — Click **Fixed** if you want all frames to have the same address.

- Click **From within range** if you want addresses chosen from a range which you specify.
- Click **Sequential** to have addresses chosen sequentially from within the range, starting within the range's lower bound.
- Click **Random** for random selection of addresses from within the range.
- Click **From list** if you want addresses chosen from a list that you specify.
- Click **Sequential** to have addresses chosen sequentially from the list, starting within the first address in the list.
- Click **Random** for random selection of addresses from the list.
- To add an address to the list either click the button or double-click beneath the last address in the list, enter the address value and pressing ENTER.
- To delete an address from the list, select it, then click the button.
- To move an address up or down in the list, select it and click the or the button.

The list of addresses is saved in the Windows registry, so it is available during future Workbench sessions.

2.11.9 Specifying an IP Header

1. Create an Ethernet IP data stream (see Section 2.11.4) or an Ethernet TCP/IP data stream (see Section 2.11.5).
2. In the **Ethernet IP Data Stream** dialog box, click **Create Frames**.
   The **Create frame(s)** dialog box appears.
3. Click **IP Header**.
4. Enter values directly into the boxes.
   - If you want the packet length to be automatically computed based on the length of the encapsulated payload, select **Computed** next to the **Packet length** box. Otherwise, the value you enter will be used without modification.
   - If you want the header checksum to be automatically computed, select **Computed** next to the **Header checksum** box. Otherwise, the value you enter will be used without modification.
   - If you are creating multiple frames and want each frame to have a different value for the **Source IP address** or **Destination IP address**, click **Advanced** next to the corresponding address box.
     The **Specify how you want...** dialog appears.

   In the **Frame size (in bytes)** area:
   - Click **Single** if you want all frames to have the same address then specify the address.
   - Click **From within range** if you want addresses chosen from a range which you specify. To have addresses chosen sequentially from within the range, starting within the range's lower bound, click **Sequential**. For random selection of addresses from within the range, click **Random**.

- Click **From list** if you want addresses chosen from a list that you specify.
- To add an address to the list, enter the address in the box to the right of the list then click **Add Address**.
- To delete an address from the list, select it then click the  button.
- To move an address up or down in the list, select it and click the  or the  button.
- To have addresses chosen sequentially from the list, starting within the first address in the list, click **Sequential**.
- To have addresses chosen randomly from the list, click **Random**.

2.11.10 Specifying a TCP Header

1. Create an Ethernet TCP/IP data stream (see Section 2.11.5) or a PPP TCP/IP data stream (see Section 2.11.6).
2. In the **Ethernet IP Data Stream** dialog box, click **Create Frames**.
3. Click **TCP Header**.
4. Enter values directly into the boxes.
5. If you want the checksum to be automatically computed, select **Computed** next to the **Checksum** box. Otherwise, the value you enter will be used without modification.

2.11.11 Specifying a Data Payload

1. Create or edit a data stream of any type containing frames.
2. Click the **Data Payload** button to display the options for specifying the data payload for a frame.
3. Select a pattern from the **Fill pattern** list.
4. If you select an incrementing or decrementing pattern, you can specify the starting value for the fill operation in the **Hex starting value** box.
5. If you are creating multiple new frames you also have the option of having the incrementing or decrementing span the set of frames being created. For example, if the first frame is created with data 00 01 02... 4f, the second frame will have data 50 51 52..., and so on.
6. If you are editing an existing frame, you can choose to edit the data directly by clicking **Custom Data**, then editing the data fields within the box.

2.11.12 Specifying Frame Size

Specify a fixed frame size or, if you are creating multiple frames, a size which is randomly or incrementally selected from within a specified range.

To do this:

1. Create or edit a data stream of any type containing frames.

2. In the **Create Frame(s)** dialog box, go to the **Frame size (in bytes)** area and do one of the following:
   — Click **Fixed** and type the frame size in the **Fixed** box.
   — Click **Random** and type the **from** and **to** values in the boxes to the right.
   — Click **Increment** and type the **from** and **to** values in the boxes to the right.

2.12 Debugging

Using the Workbench, you can debug microcode either in **Simulation mode** or in **Hardware mode** (using the Development platform or compatible hardware).

When in Simulation mode, the Transactor provides debugging support to the Workbench. In Hardware mode, the Microengine Debug Library (debug_2000.a) running as part of an Intel® XScale™ core application program communicates with the Workbench and relays debugging operations between the Workbench and the Microengines. The application may either be one that is supplied with the Development Platform or one that is independently developed.

The Workbench menus and toolbar selections provide the following capabilities:

- Set breakpoints and control execution of the microcode.
- View source code on a per-thread basis.
- Display the status and history of Microengines, threads, and queues.
- View and set breakpoints on data, registers, and pins.

Some of the debugging operations are either disabled when debugging in Hardware mode or available in a limited fashion. The descriptions in the sections that follow include any limitations that apply in Hardware mode. Table 1 summarizes which debugging features are available in Hardware and Simulation modes.

Table 1. Simulation and Hardware Mode Features (Sheet 1 of 2)

Feature	Simulation	Hardware
System Configuration		
Starting and Stopping Debug	X	X
Command Line Interface	X	
Script Files	X	
Command Scripts	X	
Thread Windows		
• Display Microword Address	X	X
• Instruction Markers	X	X
• View Instructions	X	X
Run Control	X	X[1]
Breakpoints	X	X[1]
Examine Registers	X	X
Watch Data		

Table 1. Simulation and Hardware Mode Features (Sheet 2 of 2)

Feature	Simulation	Hardware
• Enter New Data Watch	X	X
• Watch CSRs and Pins (Pins are unavailable for PR-5)	X	X[1]
• Watch GPRs and XFER	X	X
• Deposit Data	X	X[1]
Watch Memory	X	X
• Break on Data Change	X	
Performance Statistics	X	
Execution Coverage	X	
Thread History	X	
Queue History	X	
Queue Status	X	
Thread Status	X	X
Packet Simulation Status	X	

NOTES:1.With restrictions.

Note: When debugging a mixed C and assembler microengine, the thread window and execution coverage windows can toggle between source file view and list file view. When the source file view is showing the user-written assembler code, the popup context menu for the thread window does not contain the “Set Data Watch for…” options, and datatips are not available. The user must toggle to list file view in order to establish data watches and perform datatips. This restriction is caused by the fact that the debug data generated for the user-written assembler code has no block scope data; hence the debug data register names are mangled to make them unique within the global scope

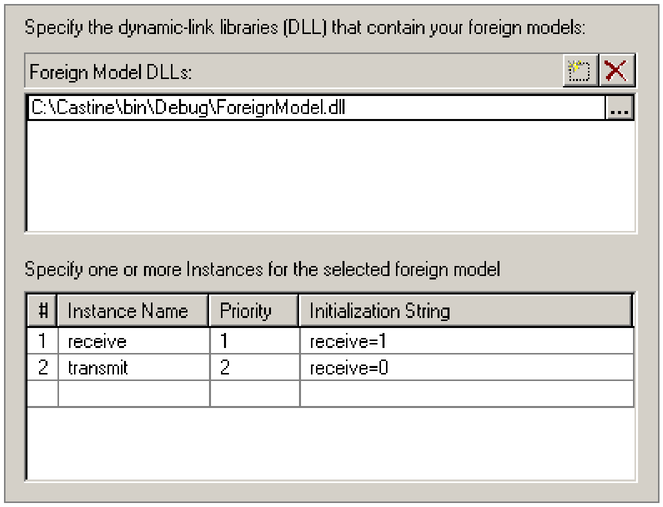
The Workbench supports debugging in four different configurations:

Mode	Foreign Model	IXP2400 and IXP2800	Comments
Local Simulation	None	Default. No special setup necessary.	The Workbench and the simulator (Transactor) both run on the Windows platform.
Local Simulation	Local	See Section 2.12.1	The Workbench, the Transactor and the foreign model Dynamic-Link Libraries all run on the same Windows platform.
Local Simulation	Remote	See Section 2.12.1	The Workbench and the Transactor both run on the same Windows platform and communicate over the network with a foreign model running on a remote system.
Hardware	None	N/A	The Workbench runs on a Windows host and communicates over a network or a serial port with a subsystem containing an actual network processor.

2.12.1 Local Simulation Debugging with a Local Foreign Model

The IXP2400 and IXP2800 Transactors support connecting multiple foreign model DLLs. The Workbench allows you to specify an unlimited list of DLL file paths for foreign model DLLs. For each DLL, you can specify an unlimited number of instantiations.

To specify the dynamic-link libraries that contain your foreign models:



1. On the **Simulation** menu, click **Options.**

   The **Simulation Options** dialog box appears.

2. Click the **Foreign Model** tab.

3. Click the  button to insert the path to your DLL.

4. Type in the complete path, or
   Click the  button to browse for the file.

When the path is set up, you must specify at least one instance.

5. Double-click the blank cell under **Instance Name**.

   Type the name of the first instance and press ENTER.The name distinguishes this instantiation when executing commands from the command line from an instantiation executing from a script.

6. Do the same for **Priority.**

   The priority can be any positive integer.

   The priority determines the order in which the Network Processor model calls the instantiations for the initialize, pre-simulation, post-simulation, and other callbacks. The instantiation with the highest priority number is called first, the next highest is called next, and so on. If more than one instantiation has the same priority number, the order among them is arbitrary.

7. You may or may not need to type a value under **Initialization String**. It depends on the requirements of the DLL.

8. Specify as many instances as you wish for the DLL.

When you have finished specifying the DLL path and instances, you can then specify as many additional DLLs as you like. Just remember that each DLL must have at least one instance.

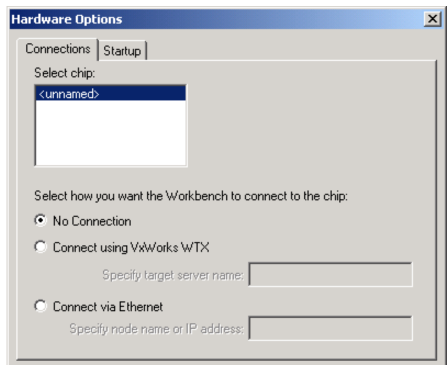
2.12.1.1 Local Simulation Debugging with a Remote Foreign Model

Running IXP2400 or IXP2800 network processors using a remote foreign model is the same as running them with a local foreign model because the DLL controls the location of the foreign model. Use the procedure in Section 2.12.1 and make sure that the PortMapper is running (see Section 2.12.1.3).

2.12.1.2 Hardware Debugging

To debug hardware, you must specify how to connect to the subsystem(s) containing the network processor.

1. On the **Debug** menu, select **Hardware.**
2. On the **Hardware** menu, click **Options.**
   The **Hardware Options** dialog box appears.



3. Click the **Connections** tab.
4. Select a chip from the **Select a chip** list box.
5. Enable the type of connection to the selected chip by clicking on the appropriate button:
   — **No Connection** - If you have multiple chips in your project, you can specify that one or more not be connected. However, at least one must be connected

   — **Connect using VxWorks WTX** - You must specify the name of the server where the hardware is located.

   — **Connect via Ethernet** - You must specify the name of the node (IP address) where the hardware is located.

2.12.1.3 Portmapper

Portmapper is automatically installed as part of the IXA SDK installation process. To ensure that Portmapper is installed and running:

1. On the Window's task bar, click **Start**, point to **Settings**, and then click **Control Panel**.
2. Double-click **Administrative Tools**, and then double-click **Services**.
3. Look for the IXP2000 Portmapper. It should indicate that it has been "Started".
4. If it is not running, select IXP2000 Portmapper, right-click and then click **Start**.

Note: The executable is installed as C:\IXA_SDK_3.1\me_tools\bin\portmapper.exe.

2.12.2 Starting and Stopping the Debugger

Starting:

To enter debug mode:

On the **Debug** menu, click **Start Debugging**, or

Press F12, or

Click the  button.

Once the debugger begins, you can interact with it through the command line window and by using the **Debug** menu and toolbar selections that become activated. (See Table 7 and Table 9.)

Stopping:

To exit debug mode:

- On the **Debug** menu, click **Stop Debugging**, or
  Press CTRL+F12, or
  Click the  button.

Project debug settings such as breakpoints are automatically saved in a debug options file (.dwo) when you save a project.

2.12.3 Changing Simulation Options

2.12.3.1 Marking Instructions

You can select how instructions are marked in a thread window when a thread execution is stopped, such as at a breakpoint (see Figure 27).

To modify the instruction marker:

1. On the **Simulation** menu, click **Options**.
2. Click the **Markers** tab.

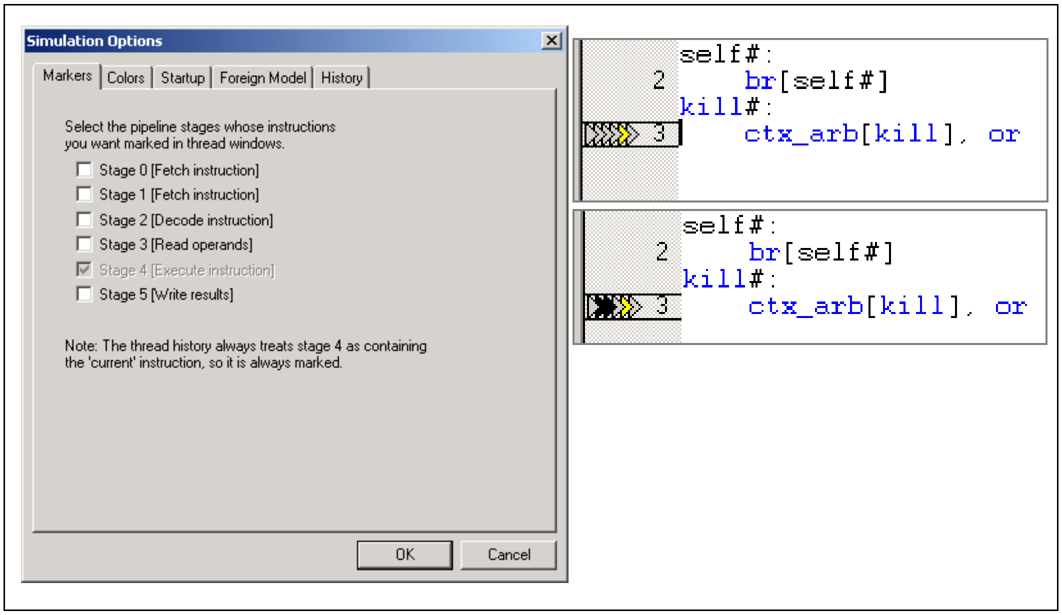
Note: For more information on thread windows, see Section 2.12.8.

By default, the stage 4 instruction is marked as the current instruction. It is highlighted by horizontal black lines above and below it. The thread window is automatically scrolled so that the current instruction marker is visible when execution stops.

If the line containing the current instruction is displayed, then the instruction marker points to it. If the line is hidden because it is in a collapsed macro, then the instruction marker points to the line containing the collapsed macro.

You can optionally choose to have multiple instructions marked in addition to the current instruction when thread execution stops. The Workbench marks any combination of instructions that are in one of the 6 pipeline stages. To add the stages you want marked, click the appropriate check boxes in the Markers tab.

Figure 27. Marking Instructions for the Network Processor



For more information on instruction markers, see:

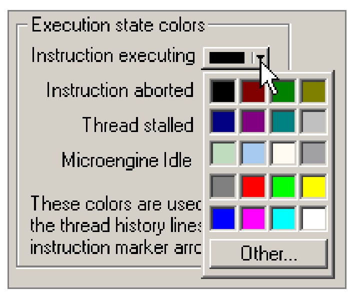
Section 2.12.3.2, "Changing the Colors for Execution State"

Section 2.12.8.8, "Instruction Markers"

Section 2.12.8.9, "Viewing Instruction Execution in the Thread Window"

2.12.3.2 Changing the Colors for Execution State

To customize the colors used to indicate the execution state:



1. On the **Simulation** menu, click **Options**.

   The **Simulation Options** dialog box appears.
2. Click the **Colors** tab.
3. Select the color for each execution state using the corresponding list.

   For more color options, click **Other**. Select the color you want and click **OK**.
4. Click **OK** when done.

Note: The execution state colors are used for both the Pipe Stage markers and for the thread history lines.

2.12.3.3 Initializing Simulation Startup Options

When you are debugging in Simulation mode, the Transactor and its hardware model must be initialized before you can run microcode.

1. On the **Simulation** menu, click **Options**.

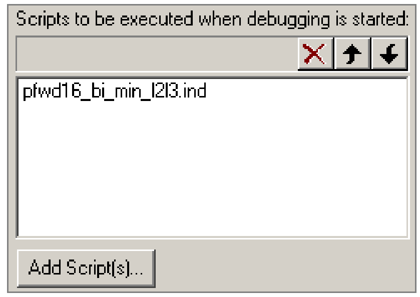
The **Simulation Options** dialog box appears.

2. Click the **Startup** tab.

This property page specifies how the Workbench behaves when you start debugging and when you reset the simulation.

Startup Scripts:

To have the Workbench execute one or more scripts at startup, after initialization:



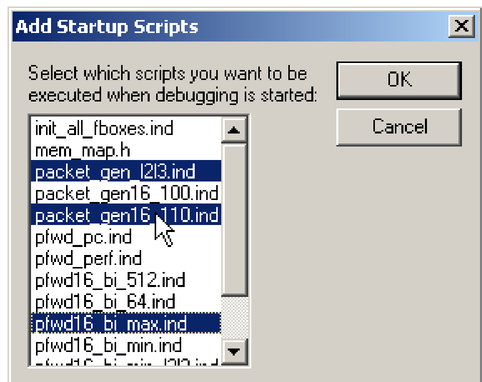
1. On the **Startup** tab, click **Add Script**(s).

   The **Add Startup Scripts** dialog box appears.

2. Select the script(s) that you want executed at startup.

   Note that the scripts must be part of your project (in the Script File folder) to appear in this list. Otherwise the list is blank.

3. Click **OK** when done.



The Workbench executes the scripts in the order in which they appear in the **Scripts to be executed when debugging is started** box. You can change the order in which scripts are executed by selecting the script and using the **Up** and **Down** arrow buttons. You can delete a script from the list with the **Delete** button.

2.12.3.4 Using Imported Variable Data

The user can defer the specification of integer values used by the microcode until load time by using the **.import_var** directive in the assembler or the **__LoadTimeConstant( )** function in the Microengine C compiler. These define a variable that can then be associated with an integer value at load time. On hardware, this association is done by an XScale application.In simulation, this association is done through an imported variable data (**.ivd**) file that gets processed by the loader.

The user specifies the path for the .ivd file using the console function, **loadImportVarData( )**, which is defined by the loader. This function must be called before the load_uof() console function is called

The Workbench allows the user to specify the .ivd file as one of the simulation startup options. If the user selects **Options** from the **Simulation** menu then selects the **Startup** tab, the property page shown in Figure 28 appears. If the user specifies an ivd file, the Workbench automatically invokes the loadImportVarData() console function at the appropriate time in the startup sequence.

The Workbench also allows the user to specify the .ivd file as one of the hardware startup options. If the user selects **Options** from the **Hardware** menu then selects the **Startup** tab, the property page shown in Figure 29 appears. When the user clicks **Start Debugging** in hardware mode, the

Workbench opens the .ivd file. As it is loading microcode into a chip, it parses the .ivd file and sends the lines that pertain to that chip to the loader. Note that if the user selects the Hardware option to assume that the microcode is already loaded, then the imported variable data is not sent to the loader.

The .ivd file is an ASCII file with one variable defined per line. The format of each line is

chip_name image_name symbol value

Where the values are:

- *chip_name* is the name of the chip and must not contain any whitespace. An empty chip name must be specified by two double quotes ("").
- *image_name* is the name of the UOF image, by default the list-file without directory or type. The string must be enclosed within double quotes if it contains any whitespace.
- *symbol* is the name of the imported variable. No embedded whitespace is allowed.
- *value* is the integer value to assign to the imported variable.

Figure 28. Using Imported Variable Data at Startup in Simulation Mode

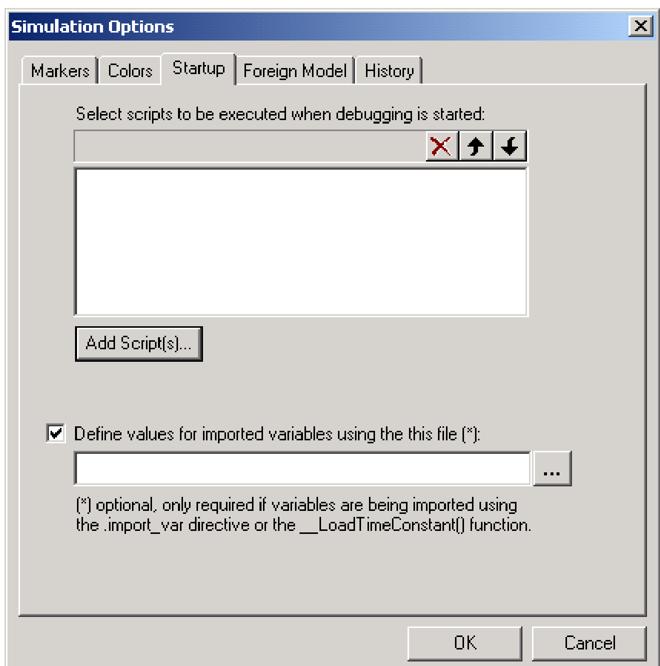
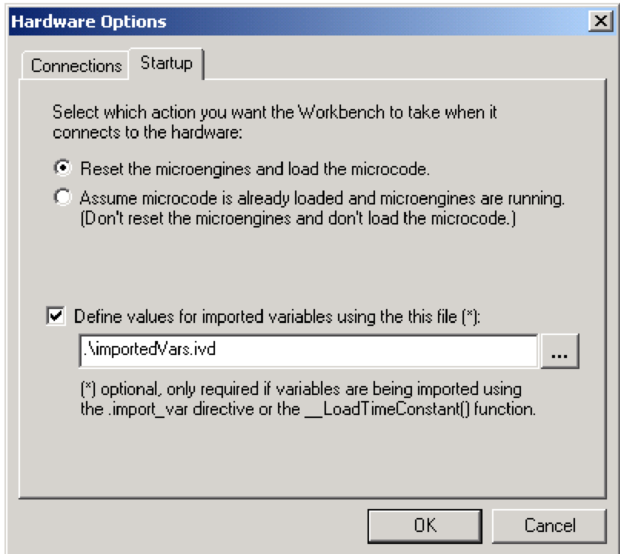


Figure 29. Using Imported Variable Data at Startup in Hardware Mode



2.12.4 Exporting the Startup Script

To create a text file containing all the commands that the Workbench sends to the Transactor during simulation startup:

1. On the **Simulation** menu, click **Export Startup Script**.
   The **Export Simulation Start Script** dialog box appears.
2. Browse to the folder to save the file.
3. Type the name of the script file in the **File name** box.
4. Click **Save**.

The default .ind file extension for script files is added to the name that you typed. See Section 2.5.9.1 to insert the script file into the project.

2.12.5 Changing Hardware Options

2.12.5.1 Specifying Hardware Startup Options

To specify whether or not the Workbench loads microcode when hardware debugging is started:

1. On the **Hardware** menu, click **Options**.
   The **Hardware Options** dialog box appears.
2. Click the **Startup** tab (see Figure 29).
3. Select or clear the action that you want the Workbench to take when it connects to the hardware:
   — **Reset the microengines and load the microcode**

This option causes the Workbench to reset the Microengines and then load microcode into all the assigned Microengines. The Microengines are left in a paused state from which you can start or step them.

— **Assume microcode is already loaded and microengines are running. (Don't reset the microengines and don't load the microcode.)**

This option causes the Workbench to connect only to the debug library on the Intel® XScale™ core. The Microengines are not affected in any way. This would be useful if you want to connect to a running system to examine its state.

4. Click **OK**.

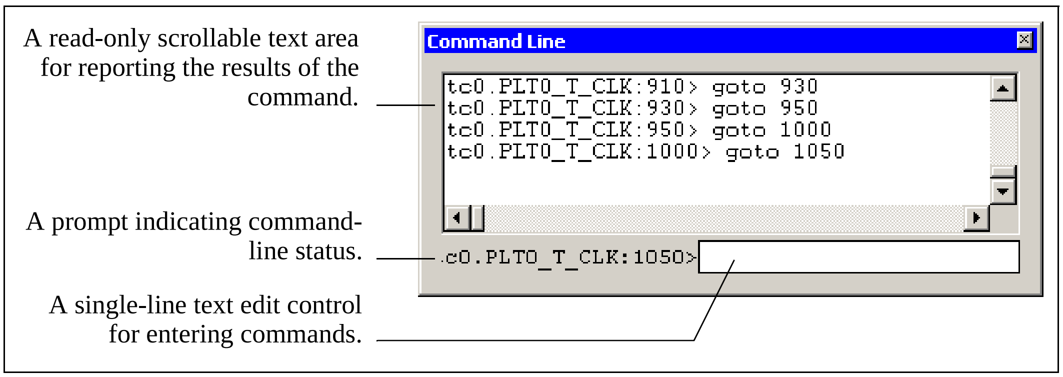
If you choose not to have the Workbench load microcode automatically at startup, you can:

- Load microcode manually by selecting **Load Microcode** on the **Debug** menu.

  (You can also click the button. This button is not on the default **Build** menu. To put this button there, see Section 2.2.3.4.)

2.12.6 The Command Line Interface

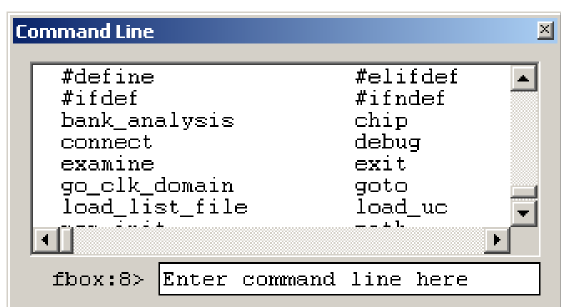
The command line interface (CLI) comprises:



Simulation Mode:

If you are debugging in Simulation mode, the command line is an interface to the Transactor command line. Commands entered on the command line are passed to the Transactor. The command and the Transactor responses are logged into the command line output area. Also, when you perform a simulation operation using GUI controls, the Workbench sends the appropriate command to the Transactor, as if you had typed in on the command line.

CLI implementation



The CLI is a dockable window.

To view it:

1. On the **View** menu, click **Debug Windows.**
2. Select **Command Line**, or

   Click the button on the **View** toolbar.

This makes the **Command Line** window visible. To hide the **Command line** window, clear the **Command Line** check box.

2.12.7 Command Scripts

The Workbench supports the creation of command scripts for frequent execution of a set of Transactor or hardware commands. Command scripts are numbered 1 through 10.

To create a command script:

1. On the **Tools** menu, click **Customize**.

   The **Customize** dialog box appears.

2. Click the **Command Scripts** tab.
3. From the list on the left, select the command script number that you want assigned to the script.
4. In the box on the right, enter the Transactor commands you wish to have executed, just as you would enter them on the Transactor command line.
5. In the **Script name** box, enter the name you want associated with the command script. This name will be displayed in the tool tip and fly-by text when you position the mouse cursor over the corresponding command script toolbar button.
6. Click **Assign**. This assigns the commands and name to the selected command script number.
7. Repeat steps 3 through 6 for as many command scripts as you wish to assign (up to a total of 10).
8. Click **OK**.

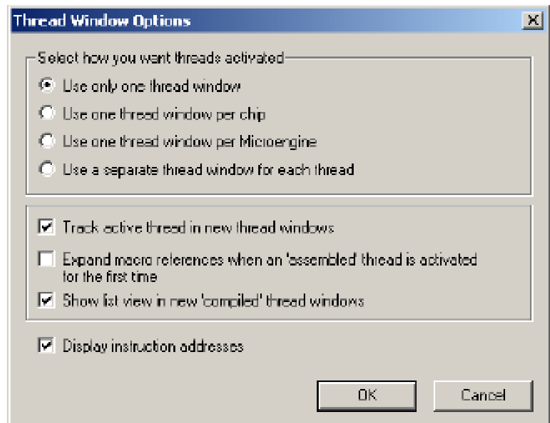
Each command script (1-10) has an associated debug toolbar button . To place a command script button in a toolbar, see Section 2.2.3.4.

2.12.8 Thread Windows

Thread windows differ from normal document windows in that they have a toolbar across the top (see Figure 30 and Figure 31). Setting and clearing breakpoints (see Section 2.12.10), displaying register or variable contents (see Section 2.12.11), and viewing the instruction(s) currently being executed (see Section 2.12.8.9) are all done in thread windows.

2.12.8.1 Controlling Thread Window Activation

You can control how the thread windows are activated using the **Thread Window Options** dialog box.



To do this:

1. On the **Debug** menu, click **Thread Window Options**, or

   Click the  button in the toolbar of an open thread window.

   The **Thread Window Options** dialog box appears.

2. Select how you want threads activated. Select one of the following:

   a. **Use only one thread window**.
   Always reuse the currently open thread window and bring it to the top.

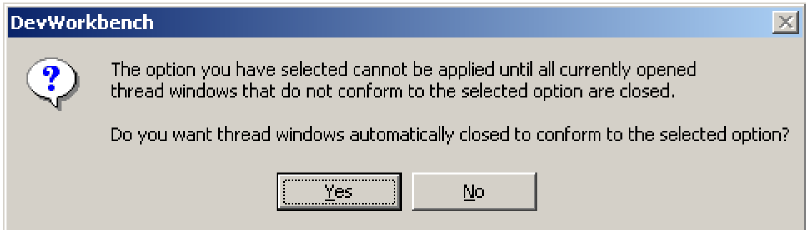
   b. **Use one thread window per chip**.
   Reuse a currently open thread window only if it displays a thread in the same chip as the thread being activated.

   c. **Use one thread window per Microengine**.
   Reuse a currently open thread window only if it displays a thread in the same Microengine as the thread being activated.

   d. **Use a separate thread window for each thread**.
   Always open a new thread window unless one is already open for the thread being activated, in which case the open window is brought to the top.

Whether or not you can select an option depends on the current thread window configuration. For example:

- If you have one or no thread window open, then all options are allowed.
- If you have two thread windows open, you cannot select option (a).
- If you have two or more thread windows open for different Microengines in the same chip, you cannot select option (a) or (b).
- If you have two or more thread windows open for different threads in the same Microengine, you cannot select option (a), (b) or (c).
- If you select an invalid option and click **OK**, the following message box appears:



— If you click **No** the option reverts to the previous selection.

— If you click **Yes** the Workbench closes the appropriate thread windows.

The activation option you select determines the behavior of the thread-selection toolbar.

- If you select option (d), all combo boxes are disabled.
- If you select option (c), the chip and Microengine combo boxes are disabled, allowing you to select a different thread.
- If you select option (b), the chip combo box is disabled, allowing you to select a different Microengine and thread.
- If you select option (a), all combo boxes are selected, allowing you to select a different chip, Microengine and thread.
- If the open project has only one chip, the chip combo box is hidden in order to save toolbar space.

In the next area of the **Thread Window Options** dialog box:

3. Select **Track active thread in new thread windows** if desired (not available if you selected to view each thread in its own window).
4. Select **Expand macro references when an ‘assembled’ thread is activated for the first time** if desired.
5. Select **Show list view in new ‘compiled’ thread windows** if desired.
6. Select **Display instruction addresses** if desired (in list view only).
7. Click **OK** when done.

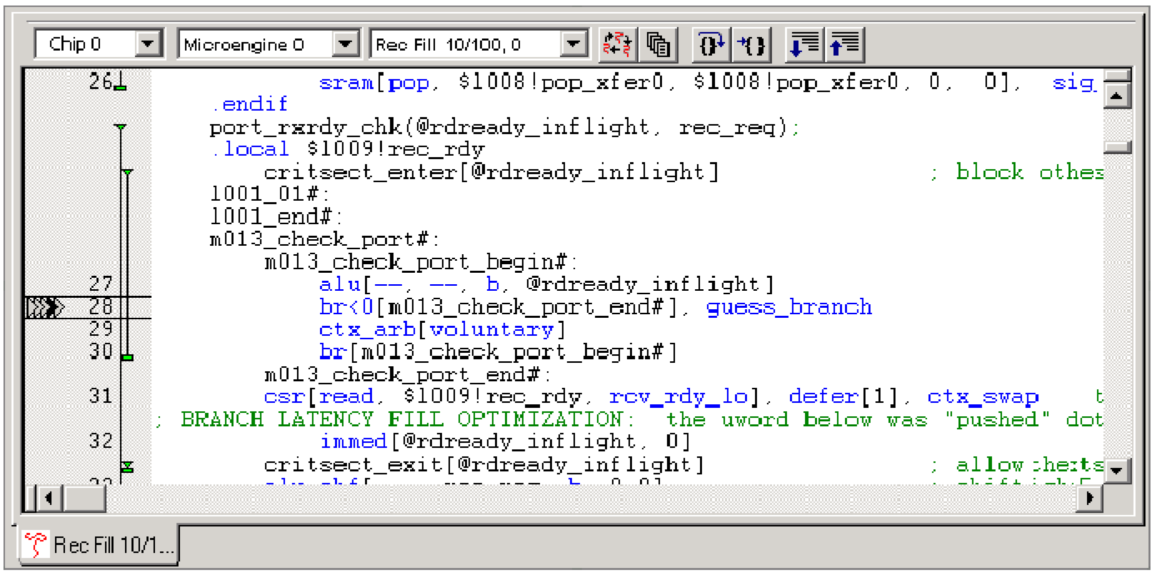
2.12.8.2 Thread Window Controls

Assembled Thread Windows:

If a thread is in a Microengine whose list file is generated by the Assembler, then its thread window displays a 'list' view. This represents flattened code for the entire microstore, as contained in the .list file.

Note: Setting and clearing breakpoints (see Section 2.12.10), displaying register contents (see Section 2.12.11), and viewing the instruction(s) currently being executed are also done in assembled thread windows.

Figure 30. The Assembler Thread Window



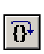
When a thread is being displayed in an assembled thread window, the toolbar contains the following controls:

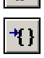
- The Chip list box. This control is not visible if your project has only one chip.
- The Microengine list box. This control is disabled if you are using one thread window per Microengine or you are using a separate thread window for each thread.
- The Thread list box. This control is disabled if you are using a separate thread window for each thread.

The toolbar contains following buttons:

 **Track Active Thread** (see Section 2.12.8.3).

 **Display the Thread Window Options dialog box** (see Section 2.12.8.1).

 **Step Over** (see Section 2.12.9.3).

 **Run to Cursor** (see Section 2.12.8.4).

 **Expand macros** (see Section 2.12.8.6).

Collapse macros (see Section 2.12.8.6).

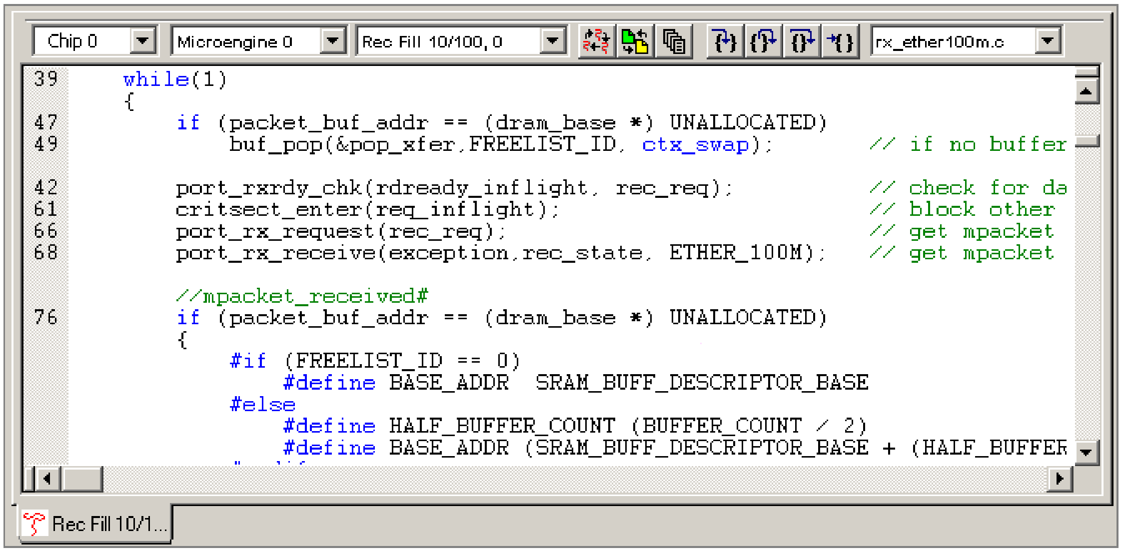
Note: Not all buttons are available in hardware mode.

Compiled Thread Windows:

Compiler thread windows look the same as Assembler thread windows but have some differences. Display options are similar (see Section 2.12.8.1).

Note: Setting and clearing breakpoints (see Section 2.12.10), displaying register contents (see Section 2.12.11), and viewing the instruction(s) currently being executed are also done in Compiler thread windows. They cannot be performed in the source file windows.

Figure 31. The Compiled Thread Window



When a thread is being displayed in a compiled thread window, the toolbar contains the following controls:

- The Chip list box. This control is not visible if your project has only one chip.
- The Microengine list box. This control is disabled if you are using one thread window per Microengine or you are using a separate thread window for each thread.
- The Thread list box. This control is disabled if you are using a separate thread window for each thread.
- The Source list box (if in source view). Here you can view any of the *.c files used to generate the .list file. When Microengine execution starts then stops, the Workbench changes the displayed source file to the one that generated the current instruction.

The toolbar contains following buttons:

Track the active thread (see Section 2.12.8.3).

Display the **Thread Window Options** dialog box (see Section 2.12.8.1).

 Step Into (see Section 2.12.9.4).

 Step Out (see Section 2.12.9.5).

 Step Over (see Section 2.12.9.3).

 Run to Cursor (see Section 2.12.8.4).

Note: Unlike the assembled thread window, you cannot expand or collapse the display in a compiled thread window in list view.

Note: Not all buttons are available in hardware mode.

2.12.8.3 Tracking the Active Thread

You can specify that you want tracking of the active thread by clicking the  button in the thread window toolbar. This feature is available only if you have specified that you want only one thread window or one thread window per chip or per Microengine.

When Microengine execution is started then stopped and a different thread in the same Microengine becomes active, the thread window is automatically changed to display the active thread.

In the **Thread Windows Options** dialog box, specify whether new thread windows are opened with active thread tracking enabled by selecting or clearing Track active thread in new thread windows.

2.12.8.4 Running to Cursor

If you are debugging in Simulation mode, you can place the cursor at a point in the code and then you can Run to Cursor.

To do this:

1. Place the cursor on a line in a thread window by clicking in that line.
2. On the **Debug** menu, click **Run Control**, then click **Run To Cursor**, or

   Click the  button in the **Thread window**.

Or:

   Right-click in the line to which you want to run, then select **Run To Cursor** from the shortcut menu.

If the line is in the source view of a compiled thread or if it contains a collapsed macro reference in an assembled thread, then the simulation runs until the first generated instruction is reached.

Run to Cursor can be performed only on lines that generated instructions.

2.12.8.5 Activating Thread Windows

Once microcode is loaded, you can directly access the execution state of all the threads in the project.

To explicitly activate a thread window, do this:

1. Double-click the thread name in the **ThreadView** or the **Thread Status** window, or

   Right-click the desired thread in the **ThreadView** or the **Thread Status** window and click **Open Thread Window**, or

   Change the selection(s) in the thread-selection toolbar in the thread window.

The thread window is implicitly activated by:

- **Stopping at a breakpoint**. The thread in which this occurred is activated.
- Selecting **Go To Instruction** from the shortcut menu in either the thread history or queue status window. The thread in which the instruction was executed is activated.

When Microengine execution stops for any reason other than a location breakpoint—such as, a break-on-change occurs, or you click **Stop**, or a packet count limit was reached by the bus device simulation—the Workbench determines the active thread for each Microengine and does one of the following:

- If the active thread is already activated in a thread window, the window is simply scrolled to display the current instruction or source line.
- If the active thread isn't already activated and there is a thread window in which a different thread in the same Microengine is activated, then the active thread is activated in that window if you have specified that you want tracking of the active thread. (A toolbar button in the thread window allows you to enable or disable this feature.)
- If the active thread isn't already activated and there are no thread windows in which a different thread in the same Microengine is activated, then the active thread is not activated.

Thread Window Title Bar:

The title displayed on the thread window shows the Microengine address and thread name. Also displayed is the currently executing PC and whether the thread is active or swapped out.

Thread Window Contents:

A thread window displays the output of the Assembler as opposed to original source code. The Assembler output differs from source code in that:

- Symbols are replaced with actual values.
- Instructions may be reordered due to optimization.
- Names of local register are adorned by a prefix, etc.
- If you built a Microengine image from multiple source files by using the #include directive, then the associated thread window displays the modified output from the combined sources.

2.12.8.6 Displaying, Expanding, and Collapsing Macros (assembled threads only)

By default, all macros are collapsed. A green triangle to the left of the instruction indicates that the instruction is a fully collapsed macro (see Figure 32).

