



Netronome Network Flow Processor 6xxx

NFP SDK version 6.0

Development Tools User's Guide

- Proprietary and Confidential -

b677.dr6296

**Product code
030-00022-004**

Netronome Network Flow Processor 6xxx: Development Tools User's Guide

Copyright © 2008-2016 Netronome

COPYRIGHT

No part of this publication or documentation accompanying this Product may be reproduced in any form or by any means or used to make any derivative work by any means including but not limited to by translation, transformation or adaptation without permission from Netronome Systems, Inc., as stipulated by the United States Copyright Act of 1976. Contents are subject to change without prior notice.

WARRANTY

Netronome warrants that any media on which this documentation is provided will be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of shipment. If a defect in any such media should occur during this 90-day period, the media may be returned to Netronome for a replacement.

NETRONOME DOES NOT WARRANT THAT THE DOCUMENTATION SHALL BE ERROR-FREE. THIS LIMITED WARRANTY SHALL NOT APPLY IF THE DOCUMENTATION OR MEDIA HAS BEEN (I) ALTERED OR MODIFIED; (II) SUBJECTED TO NEGLIGENCE, COMPUTER OR ELECTRICAL MALFUNCTION; OR (III) USED, ADJUSTED, OR INSTALLED OTHER THAN IN ACCORDANCE WITH INSTRUCTIONS FURNISHED BY NETRONOME OR IN AN ENVIRONMENT OTHER THAN THAT INTENDED OR RECOMMENDED BY NETRONOME.

EXCEPT FOR WARRANTIES SPECIFICALLY STATED IN THIS SECTION, NETRONOME HEREBY DISCLAIMS ALL EXPRESS OR IMPLIED WARRANTIES OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE.

Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to some users of this documentation. This limited warranty gives users of this documentation specific legal rights, and users of this documentation may also have other rights which vary from jurisdiction to jurisdiction.

LIABILITY

Regardless of the form of any claim or action, Netronome's total liability to any user of this documentation for all occurrences combined, for claims, costs, damages or liability based on any cause whatsoever and arising from or in connection with this documentation shall not exceed the purchase price (without interest) paid by such user.

IN NO EVENT SHALL NETRONOME OR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THE DOCUMENTATION BE LIABLE FOR ANY LOSS OF DATA, LOSS OF PROFITS OR LOSS OF USE OF THE DOCUMENTATION OR FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, EXEMPLARY, PUNITIVE, MULTIPLE OR OTHER DAMAGES, ARISING FROM OR IN CONNECTION WITH THE DOCUMENTATION EVEN IF NETRONOME HAS BEEN MADE AWARE OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL NETRONOME OR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THE DOCUMENTATION BE LIABLE TO ANYONE FOR ANY CLAIMS, COSTS, DAMAGES OR LIABILITIES CAUSED BY IMPROPER USE OF THE DOCUMENTATION OR USE WHERE ANY PARTY HAS SUBSTITUTED PROCEDURES NOT SPECIFIED BY NETRONOME.

Revision History

Date	Revision	Description
20 April 2016	004	Updated for NFP SDK 6.0 Beta 1
18 December 2015	003	Updated for NFP SDK 6.0 Preview
20 January 2014	002	Updated for NFP SDK 5.0 Beta
4 June 2013	001	Updated for NFP SDK 5.0 Alpha 1

Table of Contents

1. Introduction	12
1.1. Scope	12
1.2. Related Documents	12
1.3. Terminology	13
2. Programmer Studio	15
2.1. Overview	15
2.1.1. Features	15
2.1.2. Debugging Support	15
2.1.3. Getting Help	16
2.1.4. Programmer Studio Revision Information	16
2.2. About the Graphical User Interface (GUI)	16
2.2.1. About Windows, Toolbars, and Menus	17
2.2.2. Hiding and Showing Windows and Toolbar	18
2.2.3. Customizing Toolbars and Menus	20
2.2.4. GUI Toolbar Configurations	23
2.3. Programmer Studio Projects	23
2.3.1. Creating a New Project	24
2.3.2. Opening a Project	28
2.3.3. Saving a Project	28
2.3.4. Saving Copies of a Project	29
2.3.5. Closing a Project	29
2.3.6. Specifying a Default Project Folder	30
2.3.7. Changing Chip Type in Project	30
2.4. About the Project Workspace	31
2.4.1. FileView	31
2.4.2. ThreadView	33
2.4.3. InfoView	35
2.4.4. IR Graphs	35
2.5. Working with Files	36
2.5.1. Creating New Files	37
2.5.2. Opening Files	38
2.5.3. Closing Files	38
2.5.4. Saving Files	39
2.5.5. Saving Copies of Files	39
2.5.6. Saving All Files at Once	40
2.5.7. Working With File Windows	40
2.5.8. Printing Files	41
2.5.9. Inserting Into and Removing Files from a Project	42
2.5.10. Editing Files	43
2.5.11. Bookmarks and Errors/Tags	44
2.5.12. Find In Files	45
2.5.13. Fonts and Syntax Coloring	46
2.5.14. Macros	47
2.5.15. Functions	48
2.5.16. P4 User Config File	49
2.6. The Assembler	63
2.6.1. Root Files and Dependencies	63
2.6.2. Selecting Assembler Build Settings	64

2.6.3. Specifying Assembler Build Settings	68
2.6.4. Register Usage	74
2.7. The C Compiler for Netronome Processors	76
2.7.1. Adding C Source Files to Your Project	77
2.7.2. Selecting C Compiler Build Settings	77
2.7.3. Invoking the Compiler	86
2.7.4. Compilation Errors	87
2.8. Specifying P4/Managed C Sandbox Build Settings	88
2.8.1. Adding P4 and Sandbox C Source Files to Your Project	88
2.8.2. Selecting P4 Sandbox C/Managed C Compiler Build Settings	89
2.8.3. Invoking the P4 Compiler	92
2.8.4. Compilation Errors	93
2.9. Specifying Linker Build Settings	94
2.9.1. Customizing Linker Settings	94
2.9.2. Building and Rebuilding a Project	101
2.10. Configuring the Netronome NFP-6000 Simulation Environment	105
2.10.1. Clock Frequencies	105
2.10.2. Memory	106
2.11. Packet Streaming	107
2.11.1. Supported File Formats	109
2.11.2. Configuring Packet Streaming	110
2.11.3. Packet Streaming Statistics	112
2.12. Debugging	113
2.12.1. Hardware Debugging	115
2.12.2. Starting and Stopping the Debugger	117
2.12.3. Changing Simulation Options	117
2.12.4. Exporting the Startup Script	127
2.12.5. Changing Hardware Options	127
2.12.6. Thread Windows	129
2.12.7. Run Control	144
2.12.8. About Breakpoints	149
2.12.9. About Code Breakpoints	155
2.12.10. Displaying Register Contents	165
2.12.11. Data Watches	167
2.12.12. Memory Watch	179
2.12.13. NBI Memory Watch	187
2.12.14. Watch Scripts	195
2.12.15. NBI PM Modification Pipeline	205
2.12.16. NBI TM Packet Descriptor	206
2.12.17. Execution Coverage	207
2.12.18. ME Performance Statistics	213
2.12.19. Performance Statistics	217
2.12.20. Thread and Queue History	222
2.12.21. Queue Status	232
2.12.22. Event List View	235
2.12.23. Thread Status	237
2.12.24. P4 Inspectors	239
3. Assembler	242
3.1. Assembly Process	242
3.1.1. Command Line Arguments	242
3.1.2. Protect Macro Locals	244

3.1.3. Arithmetic Notation Feature	246
3.1.4. Assembler Steps	248
3.1.5. Case Sensitivity	249
3.1.6. Assembler Optimizations	250
3.1.7. Processor Type and Revision	250
4. C Compiler	251
4.1. Explicit Mode Command Line	251
4.1.1. Explicit Mode Supported Compilations	252
4.1.2. Explicit Mode Supported Compiler Options	252
4.1.3. Compiler Steps	257
4.1.4. Case Sensitivity	257
5. P4 Compiler	258
5.1. The P4 Front-end Compiler	258
5.1.1. Standard and Intrinsic Metadata	258
5.1.2. Target Specific P4 Pragmas	259
5.1.3. Explicit P4 Front-end Compilations	260
5.2. The P4 Back-end Compiler	261
5.3. Building P4 Applications	261
6. Linker	263
6.1. About the Linker	263
6.2. Microengine and Island IDs	263
6.2.1. Island IDs	263
6.2.2. MEIDs	264
6.3. Netronome Flow Firmware Linker (NFLD)	265
6.3.1. Usage	265
6.3.2. Command Line Options	265
6.4. Generating a Microengine Application	266
6.4.1. Using shared control store feature	266
6.4.2. Using next neighbor configurations	267
6.5. Syntax Definitions	267
6.5.1. Import Variable and Memory Symbol Definition	267
6.5.2. Microengine Assignment	269
6.6. Examples	269
6.6.1. .map File Example	269
6.7. Init-CSR	270
6.7.1. General Description	270
6.7.2. CSR Name Syntax and Scope	270
6.7.3. CSR Database Conflict Rules	271
6.8. Netronome Flow Firmware File Format	275
6.8.1. NFP ELF file header (ElfN_Ehdr)	275
6.8.2. NFP ELF section headers (ElfN_Shdr)	276
6.8.3. NFP ELF symbols (ElfN_Sym)	277
7. Command Line Tools	279
7.1. nfiv - Network Flow Import Variable Tool	279
7.1.1. Usage	279
7.1.2. Command Line Options	279

7.1.3. Key Value Import Variable Data File	280
7.1.4. UOF Import Variable Data File	280
7.2. cling - C interpreter	281
7.2.1. Invoking the C interpreter	281
7.2.2. Command Line Options	282
7.2.3. Meta Commands	283
7.2.4. Programming Interface	283
7.3. Tools common to SDK and BSP	283
7.3.1. nfp-cpp and nfp-xpb	283
7.3.2. nfp-power	284
7.3.3. nfp-mem	284
7.3.4. nfp-reg	284
7.3.5. nfp-mereg	285
7.3.6. nfp-nffw	285
7.3.7. nfp-rtsym	286
7.3.8. nfp-hwinfo	286
7.3.9. nfp-tcache	286
7.3.10. nfp-tminit, nfp-macinit	286
8. Simulator	287
8.1. Overview	287
8.2. Simulator Limitations	287
8.3. Networking	287
8.4. PCIe Simulation	288
8.5. Hardware Debug	288
8.6. State Access	288
8.7. Invoking the Simulator	289
8.7.1. Command Line Options	289
8.7.2. Environment Variables	290
8.8. Simulator Configuration File	290
8.9. Simulator API	291
9. Technical Support	292
Programmer Studio Shortcuts	293
P4 C Sandbox	298
B.1. Access to P4 data	298
B.1.1. Headers	298
B.1.2. Metadata	299
Managed C Applications	300
Run Time Environment (RTE)	303
D.1. Virtual and Physical Port Setup	305
D.2. Loading a NFP Device at System Startup	305
D.3. Setting Up Multiple NFP Devices on a Single Host	307

P4 CLI Tools Example	308
E.1. P4 Program	308
E.2. Building the P4 Program	309
E.3. Setting up a static P4 Configuration	310
E.4. Loading the Design onto the NFP	311
E.5. Some useful commands	311

List of Figures

2.1. The Programmer Studio GUI	17
2.2. Floating Window, Tool Bar, and Menu Bar	18
2.3. Specify Debug-only NFFW Files Dialog Box	27
2.4. Configure Tabs Dialog Box	44
2.5. Go To Line Dialog Box	44
2.6. Build - General Tab	65
2.7. Build - Assembler Tab	68
2.8. Register Usage Window	74
2.9. Build Settings Compiler tab	78
2.10. Selecting C Source Files to Compile	80
2.11. Selecting Assembler Source Files to Compile	82
2.12. Build Settings P4/Managed C tab	89
2.13. Build - Linker - General Tab	95
2.14. Build - Linker - List File Assignments Tab	97
2.15. Build - Linker - Memories Tab	99
2.16. NFP-6000 Clock Frequencies Options	106
2.17. NFP-6000 Memory Options	107
2.18. Packet Streaming Configuration Dialog Box	110
2.19. Packet Streaming Configuration	110
2.20. Packet Streaming Statistics Dialog Box	112
2.21. Packet Streaming Statistics Dialog	113
2.22. Marking Instructions for the Network Processor	118
2.23. Using Imported Variable Data at Startup in Simulation Mode	122
2.24. Using Imported Variable Data at Startup in Hardware Mode	123
2.25. History Dialog Box	124
2.26. The Assembler Thread Window	131
2.27. The Compiled Thread Window	132
2.28. Expanding Macros	136
2.29. Simulation Options window	142
2.30. Instruction Operand Tracing	143
2.31. Thread window list	144
2.32. Breakpoints window	150
2.33. Breakpoint Buttons and Properties	151
2.34. Breakpoint Group Management Dialog	154
2.35. Inline Function Breakpoints in Source and List Views	160
2.36. Multi-Microengine Breakpoint Dialog Box	163
2.37. Code Breakpoints Assign to Group Dialog Box	164
2.38. Add ME Watch Dialog Box	169
2.39. Add Chip CSR Watch Dialog Box	170
2.40. Data Watch for Inlined functions	171
2.41. Add Data Watch Dialog Box - User-defined Registers and Symbols	173
2.42. Insert Script Files	195
2.43. Watch Script Header Tags	196
2.44. Watch Script Header Format	196
2.45. Watch Script ps_params Section	196
2.46. Watch Script input_params Section	197
2.47. Watch Script selection Type Input Parameter	197
2.48. Watch Script main Section	199
2.49. Watch Script main Function	199
2.50. Execute Watch Script	200
2.51. Watch Script Dialog	201

2.52. Watch Script Output	201
2.53. decode_packet_tshark.c Dialog	202
2.54. decode_packet_tshark.c Dialog	203
2.55. decode_struct.c Dialog	204
2.56. decode_packet_tshark.c Dialog	205
2.57. The Execution Coverage Window	208
2.58. Report Execution Coverage for the Microengines	211
2.59. Report Execution Coverage by Line	212
2.60. Performance Statistics - Summary Tab	214
2.61. Save Performance Simulation Statistics to a File	215
2.62. Performance Statistics - Microengine Tab	215
2.63. Performance Statistics - All Tab	216
2.64. Performance Statistics - DSF-CPP Bandwidth Tab	218
2.65. Performance Statistics - Mini Packet Bus Bandwidth Tab	219
2.66. Performance Statistics - Selection Panel	220
2.67. Performance Statistics - Options Panel	221
2.68. Performance Statistics - History Toolbar	221
2.69. History Window	222
2.70. Queue Display Property Sheet	223
2.71. Display Threads Property Page	224
2.72. Customize History	228
2.73. Queue Status Window	233
2.74. Enable Display Windows for Event List View	235
2.75. Event List View	236
2.76. Event List Filtering	237
2.77. The Thread Status Window	238
3.1. Assembly Process	249
4.1. Compilation Steps	257

List of Tables

2.1. Simulation and Hardware Mode Features	114
2.2. Instruction Markers	138
2.3. List View Markers	139
4.1. Supported CLI Options	252
5.1. P4 Standard Metadata	258
5.2. P4 Intrinsic Metadata	259
5.3. NFP Target P4 Pragmas	259
5.4. Supported P4 Compiler CLI Options	260
5.5. Supported P4 Back-end Compiler CLI Options	261
5.6. Supported CLI Options for building a <code>nffw</code> file	261
6.1. NFP-6xxx Island ID Aliases	264
6.2. Linker Command Line Options	265
6.3. Init-CSR Lookup Target Map Names	271
6.4. List file Init-CSR conflict rules	272
6.5. Target and external Init-CSR conflict rules	273
7.1. <code>nfiv</code> Primary Command Line Options	279
7.2. <code>nfiv</code> Commands	279
7.3. <code>nfiv modify</code> Options	280
7.4. <code>cling</code> Command Line Options	282
7.5. <code>cling</code> Meta Commands	283
7.6. NFP SDK <code>nfp-cpp</code> Command Line Options	284
7.7. NFP SDK <code>nfp-nffw</code> Command Line Options	285
A.1. Programmer Studio Shortcuts — Files	293
A.2. Programmer Studio Shortcuts — Projects	294
A.3. Programmer Studio Shortcuts — Edit	294
A.4. Programmer Studio Shortcuts — Bookmarks	295
A.5. Programmer Studio Shortcuts — Builds	295
A.6. Programmer Studio Shortcuts — Debug	296
A.7. Programmer Studio Shortcuts — Run Control	296
A.8. Programmer Studio Shortcuts — View	297
D.1. SDk6 RTE Input Arguments and Descriptions	304

1. Introduction

1.1 Scope

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

- Chapter 2 describes Programmer Studio and its graphical user interface (GUI).
- Chapter 3 describes how to run the Assembler.
- Chapter 4 describes how to run the C Compiler for Netronome Processors.
- Chapter 5 describes how to run the P4 Compiler for Netronome Processors.
- Chapter 6 describes how to run the Linker.
- Chapter 7 provides information on how to use command line tools provided with SDK.
- Chapter 8 describes the Network Flow Simulator and its commands.
- Chapter 9 provides details on where additional information can be obtained.
- Appendix A contains a listing and description of commonly used shortcuts.
- Appendix B describes the P4 C Sandbox use and how to call into C code from P4 program.
- Appendix C describes managed C applications and how to insert user defined C code which will then be built with the dataplane provided by the SDK.
- Appendix D describes the Run Time Environment (RTE) module.
- Appendix E describes P4 command-line tool usage.

This guide is intended for use by Developers and Systems Programmers.

1.2 Related Documents

Descriptive Name	Description
Netronome Network Flow Processor 6xx0: Databook	Contains detailed reference information on the Netronome Network Flow Processor NFP-6000.
Netronome Network Flow Processor 6000: Datasheet	Provides a functional overview of the Netronome Network Flow Processor NFP-6000's internal hardware, signals and electrical and mechanical specifications.
Netronome Network Flow Processor 6000: Microengine Programmer's Reference Manual	Provides a reference for microcode programming of the Netronome Network Flow Processor NFP-6000.
Netronome Network Flow Processor 6000: Network Flow C Compiler User's Guide	Presents information, language structures and extensions to the language specific to the Netronome Network Flow C Compiler for Netronome NFP-6000.

Descriptive Name	Description
Netronome Network Flow Compiler LibC: Reference Manual	Specifies the subset and the extensions to the language that support the unique features of the Netronome Network Flow Processor NFP-6000 product line.
Netronome Network Flow Processor 6000: Network Flow Assembler System User's Guide	Describes the syntax of the NFP-6000's assembly language, supplies assembler usage information, and lists assembler warnings and errors.

1.3 Terminology

Acronym	Description
API	Application Programming Interface
ARM	ARM Holding plc
BFM	Bus Function Model
CLI	Command Line Interface
CNTL	Control
CPU	Central Processing Unit
CSIX	Common Switch Interface
CSR	Control and Status Registers
DDR	Dual Data Rate
DLL	Dynamic-link Library
DRAM	Dynamic Random Access Memory
ELF	Executable and Linkable Format
EOP	End-of-Packet indicator
FCE	Flow Control Egress
FIFO	First In First Out
FM	Foreign Model
FMI	Foreign Model Interface
GPR	General Purpose Register
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
IA	Intel Architecture
IFG	Inter-Frame Gap
IPG	Inter-Packet Gap
IVD	Import Variable Data
IXA	Internet Exchange Architecture

Acronym	Description
KV	Key Value Import Variable Data
ME	Microengine
MIP	Microcode Information Page
MSF	Media and Switch Fabric
NFAS	Network Flow Assembler System
NFCC	Network Flow Compiler
NFFW	Netronome Flow Firmware
NFIV	Network Flow Import Variable Tool
NFLD	Network Flow Linker
NFLO	Network Flow Loader
NFP	Netronome Network Flow Processor
NOP	No Operation
NTS	Network Traffic Simulator
OSSL	Operating System Services Layer
PCIe	PCI express
PDU	General term referring to packets, frames and so forth.
QDR	Quad Data Rate
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
SOP	Start-of-Packet indicator
SPI	Serial Peripheral Interface
SPI4	System Packet Interface 4.2
SRAM	Static Random Access Memory

2. Programmer Studio

2.1 Overview

Programmer Studio is an integrated development environment for assembling, compiling, linking, and debugging microcode that runs on the Network Processor Microengines.

2.1.1 Features

Important Programmer Studio features include:

- Source file editing
- Source level debugging
- Debug-only project creation mode
- P4 project creation mode
- Managed C project creation mode
- Execution history
- Statistics
- Media Bus device and network traffic simulation for the NFP6000 and NFP4000 chip
- Distributed Switch Fabric (DSF) - Command Push/Pull (CPP) Bus performance Statistics for the NFP6000 and NFP4000 chip
- Cling is provided as command line interface to the Networking Processor simulator. For more information about Cling please refer to Section 7.2
- Customizable graphical user interface (GUI) components

2.1.2 Debugging Support

Programmer Studio supports debugging in four different configurations:

- **Local simulation**, in which Programmer Studio and the Network Processor simulator both run on the same Microsoft Windows* platform.
- **Remote simulation**, in which Programmer Studio runs on a Windows host and communicates over a network with a subsystem containing Network Processor simulator. Network Processor simulator can run either on Windows or Linux platform.
- **Hardware**, in which Programmer Studio runs on a Windows host and communicates over a network with a subsystem containing actual Network Processors.

2.1.3 Getting Help

You can get help about Programmer Studio and the Network Processors in several ways:

- In the **Project Workspace** window (see Figure 2.1), click the **InfoView** tab. This give you access to documentation installed along with the Software Development Kit (SDK). See Section 2.4.3.
- On the Web, go to www.netronome.com to get more information about Netronome products.

2.1.4 Programmer Studio Revision Information

To determine the revision:

- On the **Help** menu, click **About Programmer Studio**.

The **About Programmer Studio** information box appears displaying the revision of your Programmer Studio.

2.2 About the Graphical User Interface (GUI)

The Programmer Studio GUI (Figure 2.1) 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 2.2.4).

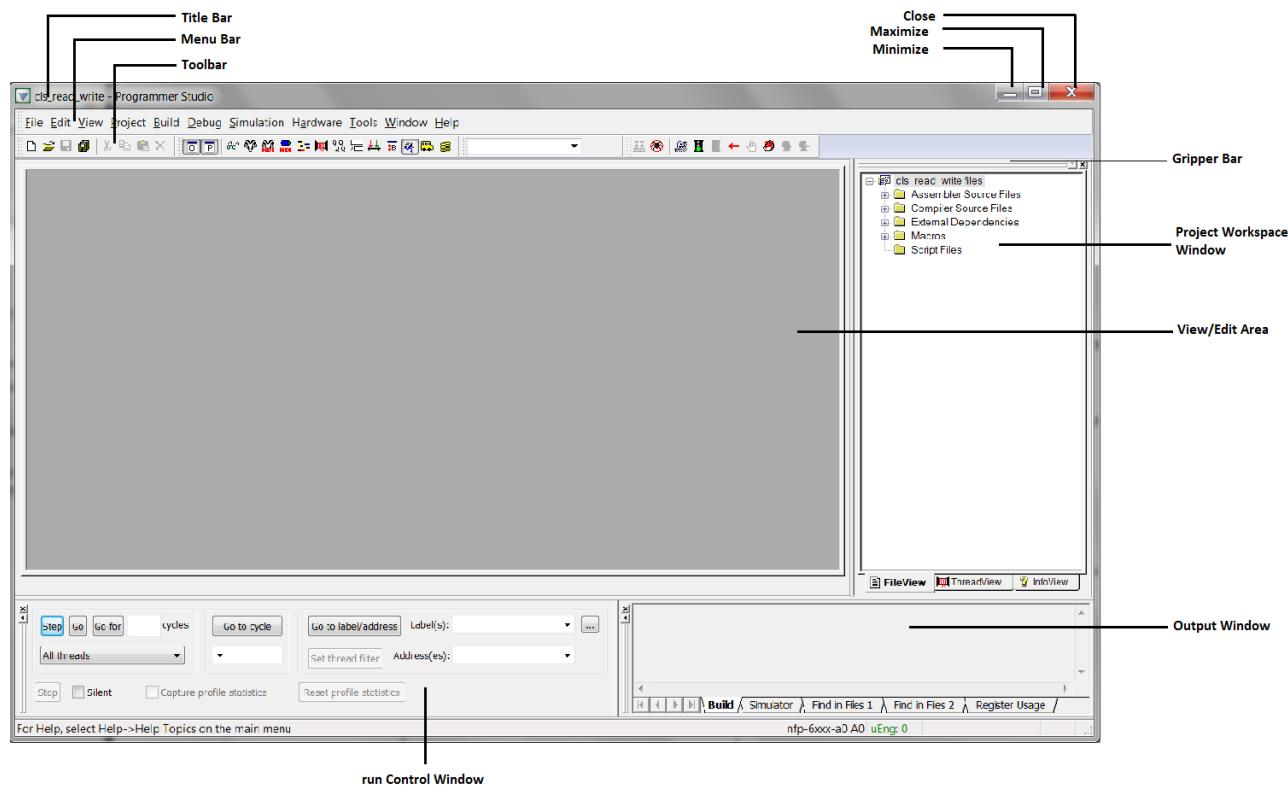


Figure 2.1. The Programmer Studio GUI

2.2.1 About Windows, Toolbars, and Menus

Dockable windows contain controls and data. You can attach them to a location on the Programmer Studio main window or you can float them over the main window. All toolbars and menu bars are dockable (see Figure 2.2).

To float, or undock, a window or toolbar, double-click its gripper bar (see Figure 2.1). To restore it to its previously docked location, double-click its title bar. You can also drag the window to a new docking location.