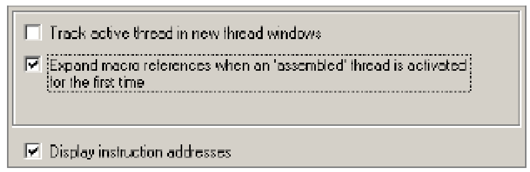
First Time Thread Activation:

You can specify whether macro references are fully expanded or collapsed when an assembled thread is activated for the first time. To do this:

1. On the **Debug** menu, click **Thread Window Options**.

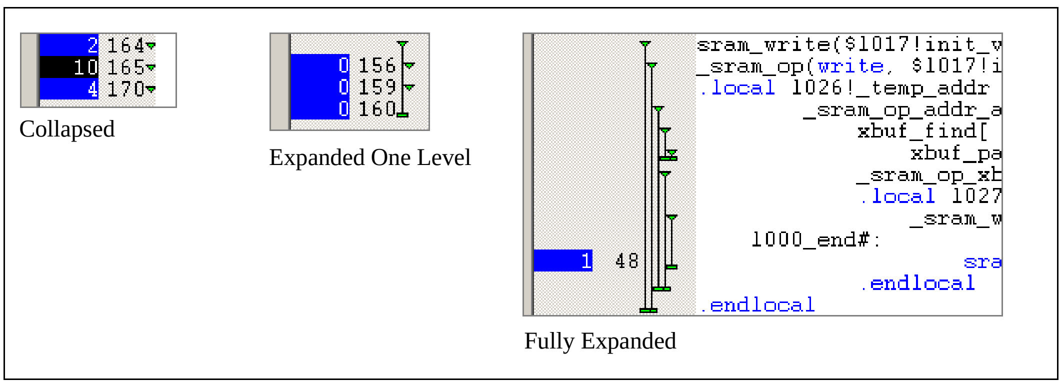
   The **Thread Window Options** dialog box appears.

2. Enable or clear the **Expand macro references when an 'assembled' thread is opened for the first time**.



When you re-activate an assembled thread, the Workbench restores the state of macro expansion that existed when the thread was deactivated. However, when you stop debugging, the macro expansion state is no longer remembered.

Figure 32. Expanding Macros



Macro Marker Display:

If you don't see the macro markers you may have to enable them.

To display macro markers:

1. Right-click the thread window.
2. Click **Display Macro Markers** on the shortcut menu.

Macro Expansion:

To expand a collapsed macro:

1. Right-click on the triangle or anywhere on the instruction line.
2. Click **Expand Macro One Level**, or,
   Click **Expand Macro Fully**.

Macro Collapse:

You can only fully collapse an expanded macro, not one level at a time. To do this:

1. Right-click anywhere on the instruction line.
2. Click **Collapse Macro**.

Expand and Collapse of all Macros at Once (Assembled Threads Only):

You can expand and collapse all macros at the same time. Do the following:

- To expand all macros one level, click the  button.
- To collapse all macros one level, click the  button.

Note that this is the only way to collapse a macro one level at a time.

Go to Source:

To go to the source line corresponding to a line in a thread window:

1. Place the insertion cursor on the line.
2. On the **Debug** menu, click **Go To Source**.
   The Workbench:
   — Opens a document window with the source file,
   — Places the insertion cursor at the beginning of the requested line, and
   — Scrolls the line into view.

You can also right-click the thread window line and click **Go To Source** from the shortcut menu.

2.12.8.7 Displaying and Hiding Instruction Addresses

To display or hide the microstore address at which each instruction in a thread window is located:

1. Right-click in the thread window.
2. Select or clear **Display Instruction Addresses** on the shortcut menu.

This toggles displaying of the addresses of the microstore instructions. You can also do this using the **Thread Window Options** dialog box.

If macro references are expanded, instruction addresses are displayed on the generated instruction lines. If references are collapsed, addresses are displayed on the macro reference lines, with the address being that of the first instruction generated by that reference.

Note: The displaying of instruction addresses affects all thread windows and is saved as a global option which is in effect across all projects.

2.12.8.8 Instruction Markers

During a typical debugging session, thread windows display several types of instruction markers. An instruction marker displays on the left side of the thread window, in the same location as bookmarks and breakpoints. The types are summarized in Table 2:

Table 2. Instruction Markers

Marker Name	Symbol	Function
Swapped Out	⊳⊳⊳⊳⊳	Marks the instruction to be executed when the thread's context is swapped back in.
History	⇨	Marks the instruction that was executing in pipe stage 4 at the cycle associated with the thread history window's cycle marker.
Pipe Stage	▶⊳⊳⊳⊳	Marks the instructions executing in each of the 6 pipeline stages.

Assembled Thread:

In an assembled thread, if line containing the current instruction is displayed, then the instruction marker points to it. If the line is hidden because it is in a collapsed macro, then the instruction marker points to the line containing the collapsed macro.

Compiled Thread:

In the list view for a compiled thread, the instruction marker points to the current instruction. In the source view for a compiled thread, the instruction marker points to the C source line that generates the current instruction.

2.12.8.9 Viewing Instruction Execution in the Thread Window

During a simulation session, the Pipe Stage marker allows you to view which instruction is inside each of the 6 stages of the pipeline. This marker contains up to 6 stacked arrowheads that correspond to each of the 6 pipeline stages. The leftmost arrowhead represents stage 0, and the rightmost arrowhead represents stage 5. The default is stage 4.

Colors:

The arrowheads are color-filled according to the state of the instruction in the pipeline stage. By default, the Workbench uses:

- **Black** - for instruction executing.
- **Yellow -** for instruction aborted.
- **Red** - for thread stalled.

If a thread has a different instruction in each of the pipe stages, then the thread window will have 6 Pipe Stage markers, one on each of the 6 instructions. Each marker will have a different arrowhead filled with the appropriate color. For example, if an instruction is executing in stage 3, then its marker will have the stage 3 arrowhead filled with black with all other arrowheads unfilled. If an instruction is aborting in stage 2, then its marker will have the stage 2 arrowhead filled with yellow with all other arrowheads unfilled. If an arrowhead is not color-filled, it means that the instruction that the marker points to is not in the corresponding pipeline stage.

Same Instruction in More Than One Pipeline Stage:

It is possible, due to branching, for the same instruction to be in more than one pipeline stage. In this case, the Pipe Stage marker on that instruction will have multiple arrowheads filled in, possibly with different colors. This also means that there will be fewer than 6 markers in the thread window.

Context Swapping Issues:

When a context swap is in progress, the latter stages of the pipeline have instructions from the context being swapped out and the early stages have instructions from the context being swapped in. In this case, the thread windows for both contexts have Pipe Stage markers displayed. However, the marker for each thread window will show arrowheads only for those stages in which the thread has instructions.

For example, if a thread only has an instruction in stage 4, then its marker will only show a single arrowhead, corresponding to stage 4. The other thread marker will show arrowheads for each of the four stages in which it has instructions.

When a context is completely swapped out, its thread window displays all five arrowheads unfilled to mark the instruction at which execution will resume when the context is swapped back in.

2.12.8.10 Document and Thread Window History

The Workbench maintains a history of previously visited document and thread windows along with their scrolled positions. When a project is opened the history is cleared. A window and its scrolled position gets added to the history when any of the following events occur

- The user changes focus from one document window to another.
- The user opens a new thread window.
- The user opens a file.
- The user creates a new file.
- The user executes the **Go To Macro** command.
- The user executes the **Go To Source** command.
- The user executes the **Go To Instruction** command.
- The user selects a different chip, Microengine, or thread to be displayed in a thread window using the combo boxes in the toolbar of the thread window. A breakpoint is hit, causing a different thread window to get focus

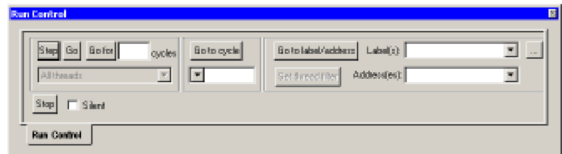
To move backwards through the history, the user selects **Back** from the **Window** menu. To move forward through the history, the user selects **Forward** from the **Window** menu. There are toolbar buttons for these commands that can be added to the toolbar by selecting **Customize** from the **Tools** menu.

If the user goes **Back** one or more times and then one of the events listed above occurs, all the history that was forward of the window that was returned to will be deleted. For example, assume the history contains windows A, B, C, D, E and F. If the user goes back to C then executes a **Go To Macro** command, windows D, E and F are deleted from the history.

2.12.9 Run Control

Run Control lets you govern execution of the Microengines. Different control operations are available from the Workbench depending on whether you are in Simulation or Hardware mode (see Table 1).

In Simulation mode, the Workbench provides the controls for running microcode in a dockable **Run Control** window.



To display the **Run Control** window:

1. On the **View** menu, click **Debug Windows**.
2. Select or clear **Run Control** to toggle visibility of the **Run Control** window, or

   Click the  button on the View toolbar.

Note: The **Run Control** window is not supported when debugging hardware.

2.12.9.1 Single Stepping

Single stepping has four variations:

Microengine Step	Performed on Microengines (see Section 2.12.9.2).
Step Into	Performed on a single thread in a compiled thread window only (see Section 2.12.9.4).
Step Over	Performed on one thread (see Section 2.12.9.3).
Step Out	Performed on a single thread in a compiled thread window only (see Section 2.12.9.5).

2.12.9.2 Stepping Microengines

To single step one cycle:

1. Click **Step** in the **Run Control** window, or

   On the **Debug** menu, click **Run Control**, then click **Step Microengines**, or

   Press SHIFT+F10, or

   Click the  button.

All Microengines:

To single step all Microengines, regardless of which threads are running:

- Select the **All threads** entry from the list under the **Step** button in the **Run Control** window.

A Specific Thread:

To single step one cycle of a specific thread:

- In the **Run Control** window, select the thread's entry from the list under the **Step** button.

Note: Stepping microengines is not supported when debugging hardware.

2.12.9.3 Stepping Over

Step Over allows you to execute as many machine cycles as it takes complete the current line in the thread window. To **Step Over**:

- On the **Debug** menu, click **Run Control**, then click **Step Over**, or

  Click the  button in the thread window's toolbar, or
  Right-click in the thread window and click **Step Over** from the shortcut menu, or
  Press F10.

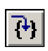
Note: When debugging hardware, only the thread in whose window the **Step Over** button is clicked gets stepped. All other microengines remain paused. Also, due to instruction sequencing restrictions, more than one instruction may get executed as part of the step operation.

2.12.9.4 Stepping Into (compiled threads only)

Step Into executes as many Microengine cycles as it takes to execute the current line in the thread window, whether it is a microinstruction line in a list view or a C source line in a source view. Stepping into is supported only for compiled threads.

To **Step Into** do the following:

- On the **Debug** menu, click **Run Control**, then click **Step Into**, or

  Click the  button in the thread window's toolbar, or
  Right-click in the thread window and select **Step Into** from the shortcut menu.

Note: Stepping into is not supported when debugging hardware.

2.12.9.5 Stepping Out (compiled threads only)

Step Out executes as many Microengine cycles as it takes to complete the thread's current function and return to the calling function. Stepping out is supported only for compiled threads.

To step out, do the following:

- On the **Debug** menu, click **Run Control**, then click **Step Out,** or

  Click the  button in the thread window's toolbar, or
  Right-click in the thread window and select **Step Out** from the shortcut menu.

Note: Stepping out is not supported when debugging hardware.

2.12.9.6 Executing Multiple Cycles

All Microengines:

To run for a specified number of cycles in all Microengines, regardless of which threads are running:

1. Select **All threads** in the list under the **Go for** button.
2. Type the number of cycles in the box to the right of the **Go for** button.
3. Click **Go for**.

All Microengines run until the specified thread has executed the specified number of cycles.

A Specific Thread:

To run for a specified number of cycles in a specific thread:

1. Select the thread’s entry from the list under the **Go for** button.
2. Type the number of cycles in the box to the right of the **Go for** button.
3. Click **Go for**.

Note: Executing multiple cycles is not supported when debugging hardware.

2.12.9.7 Running to a Specific Cycle

To run until a specified cycle count is reached:

1. Type the cycle count in the box under the **Go to cycle** button.
2. Click **Go to cycle**.

Note: Running to a specific cycle is not supported when debugging hardware.

2.12.9.8 Running to a Label or Microword Address

To run until a specific microcode label or microword address is reached by a thread:

1. Enter the label(s) and/or address(es) into the appropriate boxes.
2. Click **Go to label/address**.

Note: Running to a label or microword address is not supported when debugging hardware.

2.12.9.9 Running Indefinitely

To run the microcode indefinitely:

On the **Debug** menu, click **Run Control**, then click **Go**, or

Click the 

button in the **Run Control** window, or

Click the  button on the **Debug** toolbar, or

Press F5.

Microcode execution stops only if a breakpoint is reached or if you manually stop execution (see Section 2.12.9.10).

2.12.9.10 Stopping Execution

To stop microcode execution at any time:

On the **Debug** menu, click **Run Control**, then click **Stop**, or

Click 

in the **Run Control** window, or

Click the  button on the **Debug** menu, or

Press SHIFT+F5.

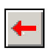
When debugging hardware, if a thread running on a microengine does not swap out, it will continue to run. The Workbench displays a message indicating which microengines did not stop.

2.12.9.11 Resetting the Simulation

Reset executes a Transactor sim_reset command, which puts the simulation back to the state after the original initialization was done. After the reset, the Workbench re-executes the options specified by the Startup page of the **Simulation Options** dialog box. However, if you specified that the Workbench should initialize the model, it doesn't repeat the chip and memory definition commands and the init command since they are unnecessary.

To reset the simulation:

1. On the **Debug** menu, click **Run Control**, then click **Reset**, or

   Click the  button on the **Debug** toolbar, or

   Press CTRL+SHIFT+F12.

This executes a Transactor sim_reset command, which puts the simulation back to the state after the original init was done. After the reset, the Workbench re-executes the options specified by the **Simulation Startup** page of the **Simulation Options** dialog box. However, if you specified that the Workbench should initialize the model, it doesn't repeat the chip and memory definition commands and the init command since they are unnecessary.

2.12.10 About Breakpoints

Note: You cannot set a breakpoint while a simulation is running or when Microengines are running in hardware.

In the source view, when you set a breakpoint on a line, a breakpoint marker appears on that line and a breakpoint is set on the first instruction that it generates. In the list view, it is possible to set breakpoints on multiple lines that are generated by the same C source line. In this case, the marker that is displayed on the source line in the source view depends on the state of the breakpoints on the generated lines.

- If all breakpoints have the same status, then the source line marker reflects that status. In this situation, you can perform a breakpoint action on the source line and the action is performed on all the breakpoints on the generated lines. For example, if all are disabled, then the disabled-breakpoint marker is displayed on the source line. If you enable the breakpoint on the source line, all breakpoints on generated lines are enabled.

- If the breakpoints do not have the same status, then a distinct marker consisting of a red circle filled with dark gray is displayed on the source line. In this situation, the only supported action is to enable all the breakpoints by executing an Insert/Remove Breakpoint command.

In an assembled thread, when you set a breakpoint on a line containing a collapsed macro reference, a breakpoint marker is displayed on that line and a breakpoint is set on the first instruction that the macro generates. If you then expand that macro reference, the breakpoint marker is displayed on the generated line.

For situations where there are breakpoints on multiple lines generated by a collapsed macro, the breakpoint marker and supported actions are the same as described above for the C source line.

Thread Window Action Taken:

When a breakpoint is reached during execution:

- The thread window that reached the breakpoint is activated.
- The appropriate instruction is displayed and marked.
- A message box appears (if set) indicating the breakpoint was reached.

  To disable the display of this message box, on the **Debug** menu, select or clear the Report Breakpoint Hit option.

Conditional and Unconditional:

A breakpoint can be unconditional or conditional.

- An unconditional breakpoint on an instruction halts execution when any context in the Microengine reaches that instruction.
- A conditional breakpoint halts execution only when one of the specified contexts reaches that instruction.

Soft Breakpoint Support:

The Workbench supports soft breakpoints, which are inserted into assembler code using the **ctx_arb[bpt]** instruction and into Microengine C code using the **_assert macro**, which in turn inserts a **ctx_arb[bpt]** instruction. When this instruction is executed, the Workbench is notified and displays a message box indicating where the breakpoint occurred, i.e., the chip, Microengine, thread, and instruction address. In simulation mode, the Workbench stops the simulation. In hardware mode, the Workbench stops all the other Microengines. In either case, the user can resume execution by clicking **Go, Step Over,** etc.

2.12.10.1 Setting Breakpoints in Hardware Mode

Restrictions:

You can set breakpoints in Hardware mode with the following restrictions:

- Each breakpoint you insert causes the Debug library in the Intel® XScale™ core to place a breakpoint routine in unused Control Store space within the Microengines. Consequently, the number of breakpoints you can insert will be limited by the size of your microcode image.

- You may be prevented from setting a breakpoint on certain instructions because processing the breakpoint will adversely affect the thread's execution state or register contents.

  If this occurs, you see the following message when you attempt to insert the breakpoint: "Breakpoint can't be set at line xx because of Microcode sequencing restrictions".

If breakpoints are set in multiple Microengines, it is possible to hit more than one breakpoint before all of the Microengines have paused.

As with step processing, if a thread running on a Microengine does not swap out, it will continue to run. You should check the **Thread Status** window after a breakpoint has been reached to determine if any threads are still active.

2.12.10.2 About Breakpoint Markers

When a breakpoint is set on a microword address, the Workbench displays a breakpoint marker in the left-hand margin of the corresponding line in the thread window. The marker's appearance depends on the properties of the breakpoint.

The different markers and their meanings are described below:

	Unconditional breakpoint	Enabled in all threads in Microengine.
	Unconditional breakpoint	Disabled in all threads in Microengine.
	Conditional breakpoint	Set and enabled in this thread but not set in all threads in the Microengine.
	Conditional breakpoint	Set but disabled in this thread but not set in all threads in the Microengine.
	Conditional breakpoint	Not set in this thread but set and enabled in one or more other threads in the Microengine.
	Conditional breakpoint	Not set in this thread but set and disabled in one or more other threads in the Microengine.
	Special breakpoint	The states of two or more breakpoints in the generated code are different, so the corresponding line in the source code gets a special breakpoint marker. In this situation, the only supported action is to enable all the breakpoints by executing and Insert/Remove Breakpoint command.

2.12.10.3 Inserting and Removing Breakpoints

To insert a breakpoint in a Microengine:

1. Open a thread window for one of the threads in the Microengine.
2. Place the insertion cursor on the line where you wish to insert the breakpoint.
3. On the **Debug** menu, click **Breakpoint**, then click **Insert/Remove**, or

   Click the  button on the **Debug** toolbar, or

   Press F9.

Or:

1. Right-click the line at which you wish to insert the breakpoint.
2. Click **Insert/Remove Breakpoint** from the shortcut menu.

Conditional Breakpoints:

By default, the breakpoint is inserted as unconditional. To make the breakpoint conditional you must change its properties (see Section 2.12.10.4).

Hardware Mode Restrictions:

When debugging in Hardware mode, you cannot set breakpoints on instructions that:

- Are in defer shadows, or
- Are indirect branches.

Breakpoint Removal:

To remove a breakpoint in a Microengine:

1. Open a thread window for one of the threads in the Microengine in which the breakpoint is set.
2. Place the insertion cursor on the line at which you wish to remove the breakpoint.
3. On the **Debug** menu, click **Breakpoint**, then click **Insert/Remove**, or

   Click the  button on the **Debug** toolbar.
   Press F9.
   or

1. Right-click the line at which you wish to remove the breakpoint.
2. Click **Insert/Remove Breakpoint** from the shortcut menu.

The breakpoint is removed in all contexts, regardless of whether it was conditional or unconditional.

Removal of all Breakpoints:

To remove all breakpoints in all Microengines:

- On the **Debug** menu, click **Breakpoint**, then click **Remove All**, or

  Click the  button on the **Debug** toolbar.

2.12.10.4 Enabling and Disabling Breakpoints

To enable or disable breakpoints on code locations, do the following:

1. Place the insertion cursor on the line at which you wish to enable/disable a breakpoint.
2. On the **Debug** menu, click **Breakpoint**, then click **Enable/Disable**, or

   Click the  button on the **Debug** toolbar. (This button is not on the default **Debug** menu. To put this button there, see Section 2.2.3.4.), or
   Press CNTRL+F9.

Or:

1. Right-click the line at which you wish to enable/disable a breakpoint.
2. Click **Enable/Disable Breakpoint** from the shortcut menu.

To disable all breakpoints:

- On the **Debug** menu, click **Breakpoint**, then click **Disable All**, or

  Click the button on the **Debug** toolbar. (This button is not on the default **Debug** menu. To put this button there, see Section 2.2.3.4.).

To enable all breakpoints:

- On the **Debug** menu, click **Breakpoint**, then click **Enable All**, or

  Click the button on the **Debug** toolbar. (This button is not on the default **Debug** menu. To put this button there, see Section 2.2.3.4.).

2.12.10.5 Changing Breakpoint Properties

To change the breakpoint properties,

1. Open a thread window for one of the threads in the Microengine where the breakpoint is set.
2. Place the insertion cursor on the line where you want to change breakpoint properties.
3. On the **Debug** menu, click **Breakpoint**, then click **Properties**.
   The **Breakpoint Properties** dialog box appears. The dialog box shows the chip, Microengine, microword address, and thread window line number where the breakpoint is set.

In this dialog box you can:

- Select or clear **Enabled** to enable or disable the breakpoint.
- Click **Unconditional** to make the breakpoint unconditional.
- Click **Conditional** to make the breakpoint conditional (that is, it applies to selected contexts in the Microengine)
  — Select the boxes for the contexts for which you want to apply the breakpoint.
- Click **Remove** in the **Breakpoint Properties** dialog box to remove the breakpoint.

2.12.10.6 About Multi-Microengine Breakpoint Support

Support for the ability to manipulate a breakpoint in multiple Microengines simultaneously has been added to the Workbench. When the user right-clicks on a code line in a thread window, the context menu that gets displayed contains a new item labeled **Multi-Microengine Breakpoint**. If the user selects this item, the Workbench displays the dialog box shown in Figure 33.

A list box displays the Microengines that meet the following criteria:

- The Microengine is in the same chip as the Microengine that contains the thread whose window was clicked in.
- The Microengine has code loaded in it and the code was generated using the same source file that generated the line of code that was clicked on.

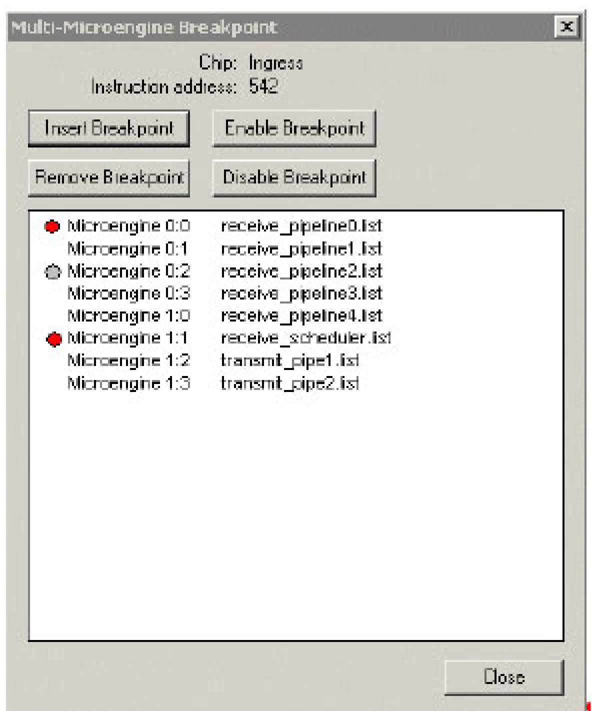
Next to each ME is the name of the list file that is loaded into that ME. If an ME already has a breakpoint set at the line that was clicked on, then the appropriate breakpoint marker is displayed next to it. A solid red marker indicates the breakpoint is unconditional and is enabled in all threads in the ME. A gray marker indicates the breakpoint is unconditional and is disabled in all threads in the ME. A red marker with a white dot inside indicates the breakpoint is conditional (not set in all contexts) and is enabled in one or more contexts in the ME. A gray marker with a white dot inside indicates the breakpoint is conditional (not set in all contexts) and is disabled in one or more contexts in the ME. A marker with a red border and gray interior indicates a 'special' breakpoint is set. This means that the line generates multiple lines of code, e.g., a macro or a C source line, and more than one generated line has a breakpoint but they are not all in the same state

The user selects one or more MEs from the list and clicks on the appropriate button to perform the desired operation. The operation is performed on all contexts in those MEs in the selected group for which the operation makes sense. For example, if three MEs are selected and two of them have disabled breakpoints and the user clicks **Enable Breakpoint**, then the two disabled breakpoints become enabled but the ME without a breakpoint is unaffected.

Depending on the breakpoint status in the selected MEs, some of the buttons may be disabled. For example, if none of the selected MEs have a breakpoint set, then **Remove Breakpoint**, **Enable Breakpoint** and **Disable Breakpoint** are disabled

Also, there is a **Multi-Microengine Breakpoint** item in the **Breakpoint** submenu in the **Debug** menu in the main menu bar. It operates on the line at which the insertion cursor is located in the active thread window. If no thread window is active, the item is disabled.

Figure 33. Multi-Microengine Breakpoint Dialog Box



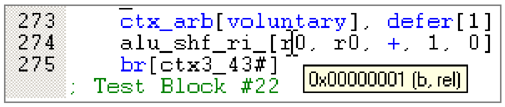
2.12.11 Displaying Register Contents

When program execution is stopped, you can display register contents directly from instruction context in a thread window.

To do this:

1. Position the cursor over the register symbol.
2. Wait for a moment.

   The Workbench displays the contents of the register assigned to that symbol as a pop-up window beneath the cursor.



Hex or Decimal Display:

To control whether the data is displayed in decimal or hexadecimal format:

1. Right-click in the thread window.
2. Select or clear **Hexadecimal Data** on the shortcut menu.

Register History:

To go along with the thread history, the Workbench will record register history. The values for all GPRs, transfer registers and neighbor registers in each Microengine will be remembered for the same cycle extents as thread history. The information displayed in a thread window datatip (the “pop-up” window described in the previous section) for a register or a C variable will be based on the register’s value at the cycle that is currently active in the history window. Similarly, a data watch for a register or a C variable that is stored in a register will display the register’s value at the active cycle. The active cycle can be changed by

- Clicking on the left or right arrows in the history window or data watch window.
- Dragging the cycle marker to the desired cycle count.
- Double-clicking in the history window at the desired cycle count.
- Right-clicking in the history window and selecting **Go To Instruction** from the context menu

Whenever simulation stops the active cycle is automatically set to be the most recently simulated cycle. This means that datatips and data watches will display the current register values. The history PC marker in all thread windows is hidden at this time.

When the active cycle is changed a non-current cycle, data watches and datatips of non-register states and variables will display an appropriate message to indicate that the historical value is not available.

2.12.12 Data Watch

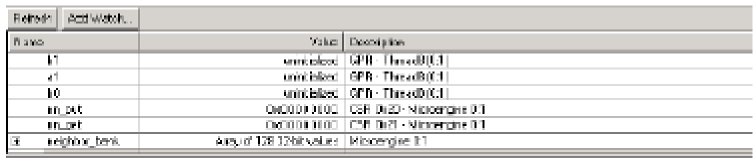
In debug mode, you can monitor the values of simulation states using the **Data Watch** window.

To do this:

- On the **View** menu, click **Debug Windows**, then click **Data Watch**, or

  Click the button on the **View** toolbar.

This toggles visibility of the **Data Watch** window. The window contains a list with three columns:



Name Contains the name of the state being watched.

Value Displays the state's value.

Description Contains information such as which chip, Microengine or thread the watch pertains to.

Note: GPR names cannot be entered directly into the data watch window.

Values Updates:

Watch values are updated whenever microcode execution stops. To force updating the values at other times, click **Refresh**.

2.12.12.1 Data Watches in C Thread Windows

In C thread windows, data watches can be set for C variables by right-clicking on the variable in the thread window and selecting **Set Data Watch for:<variable_name>**.

If the **Data Watch** window is not visible, go to Section 2.12.12.

- When the variable is in scope its value appears in the **Data Watch** window.
- When the variable is out of scope, the phrase *Out of scope* appears in the **Value** field for the watched variable.
- Variables that contain C structures are displayed hierarchically, with the member variables displayed on separate lines in the watch window. You can expand and collapse the display of the member variables.

Note: Not all variables can have data watches set. Many variables are optimized away by the Compiler and the Compiler does not provide any debug data for those variables. The workbench doesn't know that the text you select is a variable if it doesn't have any debug data. If this is the case, the Set Data Watch option on the shortcut menu is unavailable.

2.12.12.2 Entering a New Data Watch

To enter a new data watch:

1. Right-click anywhere in the **Data Watch** window.
2. Click **New Watch** from the shortcut menu, or
   Double-click the blank entry at the bottom of the data watch list.
3. Type the name of the state you want to watch.

Array States:

For states that are arrays, such as local memory, you must enter a range of array locations to be watched. The format for the range is the same as that for the Transactor:

[m]	to watch a single location of an array. For example, local_mem[1] watches location one in local memory.
[m:n]	to watch locations m through n inclusive. For example, local_mem[0:3] watches locations 0 through 3 in local memory. And since a data watch range can be specified in ascending or descending order, you could specify this watch as local_mem[3:0].
[m:+n]	to watch location m plus the n locations following it. For example, local_mem[5:+3] watches locations 5 through 8.

Bit Range:

You can specify a bit range to be watched. The format for a bit range is:

<m>	to watch only bit m of a state. For example, f0.ctl.p0_addr<12> watches only bit 12 of the stage 0 address.
<m:n>	to watch bits m through n of a state, with m being greater than n. For example: local_mem[0:3]<12:10> watches bits 12 through 10 of local memory locations 0 through 3.

Segments:

Data watch values are broken into 32-bit segments. For example:

A 64-bit value displays as 0xcafecafe 0xcafecafe.

Save:

Data watches are saved with project debug settings.

2.12.12.3 Watching Control and Status Registers and Pins

The Workbench recognizes the control and status register (CSR) and pin names described in the *Intel®IXP2400 /IXP2800 Network Processor Programmer's Reference Manual*. If your project contains multiple chips, you are prompted to select which chip's register to watch. Similarly, if the register is Microengine-based, you are prompted to select which Microengine register to watch.

Note: The Workbench now supports setting data watches on the PCI CSRs, but only in hardware debugging mode.

The CSR categories are:

- CAP
- Microengine
- Memory
- MSF
- PCI (only in hardware debugging mode)
- XSCale core
- Microengine Memory
- MSF Buffers

The Workbench also collects history for:

- Local Memory
- CAM
- Microengine CSRs
  - — T_INDEX
  - — NN_PUT
  - — NN_GET
  - — ACTIVE_LM_ADDR_0_BYTE_INDEX
  - — ACTIVE_LM_ADDR_1_BYTE_INDEX
  - — CTX_ENABLES

Named Elements:

To add a data watch by selecting a named element from a list:

1. Click **Add Watch** or right-click in the **Data Watch** window and click **Add Watch** from the shortcut menu.
   The **Add Data Watch** dialog box appears.
2. Click the category of named element that you want listed.
   A list of recognized element names appears on the right.
3. In the list, select one or more elements you want to watch.

Note: Use the left mouse button in combination with the SHIFT or the CTRL keys to select multiple elements.

4. Click **Add Watch** to have a watch added for each selected element.
   If you select from the list of MicroEngine CSRs, you are prompted with a dialog box to select in which Microengine you want the watch to be done. Similarly, if you select from a list of chip-specific elements, such as the Memory CSRs, and your project contains multiple chips, you are prompted to select a chip.
5. When you are finished adding your watches, click **Close**.

2.12.12.4 Watching General Purpose and Transfer Registers

To watch general purpose registers (GPRs) and transfer registers whose symbols are defined in your microcode:

1. Open the thread window containing the microcode.
2. Right-click the register name.
3. Click **Set Data Watch for**: from the shortcut menu.

Alternatively, you can add a watch by selecting a register name from a list:

1. Click **Add Watch** or right-click in the **Data Watch** window and click **Add Watch** from the shortcut menu.

   The **Add Data Watch** dialog box appears.
2. Select the chip and Microengine in which you want to watch registers.
3. Select whether you want GPRs listed by selecting or clearing **List GPRs**.
4. Select whether you want transfer registers listed by selecting or clearing **List Transfer Regs**
5. Select whether you want Next Neighbor registers listed by selecting or clearing **List Neighbor Regs**.

   Based on your selections, the relative registers are listed in the **Relative registers** list box and the absolute registers are listed in the **Absolute registers** list box.
6. Select one or more registers from either or both lists and click **Add Watch** to add watches for the selected registers.

   If you select relative registers, then you must specify which threads you want to watch by selecting or clearing the four check boxes beneath the relative registers list.
7. When you are finished adding your watches, click **Close**.

Note: GPR names cannot be directly typed into the data watch window.

Note: The Workbench will only display a data value for a register being watched when it is in range. If the physical register associated with a symbolic register gets re-used and goes out of range, the Workbench will display a data value along with **(?)**. If the register is not assigned to a physical register at the currently executing instruction, the symbolic register cannot be read and the message **“out of live range”** will appear.

Aggregate Register Support:

The Assembler supports the declaration and use of registers using array notation. The Workbench handles the square brackets in register names:

- The user can establish a data watch on a single register or on the entire “array” of registers that share the same aggregate name. When the watch is for a single register, the only change is that the array index notation now appears as part of the register name. When the watch is for the entire “array” of registers, then an expandable item with the aggregate name (e.g. “a”) is added along with some number of register sub-items with indexed names (e.g. “a[0]”). Sub-items are added for register array elements zero through the highest referenced register name. In the above example, there would be three sub-items created (0, 1, 2), not four sub-items, since the register named “a[3]” was never referenced in the source code. And since “a[0]” was also never referenced in the source code, it’s data watch value always shows “out of scope”.

- When the user hovers (using mouse in the thread window list view) over a register name with array notation, the resultant datatip always shows the value of the individual register. If the user hovers over the register declaration, no value is shown since the declaration itself does not match a valid register name (i.e. “a[3]” is the highest valid indexed register for the declaration “.reg a[4]”).
- When the user right clicks the mouse on a register name with array notation, there are two data watch options shown; one to add a data watch on the individual register and another to add a data watch on the entire “array” of registers that share the same aggregate name (e.g. “a”) as described in item 1 above.
- The Add Data Watch dialog’s Microengine Registers page shows all referenced registers that use array notation (e.g. “a[1]”, “a[2]”) as well as an entry that represents the entire register array (e.g. “a”). The user can add a data watch on the individual registers and on the entire array of registers that share the same aggregate name as described in item 1 above.

The above description uses relative GPRs in the examples. The description applies equally to absolute GPRs and neighbor registers. Although transfer registers can also include array notation in their names, there is no support for adding a data watch on an array of transfer registers. Instead, the transfer order directive establishes the relationship between transfer registers, whether or not their names include array notation. A data watch added on a transfer register that is a member of a transfer order shows the associated registers in the transfer order as an expandable sub-item of the read and/or write side of the transfer register.

2.12.12.5 Deleting a Data Watch

To delete a data watch:

- Right-click the watch to be deleted in the **Data Watch** window, and click **Delete Watch** from the shortcut menu, or

  Select the watch to be deleted, then, on the **Debug** menu, click **Data Watch**, then click **Delete**, or

  Select the watch to be deleted and press DELETE.

2.12.12.6 Changing a Data Watch

To change a data watch:

1. Right-click the watch to be changed in the **Data Watch** window and select **Edit Name** on the shortcut menu, or

   Select the data watch whose name you want to change, then, on the **Debug** menu, click **Data Watch**, then click **Edit Name**, or

   Double-click the name to be changed.
2. Type the state name.
3. Press ENTER.

2.12.12.7 Changing the Data Watch Radix

To select whether watch values are displayed in decimal or hexadecimal:

1. Right-click anywhere in the **Data Watch** window.
2. Click **Hexadecimal Data** on the shortcut menu to display values in hexadecimal format or clear it to display in decimal format.

2.12.12.8 Depositing Data

To change the value of a simulation state in the **Data Watch** window:

1. Right-click the value to be changed and click **Edit Value** on the shortcut menu, or

   Select the data watch whose value you want to change, then, on the **Debug** menu, click **Data Watch**, then click **Edit Value**, or

   Double-click the value to be changed.
2. Type the new value in either hexadecimal or decimal format and press ENTER. Hexadecimal values must be preceded by a '0x'.

Note: In Hardware mode, you cannot change the contents of the FIFO elements and certain CSRs.

2.12.12.9 Breaking on Data Changes

In Simulation mode, you can halt microcode execution when a state's value changes.

A red dot (the breakpoint symbol) appears before each state on which a break-on-change is set.

You can set and remove a break-on-change on aggregate state (an array) or on its individual elements. Setting or removing a break-on-change on an aggregate state affects all its elements. If some but not all of an aggregate state's elements have break-on-change set, then a half-filled breakpoint symbol is displayed on the aggregate state.

When the value changes for a state on which a break-on-change is set, microcode execution halts and a message box is displayed containing the state name along with its old and new values.

Note: Breaking on data changes is not supported in Hardware mode.

Set:

To break execution on a changed data value:

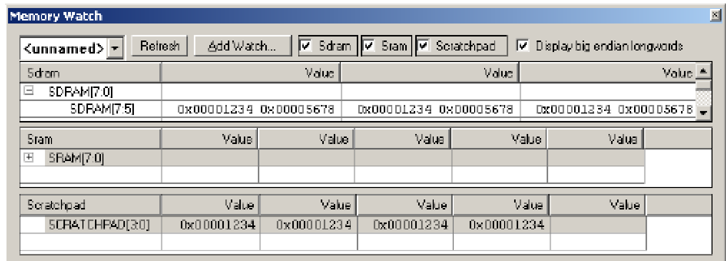
1. Right-click the name or value of the state and click **Set Break On Change** on the shortcut menu, or click the name or value of the state.
2. On the **Debug** menu, click **Data Watch**, then click **Set Break On Change**.

Remove:

To remove a break-on-change:

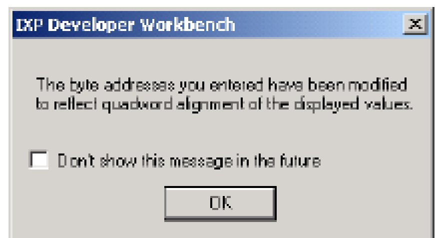
1. Right-click the name or value of the state and click **Remove Break On Change** on the shortcut menu, or click the name or value of the state.
2. On the **Debug** menu, click **Data Watch**, then click **Remove Break On Change**.

2.12.13 Memory Watch



In debug mode, you can monitor the values of DRAM, SRAM, and Scratchpad memory locations using the **Memory Watch** window.

The IXP2nnn Network Processors address memory in bytes, thus the **Memory Watch** window interprets the address to be watched as a byte-aligned. An SRAM or scratchpad byte address will be rounded to the next lower longword and the data will be displayed in longwords. A DRAM byte address will be rounded to the next lower quadword and the data will be displayed in quadwords. For example, if the user specifies a watch of dram[3:8], the watch is shown as dram[0:15]



When the user enters the address in the **Add Memory Watch** window and clicks the **Add watch** button, the following message box pops up to inform the user of the byte alignment adjustment. You may disable the message box by clicking the **Don't show this message in the future**.

Visibility:

To toggle visibility of the **Memory Watch** window:

- On the **View** menu, click **Debug Windows**, then click **Memory Watch**, or

  Click the  button on the **View** toolbar.

Chip Selection:

You select which chip's memory is watched using the list in the upper left corner of the **Memory Watch** window. Chip selection is synchronized with chip selection in the **History**, **Thread Status** and **Queue Status** windows.

Subwindows:

The **Memory Watch** window comprises three subwindows, one for each memory type. Each memory type has a check box at the top of the **Memory Watch** window to control visibility of the subwindow.

Each subwindow contains a multicolumn tree. The first column contains the range of the location(s) being watched, e.g. DRAM[0:3]. The values of the locations are displayed in columns 1 through n. The number of value columns varies with the width of the **Memory Watch** window, with a minimum of one value column.

Endianness:

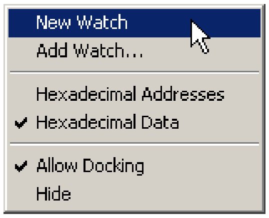
The **Display big endian longwords** check box enables DRAM quadword values to be displayed with the longwords swapped.

Values Updates:

Watch values are updated whenever microcode execution stops. To force an update of the values at other times, click **Refresh** at the top of the **Memory Watch** window.

2.12.13.1 Entering a New Memory Watch

To enter a new memory watch:



1. Right-click anywhere in the name or value column of the **Memory Watch** subwindow (either SRAM. DRAM, or Scratchpad) and click **New Watch** on the shortcut menu, or

   Double-click the blank entry at the bottom of the data watch list for that subwindow.
2. Type the range of memory locations you would like to watch.

   The format for the range is the same as that for the Transactor:

[m] To watch a single location. For example, sram[1] watches location one in SRAM.

[m:n] To watch locations m through n inclusive. For example, dram[0:3] watches locations 0 through 3 in DRAM. And since a range can be specified in ascending or descending order, you could specify this watch as dram[3:0].

[m:+n] To watch location m plus the n locations following it. For example, dram[5:+3] watches locations 5 through 8.

3. Optionally, you can specify a bit range to be watched. The format for a bit range is:

<m> To watch only bit m of a state. For example, dram[0]<12> watches only bit 12 of the dram location 0.

<m:n> To watch bits m through n of a state, with m being greater than n. For example, dram[0:3]<12:10> watches bits 12 through 10 of dram locations 0 through 3.

Note: SRAM and Scratchpad locations are 32-bit values, while DRAM locations are 64-bit values.

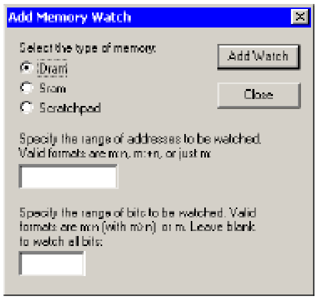
2.12.13.2 Adding a Memory Watch

To add a memory watch:

1. Click **Add Watch** at the top of the **Memory Watch** window, or

   Right-click in the name or value column window and click **Add Watch** on the shortcut menu.

   The **Add Memory Watch** dialog box appears.
2. Select **Sram**, **Dram**, or **Scratchpad** under **Select the type of memory**.
3. Type the range of addresses to be watched in the **Select the range...** box.



You can specify a range of bits to be watched in the **Specify the range of bits...** box. By default, the entire location is watched.

4. Click **Add Watch**.

   When you are finished adding your watches,

5. Click **Close**.

2.12.13.3 Deleting a Memory Watch

To delete a memory watch:

1. Right-click the watch to be deleted in the **Memory Watch** window.
2. Click **Delete Watch** on the shortcut menu.

   or

1. Select the watch to be deleted.
2. Press DELETE.

2.12.13.4 Changing a Memory Watch

To change a memory watch:

1. Right-click the watch to be changed in the **Memory Watch** window and click **Edit Address** on the shortcut menu, or

   Double-click the mouse button on the watch to be changed.

2. Edit the address range (and bit range, if applicable).
3. Press ENTER.

2.12.13.5 Changing the Memory Watch Address Radix

To select whether memory addresses are displayed in decimal or hexadecimal:

1. Right-click anywhere in the **Memory Watch** window.
2. Click **Hexadecimal Addresses** on the shortcut menu to display addresses in hexadecimal format or clear it to display in decimal format.

2.12.13.6 Changing the Memory Watch Value Radix

To select whether memory values are displayed in decimal or hexadecimal:

1. Right-click anywhere in the **Memory Watch** window.
2. Click **Hexadecimal Data** on the shortcut menu to display values in hexadecimal format or clear it to display in decimal format.

2.12.13.7 Depositing Memory Data

Simulation Mode:

In Simulation mode, you can change the value of any entry in the **Memory Watch** window.

Hardware Mode Restrictions:

In Hardware mode, you can change only those entries that were added as a single memory location. You cannot change any value that is displayed as a result of adding a range of memory locations.

Values Changes:

To change the value of a memory location that you are watching:

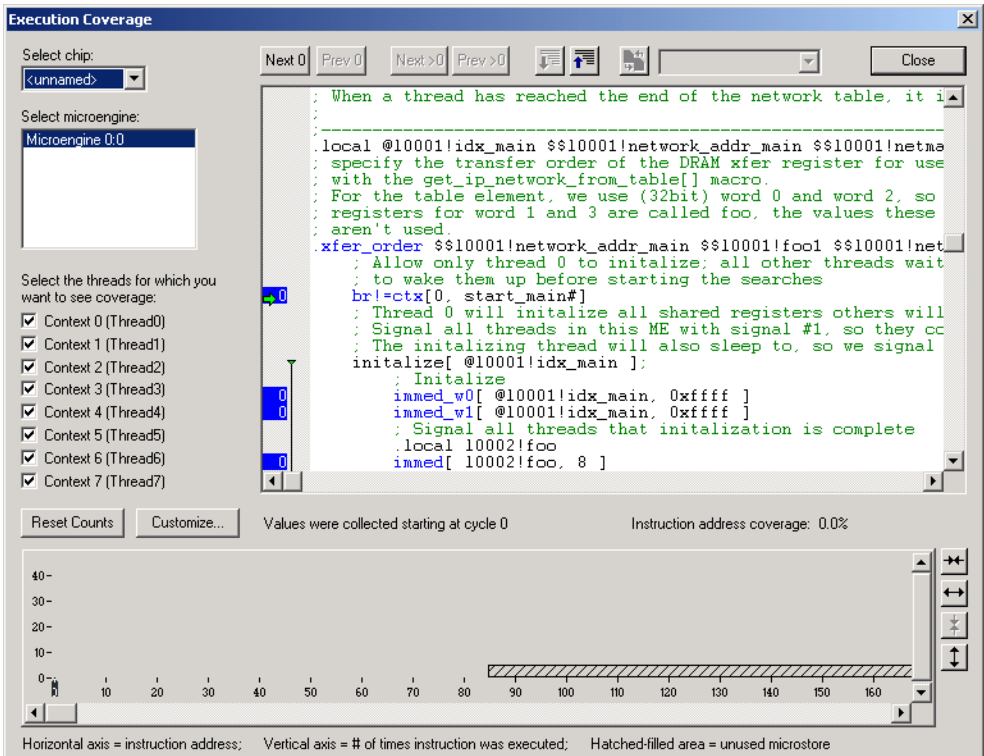
Breaking on Memory Changes:

1. Right-click the value to be changed and click **Edit Value** on the shortcut menu, or
   Double-click the value to be changed.
2. Type the new value in either hexadecimal or decimal format. (Hexadecimal values must be preceded by a '0x'.)

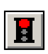
2.12.14 Execution Coverage

The code window in the **Execution Coverage** dialog box mimics the display in a thread window. That is, for a Microengine that contains C code, you can select a source view or a list view. In the source view you can select which source to display. In the source view, source lines show the sum of the number of times each generated instruction was executed. For example, if a source line generates three instructions and each instruction was executed 10 times, then the execution count displayed on that source line would be 30. Similarly, for assembled threads, the execution count for a collapsed macro reference is the sum of the execution counts for all instructions generated by the macro. The units for the horizontal axis on the bar graph remain instruction addresses.

Figure 34. The Execution Coverage Window



To display the **Execution Coverage** dialog box (see Figure 34):

1.  Stop simulation (if necessary).
2. On the **Simulation** menu, click **Execution Coverage**.

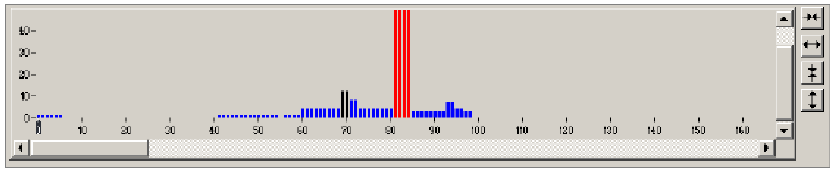
   The **Execution Coverage** dialog box appears.
3. If your project contains multiple chips, select the chip whose coverage data you wish to view from **Select a chip** list in the upper left corner of the dialog box.
4. Select a Microengine from the **Select a Microengine** list.

The microcode that is loaded in that Microengine appears in the code window. This display is the same as is shown in the thread window.

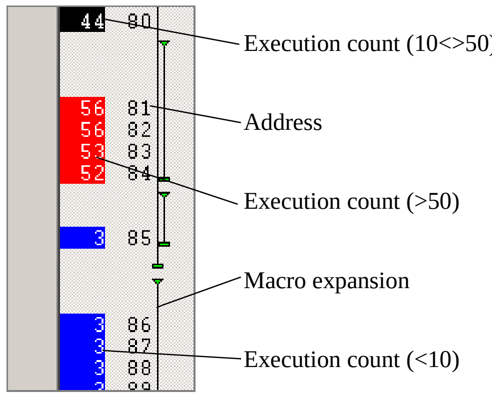
Execution Count:

The number to the left of each instruction displays the number of times each instruction was executed. The background is color-coded to indicate a range of execution counts. (see Section 2.12.14.1).

Bar Graph



At the bottom of the dialog box is a bar graph that shows the execution coverage. The instruction addresses are represented along the horizontal axis, and the execution counts are represented by the vertical axis. The bars are color-coded using the same colors and ranges as in the code window.



By default, the execution counts are the total for all contexts in the Microengine. You can see the execution counts for any subset of contexts by selecting or clearing the check boxes beneath the Microengine list.

The execution count for a collapsed macro reference is the sum of the execution counts of all instructions generated by the macro.

2.12.14.1 Changing Execution Count Ranges and Colors

By default, the colors and ranges for execution counts are:

Blue - Instruction was executed less than 10 times.

Black - Instruction was executed between 10 and 50 times, inclusive.

Red - Instruction was executed more than 50 times.

To change the colors and ranges, in the **Execution Coverage** window, click **Customize**.

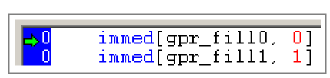
2.12.14.2 Displaying and Hiding Instruction Addresses

To toggle displaying and hiding instruction addresses in the code window:

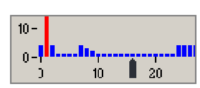
1. Right-click in the microcode window.
2. Select **Display Instruction Addresses** on the shortcut menu to display the addresses or clear to hide the addresses.

2.12.14.3 Instruction Markers

To synchronize viewing between the code window and the bar graph, the Workbench displays an instruction marker. The ‘current’ instruction is marked in the code window by a horizontal green arrow in the leftmost gutter.



In the bar graph window, it is marked by a black vertical marker on the horizontal axis.



Marker Movement:

Above the code window are four buttons that move the instruction marker:

Moves the marker to the next instruction that has an execution count of 0.

Moves the marker to the previous instruction that has an execution count of 0.

Moves the marker to the next instruction that has an execution count that is greater than 0.

Moves the marker to the previous instruction that has an execution count that is greater than 0.

You can also double-click an instruction in the code window or an address in the bar graph window and the marker moves to that instruction.

2.12.14.4 Miscellaneous Controls

Other controls in the **Execution Coverage** dialog box are:

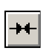
Expand macros (see Section 2.12.8.6).

Collapse macros (see Section 2.12.8.6).

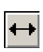
Select the source file to view.

2.12.14.5 Scaling the Bar Graph

To the right of the bar graph window are four buttons for scaling. They are:



Horizontal zoom out.



Horizontal zoom in.

Vertical zoom out.

Vertical zoom in.

2.12.14.6 Resetting Execution Counts

By default, execution counting starts at the first simulation cycle. If you have initialization code that you don't want included in the counts:

1. Run the simulation until it completes the initialization.
2. On the **Simulation** menu, click **Execution Coverage**.
   The **Execution Coverage** dialog box appears.
3. Click **Reset Counts**.

Execution counting restarts from the cycle at which you clicked **Reset**.

2.12.15 Performance Statistics

The Workbench provides the ability to gather and display statistics on simulation performance. Statistics gathering is available only in Simulation mode.

2.12.15.1 Displaying Statistics

To display **Performance Statistics**:

1.  Stop debugging (if necessary).
2. On the **Simulation** menu, click **Statistics**.

   The **Performance Statistics** information box appears.
3. Click the **Summary** tab.

   The **Summary** page shows the percentage of time that each Microengine and memory unit is active and the rate that this activity represents.
4. Click the **Microengine** tab.

   The Microengine statistics page contains a multicolumn hierarchical tree displaying the statistics. The first column identifies the component for which the statistics apply. The next four columns show the percentage of time that the component was executing, aborted, stalled, idle, and swapped out. You can expand and collapse the tree by clicking on the + sign to the right of a component, or by double-clicking on the component.
5. Click the **All** tab.

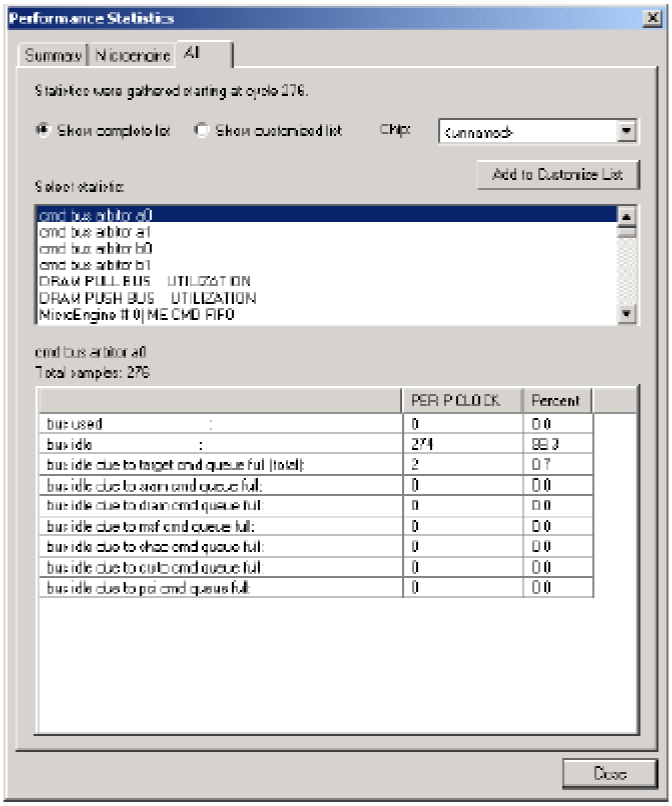
   The **All** statistics page displays as shown in Figure 35.

The **All** statistics page allows you to look at all of the statistics gathered by the Transactors. By default, all available statistics titles are listed in the top list box. Click a title to have the associated statistics displayed in the bottom list box. The **Chip** list box allows you to select which chip's statistics are displayed. You can create a customized list of statistics titles by selecting the title and clicking **Add to Customized List.**

- To display your customized list, click **Show customized list.**
- To delete a title from your list, click **Remove.**

The Workbench saves your customized list along with the project debug settings.

Figure 35. Performance Statistics - All Tab



2.12.15.2 Resetting Statistics

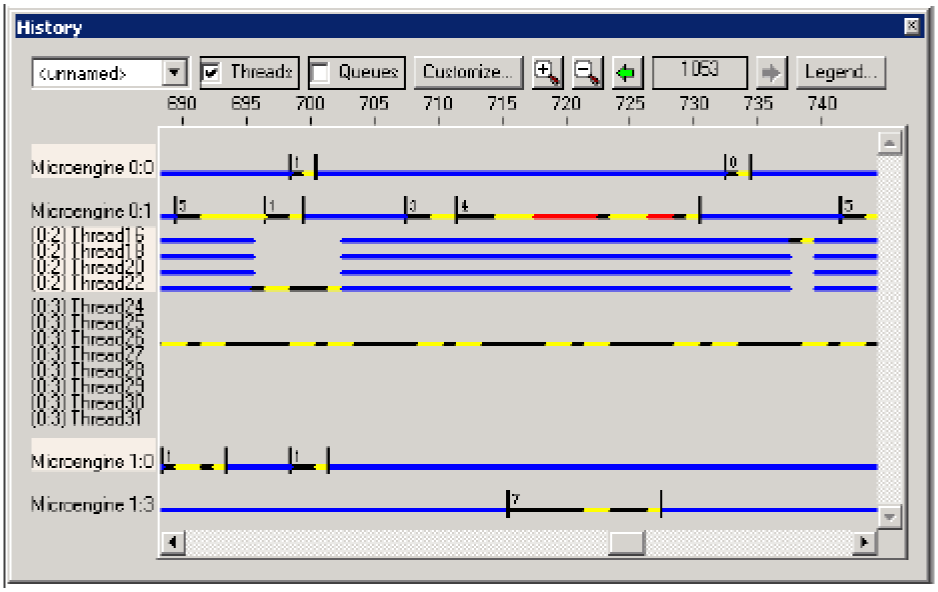
By default, statistics are gathered starting at cycle 1 of a simulation. However, you can reset the statistics at any time. The statistics are then gathered from the current cycle forward.

- To reset statistics, on the **Simulation** menu, click **Reset Statistics**.

2.12.16 Thread and Queue History

Thread and queue history enables you to look at the status of all threads and numerous queues in a chip at the same time. It provides a high level view of how your microcode is executing, enabling you to quickly locate performance bottlenecks.

Figure 36. History Window



The **History** window does not display each thread on a separate line by default. It displays all threads in a Microengine on the same line. To display the threads separately:

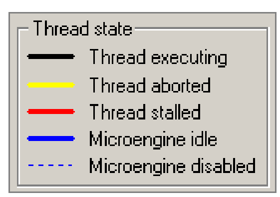
1. Right-click the Microengine whose threads you want to display.
2. Click **Expand Threads for Microengine *n***, where n is the Microengine you clicked on, or double click on a Microengine name.

Note: Only Microengines that have microcode loaded appear in the left-hand column.

To display all the threads in a Microengine on a single line:

1. Right-click on any of the thread in the Microengine.
2. Click **Collapse Threads for Microengine *n***, or double click on a Thread name.

Threads (and queues) appear on a timeline that represents the number of cycles executed. A thread's history is depicted by line segments that change color depending on whether an instruction is executing (black); aborted (yellow); stalled (red); its Microengine is idle (blue); its Microengine is disabled (dotted blue).



You select which chip's history is displayed using the box in the upper left corner of the **History** window. Chip selection is synchronized with chip selection in the **Thread Status**, **Queue Status** and **Memory Watch** windows.

2.12.16.1 Displaying the History Window

- On the **View** menu, click **Debug Windows**, then select **History**, or

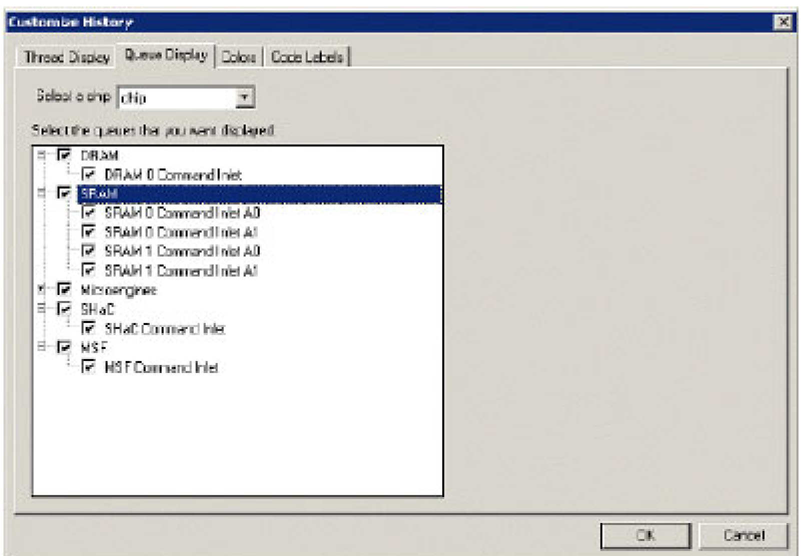
  Click the button on the **View** toolbar.

This toggles visibility of the **History** window. You must be in debug mode to view history.

2.12.16.2 Displaying Queues in the History Window

The **Queue Display** tab on the **Customize History** property sheet of the **History Window** allows for the hiding and showing of queues in the queue history section of the **History Window**. There are five queue groups - DRAM, SRAM, Microengines, SHaC and MSF - corresponding to major units in the chip. Each group expands to display the individual queues in that group. Checking or unchecking a group box checks or unchecks the boxes for all queues in that group. If all queues in a Microengine have their boxes checked for a given item, then the group's box is also checked. Conversely, if they are all unchecked then the group's box is unchecked. If some are checked and some are unchecked, then the group's box is shown as checked but grayed. If the user clicks on a grayed group box, it and all the contained queue's boxes become checked.

Figure 37. Queue Display Property Sheet



2.12.16.3 Hardware Debugging Restrictions

When debugging in the hardware configuration, thread and queue history is not supported.

2.12.16.4 Scaling the Display

To control the horizontal scale for the history display, use the zoom in and zoom out buttons.

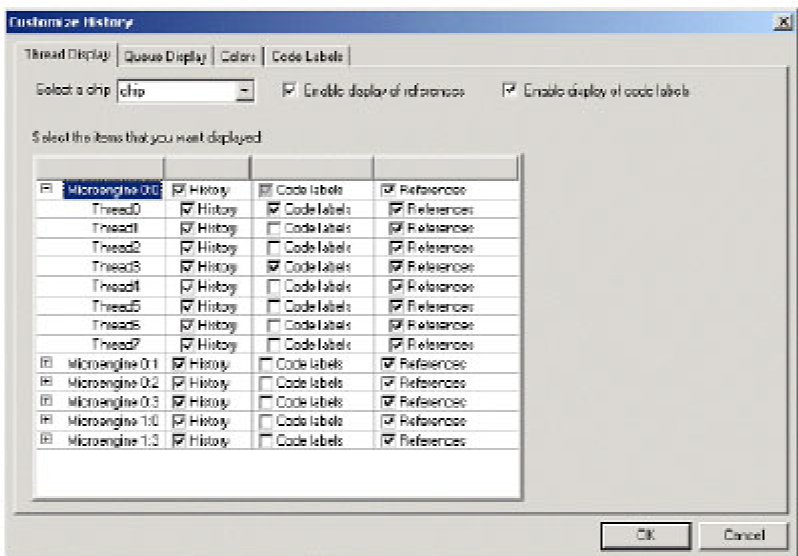
2.12.16.5 Thread Display Property Page

The **Thread Display** property page allows for convenient hiding/showing of threads, code labels and references from the **Customize Thread History** property sheet.

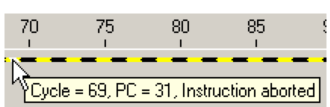
A checked box next to an item means that item gets displayed. An unchecked box means that the item is hidden. Checking or unchecking a box for a Microengine affects all the threads in that Microengine. If all threads in a Microengine have their boxes checked for a given item, then the Microengine's box is also checked. Conversely, if they are all unchecked then the Microengine's box is unchecked. If some are checked and some are unchecked, then the Microengine's box is shown as checked but grayed. If the user clicks on a grayed Microengine box, it and all the contained thread's boxes become checked.

Regardless of the state of the individual thread settings, the user can hide all references by unchecking the **Enable display of references** box. Similarly, all code labels can be hidden by unchecking the **Enable display of code labels** box.

Figure 38. Display Threads Property Page



To get details about a thread's history, position the cursor over the history line and wait for a moment. The Workbench displays the cycle count, PC, and the instruction state in a pop-up window beneath the cursor.



2.12.16.6 Displaying Code Labels

The thread history window supports the viewing of microcode labels along a thread's history line. This helps in determining what code is being executed during a certain cycle.

To specify which code labels to displayed, do the following:

1. In the **History** window, click **Customize**.

   The **Customize Thread History** dialog box appears.
2. Click the **Code Labels** tab.
3. In the **Select Microengine** box is a list of all the Microengines in the project. Click a Microengine to display all the labels in the microcode associated with that Microengine.
4. In the All labels in MicroEngine's microcode box, select the labels to be displayed in the History window by clicking the label, and then clicking Add.

   The **Labels to be displayed in thread history** box lists all the code labels you have selected to be displayed on the selected Microengine's history lines.
5. Continue this procedure for each Microengine for which you want code labels displayed.
6. To delete a label from the display list, select the label in the rightmost list box and then click the  button.
7. Click **OK** to close the **Customize Thread History** dialog box.

Code labels for a thread

Whether or not code labels are displayed on a particular thread's history line is controlled via shortcut menus in the **History** window or by the Thread Display property page (see Figure 38).

- To display code labels for a thread, right-click on the thread name or on its history line and check **Display Code Labels for** 'threadname' on the shortcut menu, where 'threadname' is the name of the thread you clicked on.
- To hide code labels for a thread, right-click and uncheck **Display Code Labels for <threadname>**.

2.12.16.7 Displaying Reference History

The **Thread History** window supports the viewing of reference history lines underneath the history lines of the thread issuing the reference.

References to the following components are displayed:

- DRAM
- SRAM
- Hash
- Scratchpad
- MSF
- CAP

Interthread references are displayed in two sections:

- The reference creation section is displayed underneath the signalling thread, with the name of the signalled thread shown above the reference line.
- The reference consumption section is displayed underneath the signalled thread, with the name of the signalling thread shown above the reference line.

There are five instances when reference history is displayed:

1. A thread issues a command and no signaling occurs.

2. A thread issues a command and gets signaled. The reference line under the issuing thread shows the referenced component and displays the markers for all the reference events that occurred. For example,.

3. A thread issues a command but a different thread gets signaled. The reference line under the issuing thread shows the referenced component and the number of the thread getting signaled. None of the events are indicated on the reference line. The reference line under the thread being signaled shows the referenced component and the number of the thread that issued the command. It also displays the markers for all the reference events that occurred. For example, if thread 30 issues a DRAM command and specifies that thread 62 gets signaled, then the reference line under thread 30's history looks like:

And the reference line under thread 62's history looks like:.

4. A thread issues a command and a different thread and the issuing thread both get signaled.
5. A thread signals another thread directly. For example, if thread 63 signals thread, then the reference line under thread 63's history looks like:

And the reference line under thread 15's history looks like:

To get details about a reference:

- Position the cursor over the reference line and wait for a moment.

  The Workbench displays the reference command, the address it is accessing, and the number of longwords being referenced in a pop-up window beneath the cursor.

The color in which each of these types of references is displayed can be customized as outlined in the section, "Changing Thread History Colors".

On a reference line, the Workbench displays markers at the cycles when reference events occur.

Displayed at the cycle when the reference is put into the queue of the unit being referenced.

Displayed at the cycle when the unit removes the reference from the queue.

Displayed at the cycle when the unit finishes processing the reference.

Displayed at the cycle when the unit signals the thread that the reference is completed. For DRAM references, there are two signals. If they occur simultaneously, the arrow is filled grey.

Shown from the reference line to the thread's history line at the cycle when the thread consumes the signal.

Displaying References:

Whether or not references are displayed on a particular thread's history line is controlled via shortcut menus in the **History** window or via the customize property sheet.

To display references for a thread:

1. Right-click the thread name or its history line.
2. Click **Display References** for 'threadname' on the shortcut menu, where 'threadname' is the name of the thread you clicked on.

The user controls the display of reference history by clicking the **Customize** button in the **History** window then clicking the **Thread Display** tab. The property page shown in Figure 39 appears. Display of all references is enabled or disabled by checking or unchecking the **Enable display of references** button. If this button is checked, then references for individual Microengines or threads can be displayed or hidden using the check boxes in the fourth column of the list box.

To hide all references for a Microengine, the user unchecks the box corresponding to that Microengine, as was done for Microengine 0:2 in Figure 39.

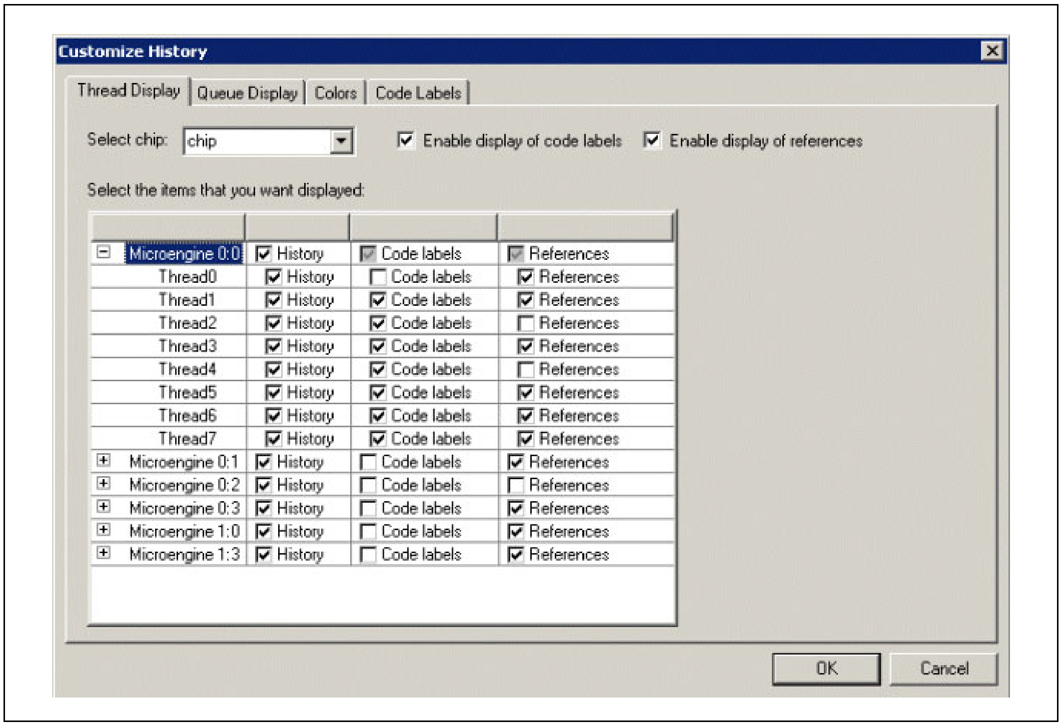
To display all references for a Microengine, the user checks the box corresponding to that Microengine, as was done for Microengine 0:1 in Figure 39.

To hide references for a thread, the user unchecks the box corresponding to that thread, as was done for Thread 2 in Figure 39. To display references for a thread, the user checks the box corresponding to that thread, as was done for Thread 0 in Figure 39. If some threads in a Microengine have references displayed and others don't, then the box for the Microengine is displayed as checked but grayed.

Checking or unchecking a Microengine's box also checks or unchecks the boxes for all thread in that Microengine.

Hiding or displaying references for a thread can also be done directly within the **History** window. If a thread's references are hidden, the user displays them by right-clicking on the thread's history line and selecting **Display References for Threadn**. If a thread's references are displayed, the user hides them by right clicking on the thread's history line and selecting **Hide References for Threadn**.

Figure 39. Customize History



2.12.16.8 Queue History

You can control whether or not a specific queue's history is displayed. This allows you to limit the display to only those queues that you are interested in. By default, all queues are displayed.

Queue's Contents:

To get information about a queue's contents:

1. Position the cursor over a vertical bar and wait for a second.
2. The Workbench displays the number of entries in the queue and the size of the queue in a pop-up window beneath the cursor.

Queue's Contents in Detail:

To get detailed information about the contents of the queues:

- Right-click and click **Show Queue Status** on the shortcut menu.

The Queue Status window appears showing details of all the queues. (See Section 2.12.17 for more information on the Queue Status window.)

2.12.17 Queue Status

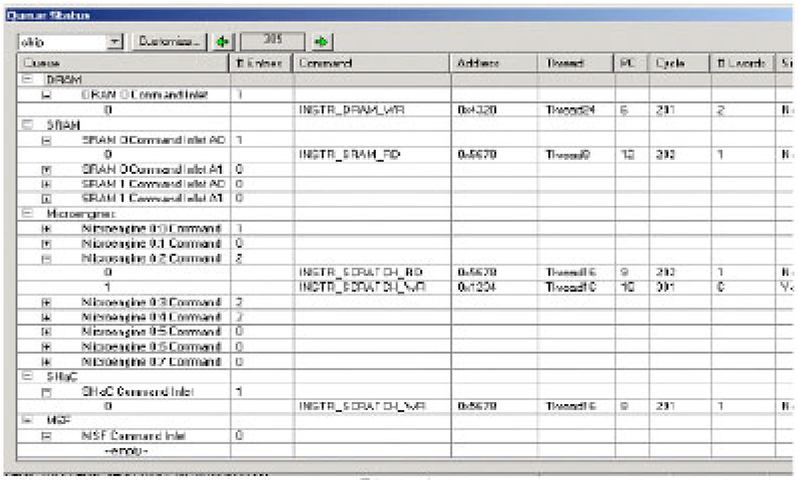
The queue status window provides current and historical information on the contents of the five queue groups: DRAM, SRAM, MSF Interface, SHAC and Microengines.

- To display the Queue Status window, on the **View** menu, click **Debug Windows**, then click **Queue Status**, or

  Click the button on the **View** toolbar.

  The Queue Status window appears (see Figure 2-17).

Figure 40. Queue Status Window



Select which chip's queue status is displayed using the list in the upper left corner. Chip selection is synchronized with chip selection in the **Memory Watch**, **Thread Status** and **History** windows.

The number of entries in the queue is displayed for each queue in the DRAM, SRAM, MSF Interface, SHAC and Microengines.

To examine the references that are in a queue, expand the queue's tree item by clicking on the + symbol or double-clicking on the item. For each reference in the queue, the Workbench displays:

- The type of reference.
- The address being referenced.
- Which thread made the reference.
- The PC of the instruction which made the reference.
- The cycle count at which the reference was made.
- The number of longwords being referenced.
- Whether a signal will be generated when the reference is completed.

Instruction Cross-reference:

If you right-click a reference and click **Go To Instruction** on the shortcut menu, the Workbench opens the appropriate thread window and displays the microcode instruction that issued the reference. A purple marker in the left margin of the thread window marks the instruction. The reference is also highlighted with the same marker in the queue status window.

2.12.17.1 Queue Status History

When the simulation stops, the queue status window is automatically updated to show the current queue contents. You can also review the contents of the queues for previously executed cycles.

To do this:

- Click the right and left arrows at the top of the queue status window.

The number between the arrows shows the cycle count at which the queue contents occurred. This historical cycle count is synchronized with the corresponding cycle count in the **History** window. Changing either one also changes the other. This allows you to move the graphical cycle marker in the **History** window to a specific cycle and view the queue contents in relation to the thread history.

2.12.17.2 Setting Queue Breakpoints

You can set a breakpoint on a queue to have simulation stop when the queue rises to or falls below a specified threshold.

To do this:

1. Right-click the queue name and click **Insert/Remove Breakpoint** on the shortcut menu.
   The **Queue Breakpoint** dialog box appears.
2. By default, the breakpoint is enabled and triggers when the queue rises to the default threshold for the queue. You can change the trigger threshold by changing the number in the Threshold box.
3. By selecting or clearing the check boxes, you can change the breakpoint properties.

When a breakpoint is set and enabled on a queue, a solid red breakpoint symbol is displayed to the right of the queue name.

To disable a breakpoint:

1. Right-click the queue name.
2. Click **Enable/Disable Breakpoint** on the shortcut menu.
   A disabled breakpoint is indicated by an unfilled breakpoint symbol.

To enable a breakpoint:

1. Right-click the queue name.
2. Click **Enable/Disable Breakpoint** or **Insert/Remove Breakpoint** on the shortcut menu.

To remove an enabled breakpoint:

1. Right-click the queue name.

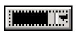
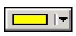
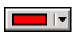
2. Click **Insert/Remove Breakpoint** on the shortcut menu.

To change a breakpoint's properties:

1. Right-click the queue name.
2. Click **Breakpoint Properties** on the shortcut menu.
   The **Queue Breakpoint** dialog box reappears where you can change properties or click **Remove** to remove the breakpoint.

2.12.17.3 Changing Thread History Colors

By default, the Workbench displays a thread history line in:

	if an instruction is executing
	f it is aborted
	if the thread is stalled
	if the Microengine is idle

By default, the reference line colors are:

SRAM	white
DRAM	black
Scratchpad	purple
CAP	green
Hash	bright blue
MSF	yellow

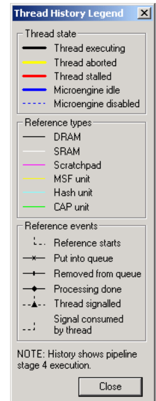
To change the colors for the thread history and reference lines, do the following:

1. In the thread history window, click **Customize**, or
   On the **Simulation** menu, click **Simulation Options**.
2. Click the **Colors** tab.
3. Click the color button next to the item whose color you want to change.
4. Select the new color.
5. Click **OK** after specifying the colors you want.

Note: For consistency, the execution state colors also apply to the instruction markers that are displayed in the thread windows.

2.12.17.4 Displaying the History Legend

To see a legend of the thread history colors and the reference event markers, click the **Legend** button in the **History** window. The **Thread History Legend** dialog box appears.



This legend displays information about the following:

- **Thread State**
- **Reference Types**
- **Reference Events**

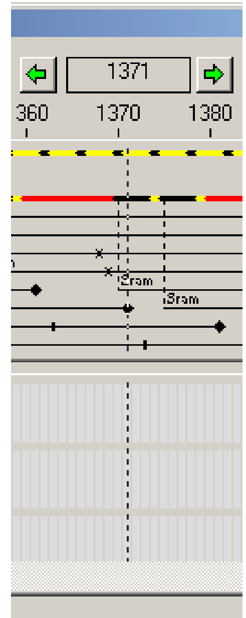
2.12.17.5 Tracing Instruction Execution

To view the instruction that a thread was executing at a given cycle count:

1. Right-click the thread’s history line at the cycle count in which you are interested.
2. Click **Go To Instruction** on the shortcut menu.

This opens or activates the thread's code window and scrolls it to show the requested instruction.

- If the requested instruction is displayed, the history instruction marker  appears on the appropriate instruction line, color coded for the execution state.
- If the requested instruction is not displayed because it is in a collapsed macro reference, the history instruction marker is displayed on the line with the macro reference.
- If the thread is compiled and the source view is being displayed, the history instruction marker is displayed on the line that generates the instruction.
- If the thread is compiled and the list view is being displayed, the history instruction marker is displayed on the appropriate instruction line.



The thread history window contains a cycle marker which marks a particular cycle count using a vertical dashed line cutting across all displayed history lines (see image at left). The cycle count at the cycle marker's position is reported in the box located between the right and left green arrow buttons in the **History** window.

There are several ways to move the cycle marker:

- To immediately move the cycle marker to a given cycle count, double-click the **History** window at the cycle where you want the marker.
- To drag the cycle marker, press the left mouse button on the marker, drag to the desired cycle and release the mouse button. As you drag, the marker snaps to cycle count positions and the cycle count is displayed.
- To move the cycle marker to the next cycle, click the button labeled with the green right arrow.
- To move the cycle marker to the previous cycle, click the button labeled with the green left arrow.

If a thread's code window is opened, movement of the cycle marker scrolls the window to show the instruction being executed by the thread at that cycle count. The instruction is marked with a color-coded arrow in the window's gutter, with the color indicating the execution state - executing, aborted, stalled, or swapped out. This enables you to trace past program flow by going to a specific cycle and incrementing the cycle marker.

2.12.17.6 History Collecting

Disable:

To disable history collecting:

1. On the **Simulation** menu, click **Options**.
   The **Simulation Options** dialog box appears.
2. Click the **History** tab.
3. Clear **Enable history collecting**.

Enable:

To collect history:

1. On the **Simulation** menu, click **Options**.
   The **Simulation Options** dialog box appears.
2. Click the **History** tab.
3. Select **Enable history collecting**.
4. Select:
   a. **Collect thread history**, or
   b. **Collect reference history**
   c. **Collect queue history**, or

d. Any combination, depending on what history you want collected.

Note: You cannot select **Collect reference history** if **Collect thread history** is not selected.

5. Specify how many cycles of history you want collected by typing a number in the corresponding box.

Note: These options must be specified before you start debugging. If you are already in debug mode, these selections are disabled.

2.12.18 Thread Status

The **Thread Status** window provides static or snapshot information on the status of each thread in a selected chip in your project. For each thread, the following information is displayed:

— Current instruction address.

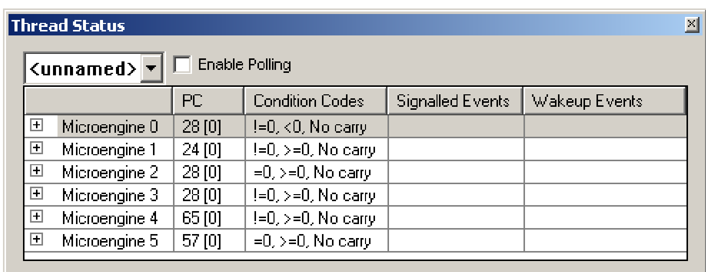
— The list of events for which the thread is waiting.

— The list of events which have been signaled to the thread.

- To display the **Thread Status** window, on the **View** menu, click **Debug Windows**, then click **Thread Status** (see Figure 41), or

  Click the  button on the **View** toolbar.

Figure 41. The Thread Status Window



In each Microengine, an arrow appears to the left of the thread that is currently executing or that is scheduled to resume execution when the Microengine resumes execution.

Select which chip's status is displayed using the box in the upper left corner of the **Thread Status** window. Chip selection is synchronized with chip selection in the memory watch, queue status and history windows.

View:

You can control which threads are displayed by expanding and collapsing the Microengine entries in the status tree. You can expand the tree so that all threads of the selected chip are displayed by right-clicking and selecting **Expand All** on the shortcut menu.

Update:

The status display is updated whenever Microengine execution stops—when you stop execution or when you hit a breakpoint.

Polling:

You can also have the Workbench poll the threads and update the status at regular intervals. To enable or disable thread status polling and to change the polling interval:

1. On the **Debug** menu, click **Status Polling**, or

   Right-click within the **Thread Status** window and click **Status Polling** on the shortcut menu.

   The **Status Polling** dialog box appears.
2. Select **Poll thread status** to enable polling or clear it to disable polling.

Note: You can also enable and disable polling in the **Thread Status** dialog box by selecting or clearing Enable Polling.

Polling Interval:

If you enable polling, specify the polling interval by:

- Typing the number of seconds between polls in the Polling interval (sec) box. You can also use the spin controls to increment or decrement the number in the box.

The value that you type in must be an integer.

2.13 Running in Batch Mode

Workbench Batch Files:

A Workbench batch file is an ASCII text file.

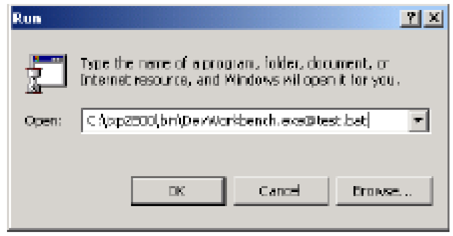
- The first line must contain the complete path for a Workbench project file, for example, c:\mydir\router.dwp.
- The Workbench opens the specified project and performs a build operation.
- The second line must contain the keyword hardware or simulation to specify the debug configuration.
- The Workbench starts debugging in the specified configuration.
- All subsequent lines are executed by the command line interface.
- To have the Workbench exit when it completes executing the batch file, place an exit command as the last line in the batch file.

Here is an example of a batch file:

```
c:\mydir\router.dwp
simulation
sfoobar.ind
go 100
exit
```

Batch Mode:

You can run the Workbench in batch mode by specifying a Workbench batch file preceded by an @ as a program argument when starting the Workbench.



For example:

1. On the Windows task bar, click **Start**, and then click **Run.**
2. Type `c:\ixp2800\bin\DevWorkbench.exe @test.bat` in the **Open** box.
3. Click **OK.**

Windows launches the Workbench and executes the batch file `test.bat`.