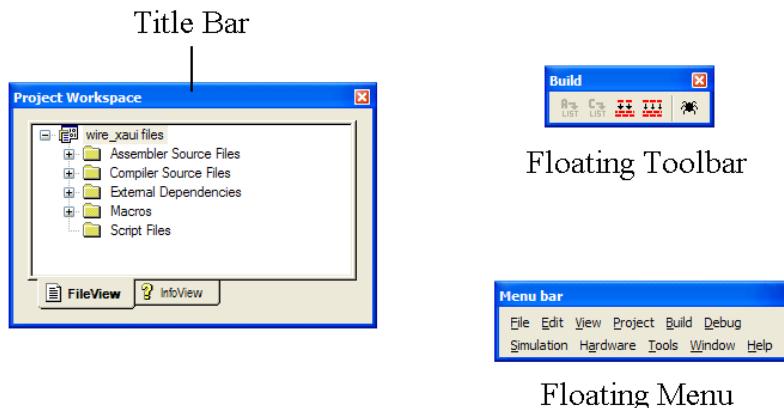


Figure 2.2. Floating Window, Tool Bar, and Menu Bar

2.2.2 Hiding and Showing Windows and Toolbar

From the **View** menu, you can toggle the visibility of the following windows in the Programmer Studio's GUI:

Toolbar If you are viewing a source file, or the view/edit area is empty, selecting **Toolbar** on the **View** menu displays the **Toolbars** dialog box. Here you can select to view, clear or 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 2.1). 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; or 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 Windows Output window displays the results of Find in Files, assembly and compile results, build results and other messages (see Figure 2.1).

Click the  button to show or hide this window.

IR Graphs window (only for P4 projects) displays parse state transitions and ingress/egress control flow in P4 application. For more information about **IR Graphs** see Section 2.4.4

Debug Windows Cling command Line - see Section 7.2.

Data Watch - see Section 2.12.11.

Memory Watch - see Section 2.12.12.

NBI Memory Watch - see Section 2.12.13.

Watch Scripts - see Section 2.12.14.

NBI PM Modification Pipeline - see Section 2.12.15.

NBI TM Packet Descriptor - see Section 2.12.16.

History - see Section 2.12.3.6 and Section 2.12.20.

Thread Status - see Section 2.12.23.

Queue Status - see Section 2.12.21.

Run Control - see Section 2.12.7.

Event List - see Section 2.12.22.

Breakpoints - see Section 2.12.8.

Packet Streaming Statistics - see Section 2.11.3.

P4 Inspectors (only for P4 projects) - see Section 2.12.24.

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 it using the same method used for windows and toolbars.

Status Bar

The status bar appears at the bottom on the Programmer Studio 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.

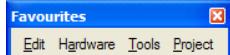
All the menus appear in the **Buttons** box.

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) 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 Programmer Studio default toolbars can be reset.

2.2.4 GUI Toolbar Configurations

Build versus Debug

Programmer Studio 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 Programmer Studio. These configurations persist from one Programmer Studio session to the next.

2.3 Programmer Studio Projects

Projects may be created in four ways: **standard**, **debug-only**, **P4** and **Managed C**.

A **standard** project consists of 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 Programmer Studio project file (.psproj).

A **debug-only** project is one in which the user specifies an externally built NFFW file for a specified 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 NFFW file builds using the Programmer Studio GUI.

A **P4** project consists of P4 source files, C source files, microcode source files, debug script files, and P4, Assembler, Compiler, and Linker settings used to build the microcode image files. This project configuration information is maintained in a Programmer Studio project file (.psproj).

A **Managed C** project consists of C source files and debug script files, and Assembler, Compiler, and Linker settings used to build the microcode image files. This project configuration information is maintained in a Programmer Studio project file (.psproj).

When you start Programmer Studio 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 2.3.5)
- Specify a default folder for creating and opening projects (see Section 2.3.6).

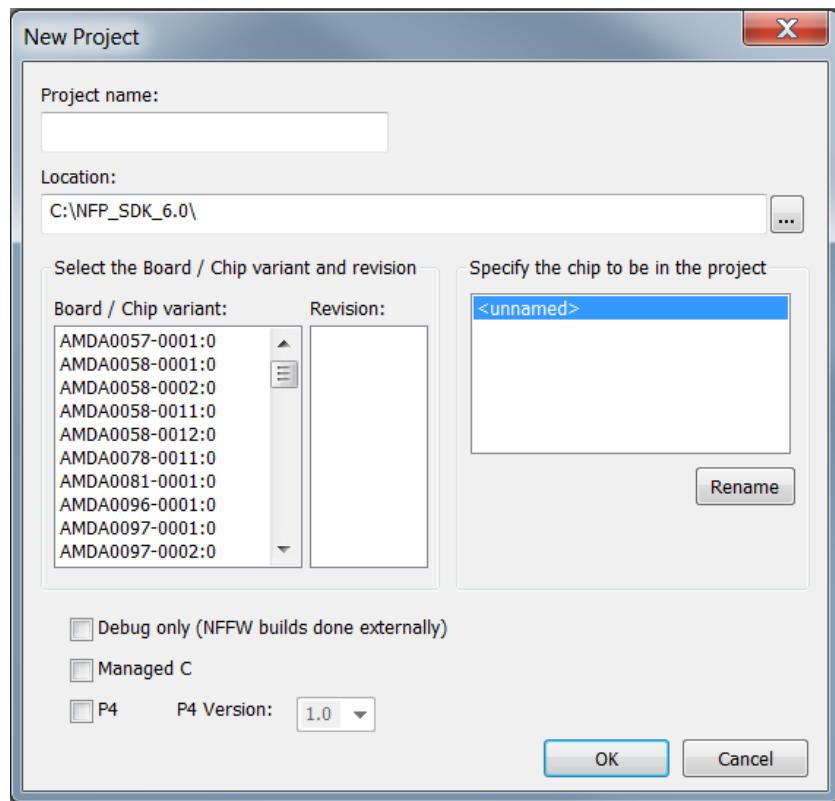
2.3.1 Creating a New Project

The processor type that you select when you create a new project, determines which Network Processor simulator is used for simulation. Programmer Studio will display only the GUI components that are relevant to the selected processor type. The processor family can be changed after a project is created, i.e. you can change your project from one processor type to a different processor type.

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, Programmer Studio creates it. You can browse to select the folder by clicking the button.

4. Select the chip family in the **Family/Variant** box.
5. Select the revision number (stepping) for the chip in the **Revision** box.
6. Specify the chip name to be used in the project. The chip name can be **<unnamed>**. To rename the chip to be used in the project, do the following:
 - Click **Rename**. The **Chip Name** dialog box appears. Type the chip's name and click **OK**.
7. If this project is to be “**debug-only**”, click the **Debug only** check box.

This specification tells Programmer Studio that the source files and list files to be used in this project will be built externally. Since the NFFW file contains the absolute paths of the source and list files, you must make sure to specify the correct locations for those files. If Programmer Studio 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** Programmer Studio buildable project. Neither can an existing **standard** project be converted to **Debug only**, although a NFFW file produced by a standard project can be used in a debug-only project.

8. If this project is to be “**P4**”, click the **P4** check box.

This specification tells Programmer Studio that the project is a P4 project. For P4 projects Programmer Studio includes P4 compiler and enables P4 source code editing and debugging capabilities in Programmer Studio. SDK now supports P4 version 1.0 and 1.1. P4 version can be selected from drop down which will be enabled when project is P4 project. By default P4 version is set to 1.0.



Caution

Once a project is created as **P4**, it cannot be converted to a **standard** project. Neither can an existing **standard** project be converted to **P4**.

9. If this project is to be “**Managed C**”, click the **Managed C** check box.

This specification tells Programmer Studio that the project is a managed C project. For more information regarding managed C projects Please refer to Appendix C of this document.



Caution

Once a project is created as **P4**, it cannot be converted to a **standard** project. Neither can an existing **standard** project be converted to **P4**.

10. When you have finished, click **OK** to create the project.

The project name you typed, by default, becomes a folder containing two files—project_name.psproj and project_name.psdbg (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.psproj.



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.



Note

If the chip instance is left unnamed, the chip name will appear as <unnamed> in all the Programmer Studio dialogs. However, in the Network Processor simulator console functions, the empty string “ ” is used wherever a chip instance name is expected.

2.3.1.1 Debug-only Projects

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

- There are no source files available, since Programmer Studio 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 NFFW files**.
- The Project Workspace **Fileview** tab does not display the tree elements **Assembler Source Files**, **Compiler Source Files**, or **Macros** since Programmer Studio does not associate source files with the **Debug-only** project.

Select **Specify Debug-only NFFW Files** from the **Project** menu. When you select the **Debug-only** option, the dialog box shown in Figure 2.3 is used to specify the NFFW file for each chip in the project.

If you try to start debugging without specifying a NFFW file, or if the NFFW or any list file identified in the NFFW 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 NFFW 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 NFFW file was created or if the build was done on a different system from the one where Programmer Studio is being run.

Similarly, if you execute a **Go To Source** command but the source file cannot be found in the location specified in the NFFW file, you will be prompted to browse to the correct location for the list file.

You also have the option to delete all file paths that were saved by Programmer Studio, as previously described. This may be required if you move the list and source files to different locations. To delete saved file paths, click the **Delete Paths** button.

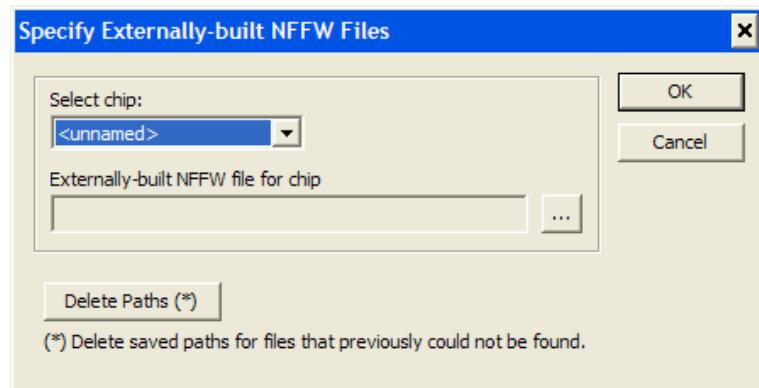


Figure 2.3. Specify Debug-only NFFW Files Dialog Box

2.3.1.2 P4 Projects

If you select the **P4** option when you create the project, there are some features added to Programmer Studio when you open the project. These features are unavailable in a **standard** project .

- The Project Workspace **Fileview** tab displays P4 specific tree elements **P4 Compiler Source Files** and **P4 User Config Files** . For more information about **Fileview** tab see Section 2.3.1.
- **P4 IR Graphs** button is added to the Programmer Studio Toolbar. For more information about **IR Graphs** see Section 2.4.4.

2.3.2 Opening a Project

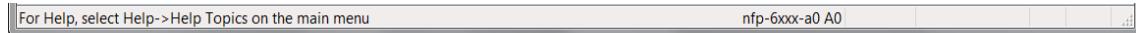
To open an existing standard project:

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

The **Open Project** dialog box appears.

2. Browse to the folder that contains the project file (*.psproj) 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 has not 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 you opened it.



Note

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

2.3.4 Saving Copies of a Project

A copy of a project can also be saved. To do this:

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

The **Project Save As** dialog box appears.

2. Browse to the folder where you want to save the project.
3. Type the new name of the project in the **File name** box.
4. Click **Save**.

If no project is opened, the **Save Project As** selection is unavailable.

This saves all project configuration information, such as debug settings, as a new project file. The existing project will be closed and the new project will be opened.

2.3.5 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 will be asked if you want to save these changes.

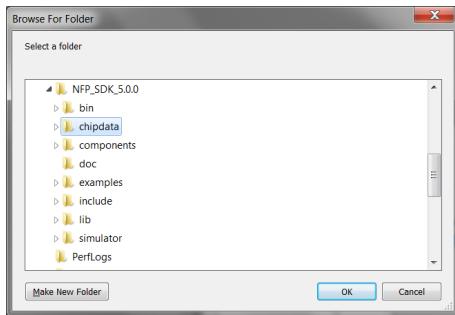
- Click **Yes** to save the file and close it, or
- Click **No** to close it without saving any changes, or
- Click **Cancel** to abort closing the project.

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

2.3.6 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 **New Project** is selected from the **File** menu.
- If **Open Project** is selected from the **File** menu.
- If no project is open, a new file is created and **Save As** is selected from the **File** menu.
- If no project is open and **Open** is selected from the **File** menu.



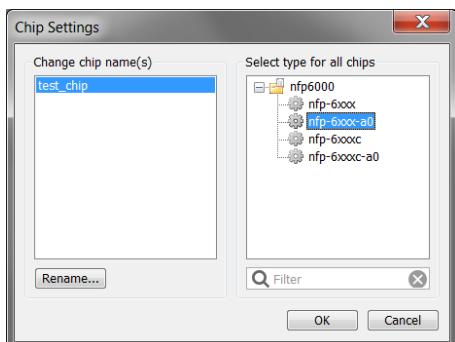
2.3.7 Changing Chip Type in Project

Programmer Studio gives you the ability to change the chip type after a project is created. The chip types supported by the SDK are NFP6xxx, NFP6xxxc, NFP4xxx and NFP4xxxc. To change the chip type for the project:

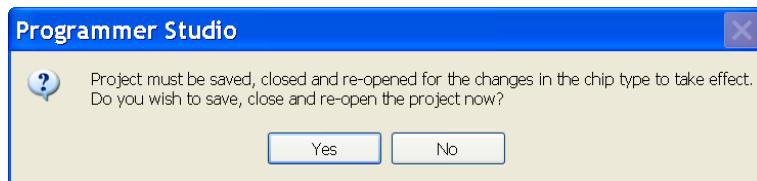
1. Open **Project** menu, click **Chip Settings**

The **Chip Settings** dialog box appears.

2. Select the chip type in the **Select type for all chips** box.
3. Click **OK**.



For the changes in the chip type to take effect, Programmer Studio asks if you would like to save the settings by saving, closing and re-opening the project. Selecting **Yes** will save, close and re-open the project. Selecting **No** will discard the changes in the chip type.



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 Programmer Studio, only **InfoView** is visible.
- When a project is open, **FileView** become visible.
- When you start debugging, **ThreadView** become visible.

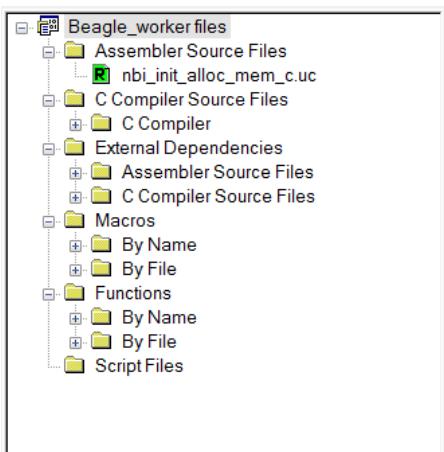
To toggle the visibility of the Project Workspace:

- On the **View** menu, select or clear **Project Workspace**, or

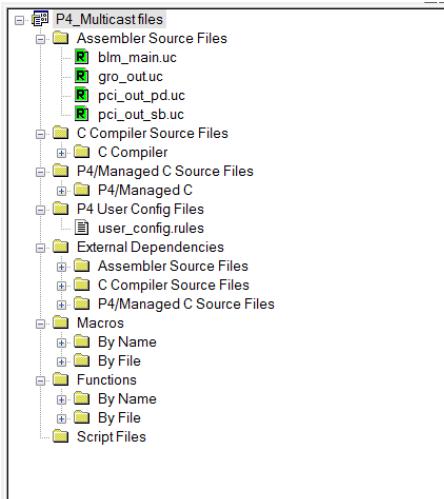
Click the button on the **View** toolbar.

2.4.1 FileView

FileView contains a tree listing your project files. The top-level item in the tree is labeled <projectname> files. Depend on the project type (P4 or standard) **FileView** contians different second-level folders



There are six second-level folders for standard projects:



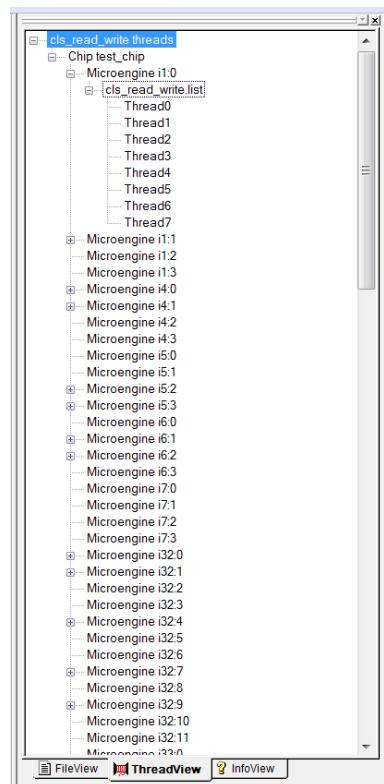
There are eight second-level folders for P4 projects:

- **Assembler Source Files**, which expands to an alphabetical list of all project Assembler source files.
- **C Compiler Source Files**, which expands to an alphabetical list of all project Compiler source files.
- **P4 Compiler Source Files**, which expands to an alphabetical list of all project P4 Compiler source files.
- **P4 User Config Files**, which expands to an alphabetical list of all project P4 user configuration files such as rules files.
- **External Dependencies**, which expands to a listing of files discovered during a dependency check and which reside in either a user-defined path or a standard include path.
- **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**.
- **Functions**, which expands to list the functions that are defined in the project's source files. This folder expands to:

- **Functions by name**, and
- **Functions by file**.
- **Script Files**, which expands to an alphabetical list of all debugging script files.

2.4.2 ThreadView

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



The top-level item in the tree is labeled <project-name> threads. There is a second-level item in the project. Each item expands to list the Microengines in the chip. Microengines are implemented in different Island. Depends on the Island type, an Island can contain either 4 or 12 Microengines. In a typical chip, there are 14 Islands, 1, 4, 5, 6, 7, 32, 33, 34, 35, 36, 37, 38, 48 and 49, with a maximum of 120 Microengines total.

Programmer Studio displays each Microengine name as **Microengine i:n** where **i** represents the island number (1 or 4 ... 49) and **n** is the number within the island.

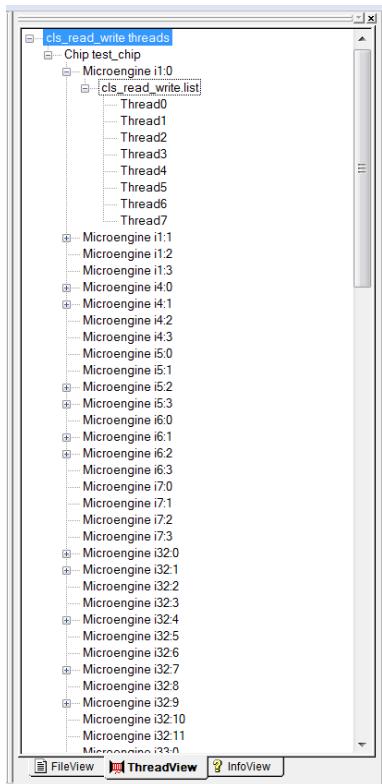
Each Microengine item expands to display the associated .list files, and then list the four or eight threads in a Microengine, but only if the threads are active in the microcode. If a Microengine is not loaded with code, no "+" sign appears to the left of the icon and it can therefore not be expanded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

By default, a chip's threads are named **Thread 0** through **Thread 7**.

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, Microengines i1:2, i1:3, i4:2, etc. cannot be expanded because they contain no microcode.

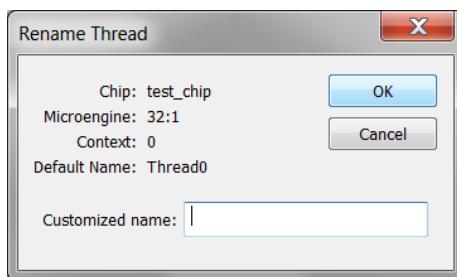
To collapse a chip's tree, double-click the chip name.

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**.

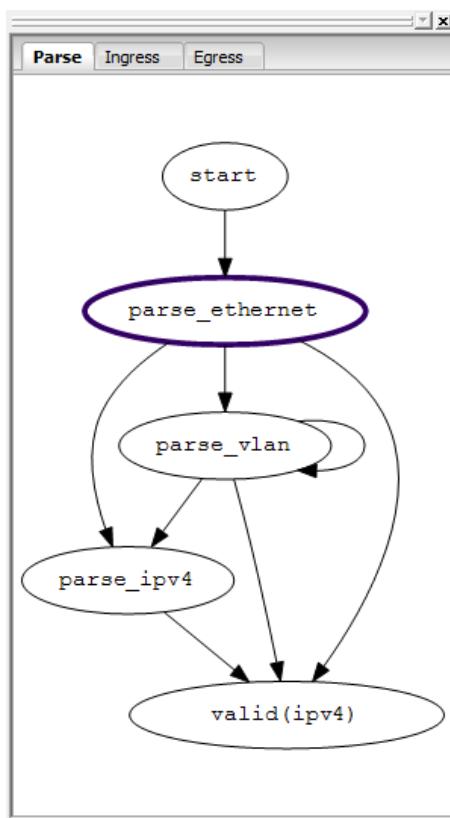
2.4.3 InfoView

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* Reader*, which then displays the document. A copy of Acrobat* Reader* is freely available from Adobe Systems Incorporated.

2.4.4 IR Graphs

This window is only available for P4 projects. To open **IR Graphs** window, click the  button on the menu button. Or select **IR Graphs** from View toolbar menu.



IR Graphs provides access to three SVG graphs that are generated when the **Generate IR** action is clicked.

1. **Parse:** P4 parse state nodes.
2. **Ingress:** Ingress control flow nodes.
3. **Egress:** Egress control flow nodes.

The text of conditional nodes are represented by the condition expression or part thereof. Some control transitions are labelled: notably omitted from this is the table result based transition.

Hovering with the mouse over any node will highlight it with a thicker dark blue border. When highlighted, clicking on the node will navigate to appropriate P4 source definition. Please note that **exit_control_flow** nodes don't have a corresponding source location and will not navigate to anywhere.

By default graphs will automatically fit to the available space in the window. If there are too many nodes in graph you may right click and turn off the **Fit to Page** checkbox for manual dragging/scrolling and zooming. To scroll, hold down the control key and use the mouse roller.

2.5 Working with Files

Programmer Studio 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)
- Find in Files (see Section 2.5.12)
- Fonts and Syntax colors (see Section 2.5.13)
- Macros (see Section 2.5.14).
- Functionss (see Section 2.5.15).

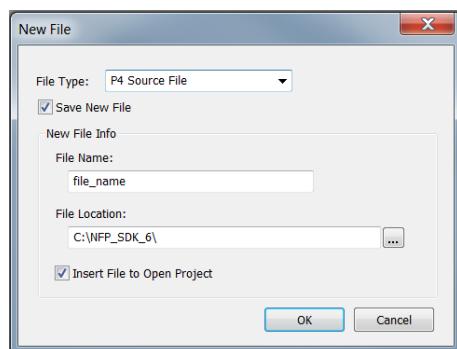
2.5.1 Creating New Files

To create a new file:

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

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

The New File dialog box appears.



2. Select which type of file you want to create from the list.

3. 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.

If the new file needs to be saved after it is created, check the **Save New File** check box. This enables the **New File Info** section.

1. Name the file that needs to be saved in the **File Name:** edit box.
2. Specify location of the new file by either clicking the  button to browse for folder. Or type the location in the **File Location:** edit box.
3. If the new file needs to be added to the open project after it is saved, check the **Insert File to Open Project** check box.



Note

Insert File to Open Project check box is enabled only when there is a project open in Programmer Studio.

2.5.2 Opening Files

To open a 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 have 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 also script files have the file type .c.

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 do not 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 Programmer Studio. The **Windows** menu deals mostly with the text file windows in Programmer Studio.

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

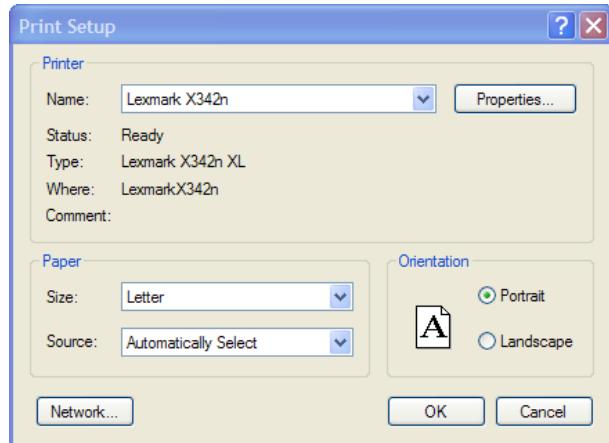
Click the  button on the minimized window that you want to restore to its previous view.

2.5.8 Printing Files

2.5.8.1 Setting Up the Printer

1. On the **File** menu, click **Print 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, C Compiler source files, P4 Compiler source files and script files into a project.

To do this:

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

Insert Assembler Source File(s), or

Insert C Compiler Source File(s), or

Insert P4 Sandbox C/Managed C Source File(s), or

Insert P4 Source File(s), or

Insert P4 User Config File(s), or

Insert Script File(s), whichever is appropriate.

The corresponding dialog box appears.

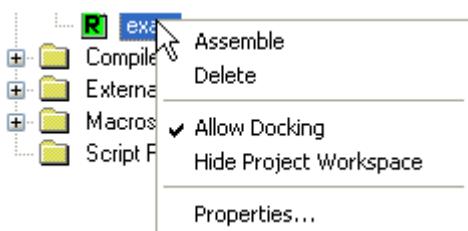
2. Browse to the desired folder and select one or more files to be inserted.
3. Click **Insert**.

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 **FileView** 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 Programmer Studio editor is similar to standard text editors. See Table A.3 for a list of Edit controls.

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

2.5.10.1 Tab Configuration

To configure tab settings, select **Options** from the **Tools** menu. The dialog box shown in Figure 2.4 appears. You can select the file type – Network Flow Assembler source, Network Flow C Compiler source, Network Flow Simulator script or Default – for which the tab settings will have effect. For the selected file type, you also specify:

- 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 nonwhitespace character in the previous line.

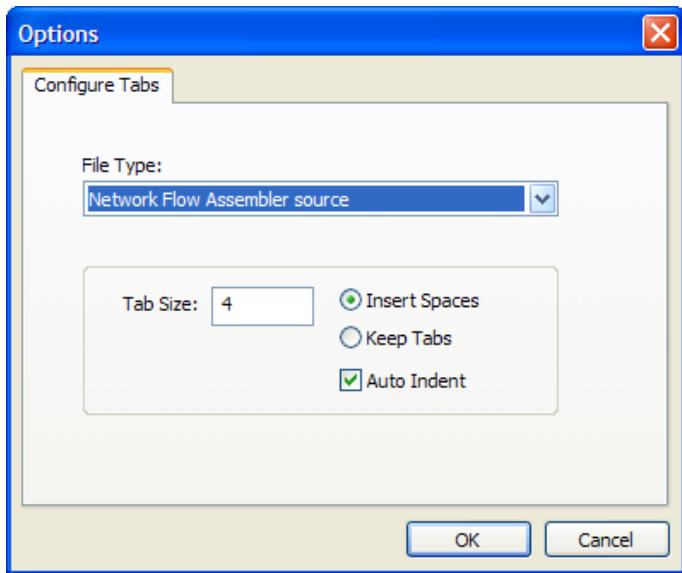


Figure 2.4. Configure Tabs Dialog Box

2.5.10.2 Go To Line

Programmer Studio allows for navigating directly to a specified line within an opened document or thread window. If you select **Go To** from the **Edit** menu, the dialog shown in Figure 2.5 appears. Enter the desired line number and click **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.

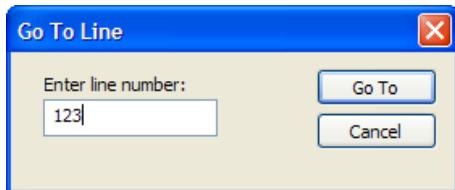
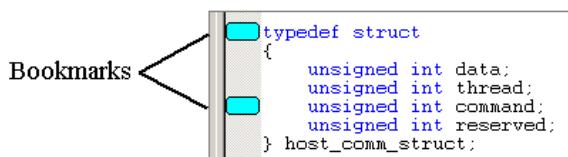


Figure 2.5. Go To Line Dialog Box

2.5.11 Bookmarks and Errors/Tags

You can mark your place in a file using bookmarks. Table A.4 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 A.6 for a list of Bookmark and Error/Tag controls.

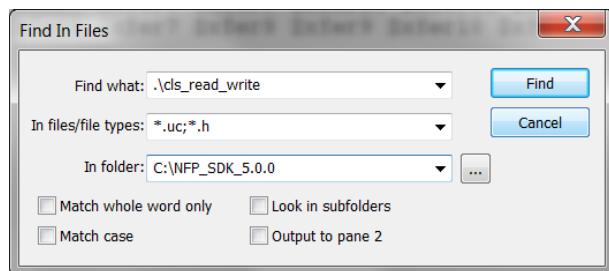
2.5.12 Find In Files

Programmer Studio 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 a file extension of “txt”.



Note

Programmer Studio also supports the following file types for drag-and-drop operations: .mer, .ppm, .xsd, .pmd and .log.

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**.

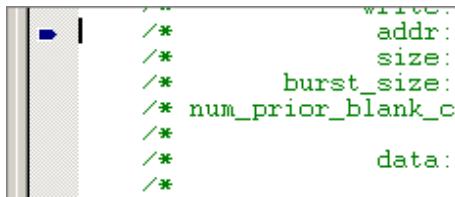
The results of the search are displayed in the **Find In Files 1** (or 2) tab of the **Output** window. For each occurrence of the search string that is found, the file name, line number, and line of text are displayed.



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.
- 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 is not already open, it is automatically opened.



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

2.5.13 Fonts and Syntax Coloring

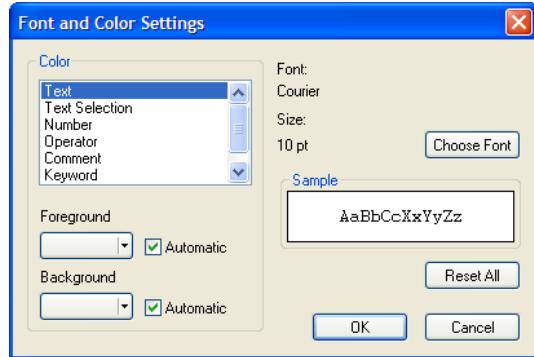
Source files, that is those files with extensions of .uc, .c, .p4 or .h, appear with syntax coloring of keywords and comments. Keywords are words that are reserved by the Assembler and Compiler are used in specific context. For example, 'alu_shf' is reserved because it is an Assembler instruction.

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 Programmer Studio executable.

2.5.14 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 in the **By File** folder according to the file that they are defined in.

Programmer Studio:

- 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.
 - 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.

2.5.15 Functions

The **FileView** tab in the **Project Workspace** has a Function folder that contains the functions that are defined in the Micro-C project's source files.

The functions are:

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

Programmer Studio:

- 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.
 - You update dependencies, by selecting **Update Dependencies** from the **Project** menu.

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

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



Note

Go To Function Definition is not available for local functions. This shortcut menu only available for functions that are defined in header files.

Go To Function Definition is available only for Micro-C projects.

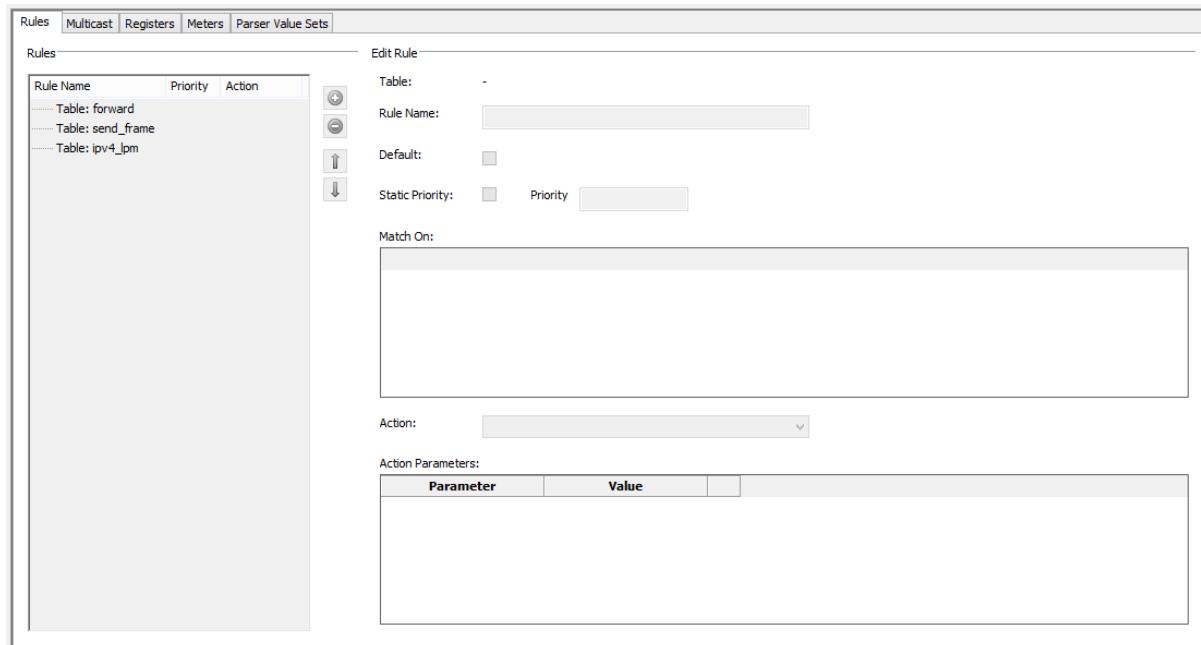
1. Right-click the function reference.

2. Click **Go To Function Definition** on the shortcut menu.

2.5.16 P4 User Config File

A P4 user config file can be edited by double clicking on its name under the **User Config** folder in the **FileView** tab in the **Project Workspace**.

The P4 user config file editor will open:



The P4 user config file editor allows you to configure the following for a P4 project:

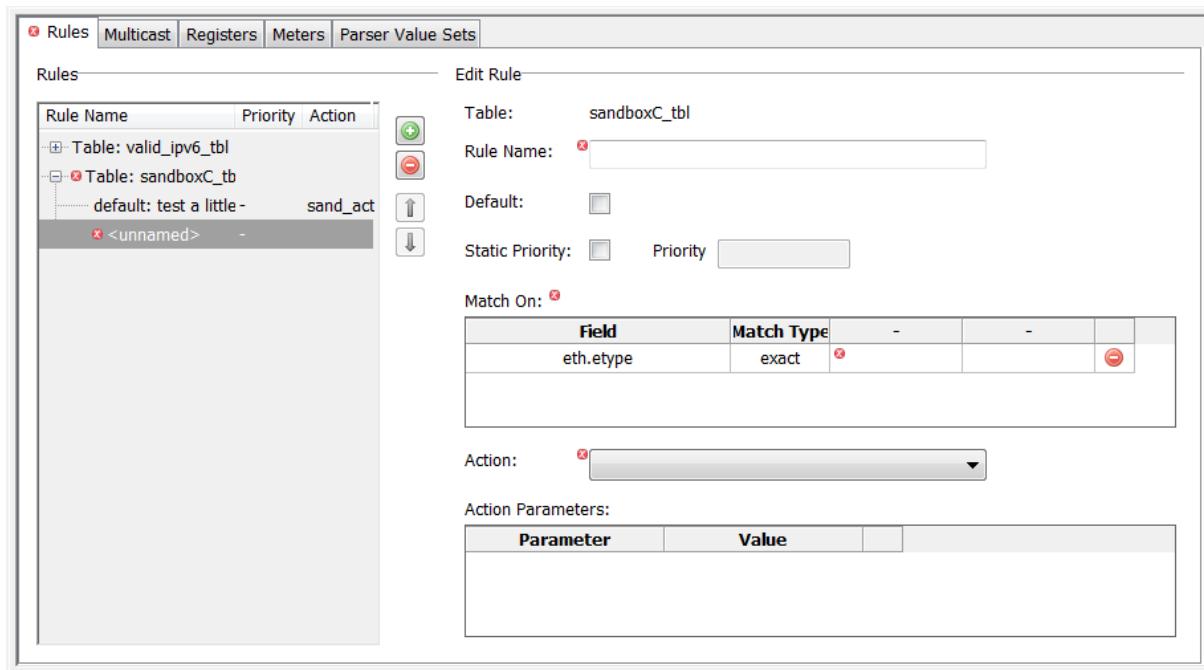
- Table rules (see Section 2.5.16.1)
- Multicast groups (see Section 2.5.16.2)
- Register initial values (see Section 2.5.16.3)
- Meter initial values (see Section 2.5.16.4)
- Parser value sets (see Section 2.5.16.5)

Look out for error icons and tooltips which will hint at input errors the editor.

2.5.16.1 Configure Table Rules

To add a rule, select a table then click the add button.

The newly added rule will be automatically selected, with all inputs empty.



Use the up and down buttons to change the order of the rules. Note that if the table has a default rule, the default rule will always be at the top.

Use the delete button to delete a rule.

The following options are configurable:

- **Rule Name**

Used to identify the rule.

- **Default**

If this option is checked this rule will be the default rule. If the table already has a default rule, the current default rule will be changed to non-default first.

- **Priority**

Priority to assign to this rule.

- **Match Values**

Match values for the rule. If this is a default rule there will not be any match values.

- **Action**

The action to be executed if the rule is matched. The combobox shows a list of actions that is allowed for this table.

- **Action Parameters**

The parameter values used to execute the action selected in the **Action** combobox.

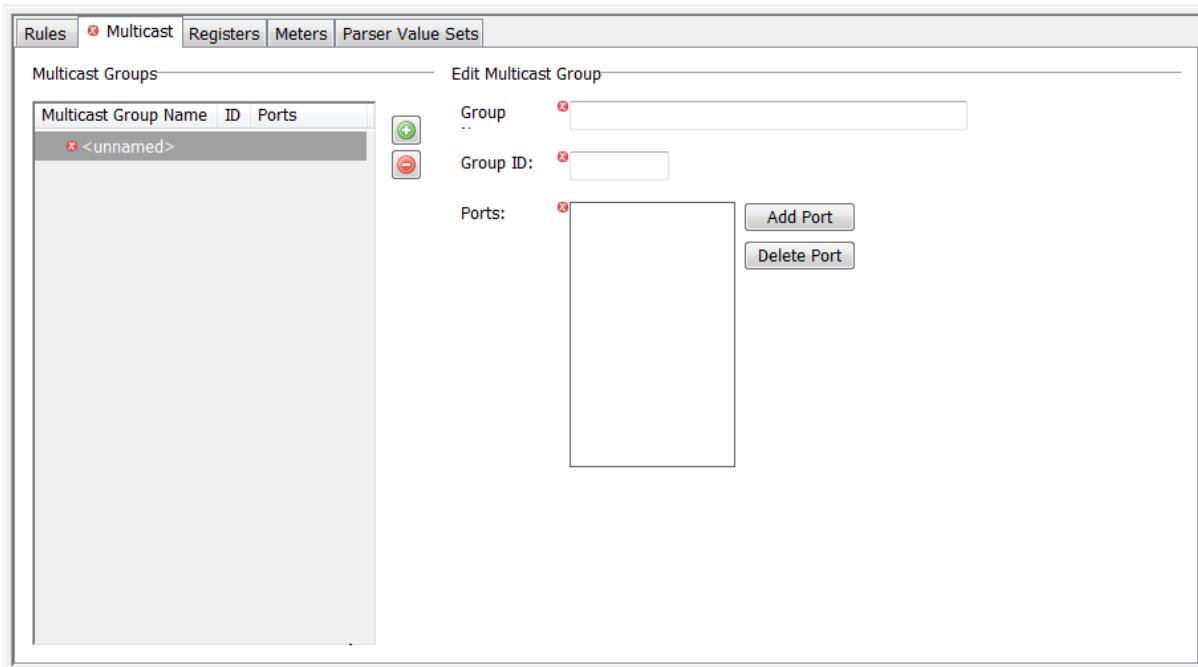
2.5.16.2 Configure Multicast Groups

Select the **Multicast** page to configure multicast groups.

Use the add button  to create a new multicast group.

The newly added multicast group will be automatically selected, with all inputs empty.

Use the delete button  to delete a multicast group.



The following options are configurable:

- **Group Name**

Used to identify the multicast group.

- **Group ID**

The multicast group ID.

- **Ports**

This is a list of at least one port forming this multicast group. Add or delete ports here. Note that you cannot add more ports than the configured multicast group size.

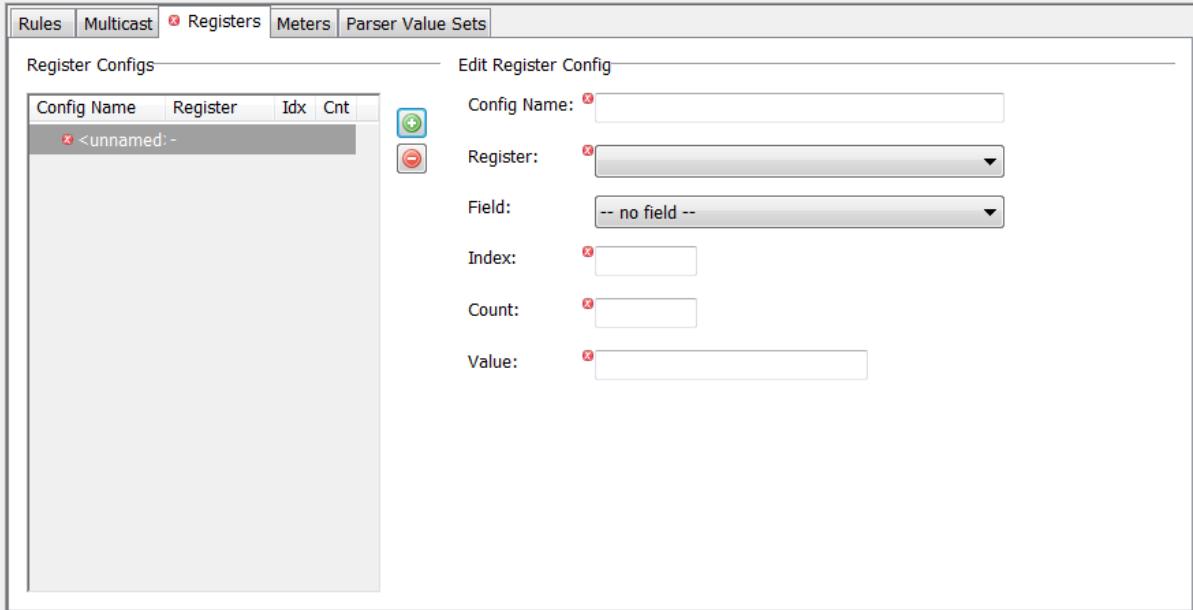
2.5.16.3 Configure Register Initial Values

Select the **Registers** page to configure register initial values.

Use the add button  to create a new register config.

The newly added register config will be automatically selected, with all inputs empty.

Use the delete button  to delete a register config.



The screenshot shows the 'Registers' tab selected in the top navigation bar. On the left, there is a list titled 'Register Configs' with one item: '<unnamed>'. To the right, there is an 'Edit Register Config' dialog box. The dialog contains the following fields:

- Config Name:
- Register: 
- Field:
- Index:
- Count:
- Value:

The following options are configurable:

- **Register Name**

Used to identify the register config.

- **Register**

The register to configure.

- **Index**

The index of the register to configure.

- **Field**

The field of the register to configure.

- **Count**

The count of registers from the index to configure.

- **Value**

The value to configure the register with. This can either be a single value to configure all fields, or specified per field. Change this option using the radio buttons.

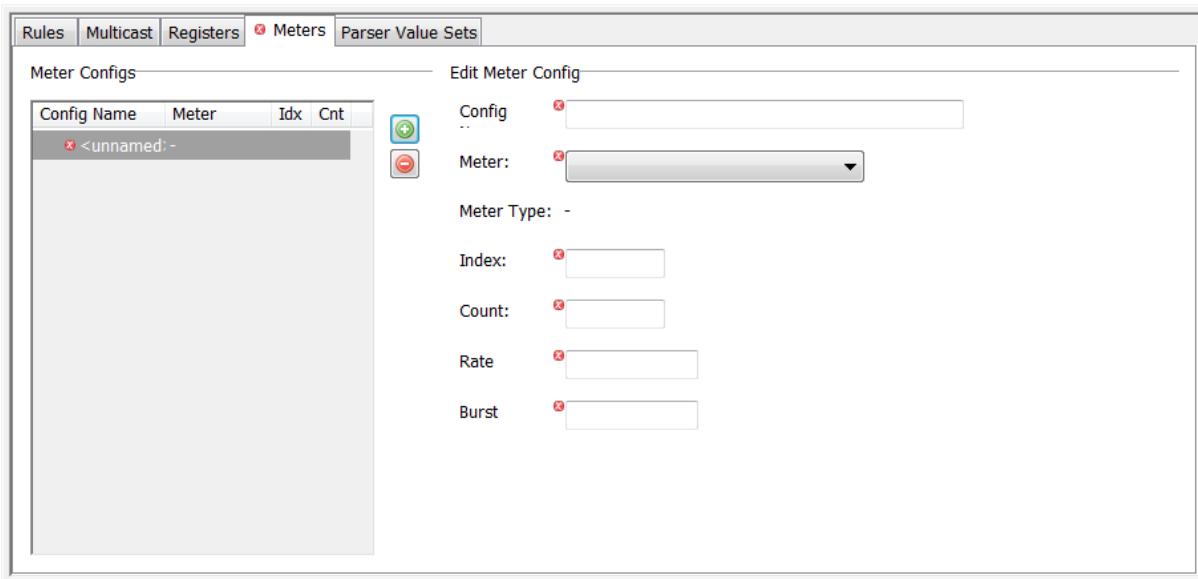
2.5.16.4 Configure Meter Initial Values

Select the **Meters** page to configure meter initial values.

Use the add button  to create a new meter config.

The newly added meter config will be automatically selected, with all inputs empty.

Use the delete button  to delete a meter config.



The following options are configurable:

- **Meter Name**

Used to identify the meter config.

- **Meter**

The meter to configure.

- **Index**

The index of the meter to configure.

- **Count**

The count of meters from the index to configure.

- **Rate**

The rate value to configure the meter to.

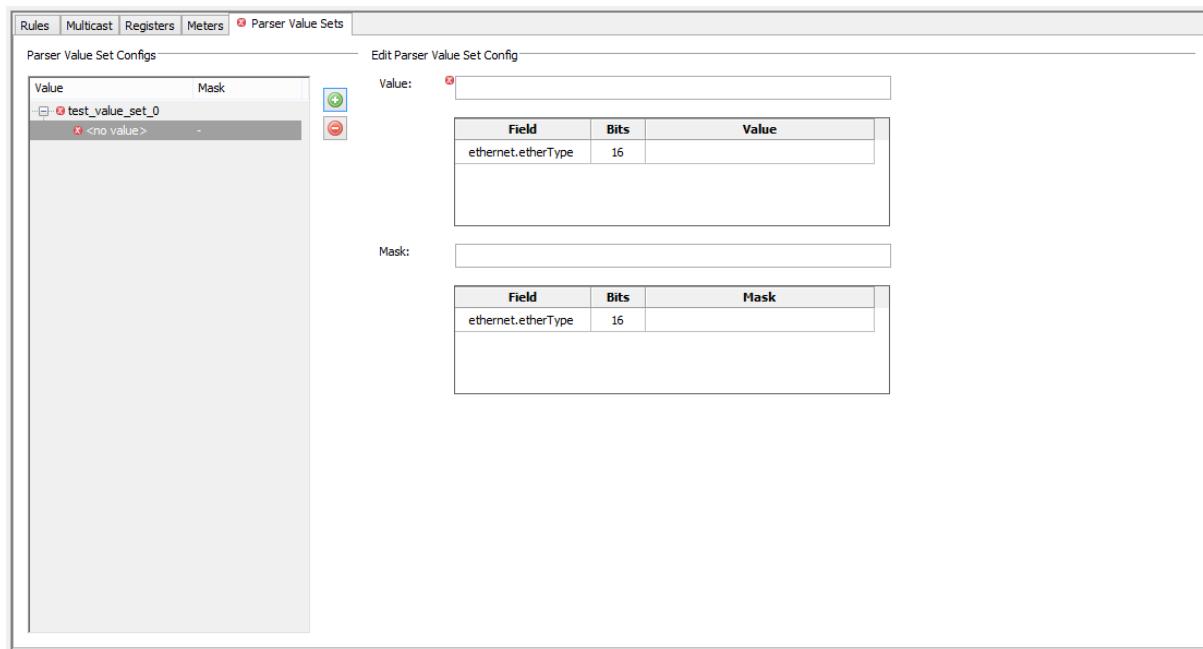
- **Burst**

The burst value to configure the meter to.

2.5.16.5 Configure Parser Value Sets

To add a parser value set value, select a parser value set then click the button.

The newly added value will be automatically selected with all inputs empty.



Use the delete button to delete the rule.

The following options are configurable:

- **Value**

The parser value set value, you can also configure individual fields in the grid below.

- **Mask**

Optional mask, you can also configure individual fields in the grid below.

2.5.16.6 User Config File Format

The p4 user config file (.p4cfg) is basically a json file with the following format:

```
{  
    "tables": {  
    },  
    "multicast": {  
    },  
    "registers": {  
    },  
    "meters": {  
    },
```

```

    "parser_value_sets": {
}
}
```

The format for each section is given below.

2.5.16.6.1 Tables

The tables section is where you configure the rules for each table. The basic format is as follows:

```

"tables": {
    "(1. Table Name)": {
        "rules": [
            (rule entries go here)
        ],
        "default_rule": {
            (default rule entry goes here)
        }
    }
}
```

A rule entry has the following format:

```

{
    "name": "(2. Rule Name)",
    "action": {
        "type": "(3. Action Name)",
        "data": {
            "(4. Action Parameter Name)": {
                "value": "(5. Action Parameter Value)"
            }
        }
    },
    "match": {
        "(6. Match Field Name)": {
            "value": "(7. Match Field Value)",
            "mask": "(8. Optional Match Field Mask)"
        }
    },
    "priority": "(9. Priority)"
}
```

Refer to the following example P4 code snippets in the format explanation below:

```

table ipv4_lpm {
    reads {
        ipv4.dstAddr : lpm;
    }
    actions {
        set_nhop;
        _drop;
    }
}
```

```
    size: 1024;  
}
```

```
action set_nhop(nhop_ipv4, port) {  
    modify_field(routing_metadata.nhop_ipv4, nhop_ipv4);  
    modify_field(standard_metadata.egress_spec, port);  
    add_to_field(ipv4.ttl, -1);  
    count(fwd_counter, 0);  
}
```

1. Table Name

The name of the table you want to add rules to. In the P4 above we will add rules to the table *ipv4_lpm*.

2. Rule Name

A unique name for the rule (unique to all the rules in this table). We will add a rule *rule_1* to the table *ipv4_lpm*.

```
"tables": {  
    "ipv4_lpm": {  
        "rules": [  
            {  
                "name": "rule_1",  
                ...  
            }  
        ]  
    }  
}
```

Note that the order in which you add rules to the table will be the priority of the rules, with the first rule you add the highest priority and the last rule you add the lowest priority.

The *default_rule* key is optional, if you don't want to add a default rule you can omit it.

3. Action Name

The action name is the name of any of the allowed actions on this table. In the P4 code example above the allowed actions are:

```
actions {  
    set_nhop;  
    _drop;  
}
```

We will select the action *set_nhop* for the rule *rule_1*.

```
"action": {  
    "type": "set_nhop",  
    ...  
}
```

```
}
```

4. Action Parameter Name

The name of the action parameter you want to set the value for. In the P4 code example above the action parameters for action *set_nhop* is *nhop_ipv4* and *port*.

```
action set_nhop(nhop_ipv4, port)
```

We will set the action parameter values for the rule *rule_1* in the example as follows:

```
"action": {
    "type": "set_nhop",
    "data": {
        "nhop_ipv4": {
            "value": "10.0.0.1",
            "port": "6556"
        }
    }
}
```

5. Action Parameter Value

The value for the action parameter. Valid values are:

```
- integer, ipv6, ipv4, MAC address, 'drop'  
- physical port: p[N] where N = (0-127)  
- vf: v0.[N] where N = 0-31  
- multicast group: mg[N] where N = 0-15
```

Note that integer values can be integers or strings in the json, while all other values must be strings ("string value")

6. Match Field Name

The name of the match field you want to set the value and optionally mask for. In the P4 code example above the match field for table *ipv4_lpm* is *ipv4.dstAddr*. The match type is *lpm* so you only have to specify a value and not a mask. A mask is only valid if the match type is *ternary* .

```
reads {
    ipv4.dstAddr : lpm;
}
```

Note that a match field is optional, if you want a field to act as a wildcard (always match on that field), then you can simply omit it.

If the rule is default (*default_rule* in the json) then you should not have a *match* key in the rule entry json. Similarly, if the rule is non-default and you haven't specified any matches then you should either add the rule under *default_rule*, or not have any other rules on the table.