Assembler 3

This chapter provides information on running the Assembler. Background information on the Assembler functions appears in the *Intel® IXP2400 /IXP2800 Network Processor Programmer's Reference Manual*.

3.1 Assembly Process

This section describes how to invoke the Assembler and the steps that it goes through in processing a microcode file.

3.1.1 Command Line Arguments

The Assembler is invoked from the command line:

```
uca [options] microcode_file microcode_file...
```

where the options consist of:

-ixp2400	Targets assembly to the IXP2400
-ixp2800	Targets assembly to the IXP2800.
-ixp2850	Targets assembly to the IXP2850.
-ixp2xxx	Targets assembly to all Microengine version 2 (MEv2) processors.

Note: Multiple processor types can be targeted by specifying multiple options; for example: -ixp2800 -ixp2850

-lm start	Define the start of local memory allocation in bytes.
-lm start:size	Define the start and size of local memory allocation in bytes.
-m *file*	Loads the microword definitions from the specified file rather than the default.
-p *file*	Loads the ucc parse definitions from the specified file rather than the default.
-o *file*	Use *file* as the generated list file. This is only valid if there is one microcode_file.
-O	Enables optimization.
-Of	Tries to automatically fix A/B Bank conflicts. Default is disabled.

Note: Note that the two optimization options are independent of each other; in other words any combination can be specified The default, **disabled**, avoids having the assembler **add** code that the programmer did not specify.

-Os	Tries to automatically spill GPRs into local memory. Default is disabled.

-g	Adds debugging info to output file.
-v	Prints the version number of the Assembler.
-h	Prints a usage message (same as -?).
-?	Prints a usage message (same as -h).
-r	Register declarations are not required.
-Wn	Set Warning Level (n=0-4)
-w	Disable warnings (sames as -W0)
-WX	Report error on any warning

The following version arguments allow assembly to be targeted for a specific chip version or range of versions, overriding the default values. The predefined Preprocessor symbols __REVISION_MIN and __REVISION_MAX will reflect the specified version range. In addition, the version range is also written to the .list file in a '.cpu_version' directive.

For the following arguments, *rev is* an upper or lower case letter (A-P) followed by a decimal number (0-15), for example -REVISION_MIN=A1 or -REVISION_MIN=B0, or an eight-bit number where bits <7:4> indicate the major stepping and bits <3:0> indicate the minor stepping, for example, -REVISION_MIN=1 or -REVISION_MIN=0x10

-REVISION=*rev*	Targets assembly to chip version *rev*. This is equivalent to setting options '-REVISION_MIN=*rev*' and '-REVISION_MAX=*rev*' with the same value.
-REVISION_MIN=*rev*	Targets assembly to the minimum chip version *rev*. (The default is 0.)
-REVISION_MAX=*rev*	Targets assembly to the maximum chip version *rev*. (The default is 15, no limit.)

The following options are passed to the preprocessor, UCP:

-P	Preprocess only into a file: microcode_file.ucp
-E	Preprocess only into stdout
-I*folder*	Add the *folder* to the end of the list of directories to search for included files
-D*name*	Define *name* as if the contained "#define *name* 1"
-D*name=def*	Define *name* as if the contained "#define *name def*"
-N	Disable the pre-processor

The **microcode_file** names may contain an explicit suffix, or if the suffix is missing, .uc is assumed. Assembling several files in one command line is equivalent to assembling each individually; the files are not associated with each other in any way.

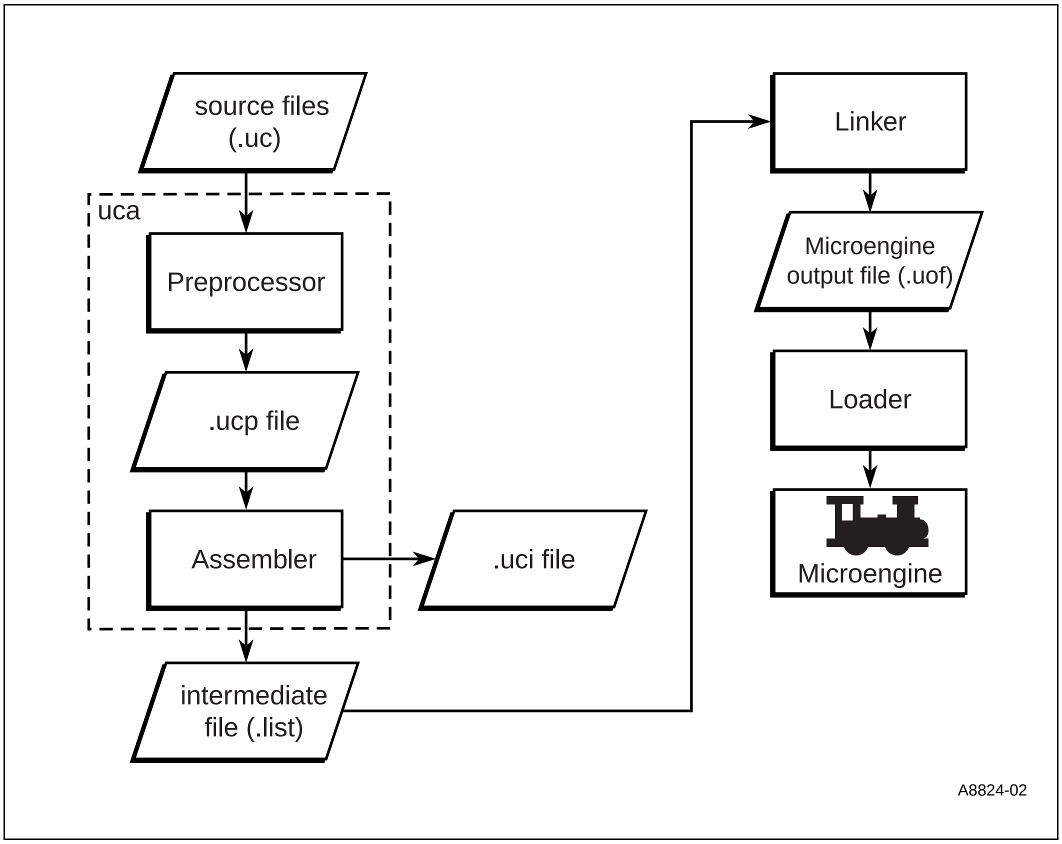
If uca is invoked with no command line arguments, then a usage summary is printed.

In Windows environments, the Assembler may also be invoked through the Workbench (see Section 2.6 for more information on running the assembler in the the Developers Workbench).

3.1.2 Assembler Steps

As shown in Figure 42, invoking the Assembler results in a two-step process composed of a preprocessor step and an Assembler step. The preprocessor step takes a .uc file and creates a .ucp file for the Assembler. The Assembler takes a .ucp file and creates an intermediate file with the file name extension of .uci. The .uci file is used by the Assembler to create the .list file and provides error information that may be used in resolving semantic problems (such as register conflicts) in the input file.

Figure 42. Assembly Process



The .uc file contains three types of elements: microwords, directives, and comments. Microwords consist of an opcode and arguments and generate a microword in the .list file. Directives pass information either to the preprocessor, Assembler, or to downstream components (e.g., the Linker) and generally do not generate microwords. Comments are ignored in the assembly process.

The Assembler performs the following functions in converting the .uc file to a .list file:

- Checks microcode restrictions.
- Resolves symbolic register names to physical locations.
- Performs optimizations (see Section 3.1.4).
- Resolves label addresses.
- Translates symbolic opcodes into bit patterns.

3.1.3 Case Sensitivity

The microcode file is case insensitive, while the command line arguments are case sensitive.

3.1.4 Assembler Optimizations

The assembler optimizer performs the following optimzations:

- It will move instructions down to fill defer shadows for instructions that support the defer token.
- It will remove unnecessary VNOPs.
- It will move instructions down to replace VNOPs that cannot be removed.

For more information, please see the *Intel® IXP2400 /IXP2800 Network Processor Programmer's Reference Manual*.

3.1.5 Processor Type and Revision

Over time, network processors will be released in different types and revisions with different features. Microcode written to take advantage of a particular processor type or revision will fail if it is run on the wrong processor.

To deal with this issue, the user can specify a type and range of revisions for which they want their microcode assembled. This is done using the **`-ixp2400, -ixp2800,-ixp2850, -ixp2xxx, -REVISION_MIN,`** **and** **`-REVISION_MAX`** command line options. For simplicity, the **`-REVISION`** option can be used to set the minimum and maximum to the same value. These options will target the assembly to a particular type and revision of processor. Several predefined preprocessor symbols will be defined according to type and revision.

For more information on the predefined symbols and on writing version specific microcode, please see the *Intel® IXP2400 /IXP2800 Network Processor Programmer's Reference Manual.*

Microengine C Compiler 4

The Microengine supports microcode compiled from C language code to support the Microengines and their threads. You can create the C code using the DWB GUI or any suitable text editor. You can then compile and link the code using the GUI or the Compiler command line.

This chapter explains the subset of the C language supported by the Microengine C Compiler and the extensions to the language to support the unique features of the processor.

For information on the Compiler functions refer to the *Intel® Microengine C Compiler Language Support Reference Manual*.

4.1 The Command Line

You can invoke the command line from a command prompt window on your system. Do the following:

1. Open a command prompt window.
2. Go to the folder containing the C Compiler files, typically:

   C:\IXP 2000\bin>
3. Invoke the C Compiler using this command:

   ```
   uccl [options] filename [filename...]
   ```

4.2 Supported Compilations

Two kinds of compilations are supported:

- To compile one or more C source files (*.c, *.i) into object files (*.obj), use:
  `uccl -c file1.c file2.c ...`
  An object containing intermediate (IL0) code is created for each C source file.
- To compile and link a microengine program, use:
  `uccl file1.c file2.obj ...`
  Use any combination of .c source file and .obj object file pairs.

Note: In the first case, you must use the -c switch in the command line.

Example: uccl -c file1.c file2.i

Note: In the second case, do not use the -c switch.

4.3 Supported Option Switches

Table 4-1 lists and defines all the supported C Compiler command line switches. The CLI warns and ignores unknown options. The CLI honors the last option if it conflicts with the previous one, for example,

```
uccl -c -O1 -O2 file.c
```

generates the following warnings and proceeds:

```
uccl: Command line warning: overriding '-O1' with '-O2'
```

Options that do not take a value argument, such as -E, -c, etc., are off by default and are enabled only if specified on the command line.

Table 3. Supported uccl CLI Option Switches (Sheet 1 of 4)

Switch	Definition
-? -help	Lists all the available options.
-c	Compiles each .c or .i file to a .obj file (rather than compile and link).
-Dname[=value]	Specifies a #define symbol. The value, if omitted is 1.
-DSDK_3_0_COMPATIBLE	Uses the IXA SDK 3.0 version of the hash intrinsics (with the read and write parameters swapped) and removes error checking for generic ("void *") typecasts in intrinsics library parameters. If possible, SDK 3.0 code should be changed to work with the new versions of the hash intrinsics and any generic typecasts should be changed to the correct types.
-E -EP -P	Preprocess to stdout. Preprocess to stdout, omitting #line directives. Preprocess to file.
-Fa<filename>	Produces a .uc file containing the generated microcode intermixed with the source program lines. The resulting assembly file is for reference only; the compiler does not guarantee that the file will pass through the assembler. If an assembler-compatible file is required, the -uc option should be used instead. This may have a negative impact on performance, however; certain optimizations cannot be performed when compiling for the assembler.
-Fo<file> -Fo<Dir/>	Name of object file or directory for multiple files.
-Fe<file>	Base name of executable (.list, .ind) file. Defaults to the base name of the first file (source or object) specified on the command line followed by the extension (.list).
-Fi<file>	Overrides the base name of the .ind file.
-FI<file>	Forces inclusion of file.
-Gx2400 -Gx2800 -Gx2850	Specifies the target processor: IXP2400, IXP2800, or IXP2850. IXP2800 is the default. The compiler adds -DIXP2400, -DIXP2800, and -DIXP2850 respectively.
-I path[;path2...]	Path(s) to include files, prepended before path(s) specified in environment variable UCC_INCLUDE.

Table 3. Supported uccl CLI Option Switches (Continued) (Sheet 2 of 4)

Switch	Definition
-link[linker options]	Calls the microengine image linker (ucld) after successful compilation, passing any specified linker options. The default linker options are: "-u 0 -sc 0x00000004:0x00003ff4 -dr 0x00000010:0x07ffffe8 -sr0 0x00000004:0x03fffffc -sr1 0x00000004:0x03fffffc -sr2 0x00000004:0x03fffffc -sr3 0x00000004:0x03fffffc"
-Obn	Inlining control: n=0, none; n=1, explicit (inline functions declared with __inline or __forceinline (default)); n=2, any (inline functions based on compiler heuristics, and those declared with __inline or __forceinline)
-On	Optimize for: n=1, size (default); n=2, speed; n=d, debug (turns off optimizations and inlining, overriding -Obn below).
-Qbigendian	Compile big-endian byte order (default). Compiler adds -DBIGENDIAN, -ULITTELENDIAN. All other command line BIGENDIAN/ LITTLEENDIAN symbol definitions and undefinitions are ignored.
-Qdefault_sr_channel=<0...3>	Specify the SRAM channel that should be used when allocating compiler-generated SRAM variables and variables that are specified as __declspec(sram). The default is channel 0.
-Qerrata	Report when the compiler-generated code triggers a known processor erratum.
-Qip_no_inlining	Turns off all inter-procedural inlining. Inter-procedural inlining is on by default.
-Qlittleendian	Compiles little endian byte order. Compiler adds -DLITTLEENDIAN -UGIBIGENDIAN. All other command line LITTLEENDIAN/BIGENDIAN symbol definitions and undefinitions are ignored.
-Qliveinfo	Same as "-Qliveinfo=all".
-Qliveinfo=gr,sr,...	Print detailed liveness information for a given set of register classes: gr: general purpose registers sr: SRAM read registers sw: SRAM write registers srw: SRAM read/write registers dr: DRAM read registers dw: DRAM write registers drw: DRAM read/write registers nn: neighbor registers (only when -Qnn_mode=1) sig: signals all: all of the above
-Qlm_start=<n>	Provides a means for user to reserve local memory address [0, n-1] (in longwords) for direct use in inline assembly. Compiler does not allocate any variables to this address range.
-Qlm_unsafe_addr	Disables the compiler's use of local memory autoincrement addressing. Used when user code writes local memory pointers with invalid values.
-Qlmpt_reserve	Reserve local memory base pointer l$index1 for user inline assembly code.
-Qmapvr	In the assembly dump of list file or uc file, this flag prints out the virtual register number for register operands. Should not be used with -uc.

Table 3. Supported uccl CLI Option Switches (Continued) (Sheet 3 of 4)

Switch	Definition
-Qnctx=<1,2,3,4, 5, 6,7, 8>	Specifies the number of contexts compiled to run; defaults to 4, or 8 if -Qnctx_mode=4.
-Qnctx_mode=<4, 8>	Specifies the context mode (either 4 context mode or 8 context mode). Defaults to 4, or 8 if -Qnctx is set to larger than 4. If -Qnctx is set to 1, defaults to 4.
-Qnn_mode=<0, 1>	Sets NN_MODE in CTX_ENABLE for setting up next neighbor access mode. (See Next Neighbor Register section in Chapter 3). 0=neighbor (default), 1 = self).
-Qnolur=<func_name>	Turns off loop unrolling on specified functions. You can supply one or more function names to the option. For example: -Qnolur="_main"; turn off loop unrolling for main(). -Qnolur="_main,_foo"; turn off loop unrolling for main() and foo(). The supplied function name must have the preceding underscore ('_').
-Qold_revision_scheme	Generates hardware revision numbers that are compatible with IXA SDK 3.0 and below.
-Qperfinfo=n	Prints performance information. n=0 - no information (similar to not specifying) n=1 - register candidates spilled (not allocated to registers) and the spill type n=2 - instruction-level symbol liveness and register allocation n=4 - function-level symbol liveness and register allocation n=8 - function sizes n=16 - local memory allocation n=32 - live range conflicts causing SRAM spills n=64 - instruction scheduling statistics n=128 - Warn if the compiler cannot determine the size of a memory I/O transfer n=256 - Display information for "restrict" pointer violations n=512 - Print offsets of potential jump[] targets
-Qrevision_min=n -Qrevision_max=m	The version arguments allow the compiler to generate code that works on a range of processor versions (steppings). 0x00=A0 (default for -Qrevision_min) 0x01=A1 0x10=B0 0x11=B1 The default revision range is 0x00 to 0xff (all possible processor versions). The default for -Qrevision_max is 0xff. The compiler adds -D__REVISION_MIN=n and -D__REVISION_MAX=m. Note: The IXP program loader reports an error if a program compiled for a specific set of processors is loaded onto the wrong processor.

Table 3. Supported uccl CLI Option Switches (Continued) (Sheet 4 of 4)

Switch	Definition
-Qspill=<n>	Selects the alternative storage areas ("spill regions") chosen when variables cannot be allocated to general-purpose or transfer registers: (LM=local memory, NN=next neighbor registers) n=0: LM (most preferred) -> NN -> SRAM (least preferred) n=1: NN->LM->SRAM n=2: NN only; halt if not enough NN n=3: LM only; halt if not enough LM n=4: NN->LM; halt if not enough LM or NN n=5: LM->NN; halt if not enough LM or NN n=6: SRAM only n=7: No spill; halt if any spilling required n=8: LM->SRAM Default is n=0. The user must set -Qnn_mode=1 to use the NN registers as a spill region. If the NN registers are used by program code, NN spilling will be automatically disabled.
-s	Changes the behavior of -uc by not calling uca to assemble the compiler produced assembly code. Only valid when combined with -uc option.
-uc	Mixing C and microcode programming. Under this option, you can compile one or more C files as well as one or more microcode files into one application. The compiler compiles all C files into one microcode file, then sends this microcode file as well as other microcode files to UCA to produce a list file.
-Wn n=0, 1, 2, 3, 4	Warning level. 0=print only errors 1, 2, 3=print only errors and warnings 4=print errors, warnings, and remarks. Defaults to 1.
-Zi	Produces debug information. The compiler generates a file with a .dbg extension for each source.

Table 4. Supported CLI Option Switches

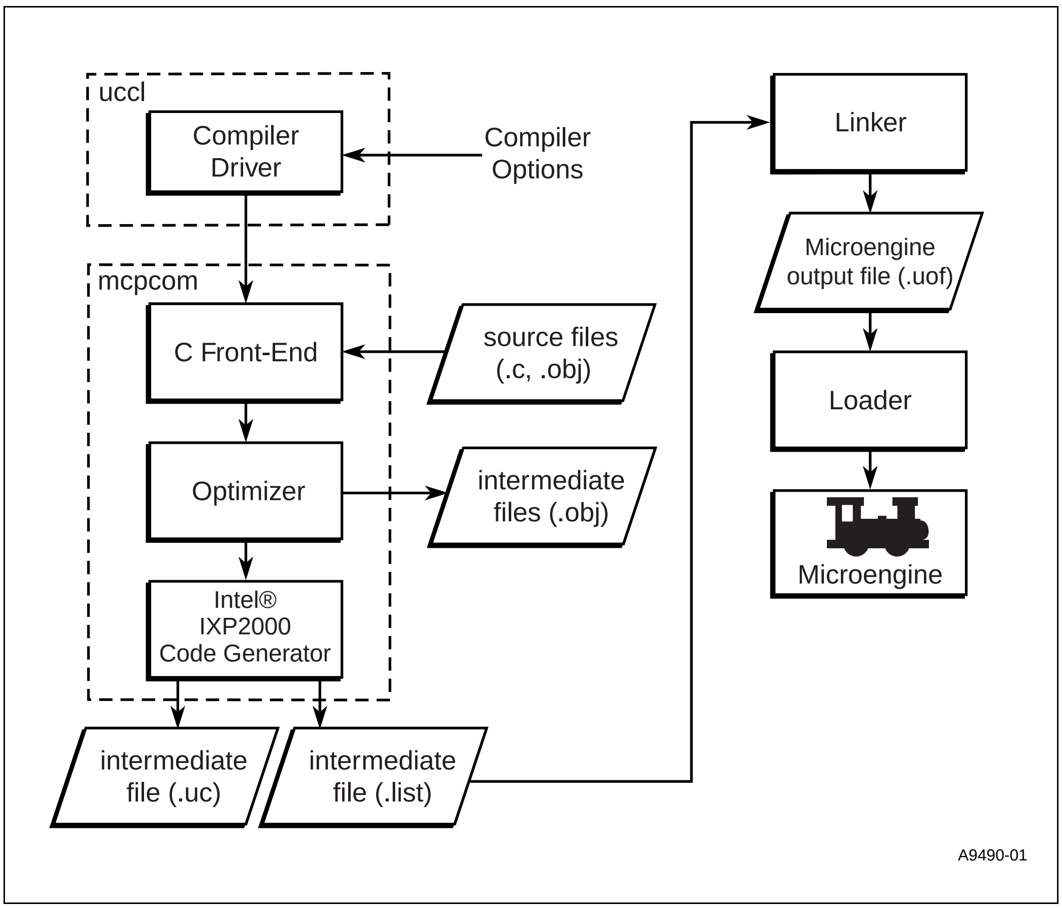
4.4 Compiler Steps

The .list file contains three types of elements: microwords, directives, and comments. Microwords consist of an opcode and arguments and generate a microword in the .list file. Directives pass information to the Linker and generally do not generate microwords. Comments are ignored in the assembly process.

The Compiler performs the following functions in converting the .c file to a .list file:

- Accepts standard C with `__declspec()` for specifying memory segments and properties and register usage {signal xfer nearest-neighbor remote}.
- Accepts restricted assembly via `__asm{ }`.
- Optimizes program in "whole program mode" where each function is analyzed and ailored according to its usage.
- Generates .list file for execution on single MicroEngine.

Figure 43. Compilation Steps



4.5 Case Sensitivity

The C language code as well as the command line switches are case sensitive.

Linker 5

The Linker is used to link microcode images. Microcode images are generated by the microcode complier or Assembler, whereas application objects are generated by a Intel® XScale™ C/C++ Compiler. The method is C/C++ Compiler independent. Shared address pointers are bound between microcode and Intel® XScale™ core application objects:

This chapter describes how to use the microcode Linker, ucld. The task of ucld is to process one or more microcode Compiler or Assembler, (uca) output files,*.list, and create an object file that can be loaded by the microcode loader. The loader, UcLo, is a library of C functions that facilitate external address pointer resolution and the loading of images to the appropriate Microengine. uca is described in Chapter 7.

5.1 About the Linker

Memory is shared between the Intel® XScale™ core and the Microengines. A common mode of design will have the Intel® XScale™ core generate and maintain data structures, while the Microengine reads the data. Common address pointers will be used for these data structures. For example, the base address of a route table will need to be shared. The solution will allow microcode and Intel® XScale™ applications to be written and compiled that can access the common address pointer.

5.1.1 Configuration and Data Accessed by the Linker

Various Microengine configuration data structures will exist in the Intel® XScale™ core. These are used to hold information about Microengines. Key repositories are:

- Microcode object. This is the binary object containing images that can be loaded into microcode.
- Microcode page/functionality map, per Microengine. Association of microcode image to Microengines. This is a list of libraries currently loaded.
- Shared address imports/externs list, per microcode image. Linker uses this to update the microcode.
- ‘C’ Compiler variables memory segments locations and sizes.

5.1.2 Shared Address Update (Flow)

1. From Assembler, get a list of externs. Get the microPC locations where these variables are used as an immediate. Get the microword format to know the offset and size of the immediate. Format this in the microcode “linked” object.
2. In the Intel® XScale™ application, load or map the microcode “linked” object.
3. In the Intel® XScale™ application, bind the external variable with a value by calling the appropriate function in the loader library.

4. The loader library updates the “immediates” with the bind-value for all occurrences of the variable in the microimage.
5. In Intel® XScale™ application, load the images to appropriate Microengines via a loader library function.

5.2 Microengine Image Linker (UCLD)

ucld is an executable that accepts a list of Microengine images (*.list) generated by the Assembler, (uca), or by the Compiler (uC), and combines them into a single object that is loadable by the core image, running on the Intel® XScale™ core processor, utilizing Microengine Loader Library (UcLo) functions.

5.2.1 Usage

```
ucld [options ...] list_file ...
```

5.2.2 Command Line Options

The following table lists the Linker command line options.

Table 5. Linker Command Line Options

Option	Definition
-h	Print a description of the ucld commands.
-c	Creates a hexidecimal representation of the output object to a ‘C’ module. The filename will be the name of the output object file but with a ‘.c’ file type.
-dr byteAddr:byteSize	Define DRAM data allocation region. Default is 0:0x80000000. (IXP2400, IXP2800, and IXP2850)
-f [fill_pattern]	Fill the unused microstore with the hexidecimal constant specifier. If the -f option is not specified, then only up to the last microword used will be output. The default fill_pattern value is the ctx_arb[kill] instruction.
-g	Include debugging information, to be used by the Workbench, in the output object file.
-o outfile	The default output file is the name of the first .list input file with a file type of .uof.
-sc byteAddr:byteSize	Define SCRATCH data allocation region. Default is 0:0x4000.
-seg file	Creates a 'C' header file defining the variables memory segments.
-sr*n* byteAddr:byteSize	Define SRAM data allocation region where ‘n’ is 0,1,2, or 3 for the specific memory bank. Default is 0:0x04000000 for byteAddr:byteSize.
-u n [n1...]	Associates a list of uEngines to subsequent uca_list_file, where 'n' is a list of uEngine numbers separated by space. All uEngines are assigned by default.
-v	Print a message that provides information about the version of the Linker being used.
-map [file]	Generate a linker .map file. The generated filename is the same as the .uof file but with the extension .map. The .map file contains the symbols and their addresses.

5.3 Generating a Microengine Application

On development system:

```
% uca ueng0.uc -o ueng0.list
% uca ueng1_5.uc -o ueng1_5.list
% ucld -u 0 ueng0.list -u 1 2 3 4 5 ueng1_5.list -o ueng.uof
```

5.4 Syntax Definitions

5.4.1 Image Name Definition

This definition associates a name to the content of the uca_list_file in the output object file. This provides the means of identifying the image within the object file, allowing referencing to particular section of the object file by name - used by the loader link library. The image name must be unique for all input files that are to be linked. The Image name is defined in the microcode source file (*.uc) use of the .image_name keyword. If the image name is not specified, it is the list file name without the extension.

Format for uca source (*.uc), and output (*.list) files:

```
.image_name name
```

5.4.2 Import Variable Definition

This scheme provisions the sharing of address pointers between Microengine images and the core image.

An application is physically comprised of two parts: a Microengine image and a core image. Microengine images are microinstructions that run on the Microengine and are created using the uca and ucld tools. Core image runs on the Intel® XScale™ core processor and is created using the ARM Compiler/Assembler and Linker tools.

Address sharing between the Microengine and core images is achieved by declaring the variables as an imported/external variable using the .import_var keyword in the microcode source file (*.uc), prior to the variables being used. The uca Assembler will create a list of microword addresses and the field bit positions within the microword where the variables are used and provided the information in its output file (*.list) in the format as described below. The ucld Linker will process the uca output files (*.list), possible one for each Microengine, and store the information in a format that is acceptable by the loader. Core application binds and resolves a value to the Import-Variable, by using the functions provided in the Microcode Loader Library (uclo.a). The binding/ resolution of import variables must be performed prior to the loading of the micro image to the Microengines.

Format for uca source (*.uc) file:

```
.import_var variable_name variable_name …
```

Format of uca output (*.list) files:

```
.%import_var variable_name page_id uword_address
<msb:lsb:rtSht_val, ...msb:lsb:rtShf_val>
```

Where:

`variable_name`	String name of the external variable as declared in the uca source file (*.uc).
`highLow_flag`	An integer 0 or 1 indicating the lower or upper respective 16 bits of the external 32-bit address.
`page_id`	String name of the page identifier
`uword_address`	An integer number between 0 and 1023 indicating the micro word address.

`<msb:lsb:rtSht_val,...msb:lsb:rtShf_val>`

A list of a maximum of four integer triad representing the bit field most and least significant bit positions, and the number of bit position to right shift the source address value.

5.4.3 Microengine Assignment

This feature allows the loading of Microengine images to the appropriate Microengine at load time. This is achieved by the specifying the -u option on the command line indicating the number of Microengine(s) to be associated with the subsequent uca-list-file. The -u option can be repeated for subsequent input files, (all of the Microengines) if the option is not specified. An error will be generated if a Microengine is assigned to multiple input files.

Format of command line to associate Microengines 0, 1, and 2 to the input file test.list, and to associate Microengines 3, 4, and 5 to the input file test2.list:

```
ucld -u 0 1 2 test.list -u 3 4 5 test2.list
```

5.4.4 Code Entry Point Definition

Code entry point definition allows the user to specify the starting point of each Microengine image instead of the default entry point of zero.

Format for uca source file (*.uc), and output (*.list) files:

```
.entry page_id uword_address
```

5.5 Examples

5.5.1 Uca Source File (*.uc) Example

```
 .entry common_code 0
 .image_name test
 .import_var rbuf rswap
 memsv_bcopyt#:
 br[memsv_bcopysubr#]
:
 memsv_bzero#:
 br[memsv_bzerosubr#]
 memsv_bcopy#:
 br[memsv_bcopysubr#]
```

5.5.2 Uca Output File (*.list) Example

```
.entry common_code 0
.thread_type 0123 service
.image_name test
 .%import_var rbuf common_code 1 <16:0:0>
.%import_var rbuf common_code 2 <31:16:16> 10<31:16:16> 21<31:16:16>
.%import_var rswap common_code 3 <16:0:0>
.%import_var rswap common_code 4 <31:16:16>
:
memsv#:
memsv_bcopyt#:
br[memsv_bcopysubr#]
.44 F8001C07 common_code
memsv_bzero#:
br[memsv_bzerosubr#]
.45 F8002707 common_code
memsv_bcopy#:
br[memsv_bcopysubr#]
.46 F8003E87 common_code
:
```

5.5.3 .map File Example

The following example shows a sample of the *.map file, which is generated when you use the -map [filename] option on the linker command line. The .map file shows the address of the symbol as well as the memory region in which it is located along with its size in bytes. The symbols are prefixed with the Microengine number in which the symbol resides followed by an exclamation point.

```
Memory Map file: test_c.map
Date: Fri May 30 12:11:03 2003

UcLd version: 3.1,  UOF: test_c.uof

Address             Region      ByteSize    Symbol
======================================
0x00000040 SRAM3     64        2           !sram3$tls
0x00000080 SRAM2    128        2           !sram2$tls
0x000000b0 SRAM0    176        2           !sram$tls
0x00000020 SCRATCH  32         2           !scratch$tls
0x00000004 DRAM     4          2           !dram$tls
0x000002c8 SRAM0    40         2           !_powers$5
0x00000000 LMEM     16         2           !_meNum
0x00000200 SRAM0    160        2           !_llpowers$5
0x00000000 LMEM     0          2           !_ctxNum
0x000002f4 SRAM0    4          2           !??_C@_01A@?6?$AA@
0x000002fc SRAM0    4          2           !??_C@_01A@?5?$AA@
0x00000000 SRAM3    64         0           !sram3$tls
0x00000000 SRAM2    128        0           !sram2$tls
0x00000000 SRAM0    176        0           !sram$tls
0x00000000 SCRATCH  32         0           !scratch$tls
0x00000000 DRAM     4          0           !dram$tls
```

```
0x000002a0 SRAM0   40          0               !_powers$5
0x00000000 LMEM    16          0               !_meNum
0x00000160 SRAM0   160         0               !_llpowers$5
0x00000000 LMEM    0           0               !_ctxNum
0x000002f0 SRAM0   4           0               !??_C@_01A@?6?$AA@
0x000002f8 SRAM0   4           0               !??_C@_01A@?5?$AA@


Image               ImportVar           Uninitialized Value
====================================================
test_c              rbuf                0x0
```

5.6 Memory Segment Usage

The following example shows a sample of the memory segment usage.

UcLd version: 3.1, UOF: test_c.uof, Date: Fri May 30 12:11:03 2003

```
*/
#ifndef __TEST_C_H__
#define __TEST_C_H__
#define SRAM0_DATASEG_BASE        0x0     /* SRAM0 data seg byteAddr */
#define SRAM0_DATASEG_SIZE        0x300   /* SRAM0 data seg byteSize */
#define SRAM1_DATASEG_BASE        0x0     /* SRAM1 data seg byteAddr */
#define SRAM1_DATASEG_SIZE        0x0     /* SRAM1 data seg byteSize */
#define SRAM2_DATASEG_BASE        0x0     /* SRAM2 data seg byteAddr */
#define SRAM2_DATASEG_SIZE        0x100   /* SRAM2 data seg byteSize */
#define SRAM3_DATASEG_BASE        0x0     /* SRAM3 data seg byteAddr */
#define SRAM3_DATASEG_SIZE        0x80    /* SRAM3 data seg byteSize */
#define DRAM_DATASEG_BASE         0x0     /* DRAM data seg byteAddr */
#define DRAM_DATASEG_SIZE         0x8     /* DRAM data seg byteSize */
#define DRAM1_DATASEG_BASE        0x0     /* DRAM1 data seg byteAddr */
#define DRAM1_DATASEG_SIZE        0x0     /* DRAM1 data seg byteSize */
#define SCRATCH_DATASEG_BASE      0x0     /* SCRATCH data seg byteAddr */
#define SCRATCH_DATASEG_SIZE      0x40    /* SCRATCH data seg byteSize */
#endif          /* __TEST_C_H__*/
```

5.7 Microcode Object File (UOF) Format

The Microcode Object File consists of a file-header and one of more sections called file-chunks. Each fileChunk is identified by a unique ID in the fileChunkHdr section of the fileHdr. A typical UOF will consisted of a UOF_OBJS file-chunk, and an optional DBG_OBJS file-chunk. The UOF data structures are described in uof.h, and dbg_uof.h.

5.7.1 File Header

The file header is mandatory and must be the first entry in the file. It is used to identify the file format, and to locate file-chunks within the file. This header consists of a fixed 18-byte section of type uof_fileHdr_T, and a variable section of type uof_fileChunkHdr_T. The variable section must consist of at least MaxChunks of uof_fileChunkHdr_T and immediately precedes the fixed section in the file.

fileId	2 bytes: File id and endian indicator.
reserved1	2 bytes: Reserved for future use.
minVer	1 byte: File format minor version.
majVer	1 byte: File format major version.
reserved2	2 bytes: Reserved for future use.
maxChunks	2 bytes: Maximum possible chunks that the file can contain—specify at the creation of the file.
numChunks	2 bytes: Number of chunks currently being used.

5.7.2 File Chunk Header

chunkId	8 bytes: A unique value identifying the chunk. Currently the values are the literal **UOF_OBJS,** or **DBG_OBJS**
checksum	4 bytes: CRC checksum of chunk.
offset	4 byte: Offset into the file where the chunk begins.
size	4 bytes: Size of the chunk.

5.7.2.1 UOF Object Header

This object header describes the IXP system that the UOF can execute on, and it contains the locations of the object-chunks within the file. This header, of type uof_objHdr_T, must be at the beginning of the object and precedes at least `MaxChunks of object-chunk-header (uof_chunkHdr_T).`

cpuType	4 bytes: CPU family type -- IXP2400=2, IXP2800=4. This is a resolution of all the cpu types from images (list files).
minCpuVers	2 bytes: The minimum CPU revision that the UOF will run on.
maxCpuVers	2 bytes: The maximum CPU revision that the UOF will run on.
maxChunks	2 bytes: The maximum of chunks that can be contained in an UOF chunk.
numChunks	2 bytes: Number of chunks currently being used.
reserved1	4 bytes: Reserved for future use.
reserved2	4 bytes: Reserved for future use.
UOF-object headers	MaxChunks * sizeof(uof_ChunkHdr) contiguous bytes of uof object headers.

5.7.2.2 UOF Object Chunk Header

This is the variable portion of the obj-header and it must immediately precede the fixed section (uof_uof_objHdr_T) in the file. This header, of type uof_chunkHdr_T, identifies and provides the location of the object-chunks. The linker (UcLd) currently creates object-chunks that are identified by the following literals: UOF_STRT, UOF_GTID, UOF_IMAG, UOF_MSEG, UOF_IMEM.

chunkId	8 bytes: A unique value identifying the chunk.
offset	4 byte: Offset of the chunk relative to the begining of the object.
size	4 bytes: Size of the chunk in bytes.

5.7.2.3 UOF_STRT

This object-chunk identifies the string table within the object and contains all the strings that are used by the other object-chunks. The strings are NULL terminated, and referenced by a value that is less than `tableLength as an offset` into this table. The strings in this table should not be altered. This object is represented by the uof_strTab_T data type.

tableLength	4 bytes: Total length of the table in bytes.
*strings	tableLength of bytes: NULL terminated strings.

5.7.2.4 UOF_IMEM

This object-chunk, of type uof_initMem_T, contains the memory initialization values and locations. The values are stored and written to memory as a sequence of bytes, therefore, the attributes are only used when the byte-order of the values needs to be switched

symName	4 bytes: Symbol name string table offset.
region	1 byte: Memory region -- uof_MemRegion.
scope	8 bytes: 0 = global, 1 - local
reserved1	2 bytes: Reserved for future use.

addr	4 bytes: The start address of memory to initialize with the byte-values.
numBytes	4 bytes; Number of bytes of consisting of the values.
numValAttr	4 bytes: The number of value attributes

5.7.2.5 Memory Initialization Value Attributes

This object-chunk, of type uof_memValAttr_T, describes the attributes of memory initialization values.

byteOffset	4 bytes: Byte offset from allocated memory.
value	4 bytes: data value.

5.7.2.6 uof_initRegSym

This object-chunk contains register or symbol initialization information. The value could be an integer constant, postfix-expression, or a register standard value.

symName	4 bytes: Symbol name string table offset.
initType	1 byte: 0=symbol, 1=register.
valueType	1 byte: EXPR_VAL, STRTAB_VAL.
regType	1 bytes: The register type -- ixp_RegType_T.
reserved1	1 byte: Reserved for future use.
regAddrOrOffset	4 bytes: The register address, or the offset from the symbol.
value	4 bytes: integer value, or expression string table offset.

5.7.2.7 UOF_MSEG

This object-chunk, of type uof_varMemSeg_T, contains the starting address and the byte size of the allocated memory region.

sram0Base	4 bytes: uC variables SRAM0 memory segment base address.
sram0Size	4 bytes: uC variables SRAM0 segment size bytes.
sram1Base	4 bytes: uC variables SRAM1 memory segment base address
sram1Size	4 bytes: uC variables SRAM1 segment size bytes.
sram2Base	4 bytes: uC variables SRAM2 memory segment base address
sram2Size	4 bytes: uC variables SRAM2 segment size bytes
sram3Base	4 bytes: uC variables SRAM3 memory segment base address
sram3Size	4 bytes: uC variables SRAM3 segment size bytes
sdramBase	4 bytes: uC variables SDRAM memory segment base address.

sdramSize	4 bytes: uC variables SDRAM segment size bytes.
scratchBase	4 bytes: uC variables SCRATCH memory segment base address.
scratchSize	4 bytes: uC variables SCRATCH segment size bytes.

5.7.2.8 UOF_GTID

This object-chunk, of type uof_GTID_T, contains information from the tool that created the object.

toolId	8 bytes: The tool name string table offset.
toolVersion	4 bytes: The tool version.
reserved1	4 bytes: Reserved for future use.
reserved2	4 bytes: Reserved for future use.

5.7.2.9 UOF_IMAG

This object-chunk, of type uof_Image_T, contains uof image information. A uof image typically corresponds to the information derived from the list file. Therefore, there is one image for every list file that is linked

imageName	4 bytes: Image name string table offset.
meAssigned	4 bytes: The bit mask of the microengines assigned to this image.
fillPattern	8 bytes: The unused microstore fill pattern -- only the lower five bytes of the pattern will be used.
sensitivity	1 byte: Indicates the case sensativity of the image (0=insensative; 1=sensative).
reserved	1 byte: Reserved for future use.
meMode	2 bytes: Indicates the microengine modes local-memory, and context modes; unused<15:10>, locMem1<9>, locMem0<8>, unused<7:4>, ctx<3:0>. A 1 in the locMemX field indicates global addressing, and 0 indicates context relative addressinjg. The cxt field will be either 4, or 8 to indicate the context mode. All other fields are unsued.
cpuType	4 bytes: CPU family type -- IXP2400=2, IXP2800=4
maxVer	2 bytes: The maximum cpu version on which the image can run.
minVer	2 bytes: The minimum cpu version on which the image can run.
imageAttrib	2 bytes: unused<15-13>, patternFill<12>, unused<11:0>. If the patternFill bit is set, then the unused micro stores will be filled with the fillPattern.
entryPage	2 bytes: Page number entry point.
entryAddress	2 bytes: uPC entry point into the code.
numOfPage	2 bytes: The number of ustore pages associated with the image.
regTabOffset	4 bytes: Offset from the object to the register table (uof_meRegTab_T).

initRegSymTab	4 bytes: Offset from the object to the register/symbol initialization table (uof_initRegSymTab_T).
reserved2	4 bytes: Reserved for future use.
micro store pages = NumOfPages * sizeof(uof_codePage_T) contiguous bytes of code pages.	

5.7.2.10 uof_codePage

This structure, of type `uof_codePage_T`, is a container of related items.

neighRegTabOffset	4 bytes: Offset to neighbor-register table.
reserved1	4 bytes: Reserved for future use.
ucVarTabOffset	4 bytes: Offset to uC variables table
impVarTabOffset	4 bytes: Offset to import-variable table.
impExprTabOffset	4 bytes: Offset to import-expression table.
codeAreaOffset	4 bytes: Offset to microwords.
reserved2	4 bytes: Reserved for future use.