We will set the match field value for the rule *rule_1* in the example as follows:

```
"match": {  
    "ipv4.dstAddr": {  
        "value": "192.168.0.0/16"  
    }  
}
```

7. Match Field Value

The value for the match field. Valid values for each match type are (match types are in bold):

```
- ternary, lpm, exact:  
- integer, ipv6, ipv4, MAC address, 'drop'  
- physical port: p[N] where N = (0-127)  
- vf: v0.[N] where N = 0-31  
- multicast group: mg[N] where N = 0-15  
- range:  
- [N]->[M] where N,M = 0-65535 and N less than or equals M  
- header_valid:  
- 'valid', 'invalid'
```

Note that integer values can be integers or strings in the json, while all other values must be strings ("string value")

8. Optional Match Field Mask

The mask for the match field. Only integer values are valid. You can only specify a mask for a ternary match.

9. Priority

The explicit priority to assign to the rule.

2.5.16.6.2 Multicast

The multicast section is where you configure multicast groups. The basic format is as follows:

```
"multicast": {  
    "(1. Group Name)": {  
        "group_id": (2. Group ID),  
        "ports": [  
            (3. Port),  
            (add more ports here)  
        ]  
    }  
}
```

Below is an example multicast group configuration:

```
"multicast": {  
    "mg_group_1": {  
        "group_id": 1,  
        "ports": [  
            "100",  
            "200"  
        ]  
    }  
}
```

1. Group Name

A unique name for the multicast group.

2. Group ID

A unique multicast group id. The id must be an integer between 0 and the configured group count (excluded).

3. Port

The port is an integer port value forming part of this multicast group. The maximum number of added ports should not exceed the configured group size.

2.5.16.6.3 Registers

The registers section is where you configure register initial values. The basic format is as follows:

```
"registers": {  
    "configs": [  
        {  
            "name": "(1. Register Config Name)",  
            "register": "(2. Register and Field)",  
            "index": (3. Index),  
            "count": (4. Count),  
            "value": (5. Value)  
        },  
        (add more register configs here)  
    ]  
}
```

Refer to the following example P4 code snippets in the format explanation below:

```
header_type my_register_metadata_t {  
    fields {  
        field_1 : 32;  
        field_2 : 32;  
    }  
}
```

```
register my_register {
    layout : my_register_metadata_t;
    instance_count : 1;
}
```

1. Register Config Name

A unique name for the register config.

2. Register and Field

The register and/or field name to configure. To configure a specific field in a register, use the format *register_name.field_name*. To configure the whole register to a value simply omit the *.field_name*.

For example, if we want to configure a single field in the example register above, we do:

```
{
    "name": "my_register_config_field",
    "register": "my_register.field_1",
    ...
}
```

However, if we want to configure the whole register, we do:

```
{
    "name": "my_register_config",
    "register": "my_register",
    ...
}
```

3. Index

The index of the register instance to configure. This should be a positive integer value less than the register's instance count (or instance count * table size for direct registers).

4. Count

The number of register instances from *index* to configure. This should be a positive integer value, greater than 0 and *index + count* should be less than the register's instance count (or instance count * table size for direct registers).

5. Value

An integer value to initialize the register to. The bit length of this value should not be greater than the register's width.

2.5.16.6.4 Meters

The meters section is where you configure meter initial values. The basic format is as follows:

```
"meters": {  
    "configs": [  
        {  
            "name": "(1. Meter Config Name)",  
            "meter": "(2. Meter)",  
            "index": (3. Index),  
            "count": (4. Count),  
            "burst_k": (5. Burst),  
            "rate_k": (6. Rate)  
        },  
        (add more meter configs here)  
    ]  
}
```

Refer to the following example P4 code snippets in the format explanation below:

```
header_type drop_stats_t {  
    fields {  
        dropped : 32;  
    }  
}
```

```
metadata drop_stats_t drop_stats;
```

```
meter drop_meter {  
    type: packets;  
    result: drop_stats.dropped;  
    instance_count: 1;  
}
```

1. Meter Config Name

A unique name for the meter config.

2. Meter

The name of the meter you want to configure.

For example, if we want to configure the example meter above, we do:

```
{  
    "name": "drop_meter_config",  
    "meter": "drop_meter",  
    ...  
}
```

3. Index

The index of the meter instance to configure. This should be a positive integer value less than the meter's instance count (or instance count * table size for direct meters).

4. Count

The number of meter instances from *index* to configure. This should be a positive integer value, greater than 0 and *index* + *count* should be less than the meter's instance count (or instance count * table size for direct meters).

5. Burst

The burst value to initialize the meter instance(s) to. It should be an integer value, and cannot be bigger than 32-bits.

6. Rate

The rate value to initialize the meter instance(s) to. It should be a float value.

2.5.16.6.5 Parser Value Sets

The parser_value_sets section is where you configure parser value sets. The basic format is as follows:

```
"parser_value_sets": {  
    "configs": [  
        {  
            "value_set": (1. Parser Value Set),  
            "entries": [  
                {  
                    "value": (2. Value),  
                    "mask": (3. Mask)  
                },  
                (add more values here)  
            ]  
        }  
    ]  
}
```

1. Parser Value Set

The parser value set name you want to configure.

2. Value

Integer value for the parser value set.

3. Mask

Optional integer mask for the parser value set.

2.6 The Assembler

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

- How root files and dependencies are determined (Section 2.6.1)
- How to make and change Assembler build settings (Section 2.6.2)
- How to invoke the Assembler (Section 2.6.3.1)
- How to handle assembly errors (Section 2.6.3.2).

Also refer to the following sections for information on:

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

For additional details on the Assembler refer to Chapter 3, the *NFP-6000 Network Flow Assembler System User's Guide* or to the *Netronome NFP-6000 Databook* for your network processor.



Note

For P4 projects, Programmer Studio sets all required Assembler settings automatically. It is highly recommended that users not to change any Assembler build settings for P4 projects .

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 a 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 Programmer Studio 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 Programmer Studio 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, Programmer Studio automatically inserts that source file into the project. Programmer Studio automatically performs a dependency update when a project is opened. When you insert a microcode file into a project, Programmer Studio 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 **General** tab (shown in Figure 2.6) provides build settings for the compiler and the assembler.

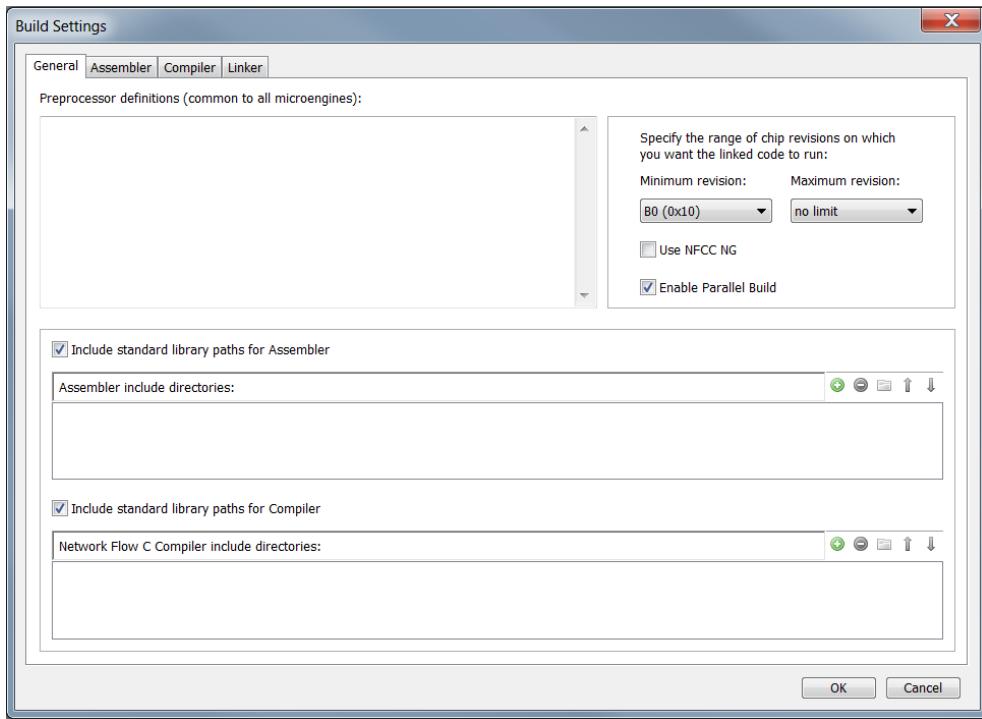


Figure 2.6. Build - General Tab

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 implies that an engine-specific preprocessor definition will override a general setting.

Preprocessor definitions (common to all microengines):
SIMULATION_BUILDAPPLICATION_NAME=Firewall



Note

A -D prefix is not necessary for the Programmer Studio declaration of preprocessor definitions, but is necessary for command line use of the Assembler.

Specifying Processor Revision Range

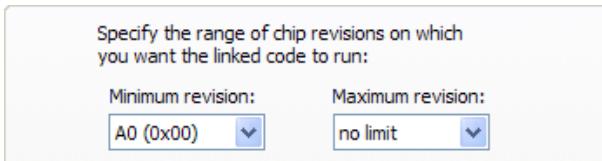
The network processors are available in different versions (steppings) with different features. You can specify a range of revisions for which you want your microcode assembled. Section 3.1.7 covers this topic in more detail.

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.

Use NFCC NG

This provides you with the option to use NFCC-NG instead of NFCC. NFCC-NG is experimental and may require changes to your code. Recommended for advanced users only. By default this option is disabled.

Use NFCC NG

Enable Parallel Build

This provides you with the option to enable or disable parallel build in PS. When enabled, PS will run multiple instances of the compiler (NFCC) if your application requires multiple list files to be created and compiled by the compiler. By default parallel build is enabled.

Enable Parallel Build

Include Standard Library Paths for Assembler

This provides you with the option of automatically including the SDK standard library paths for Assembler. Programmer Studio will set registry keys with the well known paths for the standard include libraries. For new projects, the box will be checked and the standard include directory paths will be added to the assembler settings.

Include standard library paths for Assembler

Include Standard Library Paths for Compiler

This provides you with the option of automatically including the SDK standard library paths for Compiler. Programmer Studio will set registry keys with the well known paths for the standard include libraries. For new projects, the box will be checked and the standard include directory paths will be added to the compiler settings.

Include standard library paths for Compiler

2.6.2.2 Specifying Additional Include Paths for Assembler

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.
- 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 Additional Include Paths for the Compilers

To specify the paths for the Compilers:

1. On the **Build** menu, click **Settings**.
2. Click the **General** tab.
3. To enter additional include paths for the Network Flow C Compiler explicit partitioning mode use the **Network Flow C Compiler include directories:** text box.

To specify these additional Compiler specific 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.
- 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

2.6.3 Specifying Assembler Build Settings

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 (see Figure 2.7).

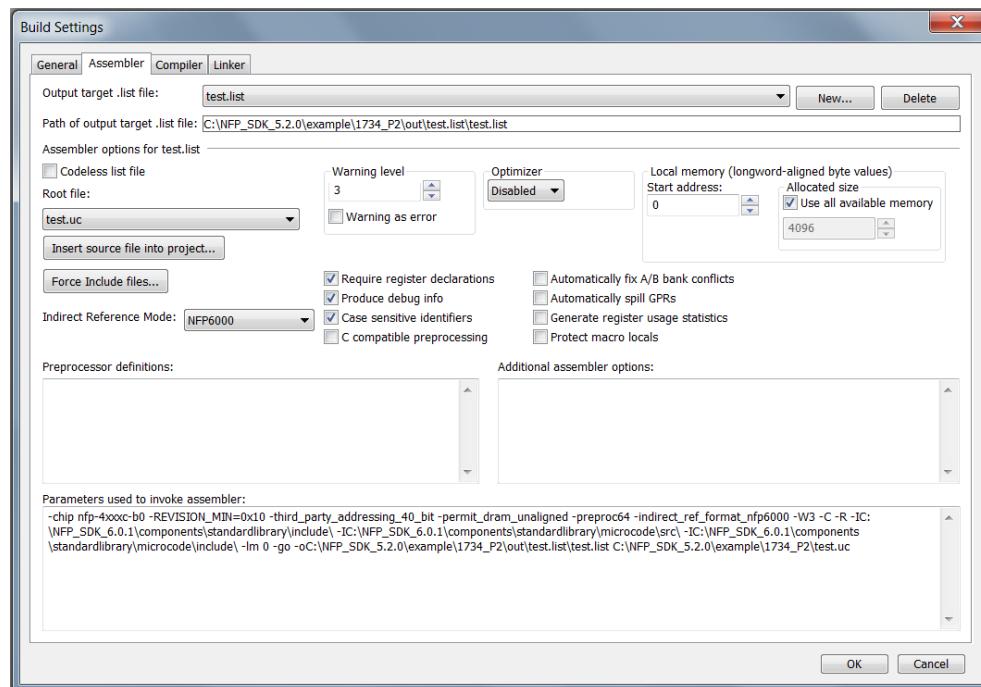


Figure 2.7. Build - Assembler Tab

Output to Target .list File

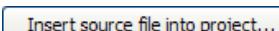
The **Output to target .list file** list allows you to select a .list file from the set of .list files that are currently defined in the project. All other controls on the page are updated according to which .list file you select in the list.

Path of target .list file

This control displays the path of the .list file that is currently selected in the **Output target .list file** pull-down **menu**.

Insert Source file into project

1. On the **Assembler** tab, click the **Insert source file into project....** button.



The Insert **Assembler Source Files** into Project dialog box appears.

2. Select a path for the .list file.
3. Type a filename.

You cannot insert a file that has already been inserted into the project.

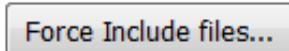
4. Click Insert.

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

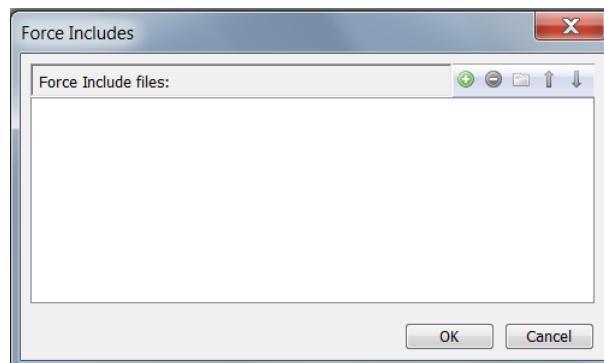
Or, you can perform the same action by going to the Programmer Studio Menu Bar and selecting **Project, Insert Assembler Source Files**.

Force Include files

1. On the **Assembler** tab, click the **Force Include files....** button.



The **Force Includes** dialog box appears.



2. Force Include files are header files that will be included with every file that is assembled.

Click the Add or Remove buttons on the dialog to manage this list.

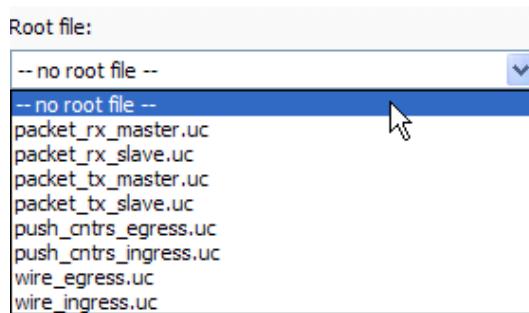
Delete File

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.



Indirect Reference Mode

There are two Indirect Reference modes: NFP-3200 mode and NFP-6000 mode. The NFP-3200 mode supports 13 formats of indirect data override with limitations. NFP-6000 mode has the most flexibility in overriding data fields with override bit per field. The format of the indirect data is specified in the instruction definition.. See "Flow Processor Core Programmer's Reference Manual" for more information.



Warning Level

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

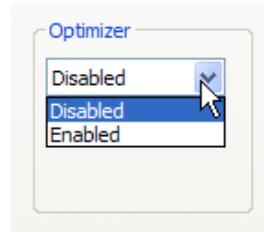
Warning as Error

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

Optimizer

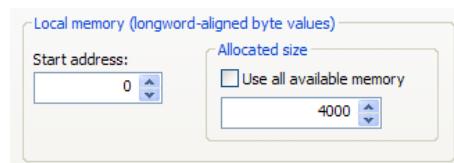
Disabled Turns off optimizations for better code troubleshooting.

Enabled Turns on optimizations for better code quality.



Local Memory

The **Local memory** settings allow you to specify the region of local memory that is available to the Assembler for allocating local memory variables.



The **Start** address is a longword-aligned byte address that specifies the start of the region. If **Use all available memory** is checked, then the region begins

at the start address and extends to the end of local memory. If **Use all available memory** is checked, then the region begins at the start address and extends for the number of bytes specified in the spin box.

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 is enabled.

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 NFFW file. Unchecking the **Produce debug info** check box causes the size of the NFFW file to be smaller at the expense of the project not being debuggable (in any fashion) through Programmer Studio.

Case sensitive identifiers

The command line arguments are case sensitive. If this option is not specified, the microcode file is case insensitive – all text in the file, with the exception of comments, is converted to lower case.

C compatible preprocessing

Select **C compatible preprocessing** to enable evaluation of undefined symbols as 0, similar to typical C preprocessors, instead of producing an error for the undefined symbol.

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.

Automatically spill GPRs

Generate register usage statistics

Force the collection of register usage statistics for a specified list file.

Protect macro locals

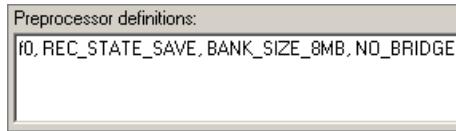
Check the **Protect macro locals** checkbox to enable the protect macros locals feature. This feature prevents names and labels inside a macro from conflicting with macro arguments. By default this feature is disabled. See the *Network Processor Databook* for details specific to the processor of your choice.

Protect macro locals

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 appending 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. Text added in this control appears 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**, Programmer Studio validates your data and does not allow the dialog box to close if there are any errors. Validation is independent of the page that 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.

2.6.3.1 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.

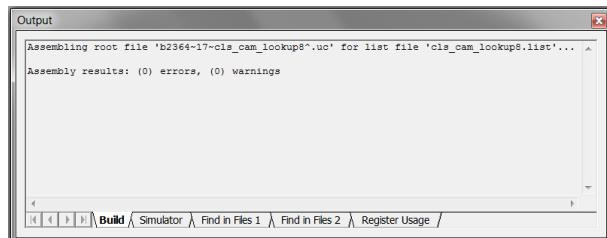
Root Files

If the file is a project source file, Programmer Studio 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, Programmer Studio 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 provided 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.

2.6.3.2 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 is not open, Programmer Studio opens it.

A blue arrow in the left margin marks the line 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.

2.6.4 Register Usage

Programmer Studio contains a new **Register Usage** tab on the **Output Window** dialog bar.



Selecting this tab displays a **Register Usage** window listing register use statistics. These statistics are reported by the assembler during the assembling for a list file. A sample **Register Usage** window is shown in Figure 2.8.

To enable **Register Usage** data collection:

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

The **Build Settings** dialog box appears.

2. Click the **Assembler** tab (see Figure 2.7).

3. Click **Generate register usage statistics** for each .list file that you want statistics for then click **OK**.

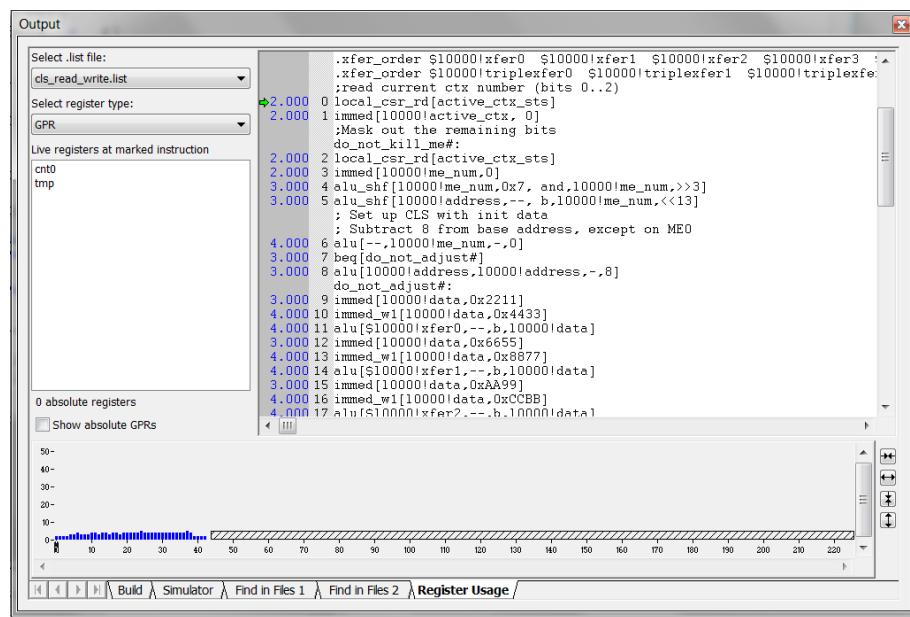


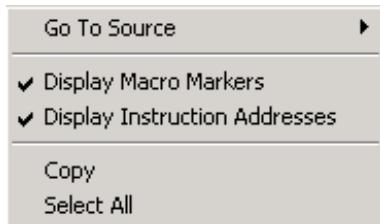
Figure 2.8. Register Usage Window

In Figure 2.8, the combo box labeled **Select a .list file**, displays the .list files for which register usage statistics have been collected. When Programmer Studio is about to invoke the assembler, it deletes the register usage statistics, if any, that it previously gathered for that list file and removes the list file from the combo box. When the assembly of a list file completes, if the assembler reported register usage statistics, Programmer Studio adds

that list file to the combo box. When a project is opened or closed, the combo box is cleared. Register usage statistics are not persistent – that is, statistics collected during previous Programmer Studio sessions are not available during the current session.

The instruction text window at the right in Figure 2.8, displays the lines of comments, instructions, etc., that is produced by the assembler for the selected .list file. In the gutter at the left side of the text window, Programmer Studio displays the usage count for each instruction – that is a line that generates a microword. To the right of the usage counts, Programmer Studio optionally displays the instruction address. Programmer Studio optionally displays macro markers which denote the beginning and end of a macro to the right of the instruction addresses.

To toggle the display of instruction addresses, right click in the text window and a popup appears.



Select **Display Instruction Addresses**. To toggle the display of macro markers, right click in the text window and select **Display Macro Markers**.

The bar graph at the bottom of the page displays a graphical representation of the usage counts. The horizontal axis depicts the instruction address while the vertical axis depicts the usage count. The buttons to the right of the bar graph are used for scaling the graph vertically and horizontally.

Because register allocation is done independently for the different types of Microengine registers, the **Select register type** combo box allows you to select which register type's usage counts are displayed in the bar graph and text window, and which live registers are displayed in the list. If usage statistics are being displayed for GPRs, then you can optionally have the live absolute registers listed in the live register list. Because all absolute GPRs are always live, this information is the same for all instruction addresses, you may not wish to clutter the live register list with the absolute GPRs.

The registers that are displayed in the live register list are the ones that are live at the instruction that is marked in the instruction text window and bar graph. To change the marker location, double-click in the bar graph at the desired instruction address.

Alternatively, double-click on a line in the instruction text window. If the line does not have an associated instruction address, the marker is moved to the next line that does. The marker location is synchronized so that the instruction text window and bar graph always point to the same instruction.

If the usage counts identify a register allocation problem at a particular point in the code, right-click on the instruction in the text window and select **Go To Source** from the popup. Programmer Studio opens the file containing the source code and scrolls to the line that generated that instruction. If the instruction is nested within macro calls, the **Go To Source** menu item will have an associated submenu that contains the instruction followed by the list of macro calls that generated it, in order from the lowest level call to the highest. This allows you to go to the source line that is most appropriate for fixing the allocation problem.

The register usage statistics are available before and during simulation or hardware mode debugging. These statistics will not be available for debug-only Programmer Studio projects, because the build is not performed under the control of Programmer Studio.

Non-interfering register groups

When an instruction is not part of a subroutine, or is part of a subroutine that is only called once in the program flow, the calculation of the number of live registers for the instruction is straight-forward. However, a subroutine can be called on a number of different paths through the program. The set of registers live at each of the subroutine calls do not necessarily conflict or interfere with each other. For example, one path to the subroutine requires register A live going in and another path requires register B. However, if A and B are never live at the same time, the assembler can allocate both of them to the same physical register. This means that together they only use up one live register – that is they only add a count of one to the usage count for any instruction within the subroutine. To decrease that usage count, you would have to delete both of those registers. To indicate this situation in the live register list, such registers are listed together under a **Non-interfering group X** heading, with each register name being indented so as to be distinguished from the other registers.

Meaning of Usage Counts

The number of available relative registers depends on the context mode. In four-context mode, there are 32 registers available. In eight-context mode, there are 16 available. Register allocation fails if more than the available number of registers are in use, or live, at a particular instruction. A relative register consumes four physical registers in four-context mode and eight physical registers in eight-context mode. An absolute register consumes only one physical register, regardless of mode. Usage counts measure the number of “equivalent” relative registers that are live at an instruction. A relative register is equal to one relative register, but an absolute register is equivalent to one-quarter of a relative register in four-context mode and one-eighth of a relative register in eight-context mode. For example, if one relative register and one absolute register are alive at an instruction address, the usage count is 1.125 for eight-context mode.

2.7 The C Compiler for Netronome Processors

Programmer Studio supports a Compiler to compile C source code into microcode for the Microengines. This compiler is available in an **Explicit Partitioning Mode**. This mode of the C Compiler is available on installing the Netronome NFP SDK Base/Encryption CD’s Tools component that also installs Programmer Studio. With this Compiler mode, the programmer explicitly controls the actions of the various microengines on the Netronome Processor chip. For more information on this Compiler refer to Chapter 4 in this document.



Note

For P4 projects, Programmer Studio sets all required Compiler settings automatically. It is highly recommended that users not to change any Compiler build settings for P4 projects .

The rest of this section focuses on using the explicit partitioning compiler. The explicit partitioning compiler (referred to as just ‘compiler’ hereafter) can compile a source file (.c or .h) or an object file (.obj).