5.7.2.11 uof_meRegTab

This table, of type `uof_meRegTab_T, contains numEntries of the register definitions uof_meReg_T. The register definitions must immediately follow this table in the file.`

numEntries	4 bytes: Number of table entries.
Table entries = NumEntries * sizeof(uof_meReg_T) contiguous bytes of objects	

5.7.2.12 uof_meReg

This data type describes microengine register definitions.

name	4 bytes: Register name string-table offset
visName	4 bytes: Register visible name string-table offset.
type	2 bytes: Register type -- ixp_RegType_T
addr	2 bytes: The register address
accessMode	2 bytes: uof_RegAccess_T: read/write/both/undef
visible	1 byte: Register visibility, 1=visible, 0=not visible.
reserved1	1 byte: Reserved for future use.
refCount	2 bytes: Number of continguous registers allocated
reserved2	2 bytes: Reserved for future use
xoId	4 bytes: Xfer order identification

5.7.2.13 uof_neighReg

Structure is same format as **uof_uwordFixup_T.**

5.7.2.14 uof_neighRegTab

This table, of type `uof_neighRegTab_T`, contains numEntries for the fixup-register definitions `uof_meReg_T`. The register definitions must immediately follow this table in the file.

numEntries	4 bytes: Number of table entries.
Table entries = NumEntries * sizeof(uof_neighReg_T) contiguous bytes of objects	

5.7.2.15 uof_importExpr

Structure is same format as **uof_uwordFixup_T.**

5.7.2.16 uof_impExprTabTab

This table, of type `uof_imExprTabTab_T`, contains objects of type `uof_importExpr_T`.

numEntries	4 bytes: Number of table entries
Table entries = NumEntries * sizeof(uof_importExpr_T) contiguous bytes of objects.	

5.7.2.17 uof_xferReflectTab

This table, of type uof_xferReflectTab_T contains objects of type uof_xferReflect_T.

numEntries	4 bytes: Number of table entries
Table entries = NumEntries * sizeof(uof_xferReflect_T) contiguous bytes of objects.	

5.7.2.18 uof_UcVar

Structure is same format as **uof_uwordFixup_T**

5.7.2.19 uof_ucVarTab

This table of type `uof_ucVarTab_T` contains numEntries of objects of type `uof_ucVar_T`.

numEntries	4 bytes: Number of table entries.
Table entries = NumEntries * sizeof(uof_ucVar_T) contiguous bytes of objects.	

5.7.2.20 uof_initRegSymTab

This table, of type `uof_initRegSymTab_T, contains numEntries of the register/ symbol initializations uof_initRegSym_T. The register/symbol initialization must immediately follow this table in the file.`

numEntries	4 bytes: Number of table entries.
Table entries = NumEntries * sizeof(uof_initRegSym_T) contiguous bytes of objects.	

5.7.2.21 uof_uwordFixup

This data structure contains microword fixup information. The fixup value can be a constant or a postfix expression

name	4 bytes: Name string-table offset.
uwordAddress	4 bytes: Micro word address(bytes 0 &1), unused (bytes 2 & 3).
exprValue	4 bytes: Postfix expression string-table.
valType	1 byte: VALUE UNDEF, VALUE NUM, VALUE EXPR
valueAttrs	1 byte: bit<0> (Scope: 0=global, 1=local), bit<1> (init: 0=no, 1=yes)
reserved1	2 bytes: Reserved for future use
fieldAttrs	12 bytes: Field pos, size, and right shift value.

5.7.2.22 uof_codeArea

This structure table contains the microwords. The microwords are stored as 5-byte values directly following this table. Therefore, this sturucture should be followed by at least `numMicroWords` * 5 bytes

numMicroWords	4 bytes: Number of microwords.
reserved	4 bytes: reserved for future use.

5.8 DBG_OBJS

This object is stored in the UOF as a file-chunk with the **DBG_OBJS** identification. This object contains sub-sections, chunks, that contain all the neccessary information pertaining to the debugging of the micro-code for all microengines. The offsets within the sub-sections, are relative to the beginning of this object and are **not** relative to the beginning of the file. The format of this object is similar to the file-header, in that it consists of a fixed header section immediately followed by variable header sections.

5.8.1 Debug Objects Header

This header, of type uof_objHdr_T, must be at the begining of the object and precedes at least `MaxChunks of debug chunk-header (dbg_chunkHdr_T).`

cpuType	4 bytes: Alway zero
minCpuVers	4 bytes: Always zero
maxCpuVers	4 bytes: Always zero
MaxChunks	2 bytes: maximum objects that can be contained in a DBG_OBJ chunk.
NumChunks	2 bytes: number of chunks currently being used.
reserved1	4 bytes: Reserved for future use.
reserved2	4 bytes: Reserved for future use.
Debug-object headers	MaxChunks * sizeof(DbgChunkHdr) contiguous bytes of debug object headers.

5.8.2 Debug Object Chunk Header

This is the variable portion of the debug-object header and must immediately precede the fixed section (uof_objHdr_T) in the file. This header, of type dbg_chunkHdr_T, identifies and provides the location of the object-chunks. The linker (UcLd) currently creates object-chunks that are identified by the following literals: **DBG_STRT, DBG_IMAG, DBG_SYMB**

chunkId	8 bytes: A unique value identifying the chunk.
offset	4 byte: Offset of the chunk relative to the beginning of the object.
size	4 bytes: Size of the chunk.

5.8.3 DBG_STRT

This debug-object chunk identifies the string table within the debug object and contains all the strings that are used by the other debug-object chunks. The strings are NULL terminated, and referenced by a value that is less than `tableLength as an offset` into this table. The strings in this table should not be altered. This object is represented by the dbg_strTab_T data type.

TableLength	4 bytes: total length of the table in bytes
Strings	NULL terminated strings

5.8.4 dbg_RegTab

This debug-object chunk of type `dbg_RegTab_T` contains objects of type `uof_meReg_T`.

numEntries	2 bytes: The number of objects in the table.
Table entries = NumEntries * sizeof(me_Reg_T) contiguous bytes of objects.	

5.8.5 dbg_LblTab

This debug-object chunk of type `dbg_LblTab_T` contains objects of type `dbg_Label_T`.

numEntries	2 bytes: The number of objects in the table.
Table entries = NumEntries * sizeof(dbg_Label_T) contiguous bytes of objects.	

5.8.6 dbg_SymTab

This debut_object chunk of type `dbg_SymTab_T` contains objects of type `dbg_Symb_T`.

numEntries	2 bytes: The number of objects in the table.
Table entries = NumEntries * sizeof(dbg_Symb_T) contiguous bytes of objects.	

5.8.7 dbg_SrcTab

This debug object chunk of type `dbg_SrcTab_T` contains objects of type `dbg_Source_T`.

numEntries	2 bytes: The number of objects in the table.
Table entries = NumEntries * sizeof(dbg_Source_T) contiguous bytes of objects.	

5.8.8 dbg_TypTab

This debug object chunk of type `dbg_TypTab_T` contains objectsof type `dbg_Type_T`.

numEntries	2 bytes: The number of objects in the table.
Table entries = NumEntries * sizeof(dbg_Type_T) contiguous bytes of objects.	

5.8.9 dbg_ScopeTab

This debug object chunk of type `dbg_ScopeTab_T` contains objects of type `dbg_Scope_T`

numEntries	2 bytes: The number of objects in the table.
Table entries = NumEntries * sizeof(dbg_Scope_T) contiguous bytes of objects.	

5.8.10 dbg_Image

This debug-object chunk, of type dbg_Image_T, contains the debug information related to a set of microengines.

lstFileName	4 bytes: List-file name string-table offset.
meAssigned	4 bytes: bit values of assigned MEs
lstFileCreatedBy	1 byte: The tool that created the list file (uca/uC)-- Ucld_LstFileToolType.
reserved1	1 byte: Reserved for future use.
ctxMode	4 bytes: The number number of contexts -- 4 or 8
endianMode	1 byte: endian: little=0, big=1
scopeTabOffset	4 bytes: Offset to the scope table
regTabSize	4 bytes: Byte size of the register table
lblTabSize	4 bytes: Byte size of the Label table.

srcTabSize	4 bytes: Byte size of the Source-line table.
regTabOffset	4 bytes: Register table offset from beginning of the debug object.
lblTabOffset	4 bytes: Label table offset from the beginning of the debug object.
srcTabOffset	4 bytes: Source table offset from the beginning of the debug object.
typTabSize	4 bytes: Variable types table byte size.
scopeTabSize	4 bytes: Scope table size.
typTabOffset	4 bytes: Variable types table offset.
instOprndTabsize	4 bytes: Instruction operands table size.
instOprnTabOffset	4 bytes: Instruction operands table offset.
reserved2	4 bytes: Reserved for future use.

5.8.11 dbg_Label

This structure defines the labels of type dbg_Label_T.

name	4 bytes: Label name debug string table offset.
addr	2 bytes: Uword address of the label.
unused1	2 bytes: Reserved for future use

5.8.12 dbg_Source

This structure defines the source code information of type dbg_Source_T.

fileName	4 bytes: Source filename debug string table offset.
lines	4 bytes: Source lines offset into the debug string table.
lineNum	4 bytes: Source file line number.
addr	4 bytes: The associated uword address.
validBkPt	1 byte: Indicates whether a breakpoint can occur at the uword.
ctxArbKill	1 Byte: This instruction is a ctx_arb[kill]
brAddr	2 bytes: Branch label address.
regAddr	2 bytes: Register address.
regType	2 bytes: Register type.
deferCount	2 bytes: this instructions's defer count.
reserved1	2 bytes: Reserved for future use.

5.8.13 dbg_Symb

This structure contains information about symbol in the debug object.

name	4 bytes: Symbol name string-table offset.
scope	1 byte: Scope -- global=0, local=1.
region	1 byte: uof_ValLocTyp: SRAM_MEM_ADDR, DRAM_MEM_ADDR, SCRATCH_MEM_ADDR.
reserved	2 bytes: Reserved for future use.
addr	4 bytes: Symbol memory location.
byteSize	4 bytes: Size of the symbol.

5.8.14 dbg_Type

Contains information regarding variable type.

name	4 bytes: Symbol name debug string-table offset.
typeId	2 bytes: Id of type -- UcId_TypeType.
type	2 bytes: Type referenced -- could be itself.
size	4 bytes: Size/bound of the type.
defOffset	4 bytes: Offset to dbg_StructDef_T or dbg_EnumDef_T.

5.8.15 dbg_StructDef

This structure defines data structure in the debug object. This structure must immediately precede `numField` of dbg_StructField_T in the debug object

numFields	2 bytes: Number of fields in the structure.
reserved	2 bytes: Reserved for future use.
fieldOffset	4 bytes: Offset to dbg_StructField_T relative to the beginning of the debug object.

5.8.16 dbg_StructField

This structure describes the fields of dbg_StructDef_T.

name	4 bytes: Field name debug string-table offset.
offset	4 bytes: This field’s offset from beginning of struct.
type	2 bytes: Field type.
bitOffset	1 byte: BitOffset.
bitSize	1 byte: BitSize.

5.8.17 dbg_EnumDef

Describes an enumeration definition.

numValues	2 bytes: Number of values.
reserved	2 bytes: Reserved for future use.
valueOffset	4 bytes: Offset to dbg_EnumValue_T.

5.8.18 dbg_EnumValue

Describes the enumeration value.

name	4 bytes: Enum value name debug string-table offset.
value	4 bytes: Enum value.
reserved	4 bytes: Reserved for future use.

5.8.19 dbg_Scope

This structure contains the variables and functions scope information.

name	4 bytes: Scope name debug string-table offset.
fileName	4 bytes: File name debug string-table offset.
type	2 bytes: Ucld_ScopeType—global, file, funct, ect...
lineBeg	2 bytes: Scope in effect at source line.
lineEnd	2 bytes: Scope stops at source line.
uwordBeg	2 bytes: Scope in effect at uword.
uworkEnd	2 bytes: Scope stops at uword.
numScopes	2 bytes: Number of dbg_Scope_T within this scope.
numVars	2 bytes: Number of variables in this scope.
scopeOffest	4 bytes: Offset to dbg_Scope_T within this scope.
varOffset	4 bytes: Offset to dbg_Variable_T within this scope.
funcRetOffset	4 bytes: Func return value offset to dbg_ValueLoc_T.

5.8.20 dbg_ValueLoc

This structure contains the location of the variable of type dbg_ValueLoc_T.

locId	4 bytes: Ucld_ValLocTyp -- reg, mem, or spill.
symbName	4 bytes: Symbol name offset to string-table.
location	4 bytes: MemAddr, regNum, or spill-offset.
multiplier	4 bytes: Spill multiplier.

5.8.21 dbg_Variable

This structure defines the variable of type dbg_Variable_T.

name	4 bytes: Variable name offset to string-table.
type	2 bytes: Type to refe.
reserved	1 byte: Reserved for future use.
locType	1 byte: Location type:- Ucld VarLocType
locOffest	4 bytes: Offset to dbg_Sloc_T, dbg_Tloc_T, dbg_RlocTab_T, or dbg_Lmloc_T

5.8.22 dbg_Sloc

This structure contains the symbol association of the variable of type dbg_Sloc_T.

symbName	4 bytes: Symbol name offset to string-table.

5.8.23 dbg_Tloc

This structure defines the variables that are local to the context of type dbg_Tloc_T.

symbName	4 bytes: Symbol name offset to string-table.
offset	4 bytes: Local mem offset.
multiplier	4 bytes: Local mem multiplier.

5.8.24 dbg_RlocTab

This structure defines variables of type dbg_Rloc_T that are located in the Register.

numEntries	2 bytes: Number of live ranges.
reserved	2 bytes: Reserved for future use.

5.8.25 dbg_Lmloc

This structure defines variables of type dbg_Lmloc_T located in local memory.

offset	4 bytes: localmemory offset

5.8.26 dbg_Liverange

This structure defines where in the range the variables of type dbg_Liverange_T are alive.

offset	4 bytes: Byte offset from var.
locId	4 bytes: Reg, or spill.
regNumOrOffset	4 bytes: Reg-num, or spill-offset.
multiplier	4 bytes: Spill multiplier.

symName	4 bytes: Spill—symbol name (sram$tls).
numRanges	2 bytes: Number of ranges.
ambiguous	1 byte: the location may or many not contain valid value
reserved	1 byte: Reserved for future use.
rangeOffset	4 bytes: Offset to dbg_Range_T.

5.8.27 dbg_Range

This structure defines the uword range where the variable of type dbg_Range_T is alive.

start	4 bytes.
stop	4 bytes.

5.8.28 dbg_InstOprnd

addr	4 bytes: Micro address of the instruction.
src1Name	4 bytes: Source operand 1 register name offset in the string table.
src1Addr	4 bytes: Source operand 1 register address offset in the string table.
src2Name	4 bytes: Source operand 2 register name offset in the string table.
src2Addr	4 bytes: Source operand 2 register address offset in the string table.
destName	4 bytes: Destination register name offset in the string table.
destAddr	4 bytes: Destination register address offset in the string table.
xferName	4 bytes: xfer register name offset in the string table.
xferAddr	4 bytes: xfer register address offset in the string table.
mask	4 bytes: <31:3> unused. <2> I/O indirect. <1> I/O read. <0> I/O write.
refCount	1 byte: I/O reference count (1 to 16)
deferCount	1 byte: Branch defer count (0-3).
reserved1	2 bytes: Reserved for future use.
reserved2	4 bytes: Reserved for future use.

Foreign Model Simulation Extensions 6

Foreign Model simulation extension is a useful tool for simulating external hardware devices, and, in developing software. It acts as an extension to the capabilities of the Transactor, which is a cycle and data accurate software model of the IXP2400, IXP2800, and IXP2850 network processors. Section 6.1 provides an overview of where foreign model simulation can be used. Section 6.2 describes how to integrate foreign models with the Transactor. Details of simulating media bus devices are discussed in Section 6.4. Section 6.5.1 contains some sample code.

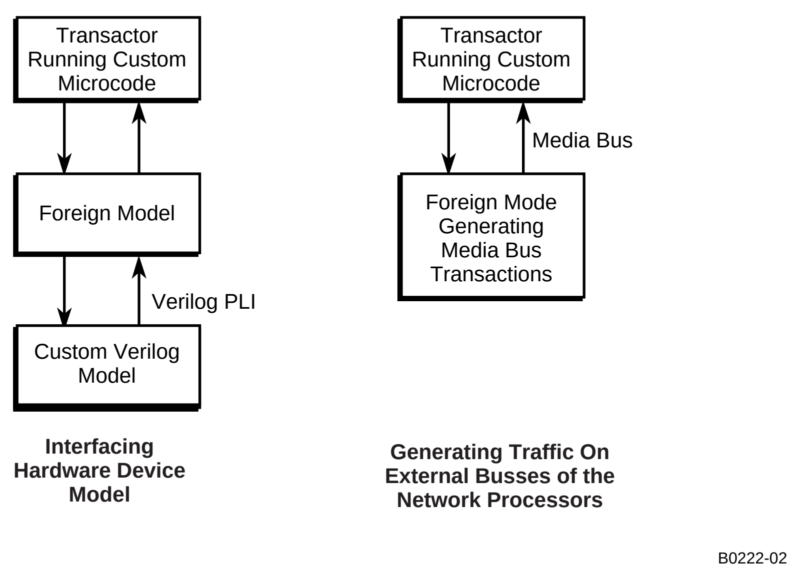
6.1 Overview

Reasons to use Foreign Model simulation:

1. To interface hardware device models, and
2. To generate traffic on external busses,

These are illustrated in Figure 44.

Figure 44. Example of Foreign Model Usage



- **Interfacing hardware device models**. Foreign modeling allows integration of models of external hardware such as a MAC device or custom chip that is connected to the Media Bus. This could be a software model of the external device or a design in a hardware simulator such as Verilog. This includes the ability to interface with hardware models (such as a Verilog model) which could reside on a remote machine.

- **Generating traffic on external busses**. Microengine software designers can use a foreign model to assist in the design and debugging of Microengine software modules by producing generic transactions of the Media Bus. This way, hardware residing on the Media Bus such as a MAC device or some custom chip can be simulated. Once the software is designed, the same foreign model interface can be used to produce the traffic typical for the application. This assists in estimating the performance of software.
- **Intel® XScale™ Software Module Prototyping**. The Foreign model interface can also be used to develop the software that will run on the Intel® XScale™. Even though Intel® XScale™software executes on a development machine, once it interfaces to the Transactor through the Transactor API, execution is cycle accurate. This reduces the simulation time and allows accurate verification of interactions between Intel® XScale™ and Microengine software.

6.2 Integrating Foreign Models with the Transactor

A foreign model provides a mechanism by which the network processor software model (Transactors) can be extended to include additional software models of hardware that interface with the network processor. The way to integrate a foreign model with the appropriate Transactor is by creating a Foreign Model Dynamic-link Library (DLL).

To activate a Foreign Model DLL, you execute the `foreign_model` command at the Transactor's command prompt (see Section 7 for more information about the transactor).

If you are running the Developer's Workbench, you specify your foreign model by selecting **Simulation->Options** then selecting the **Foreign Model** tab. The Workbench automatically executes the appropriate `foreign_model` command for you.

When the Transactor executes the `foreign_model` comand, it loads the Foreign Model DLL and calls the `GetForeignModelFunctions()` function in the foreign model to get the pointers to six foreign model functions. The Transactor calls these functions to notify the foreign model whenever the following simulation events occur:

- The model is initialized.
- Before a simulation step occurs (preSim),
- After a simulation step has completed (postSim),
- The simulation is reset (i.e., a sim_reset command is executed),
- The model is deleted (i.e., a sim_delete command is executed), and
- When the Transactor exits.

The foreign model interacts with the Transactor using the Transactor API.

6.3 Foreign Model Dynamic-Link Library (DLL)

The foreign model DLL must provide the exported function `GetForeignModelFunctions()`, that the Transactor calls to get the addresses of the six functions to interact with the foreign model.

The foreign model DLL runs in conjunction with the DLL version of the Transactor, so Xactvmod.h must be included in the foreign model DLL source files, if needed, and it must be linked against IXP2400.lib or IXP2800.lib.

To have the Transactor load a foreign model DLL, use the "foreign_model" command.

6.4 Simulating Media Devices

Simulating of devices involves the following:

- Getting states of pins,
- Determining appropriate action based on the pin states, and
- Setting the appropriate pin states

The Workbench provides media device foreign models as described in Section 2.10, "Packet Simulation. These foreign models support several protocol types for each network processor.

For streaming packets through the Transactor, refer to Section 2.10 before developing your own foreign model.

Appendix A, "Transactor States"documents the Transactor state names and a brief description of the states for various device pins including QDR and MSF devices.

6.5 Creating A Foreign Model DLL

This section contains sample code demonstrating how to create a dynamic-link library (DLL) for a foreign model. It is also available in `\me_tools\samples\ForeignModelDLL` on the distribution CD.

In your DLL you must provide the exported function `GetForeignModelFunctions( )`, which the Transactor calls to get the addresses of the six functions that it calls to interact with your foreign model. Of the six functions, the `foreign_model_initialize()` function is required but the other five are optional. If you do not need to be notified for an event, return a zero as the pointer to the function associated with that event.

If you want to call the Transactor API from your foreign model, you must include `xact_vmod.h` in your source files, and you must link against IXP2400.lib or IXP2800.lib, depending on which network processor you will be simulating.

6.5.1 DLL Sample Code.

```
// You must include xact_vmod.h in order to link correctly against
// the DLL version of the transactor.
//
#include "xact_vmod.h"
#include <stdlib.h>
#include <string.h>
#include <Windows.h>

// for testing purposes only
// remember the init_str from the foreign_model_initialize so it can be used
// by each function so the instance generating the console messages is uniquely
identifiable
// allow room for up to 255 foreign models!
static char *INIT_STRING[255] = {0};
static char *MODEL_NAME[255] = {0};

/*----------------------------------------------------------------
    foreign_model_initialize

    This function will be called to initialize the foreign model after
    the transactor "init" command has successfully executed.  Returning 0
    will result in a transactor error.

     returns:
        uses:
    modifies:
*/
int foreign_model_initialize(int model_instance_num, const char *model_name, const
char *init_str)
{
    // though we are alocating memory here, it is not deleted later
    int len = 1;
    if ( init_str != NULL )
        len = strlen(init_str)+1;

    MODEL_NAME[model_instance_num] = new char[len];
    if ( init_str != NULL )
        strcpy(MODEL_NAME[model_instance_num], model_name);
    else
        MODEL_NAME[model_instance_num] = 0;

    INIT_STRING[model_instance_num] = new char[len];
    if ( init_str != NULL )
        strcpy(INIT_STRING[model_instance_num], init_str);
    else
        INIT_STRING[model_instance_num] = 0;

    char buffer[200];

    sprintf(buffer,
            "(instance_num = %u)(instance_name = %s)(init_str = %s)
foreign_model_initialize called\n",
            model_instance_num,
```

```
            MODEL_NAME[model_instance_num],
            INIT_STRING[model_instance_num]
            );

    OutputDebugString(buffer);

    return(1);
}
/*------------------------------------------------------------------
    foreign_model_pre_sim

    This function will be called prior to each transactor simulation event.
    It can be used to deposit state values into the transactor model prior
    to simulating the next event.  Returning 0 results in an error.

     returns:
        uses:
    modifies:
*/
int foreign_model_pre_sim( int model_instance_num )
{
    char buffer[200];

    sprintf(buffer,
            "(instance_num = %u)(instance_name = %s) foreign_model_pre_sim
called\n",
            model_instance_num,
            MODEL_NAME[model_instance_num] );

    OutputDebugString(buffer);

    return(1);
}

/*------------------------------------------------------------------
    foreign_model_post_sim

    This function will be called subsequent to each transactor simulation event.
    It can be used to query transactor simulation state, in order to copy it
    into the foreign model simulator.

     returns:
        uses:
    modifies:
*/
int foreign_model_post_sim( int model_instance_num)
{
    char buffer[200];

    sprintf(buffer,
            "(instance_num = %u)(instance_name = %s) foreign_model_post_sim
called\n",
            model_instance_num,
            MODEL_NAME[model_instance_num] );
```

```
    OutputDebugString(buffer);

    return(1);
}
/*-------------------------------------------------------------------
    foreign_model_exit

    This function will be called just prior to exiting the simulator.
    The routine allows the foreign model to clean up, close files, etc
    before shutting down the program.

     returns:
        uses:
    modifies:
*/
int foreign_model_exit( int model_instance_num )
{
    char buffer[200];

    sprintf(buffer,
            "(instance_num = %u)(instance_name = %s) foreign_model_exit called\n",
            model_instance_num,
            MODEL_NAME[model_instance_num] );

    OutputDebugString(buffer);

    return(1);
}
/*-------------------------------------------------------------------
    foreign_model_reset

    This function will be called just prior to the simulator executing
    a sim_reset command. The routine allows the foreign model to perform
    required reset actions.

     returns:
        uses:
    modifies:
*/
int
foreign_model_reset( int model_instance_num )
{
    char buffer[200];

    sprintf(buffer,
           "(instance_num = %u)(instance_name = %s) foreign_model_reset called\n",
            model_instance_num,
            MODEL_NAME[model_instance_num] );

    OutputDebugString(buffer);

    return(1);
}

/*-------------------------------------------------------------------
```

```
    foreign_model_delete

    this routine will be called just after the simulator deletes all of
    its model state via the "sim_delete" command.
    The routine allows the foreign model to reset itself to stay
    in sync with the simulator.

     returns:
        uses:
    modifies:
*/
int
foreign_model_delete( int model_instance_num )
{
    char buffer[200];

    sprintf(buffer,
          "(instance_num = %u)(instance_name = %s) foreign_model_delete called\n",
           model_instance_num,
           MODEL_NAME[model_instance_num] );

    OutputDebugString(buffer);

    delete [] INIT_STRING[model_instance_num];
    INIT_STRING[model_instance_num] = 0;

    return(1);
}

/*-----------------------------------------------------------------
    GetForeignModelFunctions

    This function is exported as the sole entry point into the DLL version
    of this package.  The Developer Workbench calls it in order to get the
    foreign model entry points for the transactor.  The Workbench thens
    registers these entry points with the transactor.

     returns:
        uses:
    modifies:
*/

extern "C" __declspec(dllexport) void __cdecl
GetVmodForeignModelFunctions(
    int (**ForeignModelInitialize)(int model_instance_num, const char *model_name,
const char *init_str),
    int (**ForeignModelPreSim)(int model_instance_num),
    int (**ForeignModelPostSim)(int model_instance_num),
    int (**ForeignModelExit)(int model_instance_num),
    int (**ForeignModelReset)(int model_instance_num),
    int (**ForeignModelDelete)(int model_instance_num))
{
    *ForeignModelInitialize = foreign_model_initialize;
    *ForeignModelPreSim     = foreign_model_pre_sim;
    *ForeignModelPostSim    = foreign_model_post_sim;
```

```
    *ForeignModelExit        = foreign_model_exit;
    *ForeignModelReset       = foreign_model_reset;
    *ForeignModelDelete      = foreign_model_delete;
}
```

Transactor 7

This chapter describes the Transactor and its command line interface. The Workbench graphical user interface to the Transactor is described in Chapter 2. This chapter contains the following sections:

- Overview (see Section 7.1).
- Invoking the Transactor (see Section 7.2).
- Transactor Commands (see Section 7.3).
- C Interpreter (see Section 7.4
- Simulation Switches (see Section 7.5)
- Pre-Defined C Functions (see Section 7.6)
- Error Handling (see Section 7.7)

7.1 Overview

The C++ simulator is a cycle-based (as opposed to event-driven) 2 and 3 state simulator. It demonstrates the functional behavior and performance characteristics of a system design based on the IXP2400, IXP2800, and IXP2850 network processors without relying on the hardware.

Z state is not explicitly modeled; instead tristate nodes automatically flag floating error when 1 or more bus bits float for a period greater than the user-specified float threshold.

Two simulation modes are supported:

- "2.1" state simulation:
  Tristate nodes are 3-state (0,1,X)
  All other nodes are 2-state (0,1)
- "4" state simulation:
  All nodes are 4-state (0, 1, X, High Z)

Several commands are required at the simulator console before executing cycles. This sequence can be entered line-by-line or encapsulated inside of a text based instruction file. The example instruction file below illustrates the command sequence. The "@" symbol instructs the simulator to open and begin executing lines in the file name that follows.

```
// File: run_fifo_test.ind
// Invoke from console prompt by typing: @C:\Vmodel\run_fifo_test.ind

// Instantiate the model
inst fifo_test

// Init the model
init
```

```
// Set the clocks. Note that the parameters vary for the IXP2400
// and IXP2800

// Uncomment one of the following function calls depending on
// the network processor. Use of both will not work in the
// Transactor.

// Following are the defaults for the IXP2800
//set_clocks(sr_chip_name, 2800, 0x73E7777, 66, 0x1d301d3, 0x1d3);

// Following are the defaults for the IXP2400
//set_clocks("", 1200, 0x00040033, 66, 0x00ff0000);

// Optional - log all commands and responses to a file.
log fifo_result.log

// Your command sequences here. (go cycles, deposit, examine, watch, etc…)

go 1
...
...

// Close the log file
close fifo_result.log

// Exit the simulator. This will close the simulator window.
exit
```

Before running the first cycle, clocks need to be setup. If a model supports multiple clocks, then the phase relationship and duty cycle need to be configured. The `set_clocks` command sets all normal transactor clocks. Use the `set_clock` command to set additional clocks if needed. You may enter "`help set_clock`" at the simulator console for additional information.

The ellipses in the listing above indicate where other instructions are be inserted. This is where a `deposit` or an `examine` of model state elements takes place.

7.2 Invoking the Transactor

You invoke the Transactor by specifying the appropriate executable to run:

```
IXP2400.exe [/h] [/b] [script_file_name1 script_file_name2 ...]
or
IXP2800.exe [/h] [/b] [script_file_name1 script_file_name2 ...]
or
```

The optional fields take the following form:

Table 6. Transactor Optional Switches

`/h`	Prints basic help information then exits the program.
`/b` *`script_file_name1 ...`*	Indicates that the program is intended to run in batch mode. Using this option means that one or more script files are expected to be specified. The program will execute each script file in left to right order, then will automatically exit.
`script_file_name1 script_file_name2 ...`	Specifying script file names without the /b option tells the program to assume an interactive session. The program will execute each specified script file, but will then sit and wait for console input from the user instead of exiting automatically.

7.3 Transactor Commands

Transactor commands fall into four functional categories: initialization, simulation, debugging, and miscellaneous. The sections that follow list the Transactor commands in alphabetical order. Each command description can include:

- The symbolic command name (e.g., **examine**).
- A short description of what the command does.
- The format of the command.
- Command parameter descriptions, and where appropriate, definitions of predefined parameters.
- One or more examples illustrating the use of the command.
- Optional input strings delimited by bracket characters ({}).

The table that follows is a quick reference to the Transactor commands:

Table 7. Transactor Commands (Sheet 1 of 2)

Command Name	Command Type	Section
#define	Debugging	7.3.1
#undef	Debugging	7.3.2
@	Miscellaneous	7.3.3
benchmark		7.3.4
cd		7.3.3
close	Debugging	7.3.6
connect	Miscellaneous	7.3.7
deposit	Debugging	7.3.8
dir		7.3.9
examine	Debugging	7.3.10
exit	Miscellaneous	7.3.11
force		7.3.12
foreign_model		7.3.13

Table 7. Transactor Commands (Sheet 2 of 2)

Command Name	Command Type	Section
go	Simulation	7.3.14
go_thread		7.3.15
gop		7.3.16
goto	Simulation	7.3.17
goto_addr		7.3.18
help	Miscellaneous	7.3.19
init	Initialization	7.3.20
inst		7.3.21
load_ixc		7.3.22
log	Debugging	7.3.23
logical		7.3.24
path	Debugging	7.3.25
pwd		7.3.26
remove		7.3.27
root_init	Initialization	7.3.28
set_clock	Initialization	7.3.29
set_default_go_clk		7.3.30
set_default_goto_filter		7.3.31
set_float_threshold		7.3.32
show_clocks		7.3.33
sim_delete		7.3.34
sim_reset	Simulation	7.3.35
time	Miscellaneous	7.3.36
trace	Debugging	7.3.37
type		7.3.38
ubreak	Debugging	7.3.39
unforce		7.3.40
version	Miscellaneous	7.3.41
watch	Debugging	7.3.42

7.3.1 #define

Format:

`#define` *`macro_name`* `|` `(`*`comma_separated_macro_arguments`*`)` `|` `macro_text`

Definition:

Implements the text substitution function of the C preprocessor. It allows a user to specify text that is automatically substituted into every command line before the line is interpreted. This function allows users to customize the command interface.

macro_name If the `macro_name` has arguments associated with it, the arguments are substituted into the macro_text whenever the corresponding formal argument name matches a token of the macro_text.

`comma_separated_macro_arguments`

When a macro has substitutable arguments, three preprocessor operators may be applied to modify the resulting string. They are:

Can be inserted between two tokens in the macro_text to cause the two adjacent tokens to be concatenated.

Can be prepended onto a macro_text token causing the following token to be quoted.

Can be prepended causing the following token to be de-quoted if quote characters exist as the first and last characters of the token.

Whenever a preprocessor operator is applied, the one or two tokens associated with the operator are not expanded. In all other cases, every token in the macro_text is recursively expanded (if possible) based on other existing macro definitions. If the recursive expansion of a macro name yields a token of the same name, no further expansion occurs.

Example:

```
#define RESULT_LOW_ERROR "Mismatch of ResultLow
<31:0> with expected return value. \n"
```

Related commands:

`#undef`

7.3.2 #undef

Format:

`#undef` *`macro_name`*

Definition:

Deletes a previously defined preprocessor macro name.

Related commands:

`#define`

7.3.3 @

Format:

`@`*`cmd_file_name`*

Definition:

Executes a series of simulation commands in a specified command script file name.

Notes:

`cmd_file_name` uses an extension of ".ind"

7.3.4 benchmark

Format:

```
benchmark
```

Definition:

Prints the current CPU and sets a flag the first time it is called. On the subsequent invocation it prints the end CPU time and the total time elapsed.

Example:

```
> benchmark
Benchmark last: 4.466 secend: 4.466 secdelta: 0.000 sec
Wallclock start: 14.43:29end: 14:43:29delta; 00.00.00
```

7.3.5 cd

Format:

cd |*file_spec*|

Definition:

Changes the work directory. Analogous to the DOS “cd” command.

If the simulation is run from the vmod tool environment, a vmod logical name that prefixes the *file_spec* (starting with’$’) will be automatically translated. If no parameters are entered, cd prints the current working directory.

Examples:

```
>cd C:\Castine\Projects\Sausalito
C:\Castine\Projects\Sausalito
>cd
C:\Castine\Projects\Sausalito
```

7.3.6 close

Format:

close *log_or_trace_file_name*

close/all

Definition:

Closes a previously opened log or trace file. “close/all” closes all open log and trace files. Exiting the simulator will automatically close all open log or trace files.

7.3.7 connect

Format:

connect *top_level_net_name top_level_instantiated_pin name* [*top_level_instantiated_pin_name* ...]

Definition:

Permits a dynamic connection between instances by the creation of nets and assigning names to these nets.

Example:

The following snippet of simulator code creates two instantiations (InstA and InstB), and sets up several port connections and an instantiation-to-instantiation connection:

```
int MSB = 191;
int MID_MSB = 96;
int MID_LSB = 95;
int LSB = 0;

inst connect_test_insta InstA
inst connect_test_instb InstB

connect NEW_Output1<MSB:MID_LSB> InstA.AOutput1// note 1
connect NEW_AtoB<MID_MSB:LSB> InstA.AOutput2 InstB.Binput//note 2
connect NEW_Output1<MID_MSB:LSB> InstB.Boutput//note 3

 /////////////////////////////////////////////////////////

 Note 1: Net NEW_Output1<MSB:MID_LSB> connects to port InstA.Aoutput1
 Note 2: Net NEW_AtoB<Mid_MSB:LSB> connects InstA.Aoutput2 to InstB.Binput
 Note 3: Net NEW_Output1<MID_MSB:LSB> connects to port InstB.Boutput
```

7.3.8 deposit

Format:

dep|*osit*| |*deposit_qualifiers*|

state_spec| [*index_range*] ||<*bit_range*>| = *deposit_expr*

Definition:

Evaluates the C numeric expression, *deposit_expr*, and deposits the resulting number into the specified state. The wildcard character, “*” can be used to avoid specifying the entire state name, however, unless the “/multiple” qualifier is specified, the wildcard specification must unambiguously address only one state value.

index_range — The index_range specification is only relevant for arrays and can be a single C numeric expression or 2 numeric expressions separated by a “:” to indicate the inclusive range formed between the two numbers.

bit_range — The bit_range spec has the same form as the index_range; if it is not specified the whole field is assumed.

qualifiers

/silent — Inhibits reporting the deposit action.

/multiple — Allows a wildcarded name to deposit to multiple states.

/force[=n[:prim_clk_name]] Applies a force to the state after depositing the value. “n” represents the number of clock cycles for which the force remains in effect; if not specified, then it is forced indefinitely until it is removed by an “unforce” command or another force operation. “prim_clk”name” specifies the primary clock period which n refers to; it defaults to the clock specified in the command prompt.

Example:

```
> dep/s AEnable5<'ENABLE_LSB> = 0
```

7.3.9 dir

Format:

dir |*file_spec*|

Definition:

Analogous to the DOS “dir” command. Displays a list of files and sub-directories in a directory. If a simulation is run from the Vmod tool environment, a Vmod logical name that prefixes the *file_spec* (starting with '$') will be automatically translated

7.3.10 examine

Format:

```
ex|amine| |qualifier_list| state_spec|
[index_range]|<bit_range>|
```

Definition:

Examines the current state of one or more simulation states or user-defined variables. The wildcard asterisk character (*****) can be used in the *state_spec* to specify multiple states to be examined.

index_range — The index_range specification is only relevant for arrays and can be a single C numeric expression or two numeric expressions separated by a period (.) to indicate the inclusive range formed between the two numbers, or a “:+” to indicate the inclusive range formed by treating the second number as an offset from the first.

qualifier_list — Optional qualifiers may be applied to constrain the examination of multiple variables to specific state types defined by the qualifiers. Any number of qualifiers may be applied. They are:

/array — Data array.
/artifact — Model artifact state (does not model real hardware).
/delay — Bus transfer delay element.
/fifo — Queue structure.
/function — User-defined C function.
/register — Flip-flop or latch hardware state element of 32 bits or less.
/signal — Combinatorial hardware state element of 32 bits or less.
/statistic — Performance data collection and display facility.
/struct — User-defined C struct definition.
/ustore — Microcode control store.
/variable — User-defined C variable.
/watch — User-defined watch function.

state_spec — Any predefined simulation state or user defined state that holds a numeric value. See Appendix A for more details on states.

bit_range The bit_range spec has the same form as the index_range. If it is not specified, the whole field is assumed.

Example: In the following simulator code snippet, a user-defined variable called "pass" is created and assigned a value of "1", and then examined using the examine command.

```
>>> int pass;
>>> pass = 1;
1
>>> examine pass
pass<31:0> = 00000001 (1) (C interpreter variable)
>>>
```

7.3.11 exit

Format:

```
exit
```

Definition:

Closes all open log files and then exits the simulator. The Transactor main routine return status returns the last recorded value of sim.error_count (the number of recorded errors that have occurred).

7.3.12 force

Format:

```
force [expiration_cycles[:primary_clk]]
[state_name1[state_name2...]]
```

Definition:

Sets the data contents of specified state(s) to be unchangeable by any means. Non-model states and states that behave as unconditional clock nodes cannot be forced.

expiration_cycles

If “expiration_cycles” is specified, the forced state will be automatically removed after the specified number of cycles of the specified primary clock is simulated; otherwise the force will be held indefinitely until it is manually removed or re-forced.

Primary_clk If no primary clock is specified, the default clock shown at the prompt is used. Wildcards may be used to specify multiple states.

state_name If no state is specified, the force command lists all state that are currently forced. use “unforce” to remove the force from the state.

Example:

In the example below, a state variable is first examined, and its value is subsequently changed. Then the variable is forced, after which another state change is attempted --- unsuccessfully. Finally, the variable is unforced, and subsequently its value is successfully changed.

```
>>> examine in1
in1<31:0> = 00000000 (0) (signal:primary_input)
```

```
    >>> dep/s in1 = 0x11111111
    >>> examine in1
    in1<31:0> = 11111111 (286331153)(signal:primary_input)
    >>> force in1
    tate, "in1" is forced.
    State, "in1" is no longer forced.
    State, "in1" is forced.
    ; FORCES=1>>> dep/s in1 = 0x00000000
    ; FORCES=1>>> examine in1
   in1<31:0> = 11111111 (286331153) ======(signal:primary_input:forced_state)
    ; FORCES=1>>> unforce in1
    WARNING: Unforcing state, "in1", has no effect because it ===was not previously
f   orced.
 >>> examine in1
  in1<31:0> = 11111111 (286331153) ======(signal:primary_input)
 >>> dep/s in1 = 0x00000000
 >>> examine in1
  in1<31:0> = 00000000 (0) (signal:primary_input
```

7.3.13 foreign_model

Format:

```
foreign_model |/delete| dll_name model_inst_name
[model_init_str] [call_priority]
```

Definition:

Registers a foreign model instance with the simulator so that it will be simulated in lock step with this simulation.

/dll_name	The specified dll is assumed to adhere to the properties required for supporting foreign model simulation (see xact_vmod.h for more details)
/model_inst_name	The model_inst_name must be a unique name among all names of instantiated foreign models.
model_inst_str	the model_inst_str is a user-specified string that will be passed to the foreign model initialization routine; this enables user-specified initialization information to be passed to the foreign model.
/call_priority	The call_priority argument is an integer value that specifies the priority in which this foreign model is called relative to all other foreign models. the higher the number the earlier it is called. This argument defaults to 0.
/delete	If the “/delete” qualifier is specified, this command deletes the previously instantiated foreign model.

7.3.14 go

Format:

```
go |/silent| |/clk_domain| |cycle_count|
```

Definition:

Simulates the number of clock cycles in a given clock domain specified by “cycle_count”. If no cycle_count is given, “cycle_count” defaults to 1. If the cycle_count is specified as -1, simulation will continue indefinitely until a simulation break event occurs due to an error or assertion of the sim.halt variable.

/silent	Suppresses debug information during the simulation.
/clk_domain	Replace “/clk_domain” with any top-level primary clock node name to specify the clock domain used when simulating a cycle count. Otherwise the domain defaults to the domain specified in the “set_default_go_clk” command.
cycle_count	If “next” is specified in place of the clock name, then the simulator will simulate the next x scheduled simulation events (rather than clock cycles), where x is specified by the “cycle_count” value.

7.3.15 go_thread

Format:

```
go_thread |/silent| |/max_cycle_count| |goto_filter|
|cycle_count|
```

Definition:

Simulates the specified number of instructions in the specified thread of the specified micro-sequencer instance name. The actual number of cycles simulated may be larger than the instruction count due to context swaps.

/silent	If the “/silent” qualifier is specified, debug information is suppressed during the simulation.
/goto_filter	If “goto_filter” is not specified, the default goto filter will be used (see “set_default_goto_filter” for the goto filter syntax).
cycle_count	If no cycle_count is given, the count defaults to 1.

7.3.16 gop

Format:

```
gop |/clk_domain| |/silent| |phase_count|
```

Definition:

Simulates the specified number of clock phases (where 2 clock phases equal 1 clock cycle). If the clk_domain is not explicitly specified, the current default clock is used.

7.3.17 goto

Format:

```
goto |clk| |/silent| cycle_target
```

Definition:

The goto command simulates up until the specified target cycle number has been reached. You can specify what primary clock input to use to measure the target cycle number; otherwise the goto command will use the current default clock (set by the set_default_go_clock command) to determine the target cycle.

`/silent`	If the "/silent" qualifier is specified, debug information is suppressed.

7.3.18 goto_addr

Format:

```
goto_addr |/silent| |/max_cycle_count| |goto_filter|
uaddr_or_label ...
```

Definition:

The goto_addr command halts the simulator when one of the specified microaddresses has been reached in one of the microsequencer units specified by the goto_filter.

`/silent`	If the "/silent" qualifier is specified, debug information is suppressed during simulation.
`/max_cycle_count`	If max_cycle_count is specified, simulation will unconditionally stop once the cycle count has been reached. The max_cycle_count is assumed to be specified in the clock domain that is currently the default clock (see "set_default_go_clock" for more information).
`goto_filter`	If goto_filter is not specified, the default goto filter will be used (see "set_default_goto_filter" for the goto filter syntax).
`uaddr_or_label`	One or more uaddr_or_label specifications may be entered. A microaddress is specified by the decimal address immediately followed by "#"; a label is specified by the alphanumeric label name immediately followed by "#".

7.3.19 help

Format:

```
help |topic_or_command|
```

Definition:

Displays helpful information about a topic or command. Help is available for the following commands:

`#define`	`#undef`	`@`	`benchmark`
`cd`	`close`	`connect`	`debug`
`def_stat`	`def_stat_condition`	`deposit`	`diff_trace`
`dir`	`examine`	`exit`	
`force`	`foreign_model`	`go`	`go_thread`
`gop`	`goto`	`goto_addr`	`init`
`inst`	`load_ixc`	`load_meta`	`log`
`logical`	`path`	`profile`	`pwd`

`remove`	`root_init`	`set_bus_stats`	
`set_clock`	`set_default_go_clk`	`set_default_goto_filter`	
`set_float_threshold`	`show_clocks`	`sim_delete`	`sim_reset`
`time`	`trace`	`type`	`ubreak`
`unforce`	`version`	`watch`	

Example: `> help examine`
Get help on the examine command.

7.3.20 init

Format:

```
init
```

Definition:

This command should be executed once all cells have been instantiated using the “inst” command. “init” initializes the model for simulation. It schedules the clocks and ties all cell hierarchy together by linking port states between all parent and child instances.

7.3.21 inst

Format:

```
inst cell_name [inst_name]
```

Definition:

Instantiates the specified cell using the specified instance name. all descendent cells are also instantiated. Note that multiple instantiations must all have unique instance names.

`inst_name` The inst_name can be omitted during exactly one instantiation in which case the instance name is an empty string.

Simulation states are automatically created to connect to the port connections of the instantiated cell. The names of these states are of the form `inst_name.port_name` (unless the instance name is the empty string in which case the name is simply `port_name`).

Examples:

`inst test2` (instantiate cell "test2")
`inst test2 foo` (instantiate cell “test2” as “foo”)

7.3.22 load_ixc

Format:

```
load_ixc [useq_spec | ustore_stats_name] ucode_file_name
```

Definition:

"load_ixc" loads and validates all microcode in the control store. The control store can either be specified directly by name, or by using the chip/useq format used in other commands such as "set_default_goto_filter". If this format is used, IXP code will be loaded into all control stores that map to the specification.

7.3.23 log

Format:

```
log file_name |commands| |responses|
```

Definition:

Opens a log file of the specified name to log various simulation information. A log file may be closed using the "close" command All log files are automatically closed when the simulator exits.

`file_name`	If the file_name is not specified, the log command displays all currently open log files.
`commands`	Logs all simulator commands typed by the user.
`responses`	Logs all general simulator output. If no optional switch is specified the default assumes both "commands" and "responses".

7.3.24 logical

Format:

```
logical [logical_name] [= logical_definition]
```

Definition:

If both optional parameters are specified, the specified logical name is defined to be the specified logical definition string

If no definition string follows the "=", then the logical_name is deleted from the list of logicals. This logical definition is saved in the Win32 registry.

If only the logical_name field is specified, the current definition for this field is displayed.

If neither field is specified, all logical definitions are listed.

Examples:

(1) The following Simulator call illustrates how to list existing logicals:

```
>>> logical
Logical: $model_dir = "c:\vmod_models"
Logical: testdir = "c:\test"
Logical: newworkingdir = "c:\test\subtest"
Logical: newworkdir = "c:\test\subtest"
```

(2) The following Simulator calls illustrate how to define a logical, list it, and delete it:

```
>>> logical newtestdir = "c:\test\subtest"
>>> logical newtestdir
Logical: newtestdir = ""c:\test\subtest""
>>> logical newtestdir =
>>> logical newtestdir
No logical, "newtestdir", exists.
```

7.3.25 path

Format:

```
path |;||path_spec|\;path_spec||...|
```

Definition:

Analogous to the DOS **path** command. Allows the user to specify the search list of folders which is used to open files in the Transactor. Typing the command with no arguments displays the current path setting. Typing the command followed by a semicolon resets the path list to look only in the current folder area. Typing the command followed by a list of folder paths (separated by semicolons) specifies the list of folder paths that are searched in left to right order. The special keyword **%path%** specifies the previously existing search list.

Example:

```
>>> path c:\test;c:\vmod
```

Path search list for finding files (use "path" cmd to change this):

```
1: c:\test
2: c:\vmod
```

7.3.26 pwd

Format:

```
pwd
```

Definition:

Displays the current working directory.

7.3.27 remove

Format:

```
remove [/silent] state_name1 [state_name2...]
```

Definition:

Removes the specified simulation state(s). Note that this command will only remove user-defined states; states that are defined by the hardware model cannot be removed. Wildcard specifications may be used with this command.

/silent	Suppresses informational messages resulting from this command.

7.3.28 root_init

Format:

```
root_init
```

Definition:

This command instantiates, and then initializes, the root cell. this command works only when no previous cell has been instantiated.

7.3.29 set_clock

Format:

```
set_clock |/silent| primary_clk_name high_time low_time
[starting_offset]
```

Definition:

Sets the characteristics of the specified primary clock input provided that the clock was designated as automatically driven by the simulator (i.e. it was not marked as “undriven” in the built model). Specify the number of time units of the high and low clock levels.

/silent If set, the new clock configuration is not echoed back.

primary_clk_name high_time low_time

If no clocks are specified in the simulation, all clocks are set to a common clock waveform of 1 high time unit, 1 low time unit and a starting offset of 0. All unspecified clocks assume the waveform characteristics of the first clock to be specified.

starting offset By default the starting point of the clock at t=0 will be at its rising edge transition. You can alter this point by specifying a starting offset value relative to the rising edge of the clock

If, for example you specify an offset of 2, then the starting point of the clock at t=0 is 2 time units after its rising edge.

Note: This command can only be executed prior to model simulation; the clock frequencies cannot be changed once the simulation time has advanced past t=0.

7.3.30 set_default_go_clk

Format:

```
set_default_go_clk|primary_clk_name|
```

Definition:

Sets the default clock domain default used by the “go” and “gop” commands when simulating a cycle count. If no clock name is specified, the clock chosen by the simulator at initialization time is assumed.

7.3.31 set_default_goto_filter

Format:

```
set_default_goto_filter chip_inst_name (useq_thrd_spec ...)
```

Definition:

Specifies the default combination of microsequencers and threads that will apply to the “goto_addr” and “ubreak” commands if this specification is not explicitly included with these commands.

useq_thrd_spec One or more useq_thrd _spec specifications may be included within the parentheses delimited by spaces. The useq_thrd_spec consists of a comma-separated list of microsequencer numbers followed by a colon, followed by a comma-separated list of thread numbers. The colon and thread specification can be left out in which case all threads are assumed. Additionally, a range of microsequencer or thread numbers can be specified by 2 numbers delimited by a “-”; all microsequencers or all threads can be specified by “*”.

7.3.32 set_float_threshold

Format:

```
set_float_threshold [[clock_name:]cycle_num | 0| -1]
tristate_name [tristate_name...]
```

Definition:

Sets the tri-state floating threshold on one or more tri-state states. The tri-state floating threshold is the minimum acceptable time that can pass while a tri-state node is not driven. If this threshold is reached, an error is generated indicating that the tri-state node has been undriven too long.

Specified states that are not tri-state states are ignored.

A value of 0 implies that the state must always be driven in order to avoid an error message. A value of -1 disables the threshold check so that no error will ever be generated. A value other than 0 or -1 specifies the threshold time in terms of the clock periods of the specified clock.

If no clock is specified, the default clock shown in the prompt is used.

Wildcards:

Wildcards may be used to specify multiple tri-state nodes more efficiently.

7.3.33 show_clocks

Format:

```
show_clocks
```

Definition:

Shows the waveform characteristics of all primary clock inputs which are automatically driven by the simulator. Use `set_clock` to set the waveform characteristics of individual clocks.

7.3.34 sim_delete

Format:

```
sim_delete
```

Definition:

Deletes the entire instantiated simulation model and all associated model states. Memory for all simulation states is freed by this command. A new model maybe instantiated (via the `inst` command) and initialized (via the `init` command) after executing this command.

7.3.35 sim_reset

Format:

```
sim_reset
```

Definition:

Resets the state of the model to the point just after the model was first initialized. Therefore, all states predefined by the logic model are reset and all user-defined states are deleted. User-defined states include all int's, strings, vectors, watches, breakpoints, and functions. All open trace files are closed. Previously loaded foreign models remain in effect.

7.3.36 time

Format:

```
time
```

Definition:

Prints the current wallclock time.

Example:

```
>>> time
time: 14:20:30 date: 05/16/02
```

7.3.37 trace

Format:

```
trace |[/dino|/vcd|vcd+]|file_name name_list
```

Definition:

Opens a trace file to log simulation data over time for subsequent use by a waveform editor.

/dino	specifies the binary template format for "dinotrace".
/vcd	Specifies the ASCII Verilog trace format.
/vcd+	specifies the Verilog binary trace format.
file_name	Specifies the name of the file that contains the trace logging data.
name_list	A list of simulation states (separated by blanks). If "@" is prepended to a name, the name then refers to a file from which additional names will be derived.
Wildcards	The wildcard character, "*", may be used as a shorthand method of specifying one or more state names.
Arrays	If a signal is an array, an index range specification must be applied to address particular elements within the array.
index_range	The index_range specification (delimited by []) can be a single C numeric expression or 2 numeric expressions separated by a ":" to indicate the inclusive range formed between the two numbers.
Closing	A trace file may be closed using the "close" command. All trace files are automatically closed when the simulator is exited, or when the restore command is executed. Trace files are automatically reopened whenever a restore operation is executed which restores simulator state to the time point when that trace was first opened.

7.3.38 type

Format:

```
type file_spec
```

Definition:

Prints the contents of the specified file to the console.

7.3.39 ubreak

Format:

```
ubreak break_name |quoted_callback_function||goto_filter|
|uaddr_or_label| ...
```

Definition:

Creates a microcode breakpoint state. A ubreak breakpoint will unconditionally cause the simulator to stop when that micro-PC address is reached by a thread that is enabled to respond to that breakpoint.

The breakpoint is enabled upon creation. The breakpoint can be disabled or re-enabled by depositing a 0 or 1 respectively via the deposit command.

`uaddr_or_label` Any number of breakpoint addresses can be related to the breakpoint state by specifying 1 or more micro-addresses in the form of either number# or label#.

`goto_filter` The goto_filter specification is used to designate which threads of which micro-sequencers the breakpoint specification should apply to. If the goto_filter is not specified, the configuration set from the "set_default_goto_filter" is applied by default.

See `set_default_goto_filter` for the syntax of the goto_filter.

`quoted_callback_function`

The quoted_callback_function_name is an optional argument. If it is specified it is used to conditionalize the ubreak event. It must be a quoted string specifying the name of a pre-existing C-interpreted function name or pre-existing imported function name (do not specify parenthesis, argument types, or return types). When a potential ubreakpoint is reached this function is called with the current ubreak state information. If the return value from the function is non-zero, the ubreak is activated causing the simulation to halt; otherwise it is ignored. The absence of the quoted_callback_function_name argument causes the ubreakpoint to unconditionally stop simulation. The typedef designation for the C-interpreted callback function must be:

```
int func_name( string chip_name, int me_num, int ctx_num, int PC );
```

The typedef designation for the imported callback function must be:

```
int func_name( char *chip_name, int me_num, int ctx_num, int PC );
```

If a function is associated with a ubreak and is then subsequently removed, the ubreak will automatically be removed since it can no longer operate as it was defined.

7.3.40 unforce

Format:

```
unforce state_name1 [state_name2 ...]
```

Definition:

Sets the data contents of specified state(s) to no longer be forcible (see `force` command for related information.)

Wildcards Wildcards may be used to specify multiple states.

7.3.41 version

Format:

```
version
```

Definition:

Displays the software version and build data for the simulator.

7.3.42 watch

Format 1:

```
watch [/clk=clk_ref_name | /clear] state_name1 [state_name2...]
```

Definition:

The above format is constructed as a simulator command that takes 1 or more state names. It is used to automatically print all state transitions of the specified states. The optional clk_ref_name switch allows a primary clock name to be related to the watch. By doing so, each transition will be timestamped relative to the cycle of the specified clock. If no clk_ref_name is specified, the current default clock (set by the “set_default_go_clock” command) is used.

`/clear` Disables a pre-existing watch; if it is not specified, it is assumed a new watch is being set.

Format 2:

```
watch (watch_name, state_name1, state_name2, ..., state_namen)
C_statement
```

Definition:

The second watch format is recognized as a C function call. It is designed to allow the user to specify some arbitrarily complex action to occur on some arbitrarily complex set of conditions. The watch defines a watch state of the specified name that looks for state transitions for each of the specified state names after each simulation event completes.

If 1 or more states are found to have transitioned, then the specified C statement associated with the watch is executed; otherwise, no activity takes place.

`C_statement` The user can specify a C code block as the C_statement and thus, can further specify the exact watch trigger condition and watch function through a set of C statements that reference specified states, or any other simulation state.

Enabling Watches are automatically enabled when defined. Watches can be disabled and re-enabled by depositing a zero or non-zero value to the watch_name state (e.g. “dep watch_name = 0”).

7.4 C Interpreter

The C++ simulator contains a built-in C interpreter for scripting capability. This enables script files to run a sequence of native simulator and C Interpreter commands for regression testing a model. This help topic is designed to give a brief overview of the C Interpreter. Detailed information regarding all of the commands supported can be found in the simulator console help. Simply enter the word "help" at the transactor console to bring up the simulator on-line help system

The C interpreter supports:

- if, while, for, break, continue, user-defined C functions and many built-in predefined C functions (see on-line help for complete list)
- all unary and binary operators

The C interpreter does not support "do {} while()" or "expr ? expr1 : expr2".

Data types supported

- int: 32-bit integer defined by user in normal was (e.g. "int foo;")
- vectors: user-defined unsigned word whose width is defined by the user.

A vector is defined by the following built-in C function:

```
def_vector( name, width );
```

Arbitrary bit fields of a vector can be referenced by the following syntax:

```
vector_name(bit_position) //references the specified bit
vector_name(hi_bit,lo_bit) //references the field bounded by hi_bit and lo_bit
inclusive
```

Arbitrary precision Verilog arithmetic is supported by the vector data type.

Simulation states: all model simulation states are implicitly defined as an unsigned word of width specified by the state:

Arbitrary bit fields can be referenced in same manner as is supported by vectors. In addition, a third argument can specify a word index, if the corresponding state is an array (e.g. "foo(3, 2, 1)" references bit range [3:2] of "foo[1]").

7.4.1 C macros supported

Other non-C extensions supported by interpreter:

- Verilog constants can be specified: (e.g. 44’hdeadbeef or 8’b0x11_xx11)
- define (tic define) values defined in the model can be referenced

7.4.2 Supported Data Types

The Vmod C interpreter supports the following datatypes:

- int: The standard 32-bit signed datatype. They are defined in the normal way by "int foo;"
- model states: Any model simulation variable defined by the compiled hardware model can be referenced in a C expression simply by specifying the pre-existing state name (e.g "i.c0.cmd_req = 1;"). In addition, bit fields specifications for a model state can be specified by appending a function argument list to the end of the state name. For non-arrayed states, the argument list is: "(int msb, int lsb)". The "lsb" argument is optional and defaults to the "msb" value. Additionally, a single colon-separated argument of the form "msb:lsb" is also supported for specifying bit ranges. For arrayed states, the argument list is

```
"( int array_index, int msb, int lsb )"
```

If neither "msb" nor "lsb" is specified, the entire state width is assumed. If only "lsb" is not specified, it defaults to the "msb" value. Here are some examples

```
i.c0.cmd_req( 3 );   // bit extracts bit 3 of ti.c0.cmd_req
i.c0.cmd_req( 5, 3 ) = 5; // field inserts 5 into ti.c0.cmd_req[5:3]
i.c0.cmd_req( 5 : 3 ) = 5; // identical to above form
i.ustore( 23 );  // references whole word of i.ustore[23]
i.ustore( 23, 11:6 ) = 12; // field inserts 12 into i.ustore[23]<11:6>
i.ustore( 23, 7 );  // extracts bit 7 of i.ustore[23]
```

- vectX: A user-specified unsigned vector datatype whose bit width is specified by X (where X is a positive integer). The interpreter supports all logical, equivalence, bit-wise and arithmetic binary operators on the specified bit precision of the vector (e.g. two 100-bit vectors can be added to produce a 100-bit result). A vector declaration has two forms: "vect128 foo;" defines a 128-bit vector called foo; "vect(expr) foo;" defines a vector, foo, of width specified by the C expression. Like model states, bit field specifications may be made by appending a one or two-argument function list.
- synonym states: A synonym state can be defined in order to map another state or a particular bit field of a state to a preferred name. The synonym state name can then be used in place of the former state to reference the former state. Creation of a synonym state with its defined state mapping is done by calling the "def_syn" C function. See topic "Built-In C Functions" for more information on "def_syn".
- string: The keyword "string" defines a variable length text string container (e.g. "string foo;" declares a string called foo). Text can be assigned to a string via the "=" operator (foo = "abcd";). Text can be appended via the "+" or "+=" operators (foo = "abcd" + "xyz";). The relational operators, "==", "!=", "<", "<=", ">", and ">=" can be applied to 2 strings to produce a boolean result. In addition, many built-in string functions can be applied using the syntax "string_var_name.string_function_name(...)". The following string functions exist:

`length():` returns the number of characters in the string

`empty():` resets the string to the null string

`isempty():` returns 1 if it contains the null string or 0 otherwise

`left( int n ):` returns the left-most n characters of the string

`right( int n ):` returns the right-most n characters of the string

`mid( int n, int len ):` returns the substring starting at index n, and whose length is limited to no more than len characters. If len < 0, or if len is not supplied, the entire substring starting at n is returned.

`find( string s ):` returns the zero-based starting index of the left-most occurrence of string s in the string. If s is not found in the string, -1 is returned.

`reverse_find( string s ):` returns the zero-based starting index of the right-most occurrence of string s in the string. If s is not found in the string, -1 is returned.

`format( format_str, ... ):` Formats the contents of the string, according to the "printf" variable argument list

7.5 Simulation Switches

Several pre-defined simulation states exist that act as user setable simulation switches/parameters. All such states exist under the "sim." hierarchy. The following is a description of the states:

- `sim.error_count`: Indicates the number of unexpected errors that were flagged during the session.
- `sim.error_handle_mode`: Indicates what action should be taken when the error occurs. The valid values are:
  - **0** suppress and ignore error message
  - **10** print the error but ignore its occurrence (i.e. the sim.error_count is not incremented and the simulation continues)
  - **20** turn model errors into warnings
  - **40** print the error, increment sim.error_count and halt simulation and command line/script file execution if any is in progress
  - **100** print the error, increment sim.error_count, halt all execution and then exit the simulator. The default value is 40
- `sim.halt`: This state is normally 0. If the user sets it to 1, the model halts (if running) at the end of the current cycle. This is useful in watch statements when it is desired for the model to be halted when a specific condition has been met. This state must be reset to 0 in order to continue running the model.
- `sim.show_hidden_states:` This state is normally 0. Enable this state to see internal simulation states that are not usually useful/relevant to user simulation.
- `sim.time`: Represents current simulation time. This variable is read-only.

7.6 Predefined C Functions

The Vmod Simulation Console has a number of predefined C functions, which may prove useful when performing simulations

`cmd( quoted_cmd_string )` Executes the quoted_cmd_string as a simulation command. This allows non-C commands to be embedded inside C commands

`def_syn( syn_var, mapping_state )`: Declares a synonym state that is associated to the specified mapping state name. The mapping state specification may include a C bit specification so that the synonym maps to a subset bit field of the mapping state. For example:

— a.// synonyms "cmd_bus" to entire state, "chip.i.cmd_bus" def_syn(cmd_bus, chip.i.cmd_bus);

— b.// synonyms "xfer_reg" to "chip.i.cmd_bus[53:48]" def_syn(xfer_reg, cmd_bus(53:48));

`env_var( char *environment_variable )`: Returns 1 if the specified environment variable exists; otherwise it returns 0.

`expect_err( error_code, error_cnt, compare_cnt )`: This function is for automated QA testing and is not intended for general use. If compare_cnt = 0, the function records error_cnt as the current outstanding expected error count for the specified error code. If compare_cnt = 1, the function compares the current outstanding error count for the specified error code against the specified error_cnt value. If the comparison fails, an error is generated.

`field( state_name, field_msb, field_lsb )`: Returns the value of the data field defined by the input arguments. This function is provided for backward compatibility only. The preferred method is to append a "(int msb, int lsb)" function argument list to the state name.

`field_insert( state_name, insertion_data, field_msb, field_lsb ):` Inserts the insertion_data into the specified state at the specified bit field. The insertion_data must be <= 32 bits. The function returns 1 if successful, otherwise 0. This function is provided for backward compatibility only. The preferred method is to append a "(int msb, int lsb)" function argument list to the state name you're assigning to.

`fprintf( file_name, fmt, ... ):` Analogous to the C command fprintf, except that the first argument is a file name, not a file pointer. If the log file corresponding to the file name was not previously open, it is automatically opened by this call.

`hex_str( state_name ):` returns a hexidecimal string representing the current state value

`is_valid_state( state_name ):` Returns 1 if the specified state_name currently exists otherwise it returns 0

`log2( int ):` Returns the base 2 log of the specified number. Fractional amounts are rounded up to the next whole number.

`printf( fmt, ... )`: Analogous to the C printf command.

`project_dir()`: Returns a string representing the directory where the .vmp file existed to create this simulation model

`rand( bound1, bound2 ):` Returns a random integer between the two specified bounds (inclusive). If bound2 is not specified, it defaults to 0. If neither bound is specified, a random 32-bit result is returned

`read_only_val( var_name ):` Some system variables (e.g. sim.time) are defined with "read-only" protection. As a result, they can't be directly embedded in a C expression. The read_only_val function enables read-only variables to be read by returning the variable value.

`rec_error( fmt, ... ):` Analogous to the "printf" function, but also registers a simulation error.

`seed():` Returns the current seed for random number generation.

`sprintf( format_string, ... ):` Analogous to the C "sprintf" function, except that the format_string argument expects a variable of type "string".

`srand( number ):` Sets the seed for random number generation.

`state( string ):` Returns the predefined model state value that maps to the specified text string. If no mapping is found, an error is flagged.

`state_type( var_name ):` Returns the numerically specified state type if the specified state name was previously defined. It returns 0 if the state is undefined. The argument may be either a quoted or unquoted name. The numeric values for the state types are as follow

- Signal = 1
- Flip-Flop=2
- Latch=3
- Tristate Node=4
- Array=5
- FIFO=6
- Control Store=7
- Ucode Register=8
- Delay Element=9
- System State=10
- C "int" =11
- User-Defined=12
- Vector=13
- User-Defined Function=14
- Macro=15
- Internal=24
- Statistic=16
- Watch=17
- Model Artifact=18
- Constant=19
- Imported=20
- Ubreak=21
- Statistic=22
- String=23

`system( char *shell_cmd ):` Implements the C "system" function which passes a shell command to the OS or executing shell. The return status resulting from the execution of the command is returned by this function

uaddr(control_store_state, label_name): Returns the micro-address value of the specified label name within the specified control_store_state. If the label is not found, it returns -1

uninitialized(): Returns 1 if the model is not yet initialized; otherwise it returns 0.

valid_elements(array_state_name): Returns the number of valid elements in the specified array. If an array index is included in the array_state_name, the function returns a boolean value reflecting the validity of only the specified array element.

valid_file(file_name): Indicates if the specified file_name exists. If it does, the entire path and file name is returned; otherwise an empty string is returned. The specified file name can have an absolute or relative path included, or it can assume the default path specified by the current working directory. The file specification can include a wildcard name which will return a matched file name only if the specification matches exactly one (1) file.

7.7 Error Handling

When an error is generated, it is printed to the console and to any log files that have been configured to record simulator responses. Additionally, the built-in simulation state "sim.error_count" is incremented. The built-in simulation state, "sim.error_handle_mode", can be used to alter this default error handling behavior. See Section 7.5 for the error handling values and their corresponding modes.

The default error handle mode value is 40.

The user can write his/her own error handler that allows some user control over the handling of specific error messages. If the user defines the function:

```
int on_error( string str ); // in the interpreter
```

or

```
int on_error( char *str ); // as an imported function
```

then this function will be called with the error message as its input argument. The function should return one of the predefined sim.error_handle_mode values which will be used to handle this error event. For example, on_error would return 0 to suppress the error. If the on_error function does not take 1 string argument or does not return an int or a predefined error_handle_mode value, the on_error function will be ignored for the purpose of error handling.

7.8 Printing Statistics from the Transactor

The following console functions are associated with printing Transactor performance statistics:

7.8.1 perf_stat_set( )

This function enables or disables performance statistics collection.

int perf_stat_set(char *chip_name, char *stat_name_str, int stat_type, int enable)

chip_name chip instance name

stat_name_str name string of statistic object, use NULL for the entire set. For example, if stat_byte is set to 0x100, then entire sram stastistic object will be disabled/enabled accordingly

stat_type 1: for all statistic objects

0x100: for sram statistic objects

0x200: for dram statistic objects

0x300: for shac statistic objects

0x400: for msf statistic objects

0x500: for pci statistic objects

0x600: for gasket statistic objects

0x700: for me statistic object

enable 1: for enable

0: for disable

7.8.2 perf_stat_print( )

This console function prints the performance statistics.

int perf_stat_print(char *chip_name, int stat_type)

chip_name chip instance name

stat_type 1: for all statistic objects

0x100: for sram statistic objects

0x200: for dram statistic objects

0x300: for shac statistic objects

0x400: for msf statistic objects

0x500: for pci statistic objects

0x600: for gasket statistic objects

0x700: for me statistic object

Simulator APIs 8

This section describes the information comprising the XACT API. If you plan to access this API via a foreign model dll, include the header file ***xact_vmod.h***.

8.1 Foreign Model API

The following 6 routines must be supplied in a foreign model dll. Note that they do not have to be exported because the transactor obtains their addresses by calling `GetVmodForeignModelFunctions():`

8.1.1 FOR_MOD_INITIALIZE

This routine will be called to initialize the foreign model after the transactor "`init`" command has successfully executed. It supplies a model instance number, a model name that corresponds to it, and an initialization string that will be passed to the foreign model at initialization time. The model name was previously specified by the user when the foreign model was registered with the simulator. The model instance number is a unique non-negative number that acts as a handle to this particular foreign model instance. The model instance number will be passed in to all foreign model functions below. As a result, multiple instances of a single foreign model dll can be registered with the simulator because each call to a foreign model function specifies a particular foreign model instance number. The `init_str` argument may be NULL.

Function Prototype	`int for_mod_initialize( int model_instance_num, const char *model_name, const char *init_str )`
Returns	TRUE if successful and FALSE otherwise.

8.1.2 FOR_MOD_PRE_SIM

This routine will be called prior to each transactor simulation event. It can be used to deposit state values into the transactor model prior to simulating the next event. Returning 0 results in an error

Function Prototype	`int for_mod_pre_sim( int model_instance_num )`

8.1.3 FOR_MOD_POST_SIM

This routine will be called subsequent to each transactor simulation event. It can be used to query transactor simulation state, in order to copy it into the foreign model simulator.

Function Prototype	`int for_mod_post_sim ( int model_instance_num )`

8.1.4 FOR_MOD_EXIT

This routine will be called just prior to exiting the simulator. The routine allows the foreign model to clean up, close files, etc before shutting down the program

Function Prototype `int for_mod_exit( int model_instance_num )`

8.1.5 FOR_MOD_RESET

This routine will be called just after resetting the simulator. The routine allows the foreign model to reset itself to stay in sync with the simulator

Function Prototype `int for_mod_reset( int model_instance_num )`

8.1.6 FOR_MOD_DELETE

This routine will be called just after the simulator deletes all of its model state via the "`sim_delete`" command. The routine allows the foreign model to delete its internal state to stay in sync with the simulator.

Function Prototype `int for_mod_delete ( int model_instance_num )`

8.2 Overview of XACT API Functions

The following table comprises single-threaded APIs. No interlocks have been designed in to allow proper behavior for multiple simultaneous thread execution through this interface. If multiple threads require access to this API, it is the responsibility of those threads to synchronize their execution so that only one thread at a time is executing any of these routines. Violation of this constraint may cause unpredictable and/or catastrophic behavior.

Table 8. XACT API Functions (Sheet 1 of 3)

Function Name	Function Description
XACT_find_wildcard_state_name	Returns all state names that match the wildcard name spec.
XACT_get_handle	Returns a handle to the transactor state
XACT_delete_handle	Deletes the specified handle
XACT_get_state_info	Returns information about the state referenced by the specified handle
XACT_get_state_value	Gets the value of the state corresponding to the transactor handle
XACT_get_state_field	Returns the specified bit field of the state corresponding to the transactor handle.
XACT_get_array_state_value	Returns the value of an array state corresponding to the handle of the specified array state
XACT_set_state_value	Sets the value of the state corresponding to the transactor handle.
XACT_set_state_field	Sets the specified bit range of the state corresponding to the transactor handle
XACT_set_array_state_value	Sets the value of the array state corresponding to the transactor handle
XACT_add_sim_state	Creates a 32-bit integer simulation state

Table 8. XACT API Functions (Continued) (Sheet 2 of 3)

Function Name	Function Description
XACT_alloc_user_sim_state	Creates a user-specified simulation state
XACT_start_of_cycle	Tests the current simulation time to see if it corresponds to the time when the specified clock domain starts a new cycle
XACT_full_cycle_simulated	Tests the current simulation time to see if it corresponds to the time when the specified clock has been fully simulated
XACT_clock_cycle	Returns the clock cycle number for the specified clock
XACT_clock_cycle_with_remainder	Returns the clock cycle number for the specified clock and tests for error occurrences
XACT_get_top_level_inst	Gets the cell_name/inst_name pair names of all cells instantiated at the top level by the “inst” command
XACT_Define_Callback_Create_Chip	Calls callback when a chip of the specified name has been called
XACT_Define_Callback_Init_Sim	Calls callback when the simulation has been instantiated and initialized via the “init” command
XACT_Define_Automatic_Sim_Halt	Calls callback when the simulator has prematurely halted model for the reason specified by the input argument
XACT_Define_Callback_Sim_Reset	Calls callback when the simulation has be reset via the “sim_reset” command
XACT_Define_Callback_Sim_Delete	Calls callback when the simulation has been destroyed via the “sim_delete_ command
XACT_Define_Callback_Sim_In_Progress	Defines callback that is invoked whenever the simulator starts or stops a simulating step
XACT_Define_Callback_Default_Go_Clock_domain	Defines callback to be invoked whenever default clock domain for “go” simulation changes
XACT_Define_Callback_State_Transition	Defines a callback to be invoked when a specified state makes a transition
XACT_Define_Cancel_Callback_State_Transition	Defines the callback to cancel any further state transition callbacks when the specified state changes
XACT_Cancel_State_Transition_Callback	Cancels the predefined state transition callback
XACT_Define_Handle_Invalidation_Callback	Notifies the user when a handle is about to become invalidated
XACT_output_to_console	Prints string to transactor console output
XACT_printf	printf function which outputs to transactor console
XACT_printf_error	printf function which outputs transactor errors to transactor console
XACT_register_console_function	Registers a foreign function with the transactor’s C interpreter
XACT_register_console_function_w_arrayed_args	Registers a foreign function with the transactor's C interpreter
XACT_unregister_console_function	Unregisters a routine that has been previously registered via the XACT register console function
XACT_ExecuteCommandStr	Executes the string as a console command

Table 8. XACT API Functions (Continued) (Sheet 2 of 3)

Function Name	Function Description
XACT_alloc_user_sim_state	Creates a user-specified simulation state
XACT_start_of_cycle	Tests the current simulation time to see if it corresponds to the time when the specified clock domain starts a new cycle
XACT_full_cycle_simulated	Tests the current simulation time to see if it corresponds to the time when the specified clock has been fully simulated
XACT_clock_cycle	Returns the clock cycle number for the specified clock
XACT_clock_cycle_with_remainder	Returns the clock cycle number for the specified clock and tests for error occurrences
XACT_get_top_level_inst	Gets the cell_name/inst_name pair names of all cells instantiated at the top level by the “inst” command
XACT_Define_Callback_Create_Chip	Calls callback when a chip of the specified name has been called
XACT_Define_Callback_Init_Sim	Calls callback when the simulation has been instantiated and initialized via the “init” command
XACT_Define_Automatic_Sim_Halt	Calls callback when the simulator has prematurely halted model for the reason specified by the input argument
XACT_Define_Callback_Sim_Reset	Calls callback when the simulation has be reset via the “sim_reset” command
XACT_Define_Callback_Sim_Delete	Calls callback when the simulation has been destroyed via the “sim_delete_ command
XACT_Define_Callback_Sim_In_Progress	Defines callback that is invoked whenever the simulator starts or stops a simulating step
XACT_Define_Callback_Default_Go_Clock_domain	Defines callback to be invoked whenever default clock domain for “go” simulation changes
XACT_Define_Callback_State_Transition	Defines a callback to be invoked when a specified state makes a transition
XACT_Define_Cancel_Callback_State_Transition	Defines the callback to cancel any further state transition callbacks when the specified state changes
XACT_Cancel_State_Transition_Callback	Cancels the predefined state transition callback
XACT_Define_Handle_Invalidation_Callback	Notifies the user when a handle is about to become invalidated
XACT_output_to_console	Prints string to transactor console output
XACT_printf	printf function which outputs to transactor console
XACT_printf_error	printf function which outputs transactor errors to transactor console
XACT_register_console_function	Registers a foreign function with the transactor’s C interpreter
XACT_register_console_function_w_arrayed_args	Registers a foreign function with the transactor's C interpreter
XACT_unregister_console_function	Unregisters a routine that has been previously registered via the XACT register console function
XACT_ExecuteCommandStr	Executes the string as a console command

Table 8. XACT API Functions (Continued) (Sheet 3 of 3)

Function Name	Function Description
XACT_init_gui_console	Initializes the command line parsing done by XACT_gui_execute_command
XACT__gui_execute_command	Executes a command line in a gui (e.g. Developer Workbench) environment
XACT_Define_Callback_Output_Message	Passes transactor output strings to callback function.
XACT_Define_Callback_Set_Prompt	Registers the callback to pass the transactor prompt to an external command line
XACT_Define_Callback_Get_Console_Input	Registers the callback necessary for the transactor console function to fetch console input from an external source
XACT_start_console	Starts the transactor console
XACT_initialize	Initializes the transactor for operation when the transactor function is accessed via a library
XACT_CTRL_C_SWITCH	Enables or disables the transactor CTRL-C function
XACT_stop_execution	Stops simulation at the end of the next simulation cycle
XACT_stop_execution_at_clk	Stops simulation at the end of the next simulation cycle that aligns to the specified clock cycle
XACT_exit_transactor	Forces termination of the transactor after the next input command
XACT_gui_interface	Returns TRUE if the GUI interface is connected to the transactor

8.3 State Name Reference Routines

8.3.1 XACT_find_wildcard_state_name

This function can be iteratively called to return all state names that match the wildcard name spec ("*" is the wildcard character that can match 0 or more characters). A non-NULL wildcard name spec indicates that a new wildcard name search is initiated; a NULL wildcard name spec indicates that the next name to match the previously initiated wildcard search should be returned. Each matching state name is returned in *rtn_name_buf* provided that the specified *rtn_name_buf_length* is large enough to hold the name.

Synopsis

```
XACTAPI XACT find_wildcard_state_name(char
*wildcard_name_spec, char *rtn_name_buf,
unsigned int rtn_name_buf_length)
```

Returns

0 if no match is found

1 if match is found

-1 if match is found but not returned due to size of *rtn_name_buf*

8.3.2 XACT_get_handle

This function returns a handle to a transactor state, based on a transactor state name. If a handle corresponding to the specified state exists and is currently valid, it will be returned; otherwise, a unique handle will be created and returned. Note that the specified state name is case-sensitive.

For non-array states, the value of the array_index **MUST** be -1. For arrayed states, the array index specifies the particular element in the array that will correspond to the handle. If a value of -1 is specified for an array state, a handle is returned corresponding to the array, but which is not associated to a particular array element.

Synopsis	`XACTAPI_NORET XACT_HANDLE XACT_get_handle(char *state_name, int array_index)`
Returns	INVALID XACT HANDLE if the function failed

8.3.3 XACT_delete_handle

This function deletes the specified handle. This deletion operation disassociates the handle from the previously specified state and invalidates the handle value thereby causing a subsequent reference to that handle to fail. However, once deleted, the value of that handle may be reused when a subsequent handle is generated by a call to `XACT_get_handle()`.

Synopsis	`XACTAPI XACT_delete_handle(XACT_HANDLE handle)`
Returns	TRUE if successful
	FALSE if not successful

8.3.4 XACT_get_state_info

This function returns information about the state referenced by the specified handle

Synopsis	`XACTTAPI XACT_get_state_info(XACT_HANDLE state_handle, char *state_name, int *width, int *array_length)`
Parameters	`state_name`: name of state
	`width:` width of state (**NOTE**: a width of 0 implies that the state is not directly readable/writeable by the XACT API.)
	`array_length`: indicates the length of array ranging from 0 to *`array_length - 1`*. If this `array_length` is -1, the state is a non-array state. If this `array_length` is < -1, it indicates that the state represents a FIFO object of length `-array_length` (this precludes FIFOs from being defined with length 1).

Returning each of the 3 pieces of state information is suppressed when that argument is NULL. Note that when using a handle corresponding to an array, the function returns information corresponding to the whole array even if the handle corresponds to a particular element of an array.

Returns	1 if function is successful
	0 if function is unsuccessful

8.3.5 XACT_get_state_value

This function gets the value of the state corresponding to the transactor handle. Note that the value pointer is assumed to point to an array of unsigned ints large enough to accommodate the value of the state element. Thus, the array length must be equal to `(state_width/32) + ((state_width % 32) ? 1 : 0)`

If the specified handle corresponds to an element of an array that is currently uninitialized (e.g. an uninitialized memory location), or is currently invalid (e.g. the state of a tristate node that was not driven), the return status is set to -1 and the returned value is unpredictable.

Synopsis `XACTAPI XACT_get_state_value(XACT_HANDLE state_handle, unsigned int *value)`

Returns 1 if function is successful

0 if function is unsuccessful

8.3.6 XACT_get_state_field

This function behaves analogously to `XACT_get_state_value()` except that it gets the value of the specified bit field rather than the value of the state.