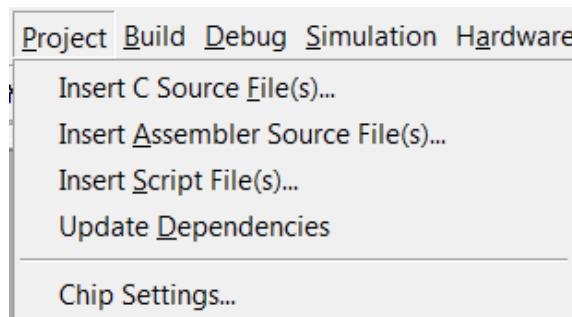
The following sub-sections detail how to use the general features of the compiler via Programmer Studio using the Explicit partitioning mode as this mode of the compiler is automatically available with Programmer Studio.

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 C Compiler Source Files ...**

The **Insert C 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 C 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 C Compiler Build Settings

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

To specify C Compiler options, do the following:

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

The **Build Settings** dialog box appears.

2. Click the **Compiler** tab (see Figure 2.9).

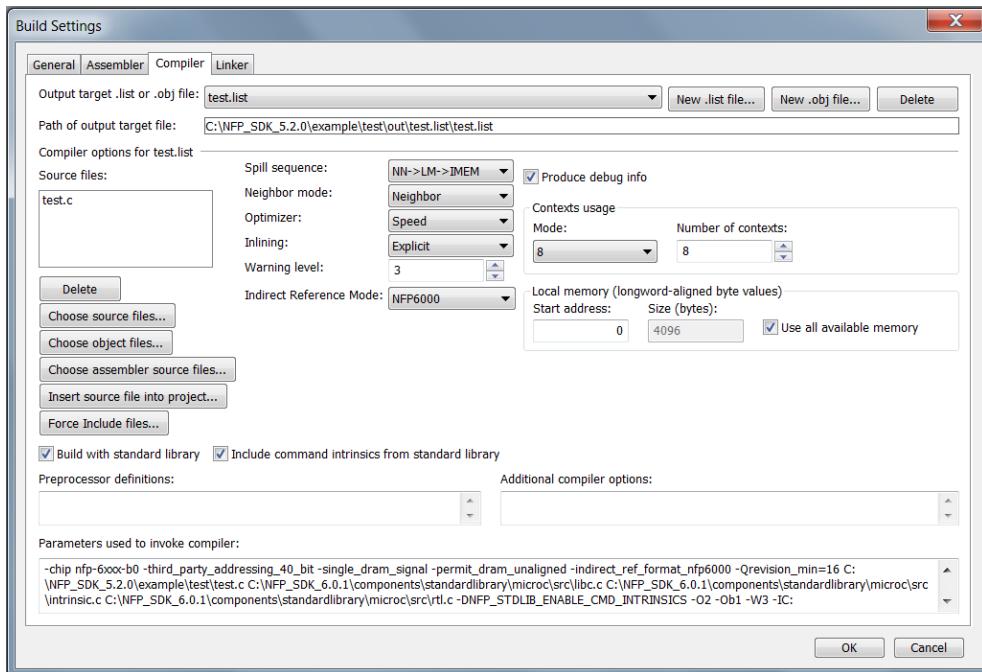


Figure 2.9. Build Settings Compiler tab



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 in order for it to 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 Programmer Studio 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 (see Figure 2.10). 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.

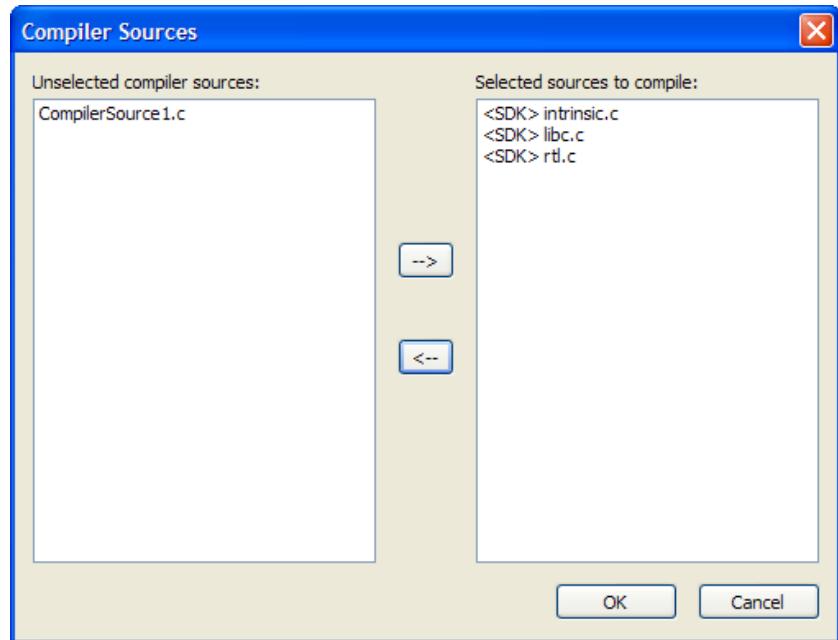


Figure 2.10. Selecting C Source Files to Compile

2.7.2.3.1 Build with standard library

To include and build with standard C source files, click on the **Build with standard library** check box on the **Compiler** tab.

Build with standard library

By default this option is selected and Programmer Studio compiles and links your application with standard C source files (intrinsic.c, libc.c, and rtl.c). These are normally located in the components\standardlibrary\microc subdirectory of the SDK installation directory. When this option is enabled, the standard C source files are not shown in the Source files list and cannot be removed. For complete control and to allow using custom versions if required, do not select this option. When this option is not selected, the standard C source files are not compiled and linked for the list file. You may add the standard C source files to the project manually and then select them for compilation and linking as if they are normal user C source files.

2.7.2.3.2 Include command intrinsics from standard library

Unless specially selected, command intrinsics are not visible within the standard library files. To access and build with these, ensure that the the **Include command intrinsics from standard library** check box on the **Compiler** tab is checked.

Include command intrinsics from standard library

2.7.2.4 Selecting Assembler Source Files to Compile

The C Compiler in Programmer Studio can assemble one or more microcode source files in addition to C source files into one .list file. You must select the assembler source files that you want to add to C source files to compile. To do this:

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

The **Sources Files** dialog box appears (see Figure 2.11). The files displayed here are all the *.uc files in your project; that is all the files in the **Assembler Source Files** folder in the **Project Workspace** window.

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

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 microcode source files that you selected to compile.

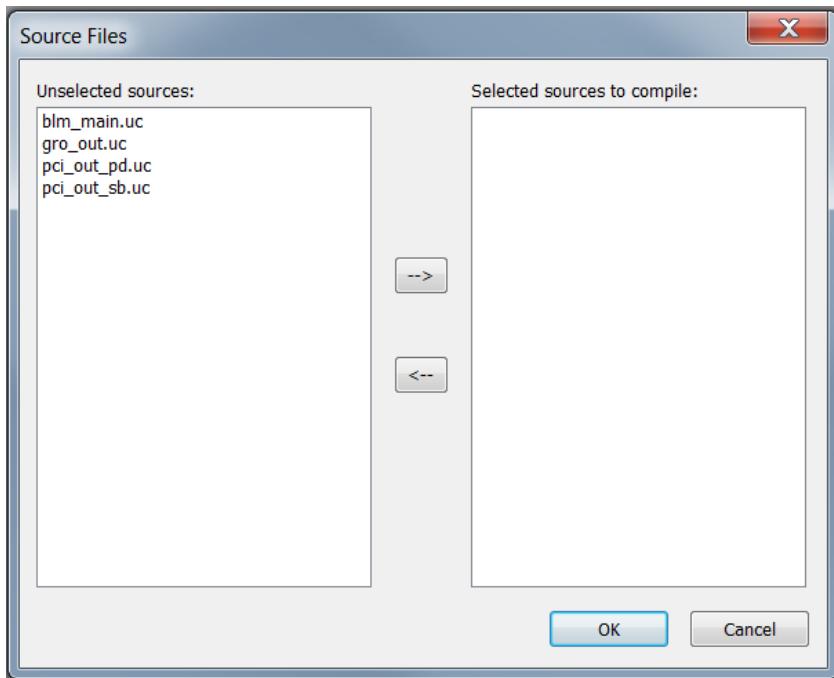


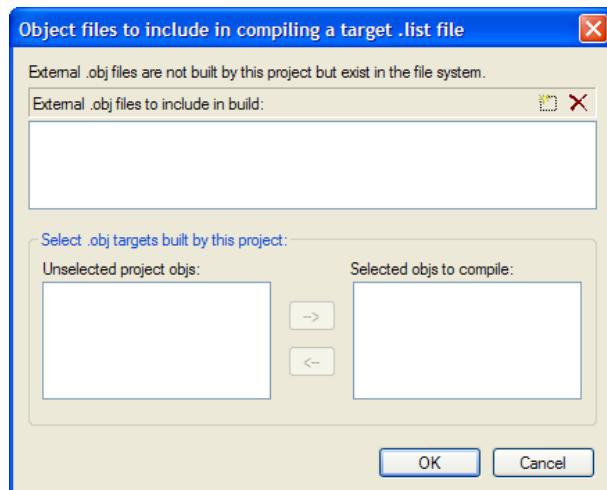
Figure 2.11. Selecting Assembler Source Files to Compile

2.7.2.5 Selecting C Object Files to Compile

The C Compiler in Programmer Studio 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 **Compiler** tab, click **Choose object files**.

The **Object file to include...** dialog box appears.



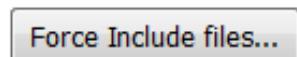
2. Enter the absolute path of any external object files you want to include in the build in the **External .obj files...** box.

3. Use the arrows to select or deselect any of your project object files you want to compile.

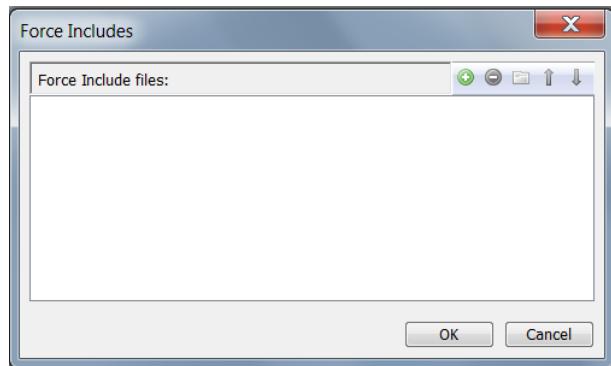
2.7.2.6 Force Include files

You can force include files in the C Compiler in Programmer Studio. To do this:

1. On the **Compiler** tab, click **Force Include files** button.



The **Force Includes** dialog box appears.



2. Force Include files are header files that will be included with every file that is compiled.

Click the Add or Remove buttons on the dialog to manage this list.

2.7.2.7 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.

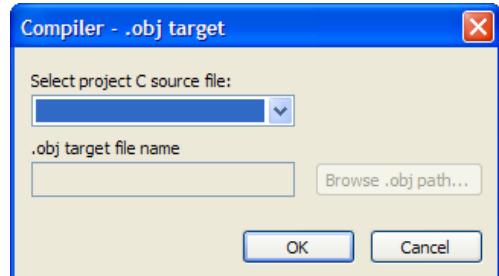
2.7.2.8 Selecting the Target .obj File

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:

- i. Click the **Browse .obj path** button.
 - ii. Select a new folder.
- b. Click **OK** when done.

2.7.2.9 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.

2.7.2.10 Selecting Compile Options

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

Spill sequence

Spill sequence:

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

Neighbor 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.
-------------	--

Optimizer

Optimizer: <input type="button" value="None"/>	
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.

Inlining

Inlining: <input type="button" value="None"/>	
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.

Warning level

Warning Level: <input type="button" value="3"/>	
0	Print only errors.
1, 2, or 3 (default)	Print only errors and warnings.
4	Print errors, warnings, and remarks.

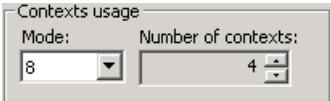
Indirect Reference Mode

Indirect Reference <input type="button" value="NFP6000"/>	
---	--

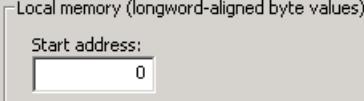
Produce debug info

<input checked="" type="checkbox"/> Produce debug info	
Select (default)	Produces debug information in the .list file. This information is needed for many of the debugging features of Programmer Studio.
Clear	No debug information is compiled into the .list file.

Contexts usage

	
Mode	Specify whether the Microengine is configured to have 4 or 8 contexts in use.
Number of Contexts	Select the number of contexts that you want to be active in the Microengine (1 through 8). All others are killed.

Local memory (longword-aligned byte values)

	
Start address	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.

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 for a complete list of options.
--

2.7.2.11 Saving Build Settings

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

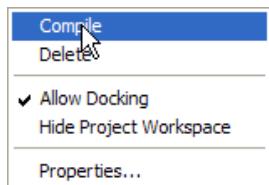
2. On the **Build** menu, click **Compile**, or

Press CTRL+SHIFT+F7, or

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

To compile a C module:

1. In the **File View** menu, select and right click the module to compile – a popup displays that includes the **Compile Module** option.



2. Select **Compile Module** and the module builds.

Only the selected module will be compiled. The Linker will not be invoked to link the project so other modules may require rebuilding before debugging.

Results

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

You can control the amount of detail provided 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.

You can also 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 is not already open, it opens.

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

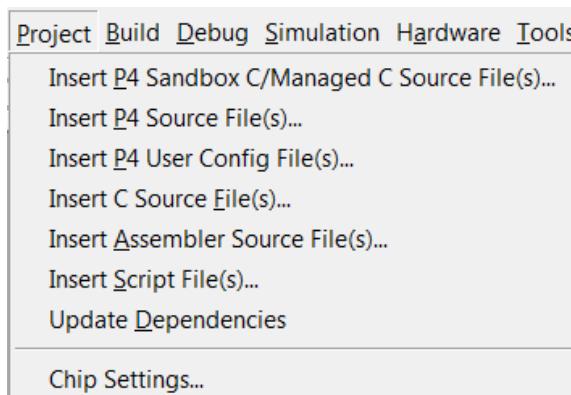
2.8 Specifying P4/Managed C Sandbox Build Settings

Programmer Studio supports a P4 Compiler to compile P4 source code into microcode for the Microengines. P4 Tools component is installed as part of Programmer Studio installation. For more information on P4 Compiler refer to Chapter 5 in this document.

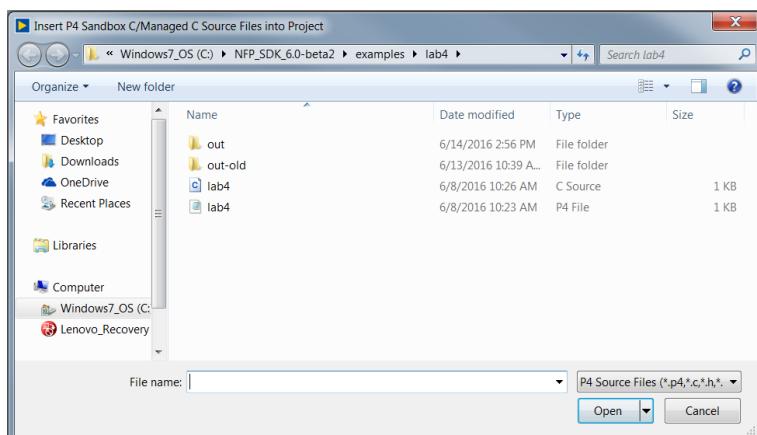
2.8.1 Adding P4 and Sandbox C Source Files to Your Project

After creating and saving P4 or C Sandbox C source files, you need to add them to your project. To do this:

1. On the **Project** menu, click **Insert P4 Sandbox C/Managed C Source File(s) ...** or **Insert P4 Source Files ...**



The **Insert P4 Sandbox C/Managed C Source Files into Project** dialog box appears.



2. From the **Look in** list, browse to the folder containing your P4 or P4 sandbox C source file(s).
3. Select the file(s) that you want to insert into your project.

4. Click **Open**.

In the Project Workspace window, to the left of the P4/Managed C 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.

2.8.2 Selecting P4 Sandbox C/Managed C Compiler Build Settings

Before building your project, you must select your P4 Sandbox C/Managed C compiler options.

To specify P4 Sandbox C/Managed C compiler options, do the following:

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

The **Build Settings** dialog box appears.

2. Click the **P4/Managed C** tab if its not already selected (see Figure 2.12).

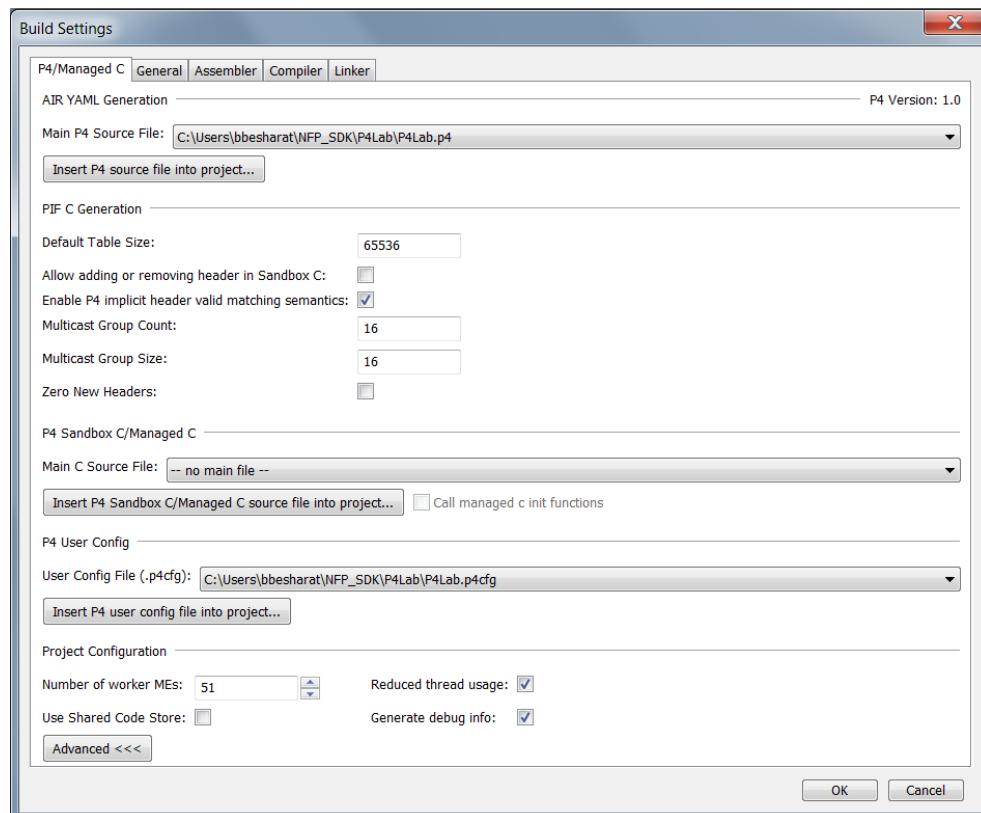


Figure 2.12. Build Settings P4/Managed C tab

2.8.2.1 Selecting Main P4 Source File

The P4 compiler needs to know which P4 file is the main file. To select main P4 source file click on:

Main P4 Source File:

and select P4 source file from drop down list.

2.8.2.2 Selecting Main P4 Sandbox C Source File

Optionally P4 projects can include a Sandbox C source file. Sandbox C files can use the PIF plugin framework and the NPE (Network Processor Environment) library to write stateful extensions for P4 applications. To select C source file click on:

Main C Source File:

and select C Sandbox source file from drop down list.

2.8.2.3 Selecting P4 User Config

To define the rules for match actions on the tables defined in your P4 source create a rules file with the Rules Editor UI and link it to your project. To select P4 rules file click on:

Rules File (.rules):

and select P4 rules file from drop down list.

2.8.2.4 PIF C Generation

The following fields can be changed to affect the PIF generated C code:

Default Table Size: The default table size in bytes for PIF generated tables.

Default Table Size:

Allow adding or removing header in Sandbox C: Disable optimization for headers that cannot be added or removed in P4 pipeline.

Allow adding or removing header in Sandbox C:

Enable P4 implicit header valid matching semantics: Enable header valid check for all packet field matches. Enabling this option will have a very slight negative impact on performance and code size.

Enable P4 implicit header valid matching semantics:

Multicast Group Count: Maximum number of multicast groups to support. Set to 0 for no multicast support.

Multicast Group Count:

Multicast Group Size: Maximum number of port entries per multicast group.

Multicast Group Size:

Zero New Headers: This turns on the strict P4 behaviour of zeroing all new headers.

Zero New Headers:

2.8.2.5 Project Configuration

Number of worker MEs: The number of worker MEs that will be assigned to the pif application list file. The maximum number depends on the chosen chip/board config.

Number of worker MEs:

Reduced thread usage: Use 4-context mode for MicroEngines; this reduces memory usage by half but reduces performance.

Reduced thread usage:

Use Shared Code Store: Use shared code store for MicroEngines; this allows the code to be loaded into two microengines sharing code store.

Use Shared Code Store:

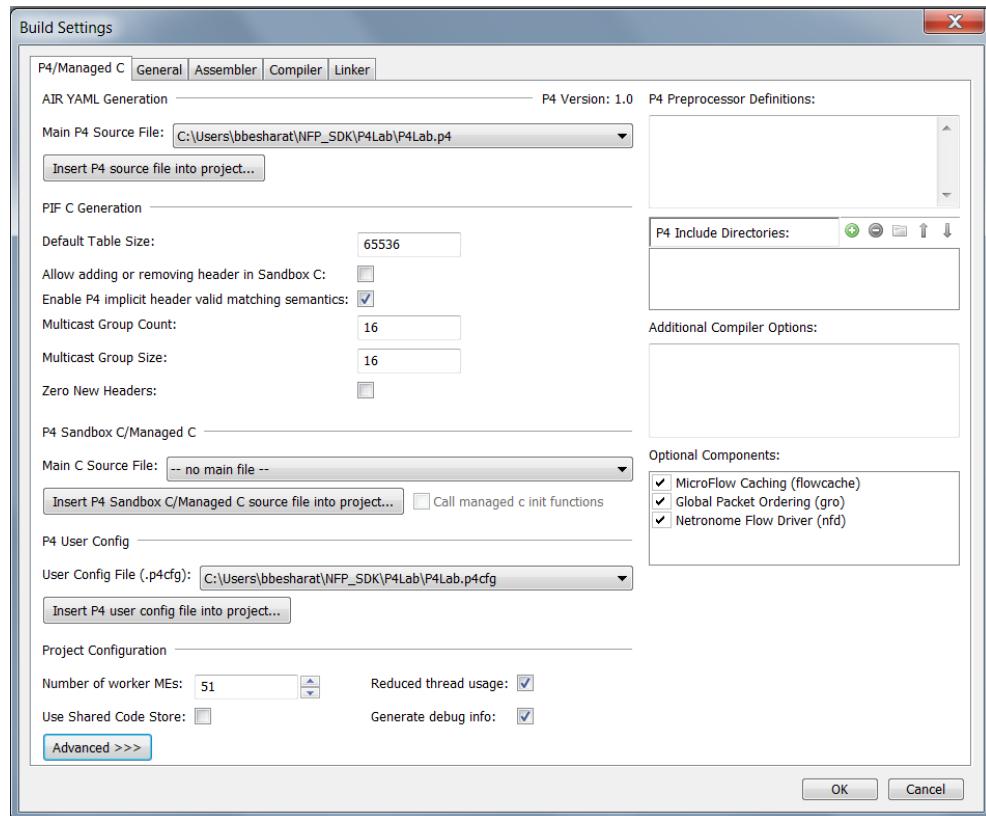
Generate debug info: This option controls the generation of debug information for generated code. When this option is disabled (unchecked), no debug information generated, therefore users can not debug the generated code. Select this option for final product with no debugging ability.

Generate debug info:

Advanced Options: To access advanced options on P4/Managed C build settings, click on the **Advanced <<<** button.

Advanced >>>

This will update the P4/Managed C build settings dialog with advance options section as shown below:



Add any additional P4 preprocessor definitions and/or additional NFCC compiler options to the respective box. Also you can specify any additional P4 include directories.

The **Optional Components** section allows you to disable or enable the following optional components that are built with P4 projects.

- **MicroFlow Caching (flowcache)**: Used to cache DCFL matches.
- **Global Packet Ordering (gro)**: Can be disabled if e.g. egress is not used.
- **Netronome Flow Driver (nfd)**: Can be disabled for NBI only applications.

2.8.2.6 Saving Build Settings

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

2.8.3 Invoking the P4 Compiler

To compile a P4 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 **Generate IR from P4**, or

Press SHIFT+F7, or

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

To check a P4 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 **P4 Check**, or

Press CTRL+ALT+F7, or

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

Results

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

You can control the amount of detail provided 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.

2.8.4 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 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.

You can also press SHIFT+F4 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 is not already open, it opens.

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

2.9 Specifying Linker Build Settings

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.



Note

For P4 projects, Programmer Studio sets all required Linker settings automatically. It is highly recommended that users not to change any Linker build settings for P4 projects .

2.9.1 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.
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 NFFW 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 **Network Flow C Compiler** tab in the **Build Settings** dialog box.

There are three tabs on Linker page.

1. **General** tab
2. **List File Assignments** tab
3. **Memories** tab

1. General Tab

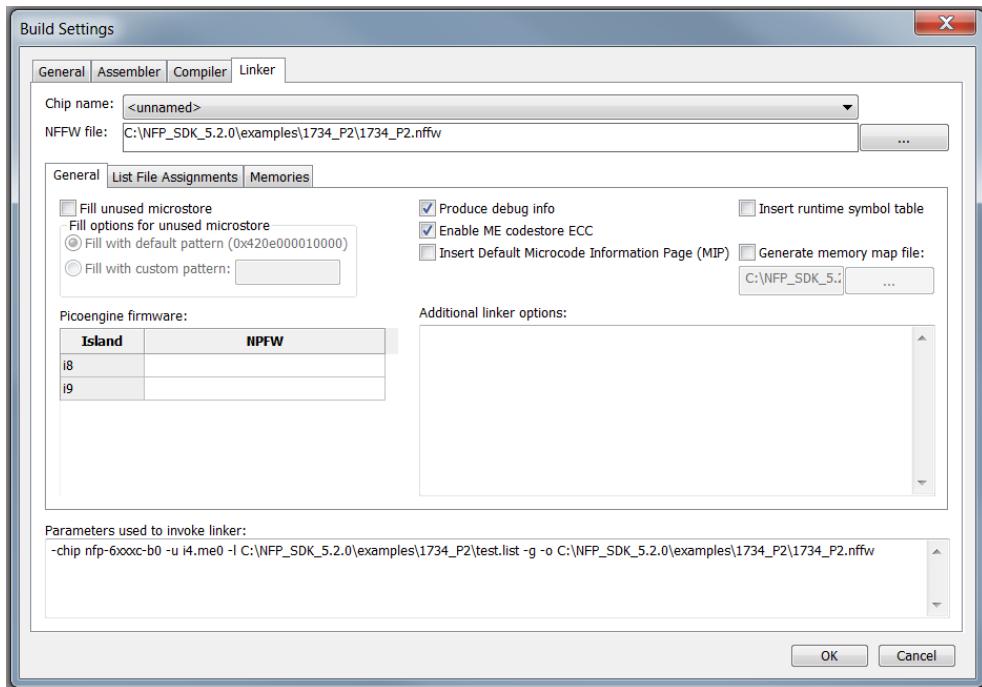


Figure 2.13. Build - Linker - General Tab

Chip name

The **Chip name** 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.