Synopsis `XACTAPI XACT_get_state_field( XACT_HANDLE state_handle, unsigned int *value, int msb, int lsb )`

Returns 1 if specified bit-field is valid

0 if specified bit-field is invalid

8.3.7 XACT_get_array_state_value

This function behaves the same as `XACT_get_state_value`, except that the handle must correspond to an array state, and a valid array index must be specified. If the specified handle corresponds to an element of an array that is currently uninitialized (e.g. an uninitialized memory location), the return status is set to -1 and the returned value is unpredictable. If the specified handle was associated to a particular element of an array, its predefined array index is ignored for the purpose of this call.

Synopsis `XACTAPI XACT_get_array_state_value( XACT_HANDLE state_handle, int array_index, unsigned int *value )`

Synopsis `XACTAPI XACT_get_fifo_state_value( XACT_HANDLE fifo_handle, int fifo_index, unsigned int *value )`

Returns 1 if valid data is returned

-1 if the addressed entry contains invalid data

0 indicating an access failure.

8.3.8 XACT_set_state_value

This function sets the value of the state corresponding to the transactor handle. Note that the "value" pointer is assumed to point to an array of unsigned ints large enough to accommodate the value of the state element. Thus, the array length must be = `(state_width/32) + ((state_width%32) ? 1 : 0 )`. If a handle to an array state is specified that was not associated to a specific array element, this function will fail.

Synopsis `XACTAPI XACT_set_state_value( XACT_HANDLE state_handle, unsigned int *value )`

Returns 1 if function is successful

0 if function is unsuccessful

8.3.9 XACT_set_state_field

This function behaves analogously to `XACT_set_state_val` with the exception that the data is field inserted into the specified bit range.

Synopsis `XACTAPI XACT_set_state_field( XACT_HANDLE state_handle, unsigned int *value, int msb, int lsb )`

Returns 1 if the bit range is valid

0 if the bit range is invalid

8.3.10 XACT_set_array_state_value

This function behaves the same as `XACT_set_state_value`, except that the handle must correspond to an array state, and a valid array index must be specified. If the specified handle was associated to a particular element of an array, its predefined array index is ignored for the purpose of this call.

Synopsis `XACTAPI XACT_set_array_state_value( XACT_HANDLE state_handle, int array_index, unsigned int *value )`

Returns 1 if function is successful

0 if function is unsuccessful

Synopsis `XACTAPI XACT_set_fifo_state_value( XACT_HANDLE fifo_handle, int fifo_index, unsigned int *value )`

8.3.11 XACT_add_sim_state

This function creates a 32-bit integer simulation state. This state type is equivalent to that created by defining a C integer at the transactor command line (e.g. "int foo;"). This state is destroyed upon executing a "sim_reset" command, unless the "/preserve" qualifier is appended to "sim_reset". This */

Synopsis `XACTAPI XACT_add_sim_state( char *state_name )`

Returns — 1 if the state was created

0 if the state pre-existed as a non-integer, non-user-defined state

-1 if the state pre-existed as a previously defined user-defined integer state */

8.3.12 XACT_HANDLE XACT_alloc_user_sim_state

This function creates a user-specified simulation state.

Synopsis

```
XACTAPI_NORET XACT_HANDLE
XACT_alloc_user_sim_state( char *state_name, int
width )
```

Returns — a handle to the created state if the function is successful

INVALID_XACT_HANDLE if the function is unsuccessful

8.3.13 XACT_start_of_cycle

This function tests the current simulation time to see if it corresponds to the time when the specified clock domain starts a new cycle.

Synopsis

```
XACTAPI XACT_start_of_cycle( XACT_HANDLE
clock_handle )
```

Returns — 1 if there is a correspondence

0 if the times do not correspond

8.3.14 XACT_full_cycle_simulated

This function tests the current simulation time to see if it corresponds to the time when the specified clock has been fully simulated,

Synopsis

```
XACTAPI XACT_full_cycle_simulated( XACT_HANDLE
clock_handle )
```

Returns — 1 if there is a correspondence

0 if the times do not correspond

8.3.15 XACT_clock_cycle

This function returns the clock cycle number (starting at 0) for the specified clock It returns -1 if an error occurred during routine execution. If the simulation time does not fall on a whole multiple of the specified clock, the remainder is ignored.

Synopsis

```
XACTAPI XACT_clock_cycle( XACT_HANDLE
clock_handle )
```

8.3.16 XACT_clock_cycle_with_remainder

This function returns the clock cycle number (starting at 0) for the specified clock. It returns -1 if an error occurred during routine execution. If the simulation time does not fall on a whole multiple of the specified clock, the percentage of the time into the partial clock cycle is returned in `percent_remainder`; otherwise this argument returns 0

Synopsis
```
XACTAPI XACT_clock_cycle_with_remainder(
XACT_HANDLE clock_handle, double
*percent_remainder )
```

8.3.17 XACT_get_top_level_inst

This function can be repeatedly called to get the `cell_name/inst_name` pair names of all cells instantiated at the top level by the "inst" command. The user supplied arguments are filled in with the string names on each successive call. The first call must be made with "`**inst_name == '\0'`". Successive calls are made by supplying the strings returned from the previous call. It is assumed that the supplied string storage is large enough to accommodate the returned string names. The `cell_name` argument can be NULL in which case no cell name data is returned.

Synopsis
```
XACTAPI XACT_get_top_level_inst( char
**inst_name, char **cell_name )
```

Returns

TRUE when a `cell_name/inst_name` is returned

FALSE on an error or when no more name pairs can be returned

8.4 Callback Creation and Deletion Functions

8.4.1 XACT_Define_Callback_Create_Chip

Calls callback when a chip of the specified name has been called

Synopsis
```
XACTAPI XACT_Define_Callback_Create_Chip(
void(*fp)( char *chip_name ) )
```

8.4.2 XACT_Define_Callback_Init_Sim

Calls callback when the simulation has been instantiated and initialized via the "init" command

Synopsis
```
XACTAPI XACT_Define_Callback_Init_Sim(
void(*fp)() )
```

8.4.3 XACT_Define_Callback_Sim_Reset

Calls callback when the simulation has been reset via the "`sim_reset`" command

Synopsis
```
XACTAPI XACT_Define_Callback_Sim_Reset(
void(*fp)() )
```

8.4.4 XACT_Define_Callback_Sim_Delete

Calls callback when the simulation has been destroyed via the "sim_delete" command

Synopsis
```
XACTAPI XACT_Define_Callback_Sim_Delete(
void(*fp)() )
```

8.4.5 XACT_Define_Callback_Restore

Calls callback when the simulation state has been reloaded via the "restore" command

Synopsis
```
XACTAPI XACT_Define_Callback_Restore(
void(*fp)() )
```

8.4.6 XACT_Define_Callback_Sim_In_Progress

Define callback that is invoked whenever the simulator starts or stops a simulating step

Synopsis
```
XACTAPI XACT_Define_Callback_Sim_In_Progress(
void (*fp)( int currently_simulating ) )
```

8.4.7 XACT_Define_Callback_Default_Go_Clock_Domain

Define callback to be invoked whenever default clock domain for "go" simulation changes

Synopsis
```
XACTAPI
XACT_Define_Callback_Default_Go_Clock_Domain(
void (*fp)( char * clk_name ) )
```

8.4.8 XACT_Define_Callback_State_Transition

Define a callback to be invoked when a specified state makes a transition. The callback priority allows the user to specify the order in which all the defined callbacks are made in; the higher the callback priority, the earlier the callback is made. The `user_context` argument allows the caller of this function to pass contextual information to the callback routine if required. Note that a callback to cancel this callback must be defined prior to executing this routine in order for this callback definition to be successful (see `XACT_Define_Cancel_Callback_State_Transition()` below). If the specified handle corresponds to an array state, the handle must also specify a particular valid element of the array. Note that only 1 state transition callback may be defined per handle. If you desire 2 more more callbacks for a particular state, acquire multiple handles to the state via "`XACT_get_handle()`" and assign 1 callback to each handle. This routine returns 1 if successful, 0 otherwise.

Synopsis
```
XACTAPI XACT_Define_Callback_State_Transition(
XACT_HANDLE transitioning_state, int
callback_priority, int(*fp)( XACT_HANDLE
transitioning_state, void *user_context, int
array_index),void *user_context )
```

8.4.9 XACT_Define_Cancel_Callback_State_Transition

Define the callback to cancel any further state transition callbacks when the specified state changes. NOTE: this callback must be defined prior to calling `XACT_Define_Callback_State_Transition()` above. This callback will be called if the transactor deletes the state associated with the HANDLE. State deletion can occur if the state element was a user-defined state (e.g. C variable, function, watch, etc.). A predefined hardware state element will never be deleted, but for consistency, this cancel callback is still required to be specified prior to defining a state transition callback. This routine returns 1 if successful, 0 otherwise

Synopsis

```
XACTAPI
XACT_Define_Cancel_Callback_State_Transition(
XACT_HANDLE state,int (*fp)( XACT_HANDLE handle,
void *user_data) )
```

8.4.10 XACT_Cancel_State_Transition_Callback

Calling this routine allows the caller to explicitly cancel the predefined state transition callback. It returns 1 if successful, -1 if no callback was associated with this state or 0 otherwise

Synopsis

```
XACTAPI XACT_Cancel_State_Transition_Callback(
XACT_HANDLE state )
```

8.4.11 XACT_Define_Handle_Invalidation_Callback

This function allows the user to be notified when a handle is about to become invalidated. Handle invalidations can occur when the user has acquired a handle to a temporary state (e.g. C variable, function, watch, etc.). The handle is valid when the specified callback is called and becomes invalid as soon as execution returns from the callback. This routine returns 1 if successful; 0 otherwise.

Synopsis

```
XACTAPI
XACT_Define_Handle_Invalidation_Callback( int
(*fp)( XACT_HANDLE handle ) )
```

8.4.12 XACT_Define_Callback_Output_Message

Passes transactor output strings to callback function. Note that the transactor will still print the string to its own console output regardless of this callback.

Synopsis

```
XACTAPI XACT_Define_Callback_Output_Message(
void(*fp)(OUTPUT_MSG_SEVERITY severity, const
char *message) )
```

8.4.13 XACT_Define_Callback_Set_Prompt

This function registers the callback to pass the transactor prompt to an external command line. It must be called after the transactor has been initialized. It returns TRUE if successful, and FALSE otherwise.

Synopsis

```
XACTAPI XACT_Define_Callback_Set_Prompt(
void(*fp)( char *prompt_str ) )
```

8.4.14 XACT_Define_Callback_Get_Console_Input

This function registers the callback necessary for the transactor console function to fetch console input from an external source. This callback should be defined prior to invoking `XACT_start_console()`

Synopsis

```
XACTAPI XACT_Define_Callback_Get_Console_Input(
char *(*console_input)() )
```

8.5 Miscellaneous Functions

8.5.1 XACT_Define_Automatic_Sim_Halt

Calls callback when the simulator has prematurely halted model for the reason specified by the input argument

Synopsis

```
XACTAPI XACT_Define_Automatic_Sim_Halt(
void(*fp)( HALT_STATUS halt_status ) )
```

8.5.2 XACT_output_to_console

Prints string to transactor console output

Synopsis

```
XACTAPI XACT_output_to_console( char *output_str
)
```

8.5.3 XACT_printf

printf function outputting to transactor console

Synopsis

```
XACTAPI XACT_printf( char *fmt, ... )
```

8.5.4 XACT_printf_error

printf function outputting to transactor console as a transactor error

Synopsis

```
XACTAPI XACT_printf_error( char *fmt, ... )
```

8.5.5 XACT_register_console_function

This function registers a foreign function with the transactor's C interpreter. The argument list of the specified function is assumed to have the following characteristics:

- only (char *) and (unsigned int) args are allowed
- all (char *) args precede all (unsigned int) args.

- number of (char *) args is <= 3

number of (int) args is <= 5SynopsisXACTAPI

```
XACT_register_console_function_w_arrayed_args( char *function_name,int (*function_ptr)( char **, unsigned int * ),int num_char_ptr_args,int num_uint_args )
```

Returns 1 if function is successful, 0 otherwise.

8.5.6 XACT_register_console_function_w_arrayed_args

This function registers a foreign function with the transactor's C interpreter.The argument list of the specified function is assumed to have the following characteristics:

- only (char *) and (unsigned int) args are allowed
- all (char *) args precede all (unsigned int) args.
- The imported function is referenced in the console as:int function_name(specified number of char * args, specified number of unsigned int args)
- The imported function_ptr is actually called as: `int (*function_ptr)( char **char_array, unsigned int *int_array )` where the char arguments are placed in the `char_array` from left to right starting at index 0 and the int arguments are placed in the `int_array` from left to right starting at index 0.

Synopsis

```
XACTAPI XACT_register_console_function( char *function_name,void *function_ptr,int num_char_ptr_args,int num_uint_args )
```

Returns 1 if function is successful, 0 otherwise.

While the `function_ptr` is passed in as a void *, it will be called by the simulator as int (*function_ptr)(specified number of char * args, specified number of unsigned int args)

8.5.7 XACT_unregister_console_function

This function unregisters a routine that has been previously registered via `XACT_register_console_function()`.

Synopsis

```
XACTAPI XACT_unregister_console_function( char *function_name )
```

Returns 1 if function is successful, 0 otherwise.

8.5.8 XACT_ExecuteCommandStr

Executes the str as a console command

Synopsis

```
XACTAPI XACT_ExecuteCommandStr( char *cmd_str )
```

8.5.9 XACT_init_gui_console

Initializes the command line parsing done by `XACT_gui_execute_command()`, i.e., clears outstanding line continuation, nested curly braces, nested conditional directives, etc.

Synopsis `XACTAPI XACT_init_gui_console()`

8.5.10 XACT_gui_execute_command

Executes a command line in a gui (e.g., Developer Workbench) environment. Command line status, such as line continuation, conditional directives, etc., is maintained between calls. Use `XACT_init_gui_console()` to initialize status.

Synopsis `XACTAPI XACT_gui_execute_command(char *command_line)`

8.5.11 XACT_start_console()

This function starts the transactor console. This function returns only when an "`exit`" command is processed by the console. When an external program is to supply the actual console I/O window, the functions: `XACT_Define_Callback_Output_Message()`, `XACT_Define_Callback_Set_Prompt()`, and `XACT_Define_Callback_Get_Console_Input()` must be called prior to invoking `XACT_start_console()`.

Synopsis `XACTAPI XACT_start_console()`

Returns TRUE if the console was successfully invoked or FALSE otherwise

8.5.12 XACT_initialize()

This function initializes the transactor for operation when the transactor function is accessed via a library. This function must be the first transactor function called. Subsequent transactor functions should only be called when this function returns with success status.

Synopsis `XACTAPI XACT_initialize()`

Returns TRUE if the initialization was successful or FALSE otherwise

8.5.13 XACT_stop_execution_at_clk

This function stops simulation at the end of the next simulation cycle that aligns to the specified clock cycle. It also stops script file execution as soon as possible by letting the current command complete, and then unwinding the command stack.

Synopsis `XACTAPI XACT_stop_execution_at_clk( XACT_HANDLE clk_handle )`

Returns TRUE if successful; FALSE otherwise

8.5.14 XACT_exit_transactor

This function forces termination of the transactor after the next input command has been received

Synopsis	`XACTAPI XACT_exit_transactor()`

8.5.15 XACT_CTRL_C_SWITCH

This function enables or disables the transactor CTRL-C function. By default, this function is enabled. Note that this is a non-blocking call (i.e. the simulation is not guaranteed to be stopped when this function returns).

Synopsis	`XACTAPI XACT_CTRL_C_SWITCH( int enable )`
Returns	TRUE if successful; FALSE otherwise

8.5.16 XACT_stop_execution

This function stops simulation at the end of the next simulation cycle It also stops script file execution as soon as possible by letting the current command complete, and then unwinding the command stack.

Synopsis	`XACTAPI XACT_stop_execution()`
Returns	TRUE if successful; FALSE otherwise

8.5.17 XACT_gui_interface

This function returns TRUE if the Workbench is connected to the transactor; otherwise, it returns FALSE

Synopsis	`XACTAPI XACT_gui_interface()`

Transactor States A

A.1 About States

The Transactor contains internal states that define the overall state of the model. The states documented in this chapter can be accessed through the Workbench command line interface using either built-in C functions or transactor access functions.

Hardware states, certain CSRs, and transactor states for QDR and MSF pins are available in this appendix.

A.1.1 State Definition Format

This section lists the Hardware Transactor states and describes each one. The state definition contains:

- The state name
- A description of the state.
- Function primitives for functions used with the state.

A.2 Hardware States

A.2.1 SRAM

SRAM memory. For all of the following functions, you must use address values that are 32-bit aligned.

Functions:

set_sram(addr, data) Use to write to SRAM memory.

get_sram(addr) Use to read SRAM memory.

watch_sram(addr) Use to watch for SRAM content changes.

watch_sram_function({code} or function ( ),addr)
Use to watch for SRAM content changes, then execute code or call function.

check_sram(addr, expect) Use to compare SRAM content with an expected value.

dump_sram(addr_lo, addr_hi) Use to display SRAM content.

init_sram(data, addr_lo, addr_hi) Use to initialize SRAM content.

init_sram_from_file(filename) Use to initialize SRAM content from file. Format is *hex_addr value* pair on each line.

A.2.2 Scratchpad

On-chip Scratchpad memory. For all of the following functions, you must use address values that are 32-bit aligned.

Functions:

set_scratch(addr, data)	Use to write to Scratchpad memory.
get_scratch(addr)	Use to read Scratchpad memory.
watch_scratch(addr)	Use to watch for Scratchpad content changes.
watch_scratch_function({code} or function( ), addr)	
	Use to watch for Scratchpad content changes and then execute code or call function.
check_scratch(addr, expect)	Use to compare Scratchpad content with expected value.
dump_scratch(addr_lo, addr_hi)	Use to display Scratchpad content.
init_scratch(data, addr_lo, addr_hi)	Use to initialize Scratchpad content.
init_scratch_from_file(filename)	Use to initialize Scratchpad content from file. Format is *hex_addr value* pair on each line.

A.2.3 DRAM

DRAM memory. For all of the following functions, you must use address values that are 32-bit aligned.

Functions:

set_dram(addr, data)	Use to write to DRAM memory.
get_dram(addr)	Use to read DRAM memory.
watch_dram(addr)	Use to watch for DRAM content changes.
watch_dram_function({code} or function( ),addr)	
	Use to watch for DRAM content changes and then execute code or call function.
check_dram(addr, expect)	Use to compare DRAM content with an expected value.
dump_dram(addr_lo, addr_hi)	Use to display DRAM content.
init_dram(data, addr_lo, addr_hi)	Use to initialize DRAM content.
init_dram_from_file(filename)	Use to initialize DRAM content from file. Format is *hex_addr value* pair on each line.

A.2.4 RBUF

Media Switch Fabric interface Receive buffer.

Functions:

set_rbuf(addr, data)	Use to write to RBUF memory.
get_rbuf(addr)	Use to read RBUF memory.
watch_rbuf(addr)	Use to watch for RBUF content changes.
watch_rbuf_function({code} or function( ), addr)	Use to watch for RBUF content changes and then execute code or call function.
check_rbuf(addr, expect)	Use to compare RBUF content with an expected value.
dump_rbuf(addr_lo, addr_hi)	Use to display RBUF content.
init_rbuf(data, addr_lo, addr_hi)	Use to initialize RBUF content.
init_rbuf_from_file(filename)	Use to initialize RBUF content from file. Format is *hex_addr value* pair on each line.

A.2.5 TBUF

Media Switch Fabric interface Transmit buffer.

Functions:

set_tbuf(addr, data)	Use to write to TBUF memory.
get_tbuf(addr)	Use to read TBUF content changes.
watch_tbuf(addr)	Use to watch for TBUF content changes.
watch_tbuf_function({code} or function( ), addr)	Use to watch for TBUF content changes and then execute code or call function.
check_tbuf(addr, expect)	Use to compare TBUF content with an expected value.
dump_tbuf(addr_lo, addr_hi)	Use to display TBUF content.
init_tbuf(data, addr_lo, addr_hi)	Use to initialize TBUF content
init_tbuf_from_file(filename)	Use to initialize TBUF content from file. Format is *hex_addr value* pair on each line.

A.3 Microengine Registers

For the following sections, Section A.3.1 through Section A.3.8, the paramenter **me** is the Microengine number. Valid **me** numbers for the IXP2400 are: 0x00 - 0x03 and 0x10 - 0x13. Valid **me** numbers for the IXP2800 are: 0x00 - 0x07 and 0x10 - 0x17. For the range of valid local memory locations, refer to either the *IXP2400/IXP2800 Programmer's Reference Manual*, as appropriate.

A.3.1 Local Memory

Local memory within a Microengine.

Functions:

set_lmem(me, addr, data) Use to write to local memory in the Microengine.

get_lmem(me, addr) Use to read local memory in the Microengine.

watch_lmem(me, addr) Use to watch for local memory content changes.

watch_lmem_function({code} or function( ), me, addr)

Use to watch for local memory content changes and then execute code or call function.

check_lmem(me, addr, expect) Use to compare local memory with an expected value.

dump_lmem(me) Use to display the Microengine's local memory content.

init_lmem(me, data) Use to initialize local memory content in the Microengine.

A.3.2 GPR A bank

A Bank General Purpose register within a Microengine.

Functions:

set_gpa(me, addr, data) Use to write to GPR A in the Microengine.

get_gpa(me, addr) Use to read GPR A in the Microengine.

watch_gpa(me, addr) Use to watch for GPR A content changes.

watch_gpra_function({code} or function( ), me, addr)

Use to watch for GPR A content changes and then execute code or call function.

check_gpa(me, addr, expect) Use to compare GPR A memory with an expected value.

dump_gpa(me) Use to display the Microengine's GPR A content.

init_gpa(me, data) Use to initialize GPR A content in the Microengine.

A.3.3 GPR B bank

B Bank General Purpose register within a Microengine.

Functions:

set_gpb(me, addr, data)	Use to write to GPR Bin the Microengine.
get_gpb(me, addr)	Use to read GPR Bin the Microengine.
watch_gpb(me, addr)	Use to watch for GPR B content changes.
watch_gprb_function({code} or function, me, addr)	Use to watch for GPR B content changes and then execute code or call function..
check_gpb(me, addr, expect)	Use to compare GPR B memory with an expected value.
dump_gpb(me)	Use to display the Microengine's GPR B content.
init_gpb(me, data)	Use to initialize GPR B content in the Microengine.

A.3.4 Transfer Register S In

S Transfer register In within a Microengine.

Functions:

set_srd(me, addr, data)	Use to write to S Transfer In register in the Microengine.
get_srd(me, addr)	Use to read S Transfer In register in the Microengine.
watch_srd(me, addr)	Use to watch S Transfer In register content changes.
watch_srd_function({code} or function, me, addr)	Use to watch for S Transfer In register content changes and then execute code or call function.
check_srd(me, addr, expect)	Use to compare S Transfer In register with an expected value.
dump_srd(me)	Use to display the S Transfer In register's content.
init_srd(me, data)	Use to initialize S Transfer In register content.

A.3.5 Transfer Register S Out

S Transfer register Out within a Microengine.

Functions:

set_swr(me, addr, data)	Use to write to S Transfer Out register in the Microengine.
get_swr(me, addr)	Use to read S Transfer Out register in the Microengine.
watch_swr(me, addr)	Use to watch S Transfer Out register content changes.
watch_swr_function({code} or function, me, addr)	Use to watch for S Transfer Out register

	content changes and then execute code or call function.
check_swr(me, addr, expect)	Use to compare S Transfer register Out with an expected value.
dump_swr(me)	Use to display the S Transfer Out register's content.
init_swr(me, data)	Use to initialize S Transfer Out register content.

Note: To display all S Transfer registers of a particular Microengine, use **dump_gprs(me)**

A.3.6 Transfer Register D In

D Transfer register In within a Microengine.

Functions:

set_drd(me, addr, data)	Use to write to D Transfer In register in the Microengine.
get_drd(me, addr)	Use to read D Transfer In register in the Microengine.
watch_drd(me, addr)	Use to watch D Transfer In register content changes.
watch_drd_function({code} or function, me, addr)	Use to watch for D Transfer In register content changes and then execute code or call function.
check_drd(me, addr, expect)	Use to compare D Transfer In register with an expected value.
dump_drd(me)	Use to display the D Transfer In register's content.
init_drd(me, data)	Use to initialize D Transfer In register content.

A.3.7 Transfer Register D Out

D Transfer register Out within a Microengine.

Functions:

set_dwr(me, addr, data)	Use to write to D Transfer Out register in the Microengine.
get_dwr(me, addr)	Use to read D Transfer Out register in the Microengine.
watch_dwr(me, addr)	Use to watch D Transfer Out register content changes.
watch_dwr_function({code} or function, me, addr)	Use to watch for D Transfer Out register content changes and then execute code or call function.
check_dwr(me, addr, expect)	Use to compare D Transfer register Out with an expected value.
dump_dwr(me)	Use to display the D Transfer Out register's content.
init_dwr(me, data)	Use to initialize D Transfer Out register content.

Note: To display all D Transfer registers of a particular Microengine, use **dump_gprd(me)**

A.3.8 Next Neighbor Registers

Next Neighbor register within a Microengine.

Functions:

set_nei(me, addr, data)	Use to write to Next Neighbor register in the Microengine.
get_nei(me, addr)	Use to read Next Neighbor register in the Microengine.
watch_nei(me, addr)	Use to watch Next Neighbor register content changes.
watch_nei_function({code} or function, me, addr)	Use to watch for Next Neighbor register content changes and then execute code or call function.
check_nei(me, addr, expect)	Use to compare Next Neighbor register content with an expected value.
dump_nei(me)	Use to display the Next Neighbor register's content.
init_nei(me, data)	Use to initialize Next Neighbor register content.

A.4 CSRs

For the following registers, use the Developer's Workbench user interface to access:

ME CSRs	Microengine Control Status registers. SRAM CSRs
SRAM CSRs	SRAM controller Control Status registers. DRAM CSRs
DRAM CSRs	DRAM controller Control Status registers. CAP CSRs
CAP CSRs	CAP unit Control Status registers. MSF CSRs
MSF CSRs	Media Switch Fabric registers. Intel® XScale™ CSRs
Intel® XScale™ CSRs	Intel® XScale™ registers.

A.5 Intel® XScale™ Memory Map Access

The following functions are used to access locations within the Intel® XScale™ memory mapping of the network processors. See the *IXP2400/IXP2800 Programmer's Reference Manual*.

For the IXP2400, IXP2800, and IXP2850:

simRead(chip_name, addr)	Returns value at the XScale *addr* in the *chip_name* chip
simWrite(chip_name, addr, data);	Writes data to the Xscale *addr* in the *chip_name* chip.

A.6 IXP2400 and IXP2800 Transactor States

In the following command and tables, *chip_name* is the name the user applies to the chip instance. In other words, the user replaces *chip_name* with their own chip_name (if there is one). If the instance is unnamed, then the *chip_name* variable is omitted.

For the QDR interface, *n* is the SRAM channel number, which can be either 0, 1, 2 or 3 (for the IXP2800). For the IXP2400, *n* may be 0 or 1.

To connect a foreign model to the QDR interface, you must add the `setup_sram_external_pin_usage` command to the setup script (this must be done for each channel). The command is not necessary if you want to connect only to the MSF interface.

```
setup_sram_external_pin_usage(chip_name, lower_address, upper_address);
```

The *lower_address* and *upper_address* denote the range in the sram memory map to use for an external model.

Table 1. IXP2400 Transactor States for QDR and MSF Pins (Sheet 1 of 3)

Transactor State Names	Datasheet Signal Name	I/O	Description
QDR Interface			
chip_name.QDR*n*_K_H[1:0] *chip_name*.QDR*n*_K_L[1:0]	S*n*_K[1:0] S*n*_K_L[1:0]	Output Output	Positive and negative clock outputs. These differential clocks are used as a reference for Address, Data Out, and Port Enable.
chip_name.QDR*n*_C_H[1:0] *chip_name*.QDR*n*_C_L[1:0]	S*n*_C[1:0] S*n*_C_L[1:0]	Output Output	Positive and negative clock outputs.
chip_name.QDR*n*_CIN_H[1:0] *chip_name*.QDR*n*_CIN_L[1:0]	S*n*_CIN[1:0] S*n*_CIN_L[1:0]	Input Input	Positive and negative clock inputs. These differential clocks are used as a reference for Data In. They are the feedback of S*n*_C & S*n*_C_L.
chip_name.QDR*n*_D_H[7:0]	S*n*_D[7:0]	Output	Data output bus
chip_name.QDR*n*_D_H[16:9]	S*n*_D[15:8]	Output	Data output bus
chip_name.QDR*n*_D_H[8]	S*n*_D[0]	Output	Byte parity for data out. D[1], D[0] corresponds to D[15:8], D[7:0] respectively
chip_name.QDR*n*_D_H[17]	S*n*_D[1]	Output	Byte parity for data out. D[1], D[0] corresponds to D[15:8], D[7:0] respectively
chip_name.QDR*n*_Q_H[7:0]	S*n*_Q[7:0]	Input	Data output bus
chip_name.QDR*n*_Q_H[16:9]	S*n*_Q[15:8]	Input	Data output bus
chip_name.QDR*n*_Q_H[8]	S*n*_Q[0]	Input	Byte parity for data in. Q[1], Q[0] corresponds to Q[15:8],Q[7:0] respectively
chip_name.QDR*n*_Q_H[17]	S*n*_Q[1]	Input	Byte parity for data in. Q[1], Q[0] corresponds to Q[15:8],Q[7:0] respectively

Table 1. IXP2400 Transactor States for QDR and MSF Pins (Continued) (Sheet 2 of 3)

Transactor State Names	Datasheet Signal Name	I/O	Description
chip_name.QDR*n*_BWS_L[1:0]	S*n*_BWE_L[1:0]	Output	Byte write enables. BW_L[1], BW_L[0] corresponds to DO[15:8], DO[7:0] respectively
chip_name.QDR*n*_RPS_L[1:0]	S*n*_RPE_L[1:0]	Output	Read Port Enable
chip_name.QDR*n*_WPS_L[1:0]	S*n*_WPE_L[1:0]	Output	Write Port Enable
chip_name.QDR*n*_A_H[23:0]	S*n*_A[23:0]	Output	Address. Depending on the operating mode, some of the address pins may also be used for RPE_L/WPE_L.
chip_name.QDR*n*_ZQ[1:0]	S*n*_ZQ[1:0]	Input	Drive Strength/Compensation
MSF Interface			
chip_name.PLMS_MR23_CLK	RXCLK23	Input	MSF Receive Clock for channel 2 and 3.
chip_name.PLMS_MR01_CLK	RXCLK01	Input	MSF Receive Clock for channel 0 and 1.
chip_name.MSPA_RXENB_WMR23H	RXENB[3:2]	Output	MSF Receive Control Pins for up to 4 Channels.
chip_name.MSPA_RXENB_WMR01H	RXENB[1:0]	Output	MSF Receive Control Pins for up to 4 Channels.
chip_name.PAMS_RXSOF_RMR23H	RXSOF[3:2]	Input	
chip_name.PAMS_RXSOF_RMR01H	RXSOF[1:0]	Input	
chip_name.PAMS_RXEOF_RMR23H	RXEOF[3:2]	Input	
chip_name.PAMS_RXEOF_RMR01H	RXEOF[1:0]	Input	
chip_name.PAMS_RXVAL_RMR23H	RXVAL[3:2]	Input	
chip_name.PAMS_RXVAL_RMR01H	RXVAL[1:0]	Input	
chip_name.PAMS_RXERR_RMR23H	RXERR[3:2]	Input	
chip_name.PAMS_RXERR_RMR01H	RXERR[1:0]	Input	
chip_name.PAMS_RXPRTY_RMR23H	RXPRTY[3:2]	Input	
chip_name.PAMS_RXPRTY_RMR01H	RXPRTY[1:0]	Input	
chip_name.PAMS_RXFA_RMR23H	RXFA[3:2]	Input	
chip_name.PAMS_RXFA_RMR01H	RXFA[1:0]	Input	
chip_name.MSPA_RXADDR_WMR01H	RXADDR[3:0]	Output	
chip_name.PAMS_RXPFA_RMR01H	RXPFA	Input	
chip_name.PAMS_RXPADL1_RMR23H	RXPADL[1]	Input	
chip_name.PAMS_RXPADL0_RMR01H	RXPADL[0]	Input	
chip_name.PAMS_RXDATA_RMR01H	RXDATA[15:0]	Input	MSF Receive Data Bus
chip_name.PAMS_RXDATA_RMR23H	RXDATA[31:16]	Input	MSF Receive Data Bus
chip_name.PLMS_MT23_CLK	TXCLK23	Input	MSF Transmit Clock for channel 2 and 3.
chip_name.PLMS_MT01_CLK	TXCLK01	Input	MSF Transmit Clock for channel 0 and 1.

Table 1. IXP2400 Transactor States for QDR and MSF Pins (Continued) (Sheet 3 of 3)

Transactor State Names	Datasheet Signal Name	I/O	Description
chip_name.MSPA_TXENB_WMT23H	TXENB[3:2]	Output	MSF Transmit Control Pins for up to 4 Channels.
chip_name.MSPA_TXENB_WMT01H	TXENB[1:0]	Output	MSF Transmit Control Pins for up to 4 Channels.
chip_name.MSPA_TXSOF_WMT23H	TXSOF[3:2]	Output	
chip_name.MSPA_TXSOF_WMT01H	TXSOF[1:0]	Output	
chip_name.MSPA_TXEOF_WMT23H	TXEOF[3:2]	Output	
chip_name.MSPA_TXEOF_WMT01H	TXEOF[1:0]	Output	
chip_name.MSPA_TXERR_WMT23H	TXERR[3:2]	Output	
chip_name.MSPA_TXERR_WMT01H	TXERR[1:0]	Output	
chip_name.MSPA_TXPRTY_WMT23H	TXPRTY[3:2]	Output	
chip_name.MSPA_TXPRTY_WMT01H	TXPRTY[1:0]	Output	
chip_name.PAMS_TXFA_RMT23H	TXFA[3:2]	Input	
chip_name.PAMS_TXFA_RMT01H	TXFA[1:0]	Input	
chip_name.PAMS_TXPFA_RMT01H	TXPFA	Input	
chip_name.PAMS_TXSFA_RMT01H	TXSFA	Input	
chip_name.MSPA_TXADDR_WMT01H	TXADDR[3:0]	Output	
chip_name.MSPA_TXPADL1_WMT23H	TXPADL[1]	Output	
chip_name.MSPA_TXPADL0_WMT01H	TXPADL[0]	Output	
chip_name.MSPA_TXDATA_WMT23H	TXDATA[31:16]	Output	MSF Transmit Data Bus
chip_name.MSPA_TXDATA_WMT01H	TXDATA[15:0]	Output	MSF Transmit Data Bus
chip_name.MSPA_TXCDAT_WMR01H	TXCDATA[3:0]	Output	CBUS Transmit Data
chip_name.MSPA_TXCSOF_WMR01H	TXCSOF	Output	CBUS Transmit Start of Frame
chip_name.MSPA_TXCSRB_WMR01H	TXCSRB	Output	CBUS Transmit Serialized Ready Bus
chip_name.PAMS_TXCFC_RMR01H	TXCFC	Input	CBUS Transmit Flow Control - FIFO Full
chip_name.MSPA_TXCPAR_WMR01H	TXCPAR	Output	CBUS Transmit Parity
chip_name.PAMS_RXCDAT_RMT01H	RXCDATA[3:0]	Input	CBUS Receive Data
chip_name.PAMS_RXCSOF_RMT01H	RXCSOF	Input	CBUS Receive Start of Frame
chip_name.PAMS_RXCSRB_RMT01H	RXCSRB	Input	CBUS Receive Serialized Ready Bus
chip_name.MSPA_RXCFC_WMT01H	RXCFC	Output	CBUS Receive Flow Control - FIFO Full
chip_name.PAMS_RXCPAR_RMT01H	RXCPAR	Input	CBUS Receive Parity

Table 2. IXP2800 Transactor States for QDR and MSF Pins (Sheet 1 of 2)

Transactor State Names	Datasheet Signal Name	I/O	Description
QDR Interface			
chip_name.QDR*n*_K_H[1:0] *chip_name*.QDR*n*_K_L[1:0]	S*n*_K[1:0] S*n*_K_L[1:0]	Output	Positive and negative clock outputs. These differential clocks are used as a reference for Address, Data Out, and Port Enable.
chip_name.QDR*n*_C_H[1:0] *chip_name*.QDR*n*_C_L[1:0]	S*n*_C[1:0] S*n*_C_L[1:0]	Output	Positive and negative output clocks to SRAM.
chip_name.QDR*n*_CIN_H[1:0] *chip_name*.QDR*n*_CIN_L[1:0]	S*n*_CIN[1:0] S*n*_CIN_L[1:0]	Input	Echo clocks. Positive and Negative input clocks. Data In is referenced to these clocks.
chip_name.QDR*n*_Q_H[17:0]	S*n*_Q[17:0]	Input	Data In. Read data and parity from SRAMs to chip.
chip_name.QDR*n*_D_H[17:0]	S*n*_D[17:0]	Output	Data Out. Write data and parity from chip to SRAMs.
chip_name.QDR*n*_BWS_L[1:0]	S*n*_BWE_L[1:0]	Output	Byte Write Enables. Asserted to enable writing each byte during writes.
chip_name.QDR*n*_RPS_L[1:0]	S*n*_RPE_L[1:0]	Output	Read Port Enable. Asserted to start a read.
chip_name.QDR*n*_WPS_L[1:0]	S*n*_WPE_L[1:0]	Output	Write Port Enable. Asserted to start a write.
chip_name.QDR*n*_A_H[23:0]	S*n*_A[23:0]	Output	Address to SRAMs.
MSF Interface			
chip_name.SPI4_TCLK		Output	SPI4 Tx Clock
chip_name.SPI4_RCLK		Input	SPI4 Rx Clock
chip_name.SPI4_TCLK_REF *chip_name*.SPI4_TCLK_REF_L		Input	SPI4 Tx reference clock
chip_name.SPI4_RCLK_REF *chip_name*.SPI4_RCLK_REF_L		Output	SPI4 Rx reference clock (buffer version of SPI4_RCLK)
chip_name.SPI4_TDAT *chip_name*.SPI4_TDAT_L		Output	SPI4 Tx data
chip_name.SPI4_RDAT *chip_name*.SPI4_RDAT_L		Input	SPI4 Rx data
chip_name.SPI4_TSTAT		Input	SPI4 Tx stat (flow control)
chip_name.SPI4_RSTAT		Output	SPI4 Rx stat (flow control)
chip_name.SPI4_TCTL *chip_name*.SPI4_TCTL_L		Output	SPI4 Tx control
chip_name.SPI4_RCTL *chip_name*.SPI4_RCTL_L		Input	SPI4 Rx control
chip_name.SPI4_TPROT *chip_name*.SPI4_TPROT_L		Output	Tx Protocol type (SPI4 or CSIX)

Table 2. IXP2800 Transactor States for QDR and MSF Pins (Continued) (Sheet 2 of 2)

Transactor State Names	Datasheet Signal Name	I/O	Description
chip_name.SPI4_RPROT *chip_name*.SPI4_RPROT_L		Input	Rx Protocol type (SPI4 or CSIX)
chip_name.SPI4_TPAR *chip_name*.SPI4_TPAR_L		Output	Tx Parity
chip_name.SPI4_RPAR *chip_name*.SPI4_RPAR_L		Input	Rx Parity
chip_name.FC_TXCDAT *chip_name*.FC_TXCDAT_L		Output	Flow Control Engress data
chip_name.FC_RXCDAT *chip_name*.FC_RXCDAT_L		Input	Flow Control Ingress data
chip_name.FC_TXCSOF *chip_name*.FC_TXCSOF_L		Output	Flow Control Engress SOF (Start Of Frame)
chip_name.FC_RXCSOF *chip_name*.FC_RXCSOF_L		Input	Flow Control Ingress SOF
chip_name.FC_TXCSRB *chip_name*.FC_TXCSRB_L		Output	Flow Control Engress SRB (Serialized Ready Bits)
chip_name.FC_RXCSRB *chip_name*.FC_RXCSRB_L		Input	Flow Control Ingress SRB
chip_name.FC_TXCPAR *chip_name*.FC_TXCPAR_L		Output	Flow Control Engress Parity
chip_name.FC_RXCPAR *chip_name*.FC_RXCPAR_L		Input	Flow Control Ingress Parity
chip_name.FC_TXCFC *chip_name*.FC_TXCFC_L		Input	Flow Control Engress FIFO full
chip_name.FC_RXCFC *chip_name*.FC_RXCFC_L		Output	Flow Control Ingress FIFO full

Developer Workbench Shortcuts B

B.1 Introduction

In the Developer Workbench there are at least three ways to initiate an action:

- A menu command
- A keyboard shortcut
- A toolbar button

The following tables summarize most these tools.

Table 3. Developer Workbench Shortcuts—Files

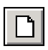
Button	Keyboard	Menu	Action	Reference
	CTRL+N	File, New	Create new file.	Section 2.5.1.
	CTRL+O	File, Open	Open a file.	Section 2.5.2.
	CTRL+S	File, Save	Save a file.	Section 2.5.4.
	ALT+F+A	File, Save As	Save copy of file.	Section 2.5.5.
	ALT+F+L	File, Save All	Save all open files.	Section 2.5.6.
	ALT+F+U	File, Print Setup	Set up the printer properties.	Section 2.5.8.1.
	CTRL+P	File, Print	Print file in active window.	Section 2.5.8.2.
	ALT+F+F	File, Recent Files	Select from the four most recently opened files.	Section 2.5.2.
	ALT+F+C	File, Close, or WIndow, Close	Close the active window.	Section 2.5.3.
	ALT+SHIFT+B		Moves back to previous window.	
	ALT+SHIFT+F		Moves forward one window.	

Table 4. Developer Workbench Shortcuts—Projects

Button	Keyboard	Menu	Action	Reference
	ALT+F+W	File, New Project	Create a new project.	Section 2.3.1.
	ALT+F+R	File, Open Project	Open a project.	Section 2.3.2.
	ALT+F+V	File, Save Project	Save project.	Section 2.3.3
	ALT+F+E	File, Close Project	Close a project.	Section 2.3.4.
	ALT+P+I	Project, Insert Assembler Source Files	Insert Assembler source files into a project.	Section 2.5.9.1.
		Project, Insert Compiler Source Files	Insert Compiler source files into a project.	Section 2.5.9.1.
	ALT+P+S	Project, Insert Script Files	Insert script files into a project.	Section 2.5.9.1.
	ALT+P+U	Project, Update Dependencies	Update project dependencies.	Section 2.6.1.
	ALT+P+C	Project, System Configuration	Specify system configuration.	Section 2.9.

Table 5. Developer Workbench Shortcuts—Edit (Sheet 1 of 2)

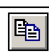
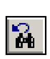
Button	Keyboard	Menu	Action	Reference
	CTRL+Z	Edit, Undo	Undo.	Section 2.5.10.
	CTRL+Y	Edit, Redo	Redo.	Section 2.5.10.
	CTRL+X	Edit, Cut	Cut.	Section 2.5.10.
	CTRL+C	Edit, Copy	Copy selected text.	Section 2.5.10.
	CTRL+V	Edit, Paste	Paste.	Section 2.5.10.
	DELETE	DELETE key	Delete.	Section 2.5.10.
	CTRL+A	Edit, Select All	Select all text in the file.	Section 2.5.10.
	CTRL+F	Edit, Find	Find text in a text file.	Section 2.5.10.
	CTRL+SHIFT+F		Find next.	Section 2.5.10.
	CTRL+SHIFT+F	Find Previous	Same as Find Next only you must click **Up** in the **Direction** area first.	

Table 5. Developer Workbench Shortcuts—Edit (Sheet 2 of 2)

Button	Keyboard	Menu	Action	Reference
	ALT+E+I	Edit, Find in Files	Find in text files.	Section 2.5.12.
	CTRL+H	Edit, Replace	In the Replace dialog box, replace the items currently selected in the file with the text in the Replace with box.	Section 2.5.10.
search text here		Search	To search for text in the active file, type the text in the **Search** box and press ENTER. The Workbench highlights the next occurrence of the text in the file. Press ENTER again to go to the next occurrence of the same text. This feature searches only the active file. You can also select previously searched text to search from the list by pressing the button on the right.	

Table 6. Developer Workbench Shortcuts—Bookmarks

Button	Keyboard	Menu	Action	Reference
	CTRL+F2	Edit, Bookmark, Insert/Remove	Insert/Remove bookmark.	Section 2.5.11.
	F2	Edit, Bookmark, Go To Next	Go to the next bookmark.	Section 2.5.11.
	SHIFT+F2	Edit, Bookmark, Go To Previous	Go to previous bookmark.	Section 2.5.11.
	CTRL+SHIFT+F2	Edit, Bookmark, Clear All	Clear all bookmarks.	Section 2.5.11.

Table 7. Developer Workbench Shortcuts—Breakpoints

Button	Keyboard	Menu	Action	Reference
	F9	Debug, Breakpoint, Insert/Remove	Insert/Remove breakpoint.	Section 2.12.10.3, .
	CTRL+F9	Debug, Breakpoint, Enable/Disable	Toggle enable/disable.	Section 2.12.10.4.
	ALT+D+B+D	Debug, Breakpoint, Disable All	Disable all breakpoints.	Section 2.12.10.4.
	ALT+D+B+A	Debug, Breakpoint, Enable All	Enable all breakpoints.	Section 2.12.10.4.
	ALT+D+B+R	Debug, Breakpoint, Remove All	Remove all breakpoints.	Section 2.12.10.3.

Table 8. Developer Workbench Shortcuts—Builds

Button	Keyboard	Menu	Action	Reference
	CTRL+F7	Build, Assemble	Assemble.	Section 2.6.3.
	CTRL+SHIFT+F7	Build, Compile	Compile.	Section 2.7.3.
	F7	Build, Build	Link.	Section 2.9.
	ALT+F7	Build, Rebuild	Rebuild.	Section 2.9.
	F4		Go to next error/tag.	Section 2.5.11, Section 2.5.12, Section 2.6.4, Section 2.7.4.
	SHIFT+F4		Go to previous error/tag.	Section 2.5.11, Section 2.6.4, Section 2.7.4.

Table 9. Developer Workbench Shortcuts—Debug

Button	Keyboard	Menu	Action	Reference
	F12	Debug, Start Debugging	Start debugging.	Section 2.12.2.
	CTRL+F12	Debug, Stop Debugging	Stop debugging.	Section 2.12.2.

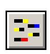
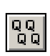
Table 10. Developer Workbench Shortcuts—Run Control (Sheet 1 of 2)

Button	Keyboard	Menu	Action	Reference
	F5	Debug, Run Control, Go	Start simulation.	Section 2.12.9.10, .
	SHIFT+F5	Debug, Run Control, Stop	Stop simulation.	Section 2.12.9.10, .
	F10	Debug, Run Control, Step Over	Step over.	Section 2.12.9.3.
	F11	Debug, Run Control, Step Into	Step into (Compiler thread only).	Section 2.12.9.4.
	SHIFT+F11	Debug, Run Control, Step Out	Step out (Compiler thread only).	Section 2.12.9.5.

Table 10. Developer Workbench Shortcuts—Run Control (Sheet 2 of 2)

Button	Keyboard	Menu	Action	Reference
	CTRL+F10	Debug, Run Control, Run To Cursor	Run to cursor.	Section 2.12.8.4.
	SHIFT+F10	Debug, Run Control, Step Microengines	Step Microengines.	Section 2.12.9.2.
	CTRL+SHIFT+F12	Debug, Run Control, Reset	Reset.	Section 2.12.9.11, .

Table 11. Developer Workbench Shortcuts—View

Button	Keyboard	Menu	Action	Reference
	CTRL+F6	Window, <filename>	Make next window active.	Section 2.2.2.
	ALT+V+O	View, Output Window	Toggle visibility of Output window.	Section 2.2.2.
	ALT+V+D+C	View, Debug Window, Command Line	Toggle visibility of Command Line window.	Section 2.12.6.
	ALT+V+D+D	View, Debug Window, Data Watch	Toggle visibility of Data Watch window.	Section 2.12.12.
	ALT+V+D+M	View, Debug Window, Memory Watch	Toggle visibility of Memory Watch window.	Section 2.12.13
	ALT+V+D+H	View, Debug Window, History	Toggle visibility of History window.	Section 2.12.16.
	ALT+V+D+T	View, Debug Window, Thread Status	Toggle visibility of Thread Status window.	Section 2.12.18.
	ALT+V+D+Q	View, Debug Window, Queue Status	Toggle visibility of Queue Status window.	Section 2.12.17.
	ALT+V+D+R	View, Debug Window, Run Control	Toggle visibility of Run Control window.	Section 2.12.9.

XScale Core Memory Bus Functional Model C

This document describes two Application Program Interfaces (API): one is embedded in the IXP2800/IXP2400 transactor, and the other provided by a Bus Function Model (BFM) which simulates the XScale Core Memory Bus (CMB) through the transactor Foreign Model Interface (FMI).

The XScale/CMB BFM is invoked by the IXP2800/IXP2400 transactor through transactor command, and synchronized with the same event control logic over the transactor. This feature enables the C-model simulation environment to run XScale native code through C/C++ API's provided by the IXP2800/IXP2400 transactor and XScale/CMB BFM.

The CMB BFM provides calls for reading and writing Transactor memory and callbacks for Transactor to XScale interrupt notifications. A user can develop an XScale C/C++ application, however the XScale execution will not be timing accurate. Using the CBM BFM, no XScale instruction fetches occur on the Transactor. The only throttling of XScale execution is the number of outstanding memory operations that can be supported by the Transactor's XScale Gasket, and the user must make all memory references to Transactor memory using CMB BFM calls. This is very different than the real environment where all XScale instruction or data cache misses would generate a Transactor (CBM BFM) memory reference, and no CBM BFM API calls would be necessary for accessing data on the Transactor, because the Transactor memory would be directly accessible to the XScale processor (i.e. it resides on the CMB).

C.1 Summary of APIs

There are two sets of APIs that were programmed to access XScale-related transactor states: one is referred to as **XACT_IO** and the other is **CMB_IO**.

The XACT_IO API is a set of C functions that could be linked to emulate the XScale transactions through a non-simulation event, i.e. the API provides means to query/change transactor state directly, without any simulation cycles. In contrast, the CMB_IO API provides a way of simulating the XScale transaction by depositing the XScale/CMB with proper signals to trigger the XScale transaction(s) through IXP2800/IXP2400 gasket such as load or store operation. The XACT_IO is embedded in each release of IXP2800 Transactor. To access the XACT_IO API, the user must link with a C header file, **XT_WB_xactio_api.h,** which includes the definition of XACT_IO API and this header file is part of transactor release.

The CMB_IO API is supported by the XScale/CMB BFM, **cmb_bfm_ixp2800.dll / cmb_bfm_ixp2400.dll**. The BFM is implemented as a Win32 DLL that supports the FMI that is defined in the IXP2800/IXP2400 transactor. To access the CMB_IO API, the program needs to be linked with the header files, **cmb_api.h** and **cmb_api_ex.h**.

C.1.1 XACT_IO API

There are ten API functions exported from the XACT_IO. Among the ten API functions, two functions are programmed to support write/read to the IXP2800/IXP2400 XScale 32-bit address space, six functions are devised to support interface for handling interrupt requests, and two functions are provided for sanity check.

C.1.2 simRead32 / simWrite32

These two functions are provided to read/write to any 32-bit address space. The address mapping is programmed according to the *IXP2400/IXP2400 Network Processor Programmer's Reference Manual*.

int **simRead32**(char **chip_name,**
unsigned int **addr,**
unsigned int ***data**)

int **simWrite32**(char ***chip_name**,
unsigned int **addr**,
unsigned int **data**)

where:

chip_name:	name of the instantiated IXP2800/IXP2400 instance,
addr:	32-bit XScale address,
data:	write data or pointer to return read data (*data),
return value:	1 for success, -1 for fail

C.1.3 simIntConnect / simIntEnable / simIntDisable cmbIntConnect/cmbIntEnable/cmbIntDisable

The simIntXXX functions are XACT_IO API, embedded in the Transactor (both IXP2800 and IXP2400), and the cmbIntXXX functions are CMB_IO API which is supported by the XScale/gasket BFM. Both sets of API are function calls that handle XScale/gasket interrupts. The difference is in the implementation.

The XACT_IO handles the interrupt service by registering a CSR watch state to transactor, which in turn calls the service routine when the interrupt CSR state is changed.

In contrast, the CMB_IO implements the same set of routines by checking the IRQ/FIQ pins on XScale/CMB BFM. The service routines will be called when the BFM detects that the respective pin is asserted.

simIntConnectIRQ/simIntConnectFIQ provide interface for user program to register a callback function to be invoked once the interrupt status gets changed.

int **simIntConnectIRQ**(char ***chip_name**,
unsigned int **intVecto**r,
void (***isrPtr**)(unsigned int **data**),
unsigned int **usrData**)

```
int simIntConnectFIQ(char *chip_name,
                     unsigned int intVector,
                     void (*isrPtr)(unsigned int data),
                     unsigned int userData)
```

where:

chip_name:	name of the instantiated IXP2800/IXP2400 instance,
intVector:	0..31 in correspond to the interrupt vector defined in the EAS(CSR description, chapter 5, section 5.10),
isrPtr:	pointer to the registered callback function,
usrData:	data to be returned with the callback function,
return value:	1 for success, -1 for fail

C.1.4 simIntEnableIRQ / simIntEnableFIQ / simIntDisableIRQ / simIntDisableFIQcmbIntEnableIRQ / cmbIntEnableFIQ / cmbIntDisableIRQ / cmbIntDisableFIQ

These four functions provide the interface to enable/disable interrupt callback service.

```
int simIntEnableIRQ( char *chip_name,
                     unsigned int intVector)

int simIntEnableFIQ( char *chip_name,
                     unsigned int intVector)

int simIntDisableIRQ( char *chip_name,
                     unsigned int intVector)

int simIntDisableFIQ( char *chip_name,
                     unsigned int intVector)
```

where:

chip_name:	name of the instatiated IXP2800/IXP2400 instance,
intVector:	0..31 in correspond to the interrupt vector defined in the EAS(CSR description, chapter 5, section 5.10),
return value:	1 for success, -1 for fail

C.1.5 IS_CMB_ADDR_RESERVED / IS_CMB_INT_RESERVED

These two functions, only defined in XACT_IO, provide a way to verify if a calling routine will access an unimplemented area. The result of such access is considered as unpredictable.

```
int IS_CMB_ADDR_RESERVED(unsigned int addr)
int IS_CMB_INT_RESERVED(unsigned int intVector)
```

where:

addr:	32-bit XScale address,
intVector:	0..31 in correspond to the interrupt vector defined in the EAS (chapter 2, section 2.21),
return value:	1 if the address/interrupt-vector is reserved, otherwise it returns 0

C.1.6 Additional CMB_IO API

The CMB_IO API also provides several routines that emulate the XScale load, store, and swap bus operations. The **cmbRead32** and **cmbWrite32** routines are similar to the simRead32 and simWrite32 routines except that the cmbXXXX routines return a request-id and the operation is completed when **cmbSetCb** callback occurs, after the appropriate simulation delay. If the operation was a read then **cmbSetCb** will also return data.

These two sets of API functions are one-to-one mapping from one set to the other. The purpose of providing two sets of API is to provide the basic functions that supports software group to develop code for XScale related library file, in which all the accesses from the XScale are 32-bit, and to support for a more generic use. i.e. A more comprehensive transaction types, such as long-burst transactions, atomic transactions, etc. are covered.

C.1.7 cmbRead32 / cmbWrite32

These two functions provide the interface for issuing a XScale load/store of single long-word (LW).

If the XScale Gasket command FIFO is full then the **CMB_ERROR_QUEUE_FULL** code is returned.

If no callback is registered then the **CMB_ERROR_NO_CB_REGISTER** code is returned.

```
int cmbRead32(char *chip_name,
              unsigned int addr)

int cmbWrite32(char *chip_name,
               unsigned int addr,
               unsigned int data)
```

where:

chip_name:	name of the instantiated IXP2800/IXP2400 instance,
addr:	32-bit XScale address,
data:	write data,
return value:	request-id (positive value, 0-n), or error code (negative value). Such as: -3: no callback function registered **CMB_ERROR_NO_CB_REGISTER**, -2: request queue is full **CMB_ERROR_QUEUE_FULL**

C.1.8 cmbSetCb

The **cmbSetCb** function provides interface to register a callback function once the XScale/CMB BFM finishes the load transaction.

```
int cmbSetCb(char *chip_name,
             void (*fnPtr)(int reqId,
             unsigned int *retData,
             unsigned int outData),
                                   unsigned int inData)
```

where:

chip_name:	Name of the instantiated IXP2800/IXP2400 instance,
fnPtr:	Pointer to the registered callback function which is invoked by the BFM when BFM received data valid signal from gasket,
ReqId:	The request-id which was returned by cmbRead32/cmbWrite32,
retData:	Pointer to returned read data,
outData:	The data passed by callee which is the same as inData
return value:	1 for success, <0 for fail.

If an error is detected, bits 15-0 of the return value will contain the request ID that failed, bits 30-16 of the return value will contain the error code, and bit 31 will be set (making the return value negative).

Possible failure codes are

- **CMB_ERROR_BUS_ERROR** – Bus Error asserted by the XScale gasket
- **CMB_ERROR_TIME_OUT** – Timed out waiting for return data.

C.1.9 cmbSwapRead32 / cmbSwapWrite32

These two functions are used for atomic operations, which are programmed to support XScale SWP/SWPB instructions.

The **cmbSwapWrite32** is used if no read back is needed for Atomic operation. XScale can send a write(store) command with the alias address but without XSOCBI_LOCK asserted. In this case the callback for the **cmbSwapWrite32** will not wait for read data to be returned, but occur as soon as the request is acknowledged.

The following rules are enforced by the XScale Gasket for Atomic commands in Transactor I/O space (EAS, XScale Gasket Unit, chapter 2, section 2.10):

- A swap operation to the SRAM/Scratch space and the Issue I/O Request signal **is not** asserted, then the Gasket generates Atomic operation command to the Transactor.
- A swap operation to all addresses other than the SRAM/Scratch space will be treated as separate read and write commands, i.e. no Atomic command is generated.
- A swap operation to the SRAM/Scratch space and the Issue I/O Request signal **is** asserted will be treated as separate read and write commands. i.e. no Atomic command is generated.

Note: The Issue I/O Request signal is asserted by the XScale core when a data cache request is for a memory region with C=0 and B=0 (Uncachable and Unbufferable). The CMB BFM API does not provide a means of asserting Issue I/O Request, so all swap commands to SRAM or Scratch memory generate Atomic operations and references to any other address space will be treated as separate read and write commands.

int **cmbSwapRead32**(char ***chip_name**,
unsigned int **addr**,
unsigned int **byteEnable**)

int **cmbSwapWrite32**(char ***chip_name**,
unsigned int **addr**,
unsigned int **data**,
unsigned int **byteEnable**)

where:

chip_name:	name of the instantiated IXP2800/IXP2400 instance,
addr:	32-bit XScale address,
data:	write data,
byteEnable:	mask bit for each byte (1: byte enable, 0: byte masked),
return value:	request-id or -1: byteEnable invalid, Write only (CMB_FAIL) -3: no callback function registered (CMB_ERROR_NO_CB_REGISTER), -2: request queue is full (CMB_ERROR_QUEUE_FULL)

The error codes are defined in **Cmb_Client.h**.

C.1.10 cmbBFMRead32 / cmbBFMWrite32

These two functions are provided to support a more generic load/store instructions – i.e. burst transfers.

int **cmbBFMRead32**(char ***chipName**,
unsigned int **addr**,
unsigned int **size**)

int **cmbBFMWrite32**(char ***chipName**,
unsigned int **addr**,
unsigned int **size**,
unsigned int **byteEnable**,
unsigned int ***data**)

where:

chip_name:	name of the instatiated IXP2800/IXP2400 instance,
addr:	32-bit XScale address,
size:	size of the request (defined by enum CmbSize), 0: one-byte, 1: 2-byte,

2: 4-byte,
3: 8-byte,
4: 12-byte,
5: 16-byte and
6 :32-byte,

byteEnable: mask bit for each byte (1: byte enable, 0: byte masked),

data: pointer to write data

return value: request-id or
-3: no callback function registered
(CMB_ERROR_NO_CB_REGISTER),
-2: request queue is full (CMB_ERROR_QUEUE_FULL)

C.2 ENUMs

The following enum translates a size (byte count) into the CMB Size (cbiSize) encoding.

```
typedef enum __cmbSize{
        byte = 0,
        word_half,
        word,
        word_x2,
        word_x3,
        word_x4,
        word_x8,
        invalid_word
} CmbSize;
```

C.3 Defines

Completion codes:

```
#define CMB_SUCCESS 1   // Successful completion
#define CMB_FAIL        -1    // Possible causes:
                                          //  Byte Enable invalid
                                    or
                                          //  read access to Write
                                          //  only location
#define CMB_ERROR_NO_CB_REGISTER-3// No callback function
                                          //  registered
#define CMB_ERROR_QUEUE_FULL-2// Request queue is full
#define CMB_ERROR_BUS_ERROR 1// Bus error asserted by
                                          //    the   memory   bus
                                    interface
#define CMB_ERROR_TIME_OUT 2// Timeout on memory bus
                                          //  request
```

PCI Bus Functional Model D

The PCI Bus Functional Model allows simulation of the PCI unit in the network processor, as a master issuing commands to a PCI slave devices modeled by the PCI BFM.

The PCI BFM functions like any other BFM: it is loaded through the "foreign_model" command and it implements a set of console functions for access. This section will give an overview to the usage of the PCI BFM—more information can be found in the **pciconfx.h** header file.

D.1 Loading the BFM

The PCI BFM can be loaded into the transactor via the Workbench Simulation Options – Foreign Model tab. If the transactor is run on the command line without the Workbench, it can be loaded manually on the command line or in an IND script.

To load the BFM from the transactor command line, use the foreign_model command:

```
foreign_model pci_bfm2400.dll <name>
```

The parameter name can be any valid identifier. This line will load the DLL into the transactor; for the IXP2800, simply substitute pci_bfm2800 for pci_bfm2400.

D.2 Initializing the BFM

The PCI BFM will initialize its console functions and internal variables during the transactor initialization phase. When running under the Workbench, initialization is automatically performed when a debugging session is started. When running on the command line console mode without the Workbench, calling @init_IXP2400 or @init_IXP2800 on the command line will lead to the initialization of the transactor and BFM. This init sequence must be called before any of the console functions are registered.

D.3 Creating a Device

The general procedure for instantiating a device consists of defining the device, setting its attributes, and connecting the device. These are normally done in an IND script. In the following example, MODE is defined as 1, indicating that the device is a slave—currently only slave devices are supported.

```
pci_define_device(iDevId, MODE);
pci_set_param(iDevId, MEM_SPACE, iMemBase, iMemRange);
pci_connect_device(sr_chip_name, iDevId);
```

Of course, an additional set parameter line would be necessary for a device that supports both IO and memory space. When the device is connected, the chip name must be passed to the function. This specifies the name of the chip and binds the get and set signal functions to that chip name. After at least one device is initialized, the PCI BFM will respond to activity on the PCI bus.

D.4 Calling Console Functions from Another Foreign Model

Another foreign model can call the PCI BFM console functions by including the header file "pciconfx.h" and linking against the library "pci_bfm2X00.lib." These files are located in "me_tools/include/" and "me_tools/lib/", respectively.

D.5 Setting a Callback Function

A foreign model, by calling the "pci_register_callback" function can register a function of its choice, one per device, for callback. Upon completion of a transaction, the PCI BFM will make a call to any functions registered with the devices involved. Details of the transaction are passed through the callback parameters.

D.6 Header file pciconfx.h

```
/** @file pciconfx.h
The PCI Foreign Model Console Functions will allow users to test
microcode doing read, write and DMA to PCI slave devices attached
to the IXP2000.


These console functions are to be executed on the transactor con-
sole command line interface.  They can also be called via the
exported C-API by a C program in the form of another foreign model
DLL.  The functions will have the following characteristics:

1. The functions can be called by IND scripts or by hand when the
simulator is in the standalone console mode or under the Work-
bench.  Additional GUI can be added onto the Workbench to automate
the calling of these functions for configuration purpose in a sim-
ilar fashion as the PacketSim configuration. These functions are
also exported from the DLL so that they can be called from yet
another DLL programmatically (C-callable).

2. The execution of these functions does not involve simulation
cycles. In other words, the execution of these functions does not
cause any clocks in the simulator to advance. They can be called at
the console command line or in a Watch function defined by IND
```

```
script.

3. The return value of the console functions are normally zero as
they cannot be returned to the caller (Workbench).  In cases where
there are return values, they are intended to be used by C-Inter-
preter expressions in an IND script or by a C program via the
exported C-API.

The following features will be supported by the console functions:

1. Ability to instantiate up to 2 slave devices that respond to
transactions in a specified address range.

2. Slave devices have distinct I/O and memory spaces.  Each loca-
tion is initialized with its address.  Simulation of 32-bit
devices.

3. Option to print details of transactions initiated by the
IXP2000 to the PCI foreign model acting as a slave.

4. Ability to read and write slave memory locations by user
scripts.

5. Ability to check slave address contents against expected value.
This will be useful in verification.

6. Option to print details of initiator transactions issued by the
PCI foreign model as a master including the response from the
IXP2000.

7. Ability to interface with another DLL to provide transaction
notification.

*/


/* //////////////////////////////////////////////////////////////
////////////
      PCI Foreign Model Console Functions
/////////////////////////////////////////////////////////////////
///////// */

#ifndef __PCICONFX_H
#define __PCICONFX_H
```

```
#ifndef DLLIMPEX
#define DLLIMPEX __declspec(dllimport)
#endif


/**
      The pci_define_device function defines a PCI slave or master
device

      @param device_id:
            A unique integer used to identify the device in subse-
quent
            function calls.  This can be any arbitrary integer
although
            it may be easier for the caller to keep track of
sequential
            numbers starting from zero.
      @param mode:
            1 = slave.
            All other values are Reserved and should not be used.
      @return 1 on succes, 0 on failure
*/
DLLIMPEX int pci_define_device(int device_id,
                  int mode);


/**
      The pci_set_param function specifies various parameters in a
      previously defined device.

      Each slave device can respond to one I/O space address range
and
      one memory space address range.

      @param device_id The device_id used in conjunction with
            pci_define_device( )
      @param io_memory 0 = I/O space; 1 = memory space
      @param start_addr Start address for the range. Though there
is
            no restriction on alignment imposed by this function,
PCI
            devices in practice must be aligned to a boundary
divisible
            by their range.  See the PCI Spec "Configuration
Space" for
```

```
            more info.
      @param size Size of address range in bytes
      @return 1 on success, 0 on failure
*/
DLLIMPEX int pci_set_param(int device_id, int io_memory, int
start_addr, int size);




/**
      The pci_connect_device function connects a device to the PCI
bus
      on a specific chip.

      The normal sequence of initialization is as follows.

      -# pci_define_device()
      -# pci_set_param(I/O space) | pci_set_param(memory space)
      -# pci_connect_device()

      @param chip_name The name of the chip
      @param device_id The device_id used with pci_define_device()
            and pci_set_param()
      @return 1 on succes, 0 on failure
*/
DLLIMPEX int pci_connect_device(char *chip_name, int device_id);

/**
      pci_remove_dev disconnects a device and then deletes it.
Essentially,
      the function performs the inverse operations of
pci_define_device
      and pci_connect_device.

      @param iDev The device ID of the device to be removed.  The
device
                        will be disconnected from the PCI Bus and
then deleted.
                        The device ID can be re-used in new calls
to
                        pci_define_device
      @return 1 on success, 0 on failure
*/
DLLIMPEX int pci_remove_dev(int iDev);
```

```
/**
      The pci_slave_read function reads a slave device's memory.
      It provides a backdoor mechanism to read memory contents
      from the device.

      Address should fall on 32-bit dword boundaries.  Those that
      do not will be rounded to the corresponding dword.  For
      example, for address 2, dword 0 (bytes 0-3) will be
returned.

      @param io_space
            0 = use I/O space,
            1 = use memory space
      @param address A 32-bit pci address within the device range.
                    This is the full PCI address, not the address
                           relative to the device's base address.
      @return The dword containing the specified address.  See
              the notes above.
*/
DLLIMPEX int pci_slave_read(int device_id,
               int io_space,
               int address);

/**
      The pci_slave_check function reads a slave device's memory,
      checks it against the expected data item, and prints an
error
      message if the data compare fails.

      Addresses should fall on 32-bit dword boundaries.  If they
do
      not, the dword line will be returned.  Ex. for address 5, the
      second dword (bytes 4-7) will be returned.  The calling
      function must use pci_slave_read, and mask the returned
      contents accordingly.

      @param io_space
            0 = use I/O space,
            1 = use memory space
      @param address The full PCI address, not the address
                           relative to the device's base address.
      @oaram data The expected data to be compared against
      @return 1 on succes, 0 on failure
```

```
*/
DLLIMPEX int pci_slave_check(int device_id,
                int io_space,
                int address,
                int data);

/**
      The pci_slave_write function writes to a slave device’s mem-
ory.

      ONLY 32-bit data and address are supported.
      It provides a backdoor mechanism to initialize memory loca-
tions
      in the device.

      @param io_space
            0 = use I/O space,
            1 = use memory space
      @param address The full PCI address, not the address
                     relative to the device’s base address.
      @param data The data to be written
      @param byte_en The Byte Enables bits [4::0] active low
      @return 1 on succes, 0 on failure
*/
DLLIMPEX int pci_slave_write(int device_id,
                int io_space,
                int address,
                int data,
                int byte_en);

/**
      The pci_set_error_level function controls the verbosity of
debug
      printout.

      @param level Uses codes from [-1 .. 2]
      <table>
            <tr><td> (-1) prints everything including debug trace
information </td></tr>
            <tr><td> ( 0) prints only informational, warning and
error messages.
                          Informational messages include all
PCI transaction details. </td></tr>
            <tr><td> ( 1) prints only warning and error messages
</td></tr>
```

```
            <tr><td> ( 2) prints only error messages </td></tr>
      </table>
      @return 1 on succes, 0 on failure
*/
DLLIMPEX int pci_set_error_level(int level);


/**
      Register a callback function from another DLL so that any
      accesses to slave devices by the IXP can be notified.

      Each PCI bus transaction will trigger one callback.

      This function is to be called by a C program from another
DLL.
      Parameters for Callback are listed below

      Callback(
            - int dev_id     (identifies which device)
            - int rw         (0=read, 1=write, 2=abnormal termina-
tion)
            - int space      (0=IO, 1=Mem)
            - int addr       (starting address)
            - int size       (number of 32-bit dwords)
      )

*/
DLLIMPEX void pci_register_callback(
            int device_id,
            void (*fnCallback)(int dev_id, int rw, int space, int
addr, int size)
);


#endif // __PCICONFX_H
```

SPI4 Bus Functional Model E

This chapter describes two Application Program Interfaces (APIs) supported through the IXP2800/IXP2850 SPI4 Bus Function Model (BFM) - which simulates the SPI4 protocol, OIF-SPI4-02.0, through the transactor Foreign Model Interface (FMI).

E.1 Overview

The SPI4 BFM is invoked by the IXP2800/IXP2850 transactor through the foreign_model transactor command and synchronized with the same event control logic as the transactor.

The SPI4 BFM, spi4_bfm.dll, provides function calls for users to configure and access FIFO status either through extended console functions or a C-API. The extended console functions are recognized by the C-interpreter embedded in the transactor, such that the programmer can invoke these functions through transactor scripts or by typing them into the transactor command line. The C-API is used mainly for an external software component, such as the Developers Workbench, to use the SPI4 BFM through C functions.

E.2 SPI4 BFM Help

To access additional information about the SPI4 BFM:

spi4_help();	This command prints out all supported console functions. The commands are: spi4_help, spi4_help_fn, spi4_define_device, spi4_set_device_rx_param, spi4_set_port_rx_param, spi4_set_device_tx_param, spi4_set_port_tx_param, spi4_create_device, spi4_connect_device, spi4_enable_device, spi4_enable_port, spi4_set_device_stop_control, spi4_set_port_stop_control, spi4_set_sim_options, spi4_get_tx_stat spi4_version, spi4_set_network_log, spi4_get_receive_stats_packet, spi4_get_receive_stats_byte, spi4_get_receive_stats_cycle, spi4_get_transmit_stats_packet, spi4_get_transmit_stats_byte, spi4_get_transmit_stats_cycle, spi4_get_rx_buffer_byte, spi4_get_rx_buffer_int32, spi4_get_tx_buffer_byte, spi4_get_tx_buffer_int32, spi4_get_rx_clock_cycle spi4_get_tx_clock_cycle, spi4_reset_stat, spi4_debug_log
spi4_help_fn(fn_name);	This command prints out the argument list and a brief explanation of the specified function. For example: spi4_help_fn("spi4_define_device"); spi4_define_device(string device_type, int device_id, int num_ports) device_type: any string device_id: unique device id num_ports: # of ports 1

E.3 Console Functions

The SPI4 BFM supports the concept of a device/port as devised by the IXP2800/IXP2850 development tool (WB). This concept allows for single instantiation of SPI4 BFM via a transactor command, ***foreign_model***, to handle multiple logical spi4 devices in the simulation environment. Each spi4 device is dubbed with a unique device id; each device has its own configuration. This configuration information includes:

- Device/Port configuration
- Flow control configuration
- Simulation control
- Statistic information access

The following sections will describe how to use each of the console functions supported by the latest **spi4_bfm.dll**.

E.3.1 Device/Port Configuration

The *spi4_define_device* function defines a spi4 device for use in the simulation.

```
int spi4_define_device(char *device_type,
                              int device_id,
                              int num_ports)
```

Where device_type: N/A

```
device_id: unique device id for each device
num_ports: number of ports in the specified device
```

The *spi4_set_device_X_param*/*spi4_set_port_X_param* sets the value of the specified parameter of the specified spi4 device/port.

```
int spi4_set_device_rx_param(int device_id,

                             int param_id,

                             int param_value)

int spi4_set_port_rx_param(int device_id,

                             int port_num,

                             int param_id,

                             int param_value)

int spi4_set_device_tx_param(int device_id,

                             int param_id,

                             int param_value)

int spi4_set_port_tx_param(int device_id,
```

int port_num,

int param_int,

int param_value)

Where device_id: unique device id for each device

port_num: port in the specified device

param_id: parameter ID (see below for the complete list)

param_value: the value for the specified parameter

Parameter ID	Parameter Description
1	Rx/Tx FIFO size (in byte)
2	Rx/Tx FIFO Low Water Mark (in byte)
3	Rx/Tx Line Rate (in Mb/s)
4	Rx/Tx Device/Port Physical address
5	Rx Maximum Burst Cycle (in 2-byte/unit)
6	Rx/Tx Inter-Packet Time (in nano-second)
7	Rx/Tx MSF Frequency (in MHz)
8	Rx/Tx Minimum Burst Size (in byte)
9	Rx/Tx SPI4 Flow Control MaxBurst1 (16-byte/unit)
10	Tx SPI4 Flow Control time interval FIFO_MAX_T (msf cycle)
11	Tx SPI4 Flow Control time interval FIFO_MAX_T (msf cycle)
12	Rx/Tx High Water Mark (in byte)

The *spi4_create_device* instantiate a new spi4 device with the assigned device_id.

```
int spi4_create_device(int device_id)
```

Where device_id: unique device id for the device

The *spi4_connect_device* sets the connection between the specified spi4 device and the specified chip (ixp2800/ixp2850).

```
int spi4_connect_device(char *ChipName, int device_id, int
direction)

Where ChipName: the instance name string of the target ixp2800/
ixp2850

      device_id: unique device id for the device direction:

            1: data flow from spi4_bfm to ixp/msf

            2: data flow from ixp/msf to spi4_bfm

            3: connect data flow from both directions
```

The *spi4_enable_X* enables/disables the specified spi4 device/port. When a device/port is disabled, there will be no data transferred from data stream to IXP/MSF nor passing data from IXP/MSF to packet_sim.dll (this is the interface between spi4_bfm.dll and external data stream generator).

```
int spi4_enable_device(int device_id,
      int state)
int spi4_enable_port(int device_id,
      int port_num,
      int state)
Where device_id: unique device id for each device
      port_num: port in the specified device
      state: 0: disable; 1: enable
```

E.3.2 Simulation Control

The *spi4_set_X_stop_control* sets simulation control over the specified spi4 device/port.

```
int spi4_set_device_stop_control(int device_id,
                                 int stop_type,
                                 int num_packets)
int spi4_set_port_stop_control(int device_id,
                               int port_num,
                               int stop_type,
                               int num_packets)
```

Where device_id: unique device id for each device

```
port_num: port in the specified device
stop_type: 0: no stop-control ; 1: stop on sending X packets
      to IXP/MSF; 2: stop on receiving X packets from IXP/
      MSF
num_packets: # of packets received/transmitted for the stop
      condition to be true
```

The *spi4_set_sim_options* sets simulation control over:

- Running in unbounded mode - spi4 mode at full-bandwidth of IXP/MSF bus
- ignoring the spi4_bfm receive buffer overflow, and
- ignoring the spi4_bfm transmit buffer underflow.

```
int spi4_set_sim_options(int run_unbounded,
                         int ignore_rx_buffer_overflow,
                         int ignore_tx_buffer_underflow)
```

```
Where run_unbounded: 1: run unbounded mode otherwise run normal
mode

      ignore_rx_buffer_overflow: 1: ignore spi4_bfm rx FIFO
overflow
      ignore_tx_buffer_underflow: 1: ignore spi4_bfm tx FIFO
            underflow
```

E.3.3 Flow Control

The spi4_set_rx_fc_info/spi4_set_tx_fc_info/spi4_set_tx_calendar/ spi4_set_rx_calendar sets the flow control information. The parameters specified in these functions are also supported by the spi4_set_device_rx_param/ spi4_set_device_tx_param.

Note: The spi4_set_rx_calendar function is only supported for the IXP2800IXP2850 B0 (or later) stepping.

```
int spi4_set_rx_fc_info(char *device_id_str,
                        int nBurst1,
                        int nBurst2,
                        int nMaxiFifoInterval,
                        int nHWM,
                        int nPortAddr)
int spi4_set_tx_fc_info(char *device_id_str,
                        int nBurst1,
                        int nBurst2,
                        int nMaxiFifoInterval,
                        int nHWM,
                        int nPortAddr)
int spi4_set_rx_calendar(int device_id,
                        int calendar_length,
                        int index,
                        int calendar);
```

```
int spi4_set_tx_calendar(int device_id,
                        int calendar_length,
                              int index,
                              int calendar);
```

```
Where device_id: unique device id for each device

      device_id_str: string representation of device_id

      port_num: port in the specified device
      nBurst1: Rx/Tx SPI4 Flow Control MaxBurst1 (16-byte/unit)
      nBurst2: Rx/Tx SPI4 Flow Control MaxBurst2 (16-byte/unit)
      nMaxiFifoInterval: Tx SPI4 Flow Control time interval
            FIFO_MAX_T (msf cycle)
      nHWM: Rx/Tx High Water Mark
      nPortAddr: physical port address
      calendar_length: MSF/Tx calendar length (the latest spi4_bfm
            uses the transactor XACT_IO interface to retrieve this
            information automatically once the Tx flow control is
            turned on)
      index: calendar index
      calendar: calendar value
```

E.3.4 Statistic Information Access

The `spi4_get_receive_stats_X/spi4_get_transmit_stats_X` returns the statistics information on rx/tx FIFO of spi4_bfm.

```
int spi4_get_receive_stats_packet(int device_id,
                                  int port_num)

int spi4_get_receive_stats_byte(int device_id,
                                  int port_num)

int spi4_get_receive_stats_cycle(int device_id,
                                  int port_num)

int spi4_get_transmit_stats_packet(int device_id,
                                  int port_num)

int spi4_get_transmit_stats_byte(int device_id,
                                  int port_num)

int spi4_get_transmit_stats_cycle(int device_id,
```

```
                                          int port_num)
Where device_id: unique device id for each device
      port_num: port in the specified device
```

The *spi4_get_rx_buffer_byte*/ *spi4_get_tx_buffer_byte* returns one-byte data from spi4_bfm rx/tx FIFO, and *spi4_get_rx_buffer_int32*/ *spi4_get_tx_buffer_int32* returns four-byte of data from spi4_bfm rx/tx FIFO.

```
int spi4_get_rx_buffer_byte(int device_id,
                              int port_num,
                              int byte_index)
int spi4_get_rx_buffer_int32(int device_id,
                              int port_num,
                              int byte_index)
int spi4_get_tx_buffer_byte(int device_id,
                              int port_num,
                              int byte_index)
int spi4_get_tx_buffer_int32(int device_id,
                              int port_num,
                              int byte_index)
Where device_id: unique device id for each device
      port_num: port in the specified device
      byte_index: index for the 1st byte in the FIFO
```

The *spi4_get_rx_clock_cycle*/*spi4_get_tx_cycle_count* returns the spi4_bfm cycle count for reference.

```
int spi4_get_rx_clock_cycle(int device_id,
                              int port_num)
int spi4_get_tx_clock_cycle(int device_id,
                              int port_num)
Where device_id: unique device id for each device
      port_num: port in the specified device
```

The *spi4_reset_stat* resets both the rx/tx statistics over spi4_bfm.

```
int spi4_reset_stat(void);
```

The *spi4_version* prints out the version and built time of spi4_bfm.dll

```
int spi4_version();
```

E.4 C-API

The following C-API provides access functions to retrieve SPI4 BFM FIFO statistics through C-function calls. They are the C-function implementations of the aforementioned statistic information access console functions. The only addition is the *RegisterMessageCallback* function, which allows the WB to register a callback routine to handle message print out in the GUI environment.

```
SPI4_WMAC_API int GetNumBytesInRxBuffer(int device_id,
                                            int port_num)
SPI4_WMAC_API void GetRxBufferBytes(int device_id,
                                    int port_num,
                                    unsigned char *buffer,
                                    int buffer_size)
SPI4_WMAC_API int GetNumBytesInTxBuffer(int device_id,
                                            int port_num)
SPI4_WMAC_API void GetTxBufferBytes(int device_id,
                                    int port_num,
                                    unsigned char *buffer,
                                    int buffer_size)
SPI4_WMAC_API int GetReceiveStats(int device_id,
                                  int port_num,
                                          int&
num_packets_received,
                                          int&
num_bytes_received,
                                          int& num_cycles)
SPI4_WMAC_API int GetTransmitStats(int device_id,
                                   int port_num,
                                   int& num_packets_received,
```

```
                                       int& num_bytes_received,
                                       int& num_cycles)
SPI4_WMAC_API bool GetRxClockCycle(int device_id,
                                       int port_num,
                                       unsigned int& cycles)
SPI4_WMAC_API bool GetTxClockCycle(int device_id,
                                       int port_num,
                                       unsigned int& cycles)
SPI4_WMAC_API void ResetStatistics()
SPI4_WMAC_API int RegisterMessageCallback(void (*fp)(char
*pMessage,
                                       int Severity))
```