Chip name:

chip_00

NFFW file

The **NFFW file** box displays the .nffw file that the Linker produces.

NFFW file:

C:\nfp\basic\basic\chip_00.nffw



Note

Programmer Studio does not support multiple NFFW files for the same chip.

To change the output NFFW file to the project for the selected chip:

1. Click the **...** button.

The **Select Name and Location for the Netronome Flow File** dialog box appears.

2. In the **Look in** box, browse to the folder where you want to put the output file.

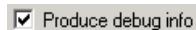
3. Type a new name in the **File name** box.

You do not have to type the .nffw extension — Programmer Studio adds it for you, but you can also type it.

4. Click **Select**.

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 NFFW file. Unchecking the **Produce debug info** check box causes the size of the NFFW file to be smaller at the expense of the project not being debuggable (in any fashion) through Programmer Studio.

Enable ME codestore ECC

This will enable or disable calculation and storage of the ECC bits for each codestore word and consequently they will be set when the NFFW file is loaded.

Insert Default Microcode Information Page (MIP)

The MIP is a small structure in DRAM containing, among other things, information about when the image file was built and what toolchain version was used to build it.

This option will insert a default MIP in the NFFW output image.

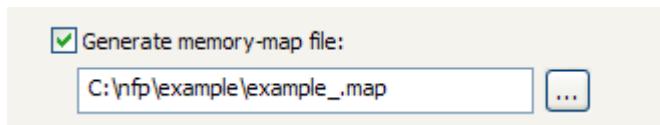


Insert runtime symbol table

This will insert a MIP entry containing the symbol table and is only valid if a MIP is present, either by microcode or by inserting the default MIP.

Generate memory map file

Select **Generate memory map file** to have Programmer Studio 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 you to specify the filename or browse to it.



Picoengine

Select **Picoengine** to have Programmer Studio program NBI Island(s) pico engines with pre defined pico code. Options available are: **ipv4_fwd**, **null**, **catamaran**, **wire**.

PICO code	
Island	Partial Firmware
i8	ipv4_fwd

2. List File Assignments Tab

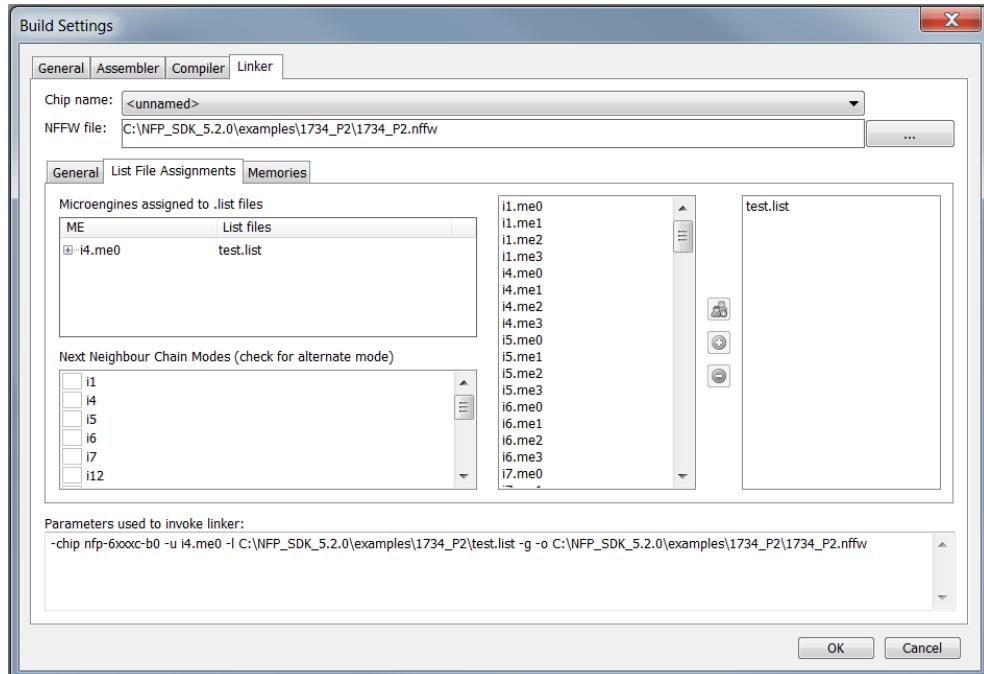


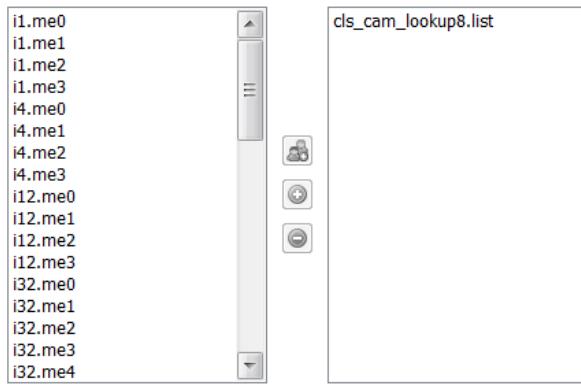
Figure 2.14. Build - Linker - List File Assignments Tab

Linker options for chip

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 NFFW file and for which Microengine.

To do this:

1. To the right of **List File Assignments** page click on one or more Microengines.



2. Click on one or more .list file from the list to the right.
3. Click the button to add selected Microengine(s) and assigned .list file(s) to the **Microengines assigned to .list files** list.

Microengines assigned to .list files	
ME	List files
i12.me0	cls_cam_lookup8.list

To remove a Microengine from the **Microengines assigned to .list files** list, simply select a Microengine from the list and click the button.

4. Do the same for remaining Microengines.

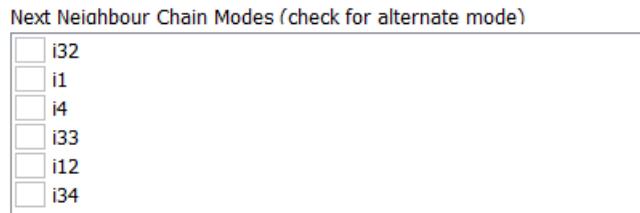
This method allows you to select any combination of .list files or no .list file for any or all the Microengines to be linked to the NFFW file.

If you do not assign microcode to at least one Microengine, you get an error.

5. To set up a **Shared Code Store** list file assignment, first select two consecutive microengines starting with an even microengine number. Then select one or more list files and click the **Group assign (Shared Code Store)** button.

Next Neighbor Chain Modes

Next neighbor chains can be controller for each Island. You can select between "Normal" or "Alternate" chaining mode.



You can select "Alternate" mode for an island by checking the check box for that specific Island. If check box is not checked, "Normal" chaining mode is selected. "Normal" chaining mode, for Island 32 as an example, will result in the sequence i32:0, i32:1, i32:2 ... i32:11. "Alternate" mode will result in i32:0, i32:2, i32:4, i32:6, i32:8, i32:10, i32:1, i32:3, i32:5, i32:7, i32:9, i32:11. Using "Alternate" mode is useful when sharing control stores for MEs in that Island.

3. Memories Tab

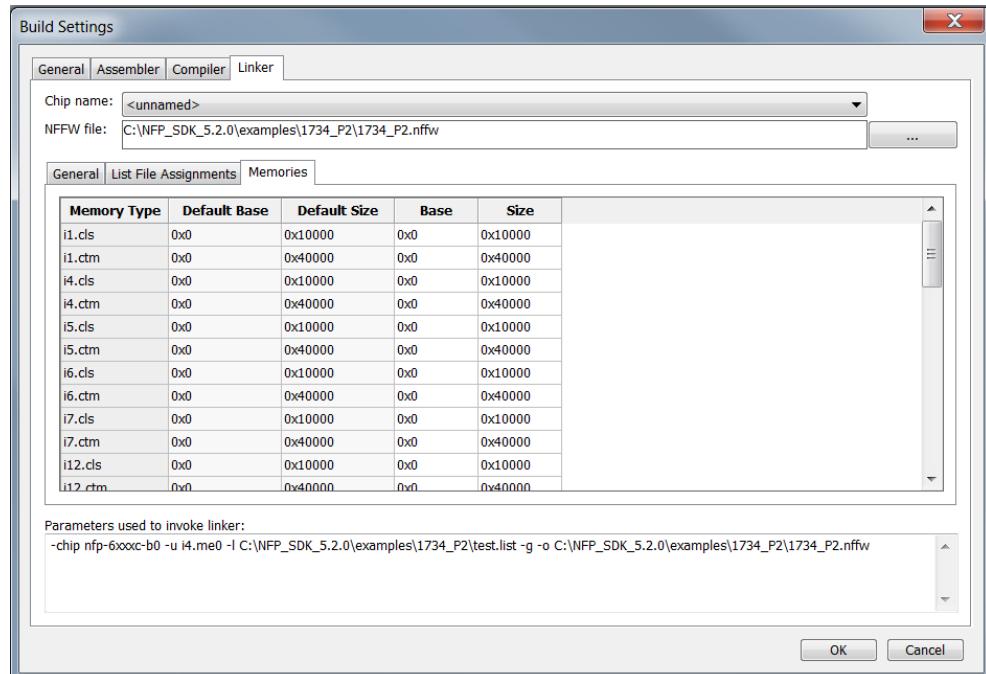


Figure 2.15. Build - Linker - Memories Tab

Reserved memory segment for variables

Memory Type	Default Base	Default Size	Base	Size
i1.cls	0x0	0x10000	0x0	0x10000
i1.ctm	0x0	0x40000	0x0	0x40000
i4.cls	0x0	0x10000	0x0	0x10000
i4.ctm	0x0	0x40000	0x0	0x40000
i5.cls	0x0	0x10000	0x0	0x10000
i5.ctm	0x0	0x40000	0x0	0x40000
i12.cls	0x0	0x10000	0x0	0x10000
i12.ctm	0x0	0x40000	0x0	0x40000
i13.cls	0x0	0x10000	0x0	0x10000
i13.ctm	0x0	0x40000	0x0	0x40000
i24.emem	0x800000	0x1ff800000	0x800000	0x1ff800000
i28.imem	0x0	0x400000	0x0	0x400000
i32.cls	0x0	0x10000	0x0	0x10000
i32.ctm	0x0	0x40000	0x0	0x40000
i33.cls	0x0	0x10000	0x0	0x10000
i33.ctm	0x0	0x40000	0x0	0x40000
i34.cls	0x0	0x10000	0x0	0x10000
i34.ctm	0x0	0x40000	0x0	0x40000
i35.cls	0x0	0x10000	0x0	0x10000
i35.ctm	0x0	0x40000	0x0	0x40000
i36.cls	0x0	0x10000	0x0	0x10000
i36.ctm	0x0	0x40000	0x0	0x40000
i37.cls	0x0	0x10000	0x0	0x10000
i37.ctm	0x0	0x40000	0x0	0x40000
i48.cls	0x0	0x10000	0x0	0x10000
i48.ctm	0x0	0x40000	0x0	0x40000
sram0	0x0	0x40000000	0x0	0x40000000
sram1	0x0	0x40000000	0x0	0x40000000
sram2	0x0	0x40000000	0x0	0x40000000
sram3	0x0	0x40000000	0x0	0x40000000

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

- | | |
|---------------------|--|
| Memory Type | Each Island that has Microengine has two types of memory: 1) CLS; 2) CTM. External memory units are Island 24, 25 and 26. Internal memory units are Island 28 and 29. Also 4 SRAM banks can be used. |
| Base Address | The Linker uses specified memory starting at the base address, allocating as much memory as needed up to the memory size for variables. |
| Size | The memory size is a parameter sent to the Linker. The Linker reserves as much DRAM as necessary for variables up to the size. |

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

Linker settings are saved when you save the project.

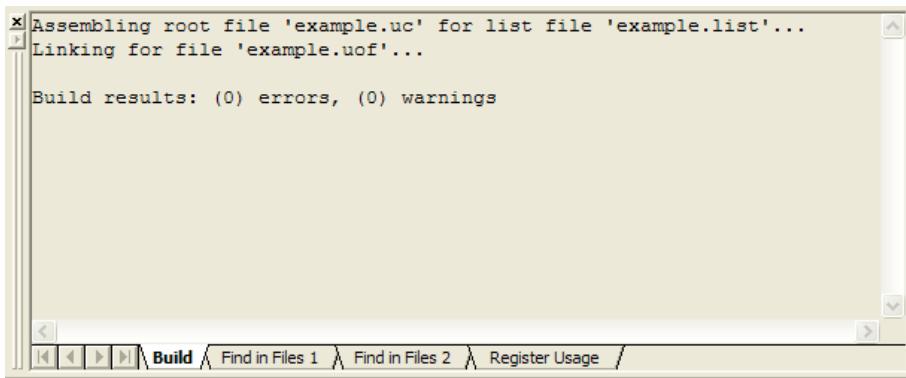
2.9.2 Building and Rebuilding a Project

Building a project

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

Click the  button on the **Build** toolbar, or Press F7.

The results of the build appear in the **Build** tab of the **Output** window, which appears automatically. For standard projects build output would be as follow:



The screenshot shows the 'Output' window with the 'Build' tab selected. The window displays assembly and linking progress: 'Assembling root file 'example.uc' for list file 'example.list'...' and 'Linking for file 'example.uof'...'. At the bottom, it shows 'Build results: (0) errors, (0) warnings'. The window has standard Windows-style scroll bars and a toolbar at the bottom with buttons for Back, Forward, Find, and Register Usage.

```
x Assembling root file 'example.uc' for list file 'example.list'...
Linking for file 'example.uof'...

Build results: (0) errors, (0) warnings
```

For P4 projects build output would be as follow:

The screenshot shows a software interface for building a project. The main window displays a build log with numerous compiler and linker messages. Key lines include:

```
Compiling for list file '...
WARNING: Token 'PPHASH' defined, but not used
WARNING: There is 1 unused token
nfd_pcio0_pci_out_me0.list compiler list file added.
flowcache_timeout_emu0.list compiler list file added.
nfd_pcio0_pd0.list assembler list file added.
pif_app_nfd.list compiler list file added.
nfd_pcio0_pci_in_gather.list compiler list file added.
nfd_pcio0_sb.list assembler list file added.
blm0.list assembler list file added.
nfd_svc.list compiler list file added.
nfd_master.list compiler list file added.
nfd_pcio0_pci_in_issue0.list compiler list file added.
gro0.list assembler list file added.
nfd_pcio0_notify.list compiler list file added.
pif_app_nfd_master.list compiler list file added.
Linker settings updated.
Assembling root file 'pci_out_pd.uc' for list file 'nfd_pcio0_pd0.list'...
Assembling root file 'pci_out_sb.uc' for list file 'nfd_pcio0_sb.list'...
Assembling root file 'blm_main.uc' for list file 'blm0.list'...
Assembling root file 'gro_out.uc' for list file 'gro0.list'...
Compiling for list file 'nfd_pcio0_pci_out_me0.list'...
Compiling for list file 'flowcache_timeout_emu0.list'...
Compiling for list file 'pif_app_nfd.list'...
Compiling for list file 'nfd_pcio0_pci_in_gather.list'...
Compiling for list file 'nfd_svc.list'...
Compiling for list file 'nfd_master.list'...
Compiling for list file 'nfd_pcio0_pci_in_issue0.list'...
Compiling for list file 'nfd_pcio0_notify.list'...
Compiling for list file 'pif_app_nfd_master.list'...
Linking for file 'P4Lab1.nffw' ...

Build results: (0) errors, (0) warnings
Updating dependencies...
Finished updating dependencies.
```

The status bar at the bottom of the interface shows the build progress: "nfp-6xxx A0 Building...".

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

The build status appears on the status bar. While running, the message indicates “Building ...” .

nfp-6xxx A0 **Building...**

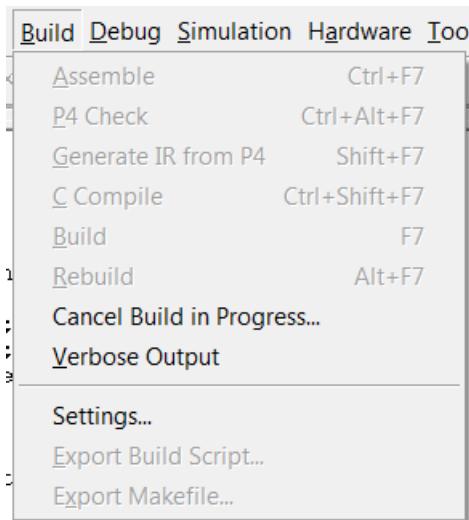
While a build is in process you can continue to perform other functions, such as editing files, but these edited files may result in an incorrect or an outdated build.

Cancelling a Build

You can terminate a Build while it is in progress. This will release all allocated resources, remove all temporary files, and close any open files.

- On the **Build** menu, click **Cancel Build in Progress**.

A popup displays indicating that the Build has been successfully canceled.



Out-of-date files

To perform a link, Programmer Studio 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, Programmer Studio 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

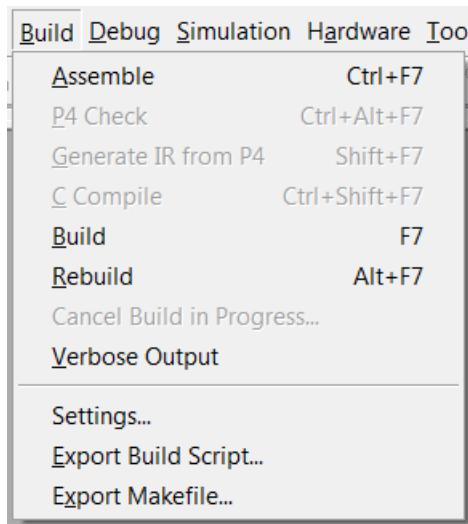
Click the  button on the **Build** toolbar, or Press Alt + F7.

Exporting the Build Script

To create a text file containing all the build commands for building the project:

- On the **Build** menu, click **Export Build Script...**

The **Export Build Script** dialog appears.



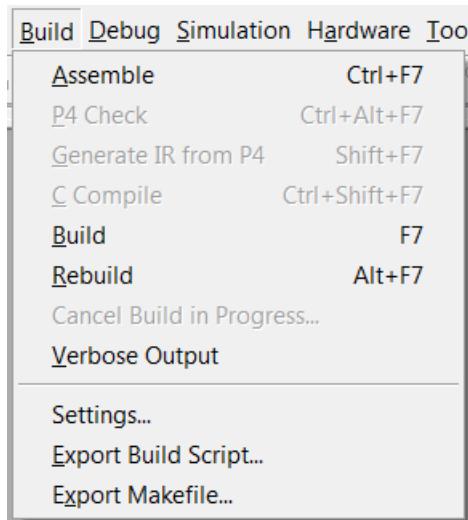
2. Browse to the folder to save the file.
3. Type the name of the script file in the **File name** box.
4. Click **Save**.

Exporting the Make File

To create a make file for the project:

1. On the **Build** menu, click **Export Makefile...**

The **Export Makefile** dialog appears.



2. Browse to the folder to save the file.
3. Type the name of the makefile in the **File name** box.
4. Click **Save**.

2.10 Configuring the Netronome NFP-6000 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 2.10.1)
- **Memory** (see Section 2.10.2)

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

2.10.1 Clock Frequencies

In this release of the NFP SDK, the clock divisors and frequencies can be selected from predefined lists.

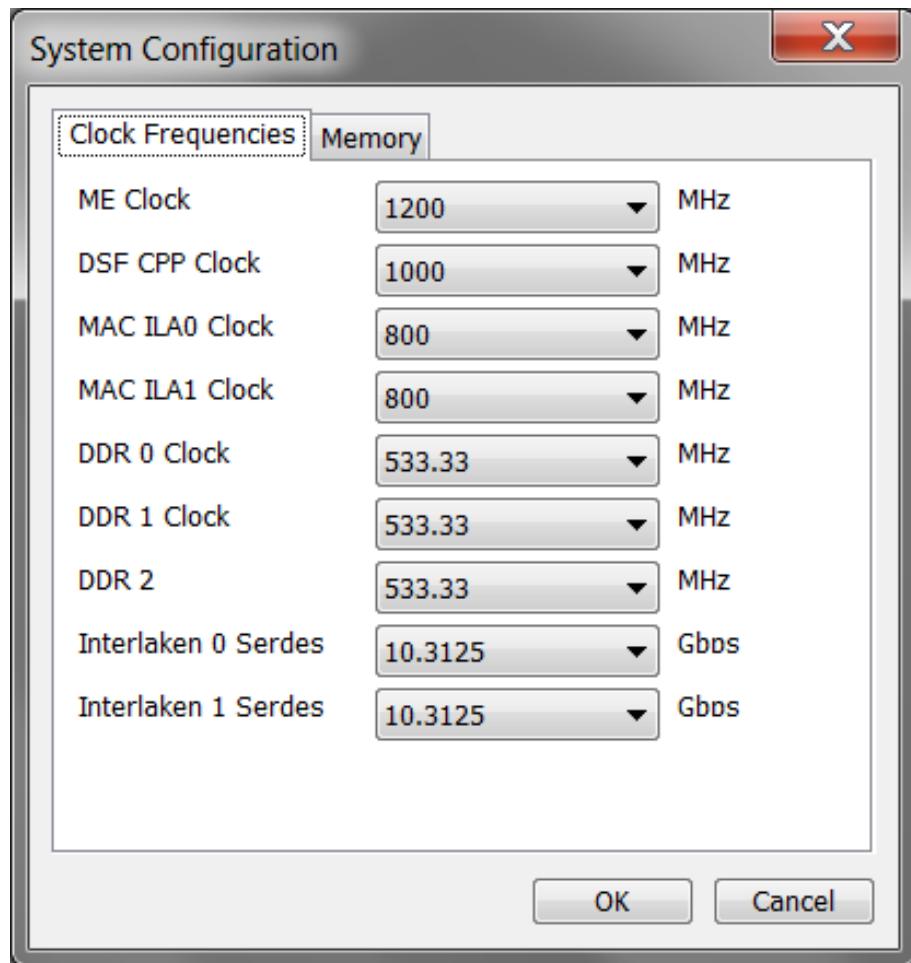


Figure 2.16. NFP-6000 Clock Frequencies Options

The complete description of clock frequencies and ratios can be found in the *Netronome Network Flow Processor NFP-6000 Databook*.

2.10.2 Memory

The **Memory** tab on the **System Configuration** property sheet supports the configuration of simulator memory (see Figure 2.17). This release of the NFP simulator supports up to 24G of memory for three external memory (DRAM) Units (Each EMEM Island has 8G of memory). Following option is supported:

- Leveling: This option allows you to enable or disable write leveling for specific memory Island. The compete description of write leveling and it's algorithm can be found in the *Netronome Network Flow Processor NFP-6000 Databook*.

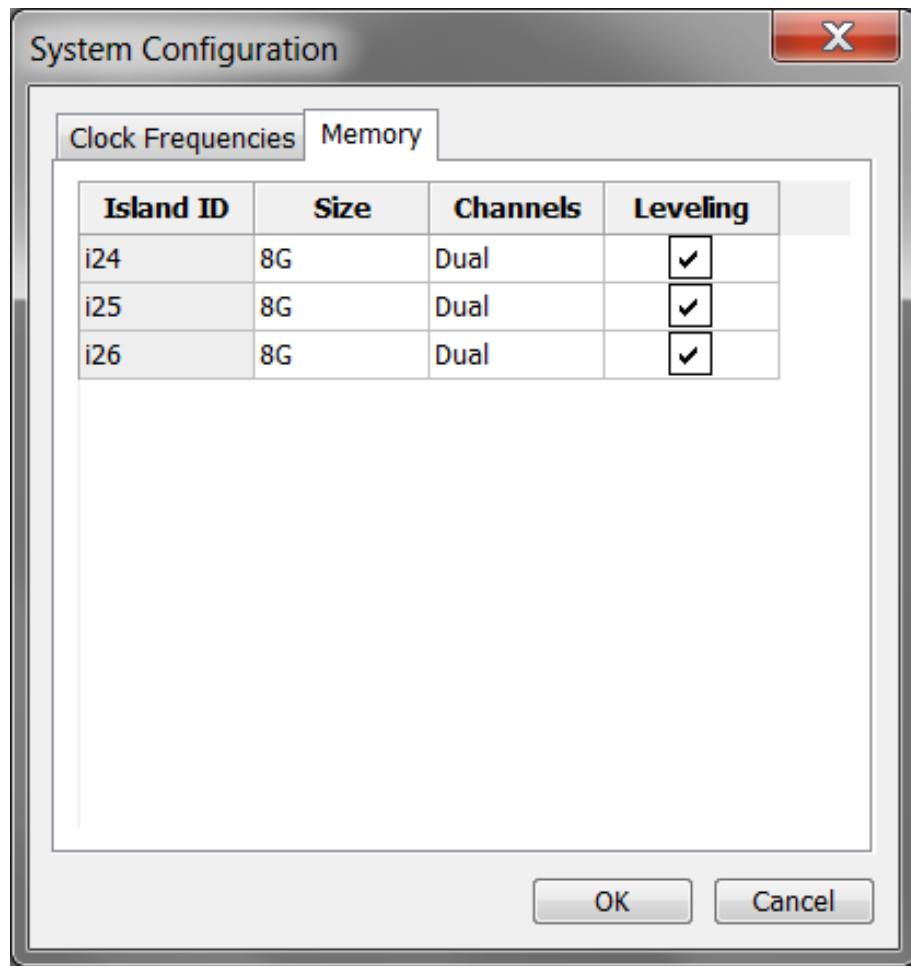


Figure 2.17. NFP-6000 Memory Options

2.11 Packet Streaming

The **Packet Streaming Configuration** and **Statistics** dialog boxes are added to Programmer Studio to set up and load packet streaming Foreign Models (FM) and also view receive and transmit statistics of the relevant interfaces. Configuration is only available when **not** in debug mode and allows a wide range of parameters to be set prior to a debug session. Statistics are only viewable while in debug mode.

Each NBI Island can be configured with one of the following port and device set-ups:

Configurations for P4 and Managed-C projects:

1. None - No ports available to configure
2. be-1x40GE-prepend - One 40 Gbps port on NBI 0 and / or NBI 1
3. be-2x40GE-prepend - Two 40 Gbps ports on NBI 0 and / or NBI 1
4. be-4x10GE-1x40GE-prepend - Five 10 Gbps ports on NBI 0 and / or NBI 1

5. be-4x10GE-prepend - Four 10 Gbps ports on NBI 0 and / or NBI 1
6. be-8x10GE-prepend - Eight 10 Gbps ports on NBI 0 and / or NBI 1
7. cdp-1x100GE-2x10GE-3x40GE-prepend - One 100 Gbps, two 10 Gbps and three 40 Gbps ports on NBI 0 and / or NBI 1
8. hy-1x40GE-prepend - One 40 Gbps port on NBI 0 and / or NBI 1
9. hy-4x10GE-prepend - Four 10 Gbps ports on NBI 0 and / or NBI 1
10. li-2x10GE-prepend - Two 10 Gbps ports on NBI 0 and / or NBI 1
11. nfp_nbi8_dma_cdp
12. nfp_nbi8_dma_debug
13. nfp_nbi8_dma_hy
14. nfp_nbi8_dma_sf
15. sf1-2x40GE-prepend - Two 10 Gbps ports on NBI 0 and / or NBI 1
16. sf1-8x10GE-prepend - Eight 10 Gbps ports on NBI 0 and / or NBI 1

Configurations for all other projects:

1. None - No ports available to configure
2. 1x10GE - One 10 Gbps port on NBI 0 and / or NBI 1
3. 1x100GE - One 100 Gbps port on NBI 0 and / or NBI 1
4. 1x100GE-2x10GE - One 100 Gbps port and two 10 Gbps ports on NBI 0 and / or NBI 1
5. 12x10GE--12x10GE - 24 10 Gbps ports on NBI 0 and / or NBI 1
6. 12x1GE--12x1GE - 24 1 Gbps ports on NBI 0 and / or NBI 1
7. 2x1GE_cdp - Two 1 Gbps ports on NBI 0 and / or NBI 1
8. 3x40GE - Three 40 Gbps ports on NBI 0 and / or NBI 1
9. 4x10GE-2x40GE - Four 10 Gbps ports and two 40 Gbps ports on NBI 0 and / or NBI 1
10. 4xIL-2x40GE - Four Interlaken devices with up to 8 channels and two 40 Gbps ports on NBI 0 and / or NBI 1
11. 4xIL-4x10GE-1x40GE - Four Interlaken devices with up to 8 channels, four 10 Gbps ports and one 40 Gbps port on NBI 0 and / or NBI 1
12. 4xIL-8x10GE - Four Interlaken devices with up to 12 channels and eight 10 Gbps ports on NBI 0 and / or NBI 1
13. 8x10GE-1x40GE - Eight 10 Gbps and one 40 Gbps ports on NBI 0 and / or NBI 1
14. 8xIL - Eight Interlaken devices with up to 8 channels on NBI 0 and / or NBI 1
15. 8xIL-1x40GE - Eight Interlaken devices with up to 8 channels and one 40 Gbps port on NBI 0 and / or NBI 1
16. 8xIL-4x10GE - Eight Interlaken devices with up to 8 channels and four 10 Gbps ports on NBI 0 and / or NBI 1
17. 12x1GE - Twelve 1 Gbps ports on NBI 0 and / or NBI 1
18. 12x10GE - Twelve 10 Gbps ports on NBI 0 and / or NBI 1

19. 12xIL - Twelve Interlaken devices on NBI 0 and / or NBI 1 with up to 64 channels
20. cdp-1x100GE-2x10GE-3x40GE - One 100 Gbps, twelve 10 Gbps and three 10 Gbps ports on NBI 0 and / or NBI 1
21. cdp-2x1GE-3x40GE - Two 1 Gbps three 40 Gbps ports on NBI 0 and / or NBI 1
22. hy-1x40GE - Three 40 Gbps ports on NBI 0 and / or NBI 1
23. li-2x10GE - Two 10 Gbps ports on NBI 0 and / or NBI 1
24. sf1-1x100GE - One 100 Gbps port on NBI 0 and / or NBI 1
25. sf1-2x40GE - Two 40 Gbps ports on NBI 0 and / or NBI 1
26. sf1-2x40GE-aux - Two 40 Gbps ports on NBI 0 and / or NBI 1
27. sf2-1x40GE - One 40 Gbps ports on NBI 0 and / or NBI 1

Using packet streaming port parameters and a pcap input stream file with useful timestamps, the input rate can be finely planned and controlled as necessary to achieve average slower rates or specific flow patterns.

2.11.1 Supported File Formats

2.11.1.1 Packet Capture (*.pcap)

Standard PCAP files (with file format version 2.4) are supported as input and output files. Input files can be obtained by capturing data on a live network (using TCPDump, WinDump, or Wireshark) or by creating / editing files (using NetDude, Scapy, dpkt, etc). Output files can be viewed using applications like NetDude, TCPDump / WinDump, or Wireshark, and can be analyzed using tools like Scapy.

2.11.1.1.1 PCAP Input Considerations

1. Nanosecond PCAP files are supported as input. Timestamps may be used to schedule input frames at the same relative time spacings. This may also be used to shape an input stream to custom requirements.
2. The file header is handled as is most common in practice. This means ignoring the *timezone* and *sigfigs* fields.
3. Any PCAP data link type may be used. For Ethernet link types, small captured frames (smaller than 64 bytes) will be padded according to IEEE 802.3-2008 and all frames will always have a valid, calculated CRC appended before being streamed into the simulator, regardless of whether or not one is present in the input file. For all other link types, the frame will be sent as is, regardless of whether this is valid on the specific interface or not.

2.11.1.1.2 PCAP Output Considerations

1. The file header is written as is most common in practice. This means *timezone* and *sigfigs* are both set to 0.
2. The output file uses either nanosecond or microsecond resolution timestamps (user configurable) and timestamps are absolute values (in seconds) based on the ME cycles since the start of the simulation. This provides an accurate capture similar to what a real interface capture might look like. Timestamps are taken at the first data byte of a frame (no preamble or control byte).

2.11.2 Configuring Packet Streaming

2.11.2.1 Packet Streaming Setup

1. On the **Simulation** menu, click **Packet Streaming Configuration....**

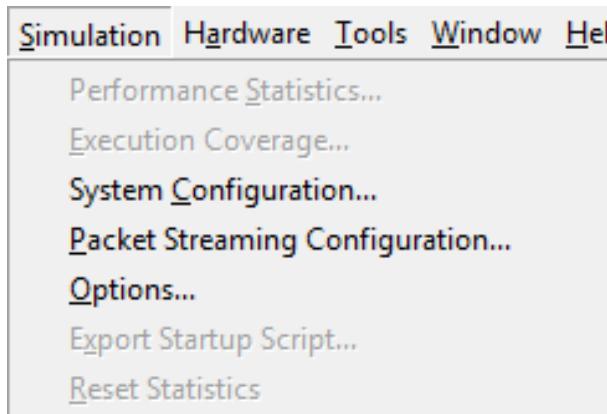


Figure 2.18. Packet Streaming Configuration Dialog Box

2. The **Packet Streaming Configuration** property page appears.

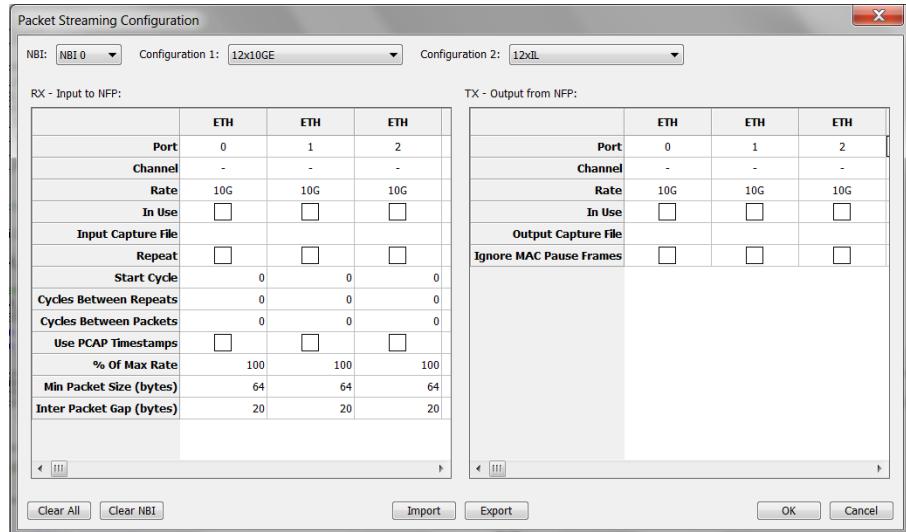


Figure 2.19. Packet Streaming Configuration

The screenshots do not show all possible configuration iterations, but the basic parameters are the same for all configurations and NBIs

2.11.2.2 Receive Port Parameters for all Device Types

2.11.2.2.1 In Use

Set to **Yes** if you want to use this port as input to the NPU. If a port is used only for receiving, output from the NPU will not be captured. All other parameters are disregarded if the port is not in use for this direction. A port must be in use if it is to be viewable in the **Statistics** window.

2.11.2.2.2 Input File

If an input file is **not** specified, but the port is in use, all parameters will be configured but no input streaming to the NPU will take place. This effectively disables the port streaming while allowing statistics gathering. This is contradictory, since an input port without streaming will have no useful statistics, but allows a user to have symmetric views of port statistics. The primary format is PCAP. Nanosecond capture files are supported and recommended for high bit rate interfaces. Legacy Datastreams are also supported, but these do not provide time stamp information.

2.11.2.2.3 Repeat

Set to **Yes** the stream will be repeated once the last packet in the stream has been received by the NPU. If the stream does not repeat, it will simply become an idle port once the end of the stream is reached.

2.11.2.2.4 Start Cycle

This is useful for delaying any streaming input until a specific point in the simulation, typically until after all initialization code is complete and MEs are ready to receive packets. This is an absolute reference to the start of the simulation.

2.11.2.2.5 Cycles Between Repeats

This is simply the number of cycles for which the input stream should be idle after the last byte of the last frame. This does not apply to a stream that is not set to repeat.

2.11.2.2.6 Cycles Between Packets

This is simply the number of cycles for which the input stream should be idle after the last byte of all frames (first, middle and last frames).

2.11.2.3 Transmit Port Parameters for all Device Types

2.11.2.3.1 In Use

Set to **Yes** if you want to use this port as output from the NPU. If a port is used only for transmitting, the assumption is that the simulated application does not require input on this port in order to send output. All other parameters are disregarded if the port is not in use for this direction. A port must be in use if it is to be viewable in the **Statistics** window.

2.11.2.3.2 Output File

If an output file is not specified, but the port is in use, all parameters will be configured but no output capturing from the NPU will take place. The primary format is PCAP.

2.11.3 Packet Streaming Statistics

This option is only available while in Debug Mode.

2.11.3.1 View Packet Streaming Statistics

1. On the **View** menu, click **Debug Windows** and then **Packet Streaming Statistics**.

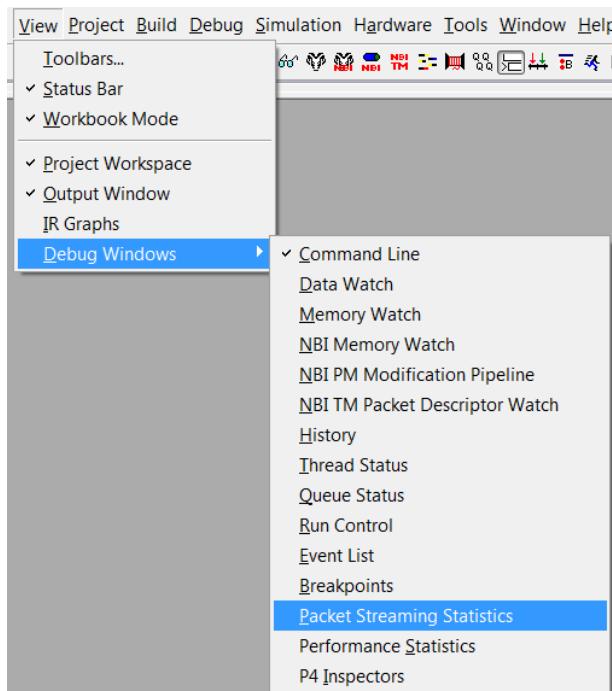


Figure 2.20. Packet Streaming Statistics Dialog Box

2. The **Packet Streaming Statistics** property page appears.

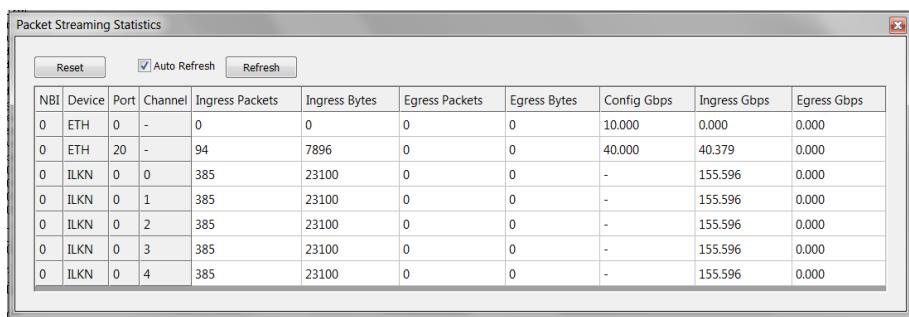


Figure 2.21. Packet Streaming Statistics Dialog

2.11.3.2 Details about Packet Streaming Statistics

Only ports that are **In Use** will be listed for viewing. If a port is in use in one or both directions, both its receive and transmit statistics will be shown.

The statistics consider line activity to be when a frame data byte is present, which excludes preamble and any control bytes as required by IEEE 802.3-2008 (on Ethernet devices) or Control Words. **Ingress Packets** specifies the amount of packets received since simulation was started. **Ingress Bytes** specifies the total number of bytes received since simulation was started. **Egress Packets** specifies the amount of packets transmitted since simulation was started. **Egress Bytes** specifies the total number of bytes transmitted since simulation was started. **Config Gbps** specifies the rate a specific port was configured for. **Ingress Gbps** specifies the current rate at which packets are currently being received. **Egress Gbps** specifies the rate at which packets are currently being transmitted.

The **Auto Refresh** check box enables automatically refreshing the displayed data every second.

2.12 Debugging

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

When in **Simulation mode**, the NPF6000 and NFP4000 simulator provides debugging support to Programmer Studio. In **Hardware mode**, the Debug Server, running on the NFP's ARM processor or on an x86 host which communicates with the NFP via a PCIe interface, facilitates debugging. It relays debugging operations between Programmer Studio and the Microengines.

Programmer Studio 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 and registers.

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 2.1 summarizes the debugging features available in Hardware and Simulation modes.

Table 2.1. Simulation and Hardware Mode Features

Feature	Simulation	Hardware
System Configuration	X	
Starting and Stopping Debug	X	X
Command Line Interface	X	X
Script Files	X	X
Command Scripts	X	
Thread Windows		
• Display Microword Address	X	X
• Instruction Markers	X	X
• View Instructions	X	X
Run Control	X	
Breakpoints	X	X ¹
Examine Registers	X	X
Watch Data		
• Enter New Data Watch	X	X
• Watch ME and Chip CSRs	X	X
• Watch GPRs and XFER	X	X
• Deposit Data	X	X ¹
Watch Memory	X	X
• Break on Data Change	X	
Watch NBI Memory	X	X
Watch Scripts	X	
• Break on Data Change	X	
ME Performance Statistics	X	
Execution Coverage	X	
Thread History	X	
Queue History	X	
Queue Status	X	
Thread Status	X	X
Packet Streaming Statistics	X	
NBI PM Modification Pipeline	X	
NBI TM Packet Descriptor	X	

Feature	Simulation	Hardware
Performance Statistics	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. You have to 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.

Programmer Studio supports debugging in three different configurations:

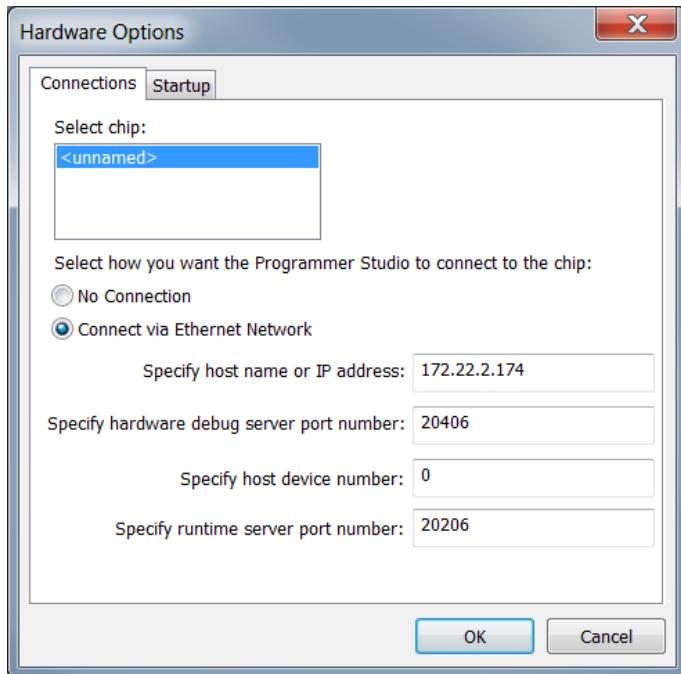
Mode	NFP6000/NFP4000	Comments
Local Simulation	Default. No special setup necessary.	Programmer Studio and the simulator both run on the Windows platform.
Remote Simulation	See Section 2.12.3.7.	Programmer Studio runs on a Windows host and communicates over a network with a subsystem containing Network Processor simulator. Network Processor simulator can run either on Windows or Linux platform.
Hardware	See Section 2.12.1	Programmer Studio runs on a Windows host and communicates over a network with a subsystem containing an actual network processor.

2.12.1 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 via Ethernet** - You must specify the name of the node (IP address) where the hardware is located. As well as hardware debug server's port, device number and RTE port (only enabled for P4 and Managed C projects) of the NFP device. By default node port is set to 20606, device number is set to 0 and RTE port is set to 20206.



Note

The hardware debug server must be running on the subsystem containing an actual network processor before trying to connect to the subsystem. For more information on how install and run debug server, please refer to the *Debug Server User's Guide*.



Note

For P4 projects, the runtime server (RTE) must be running on the subsystem containing an actual network processor before trying to connect to the subsystem. For more information on how install and run runtime server, please refer to the Appendix D of this document.

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 A.6).

Stopping

To stop debugging:

- 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 2.22).

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 2.12.6.

By default, the stage 5 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, the instruction marker points to it. If the line is hidden because it is in a collapsed macro, 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. Programmer Studio 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.

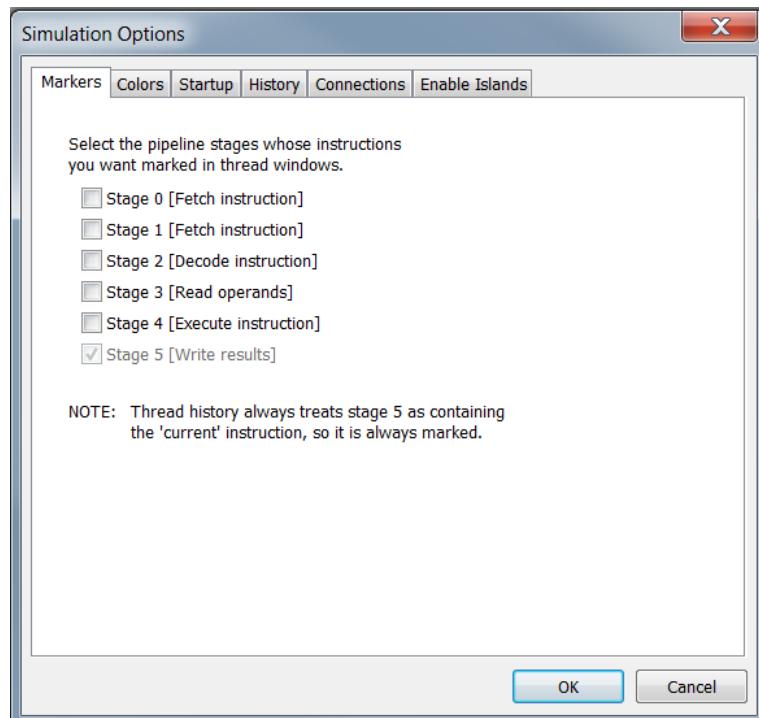


Figure 2.22. Marking Instructions for the Network Processor

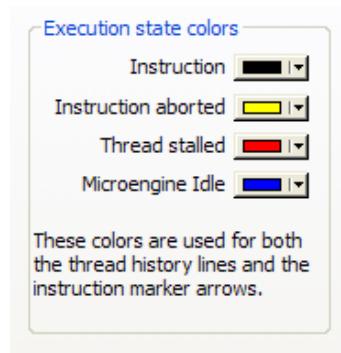
For more information on instruction markers, see Section 2.12.3.2, Section 2.12.6.9 and Section 2.12.6.11.

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 Network Flow Simulator 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.

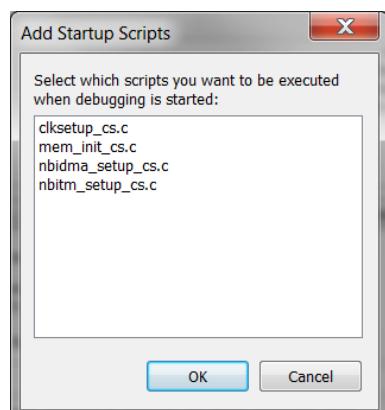
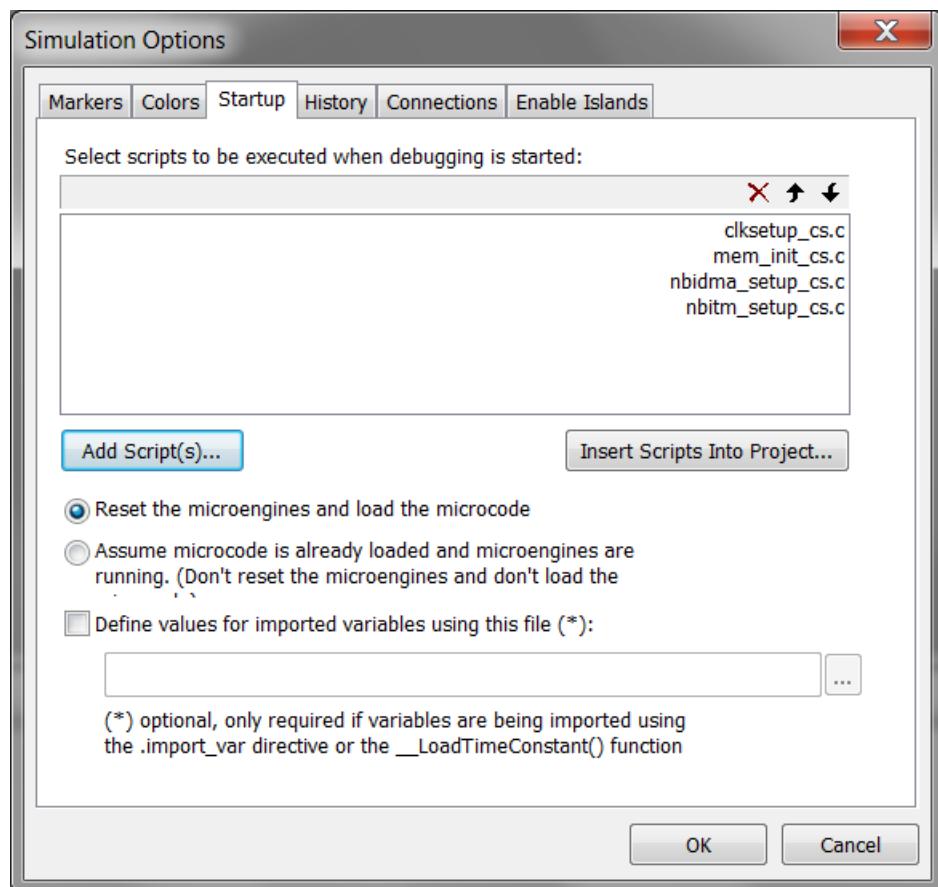
This property page specifies how Programmer Studio behaves when you start debugging and when you reset the simulation.

Startup Scripts

To have Programmer Studio 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

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.

Programmer Studio 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 also delete a script from the list with the **Delete** button.

In addition to **Startup** scripts, Programmer Studio also supports running scripts during simulation using command line tools. For more information on how to run scripts while simulation is in progress refer to Chapter 7 in this document.

2.12.3.4 Using Imported Variable Data

You 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 Compiler. These define a variable that can then be associated with a 64-bit signed integer value at prior to loading. In simulation, this association is done through an imported variable data (.ivd) file that gets processed by the loader for NFFW files. A simple Key-Value (KV) file format is also supported and recommended for new projects. If a filename does not end with .ivd, it will be assumed to be a KV file. See Section 7.1.3 and Section 7.1.4 for file formats. Alternatively, the `nfiv` command line tool can be used to manipulate the import variables of the NFFW file before loading the NFFW file on hardware or as a manual intermediate step in debugging between linking and starting debugging. On hardware, the BSP provides `nfp-mefw` to manipulate import variables prior to loading on hardware. Refer to Section 7.1.

The user specifies the path for the IVD or KV file using the console function, `loadImportVarData()`, which is defined by the loader. This function must be called before the `load_mefw()` console function is called.

Programmer Studio allows you to specify the IVD or KV file as one of the simulation startup options. If you select **Options** from the **Simulation** menu and then select the **Startup** tab, the property page appears. If you specify a file, Programmer Studio automatically invokes the `loadImportVarData()` console function at the appropriate time in the startup sequence.

Programmer Studio also allows you to specify the IVD or KV file as one of the hardware startup options. When you click **Start Debugging** in hardware mode, Programmer Studio opens the IVD or KV file. As it is loading microcode into a chip, it parses the file and relocates the microengine firmware (resolves import expressions and performs data patch-up). Note that if you select the Hardware option of assuming that the microcode is already loaded, relocation is not performed, ignoring the specified file.

See Section 7.1.3 and Section 7.1.4 for import variable file formats.

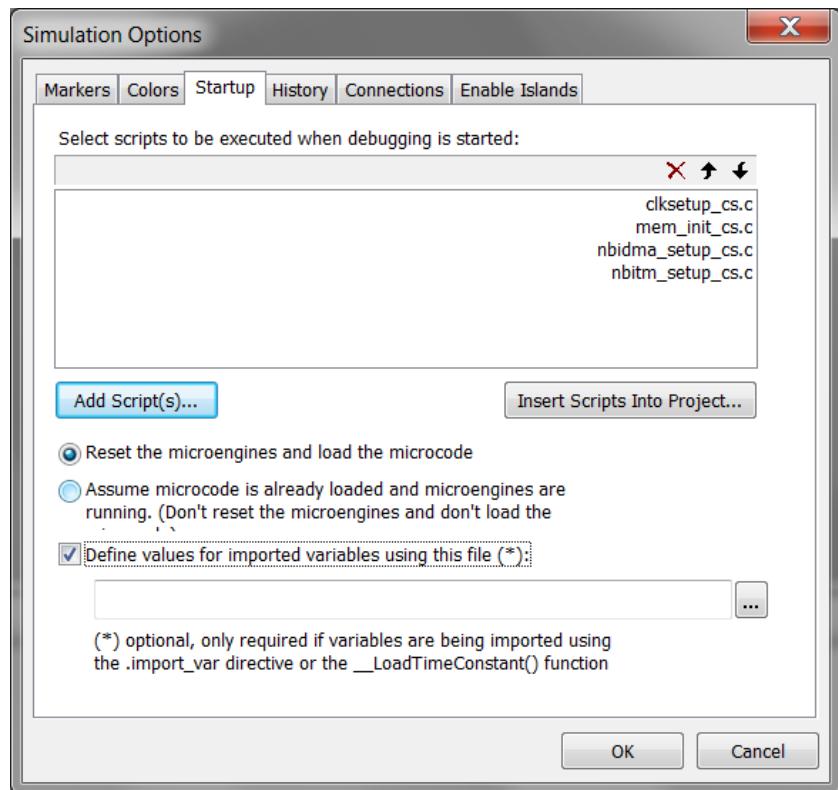


Figure 2.23. Using Imported Variable Data at Startup in Simulation Mode

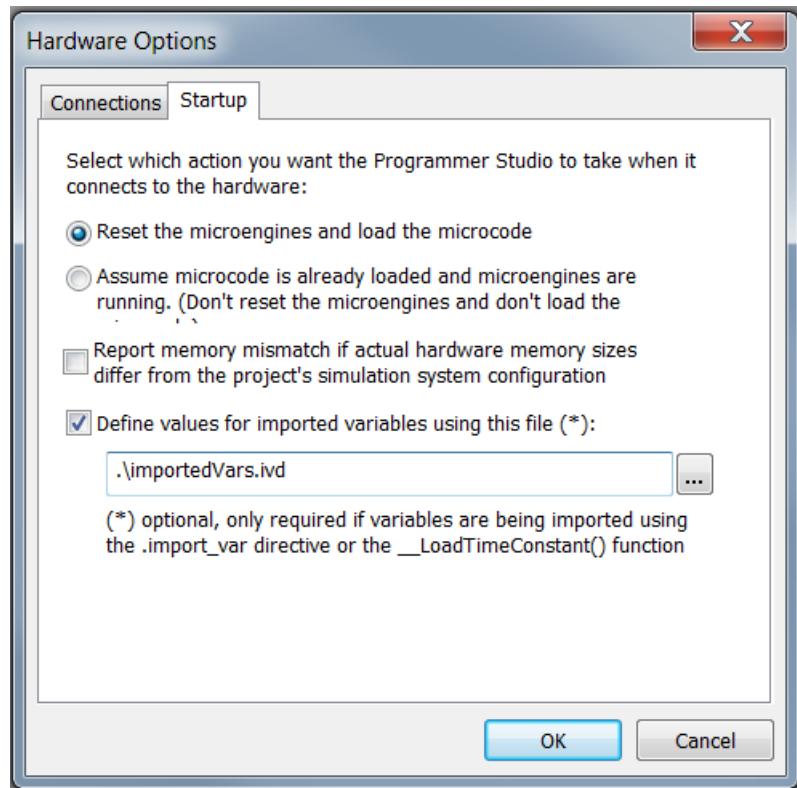


Figure 2.24. Using Imported Variable Data at Startup in Hardware Mode

2.12.3.5 Specifying Simulation Startup Options

To specify whether or not Programmer Studio loads microcode when simulation is started:

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

The **Simulation Options** dialog box appears.

2. Click the **Startup** tab.
3. Select or clear the action that you want Programmer Studio to take when it connects to the simulator:

- **Reset the microengines and load the microcode**

This option causes Programmer Studio 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 Programmer Studio to connect only to the debug library on the Simulator. The Microengines are not affected in any way. This would be useful if you want to connect to a running simulation system to examine its state.

4. Click **OK**.

If you choose not to have Programmer Studio 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, seeSection 2.2.3.4).

2.12.3.6 History Collecting

You enable or disable History collecting through the **History** dialog box (see Figure 2.25).

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**.

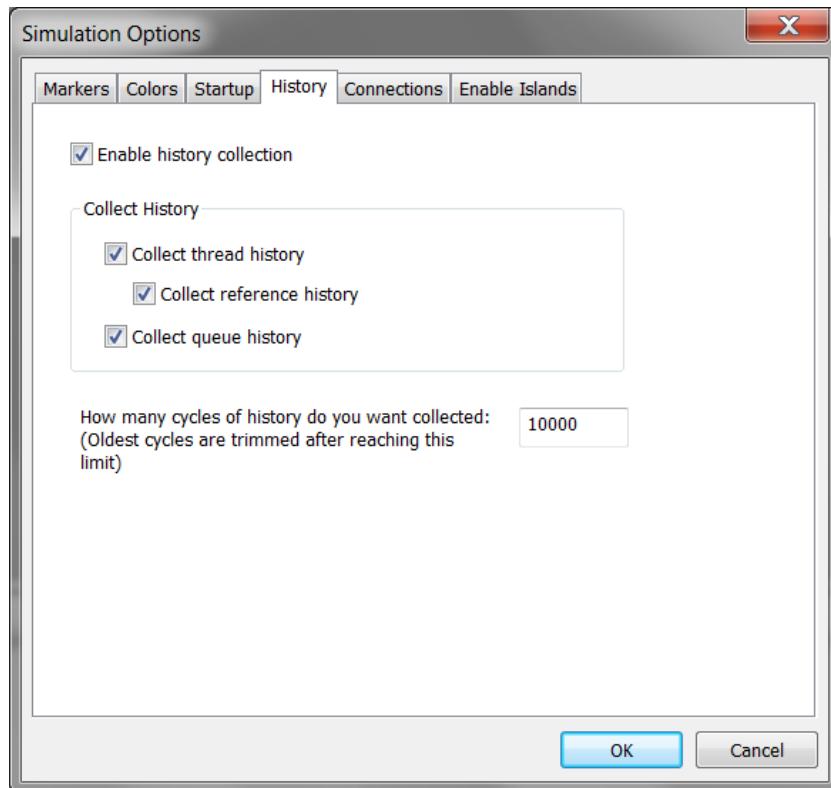


Figure 2.25. History Dialog Box

4. Select:

- **Collect thread history**, or
- **Collect reference history**, or
- **Collect queue history**, or
- Any combination, depending on what history you want collected.



Note

You cannot select **Collect reference history** if **Collect thread history** is not selected. These options must be specified before you start debugging. If you are already in debug mode, these selections are disabled.

5. Specify how many cycles of history you want collected by typing a number in the corresponding box. The maximum number of cycles that can be collected is 1,000,000. However at 1 million cycles, significant Programmer Studio process virtual memory is required for a typical reference design. For low performance hosts, a more reasonable setting is 100,000 cycles.

Disable

To disable history collecting:

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

The **Simulation Options** dialog box appears.

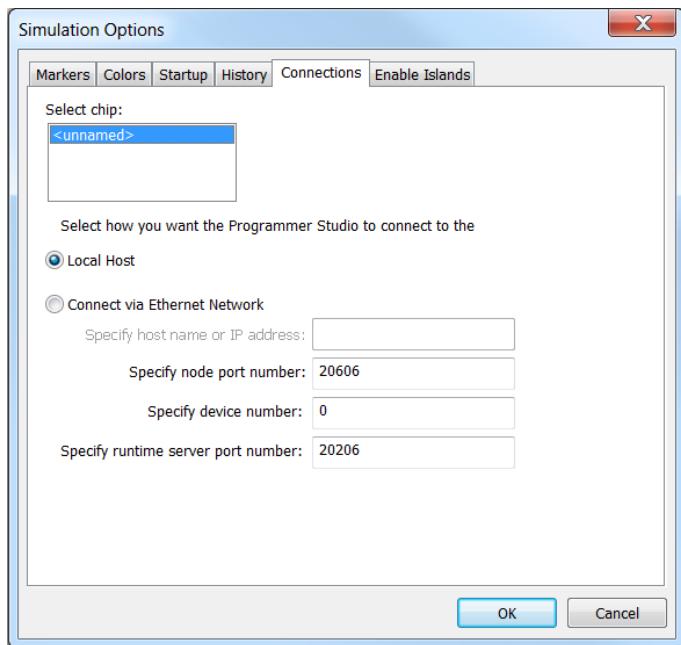
2. Click the **History** tab.
3. Clear **Enable history collecting**.

2.12.3.7 Specifying Simulation Connection Options

To debug in simulation mode, you must specify how to connect to the subsystem(s) containing the network processor simulator:

1. On the **Debug** menu, select **Simulation**.
2. On the **Simulation** menu, click **Options**.

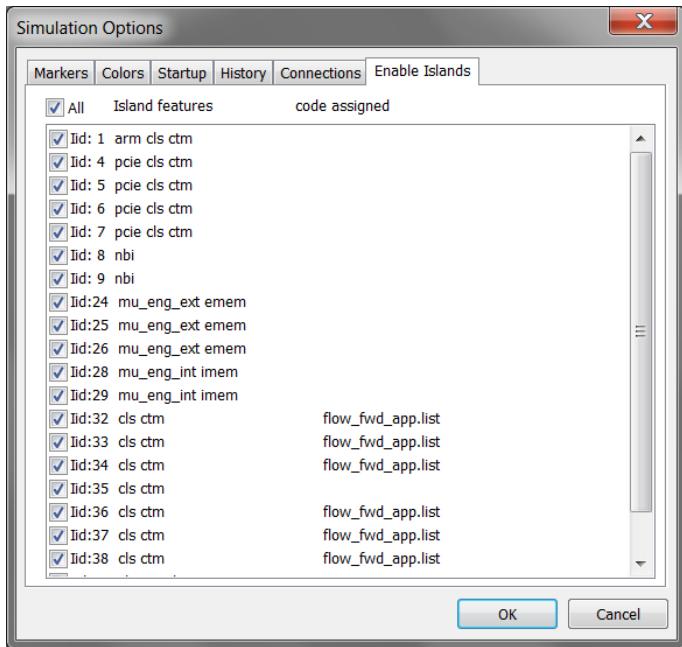
The **Simulation Options** dialog box appears.



3. Click the **Connections** tab (if its not already selected).
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:
 - **Local Host** - If you are running the simulator on the local machine.
 - **Connect via Ethernet Network** - You must specify the name of the node (IP address) where the simulator is located. As well as node port, device number and RTE port (only enabled for P4 and Managed-C projects) of the NFP device. By default node port is set to 20606, device number is set to 0 and RTE port is set to 20206.

2.12.3.8 Enable Simulated Islands Clock

This page enables users to enable and disable clocks for different Island. Simulator enables the Island by enabling the clock to that Island. To enable an specific Island, click on the check box for that Island. To disable the clocks to an specific Island, un-check the check box for that Island.



2.12.4 Exporting the Startup Script

To create a text file containing all the commands that Programmer Studio sends to the Network Flow Simulator 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. This startup script may be used to configure the Network Flow Simulator when it is run via the command line without Programmer Studio.

2.12.5 Changing Hardware Options

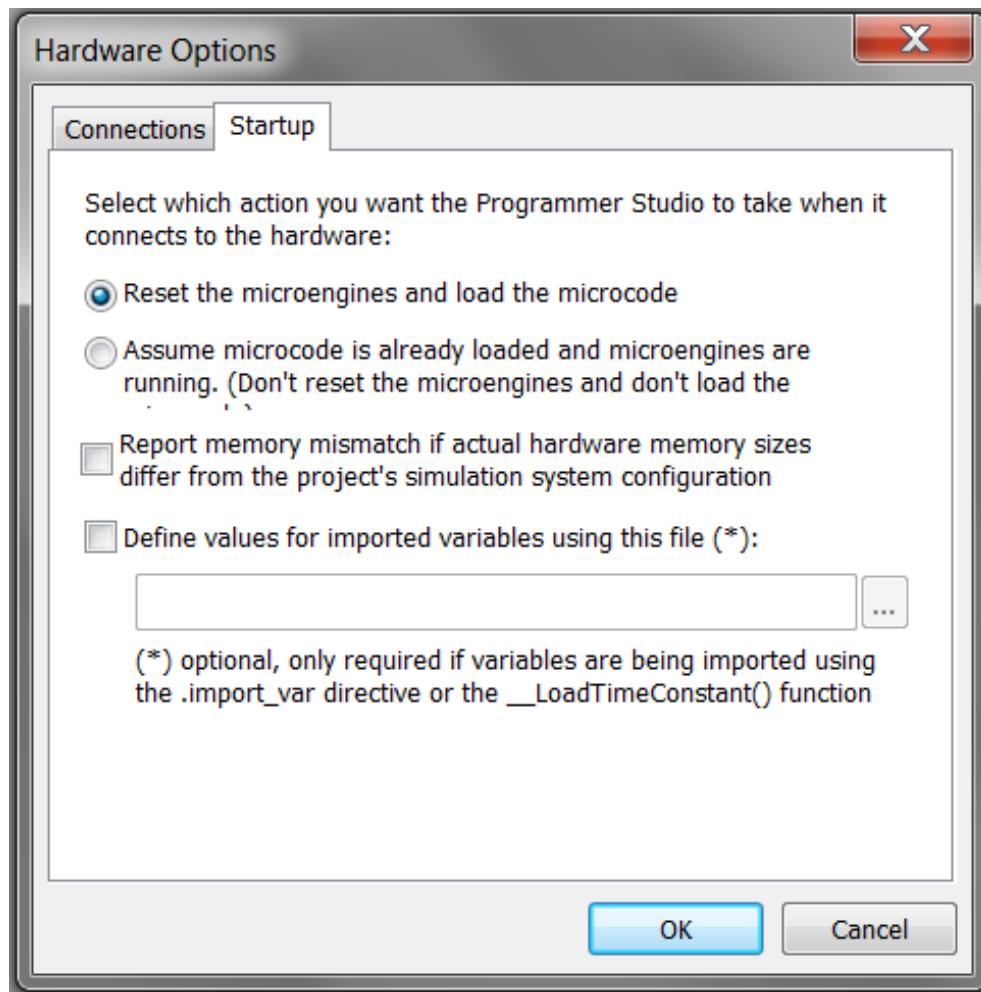
2.12.5.1 Specifying Hardware Startup Options

To specify whether or not Programmer Studio 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.



3. Select or clear the action that you want Programmer Studio to take when it connects to the hardware:

- **Reset the microengines and load the microcode**

This option causes Programmer Studio 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 Programmer Studio to connect only to the debug library on the Netronome ARM processor. 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 Programmer Studio load microcode automatically at startup, you can:

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

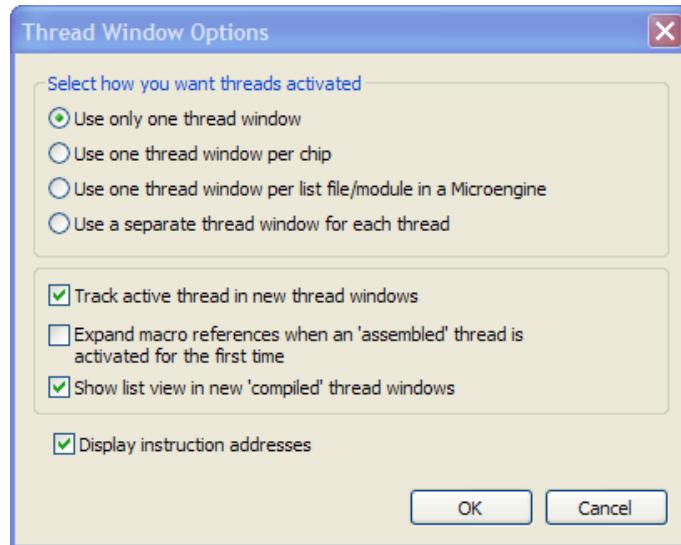
(You can also click the  button. This button is not on the default **Build** menu. To put this button there, see Section 2.2.3.4)

2.12.6 Thread Windows

Thread windows differ from normal document windows in that they have a toolbar across the top (see Figure 2.26 and Figure 2.27). Setting and clearing breakpoints (see Section 2.12.8), displaying register or variable contents (see Section 2.12.10), and viewing the instruction(s) currently being executed (see Section 2.12.6.11) are all done in thread windows.

2.12.6.1 Controlling Thread Window Activation

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



To do this, while in the debug mode:

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.**

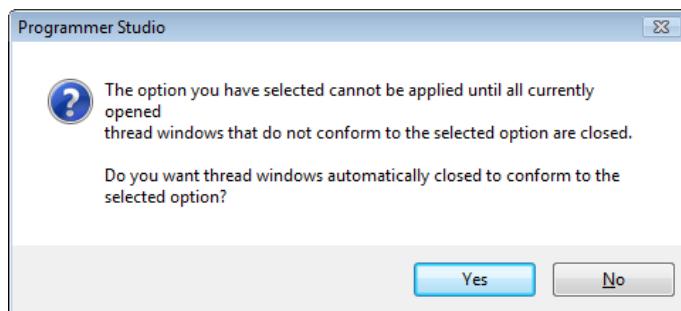
Reuse a currently open thread window only if it displays a thread in the same Microengine as the thread being activated.

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 windows 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 options (a) or (b).
- If you have two or more thread windows open for different threads in the same Microengine, you cannot select options (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**, Programmer Studio 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:

1. Select **Track active thread in new thread windows** if desired (not available if you selected to view each thread in its own window).

2. Select **Expand macro references when an ‘assembled’ thread is activated for the first time** if desired.
3. Select **Show list view in new ‘compiled’ thread windows** if desired.
4. Select **Display instruction addresses** if desired (in list view only).
5. Click **OK** when done.

2.12.6.2 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 2.12.8), displaying register contents (see Section 2.12.10), and viewing the instruction(s) currently being executed are also done in assembled thread windows.

```

261      sram[pop, $1008!pop_xfer0, $1008!pop_xfer0, 0, 0], sig
        .endif
        port_rxdy_chk(@rdready_inflight, rec_req);
        .local $1009!rec_rdy
        critsect_enter[@rdready_inflight] ; block others
1001_01#:
1001_end#:
m013_check_port#:
    m013_check_port_begin#:
        alu[--, --, b, @rdready_inflight]
        br<0[m013_check_port_end#], guess_branch
        ctx_arb[voluntary]
        br[m013_check_port_begin#]
m013_check_port_end#:
    csr[read, $1009!rec_rdy, rcv_rdy_lo], defer[1], ctx_swap
; BRANCH LATENCY FILL OPTIMIZATION: the uword below was "pushed" dot
    immed[@rdready_inflight, 0]
    critsect_exit[@rdready_inflight] ; allow cheats
    ...

```

Figure 2.26. The Assembler Thread Window

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

- Chip list box: This control is not visible if your project has only one chip.
- 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.
- 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 2.12.6.3).

 **Display the Thread Window Options dialog box** (see Section 2.12.6.1).

 **Step Over** (see Section 2.12.7.3).

 **Run to Cursor** (see Section 2.12.6.4).

 **Expand macros** (see Section 2.12.6.7).

 **Collapse macros** (see Section 2.12.6.7).



Note

Not all buttons are available in hardware mode.

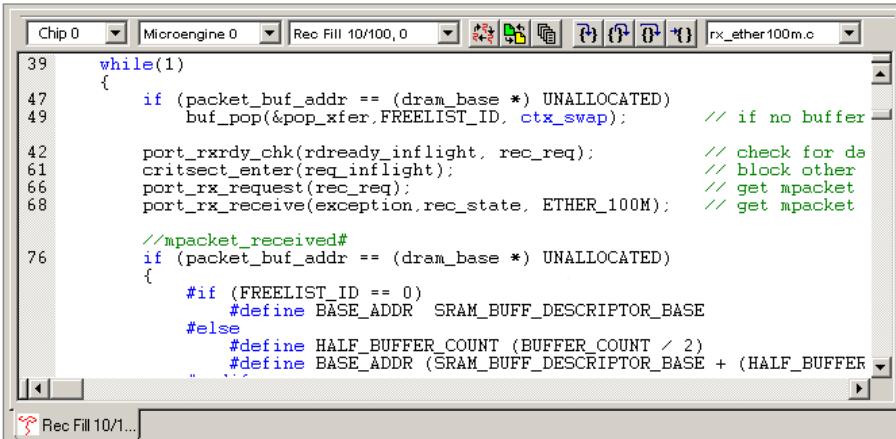
Compiled Thread Windows

Compiler thread windows look the same as Assembler thread windows but have some differences. Display options are similar (see Section 2.12.6.1).



Note

Setting and clearing breakpoints (see Section 2.12.8), displaying register contents (see Section 2.12.10), and viewing the instruction(s) currently being executed are also done in Compiler thread windows. They cannot be performed in the source file windows.



The screenshot shows a window titled "rx_ether100m.c". The code displayed is:

```
39 while(1)
{
    if (packet_buf_addr == (dram_base *) UNALLOCATED)
        buf_pop(&pop_xfer,FREELIST_ID, ctx_swap); // if no buffer
    port_rxrdy_chk(rdready_inflight, rec_req); // check for da
    critsect_enter(req_inflight); // block other
    port_rx_request(rec_req); // get mpacket
    port_rx_receive(exception,rec_state, ETHER_100M); // get mpacket
    //mpacket_received#
    if (packet_buf_addr == (dram_base *) UNALLOCATED)
    {
        #if (FREELIST_ID == 0)
            #define BASE_ADDR SRAM_BUFF_DESCRIPTOR_BASE
        #else
            #define HALF_BUFFER_COUNT (BUFFER_COUNT / 2)
            #define BASE_ADDR (SRAM_BUFF_DESCRIPTOR_BASE + (HALF_BUFFER
    }
```

Figure 2.27. The Compiled Thread Window

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

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

- 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, Programmer Studio changes the displayed source file to the one that generated the current instruction.

The toolbar contains following buttons:



Track the active thread (see Section 2.12.6.3).



Display the **Thread Window Options** dialog box (see Section 2.12.6.1).



Step Into (see Section 2.12.7.4).



Step Out (see Section 2.12.7.5).



Step Over (see Section 2.12.7.3).



Run to Cursor (see Section 2.12.6.4).



Toggle View (see Section 2.12.6.5).



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.

2.12.6.3 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 should be opened with active thread tracking enabled by selecting or clearing **Track active thread in new thread windows**.

2.12.6.4 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, the simulation runs until the first generated instruction is reached.

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

2.12.6.5 Toggle View

When debugging a C or a P4 or a mixed C and P4 or a mixed C and assembler Microengine project, the Thread Window can toggle between the Source View, the P4 Source View and the List View.

There are two ways to do this:

1. When in **List View**, use the **Select Source Combo** to automatically switch to one of the available source files.
2. When in **P4 Source View**, use the **Select Source Combo** to automatically switch to one of the available C source files.
3. Click the  button in the **Thread window**.

While in **List View**, the **Toggle View** button behaves as follows:

- When the button is enabled, the **Thread Window** switches to **C Source View** and the cursor is placed on the source line that generated the list line on which the cursor was located.

While in **P4 Source View**, the **Toggle View** button behaves as follows:

- When the button is enabled, the **Thread Window** switches to **C Source View** and the cursor is placed on the source line that generated the P4 Source code line on which the cursor was located.

While in **C Source View**, depend on the last view of the **Thread View**, the **Toggle View** button behaves as follows:

- When the last view of **Thread View** was the **P4 Source View** and when there is no number in the margin, clicking the **Toggle View** button causes the **Thread Window** to switch to **List View** and puts the cursor on line 1.
- When the last view of **Thread View** was the **P4 Source View** and when there is a black number in the margin, the **Toggle View** button causes the **Thread Window** to switch to **List View** and the cursor is placed on the line corresponding to the one that you clicked.
- When the last view of **Thread View** was the **List View** and when there is no number in the margin, clicking the **Toggle View** button causes the **Thread Window** to switch to **P4 Source View** and puts the cursor on first instruction in the source.

- When the last view of **Thread View** was the **List View** and when there is a black number in the margin, the **Toggle View** button causes the **Thread Window** to switch to **P4 Source View** and the cursor is placed on the line corresponding to the one that you clicked or to the line that is closest to the corresponding line in **C Source View**.

2.12.6.6 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.

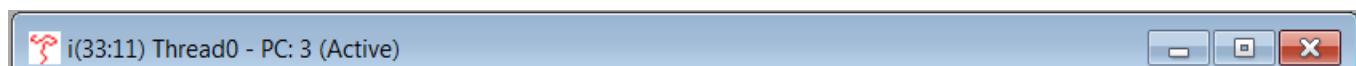
The thread window is implicitly activated by:

- **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 — Programmer Studio 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 is not 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 is not 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.

2.12.6.7 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 2.28).

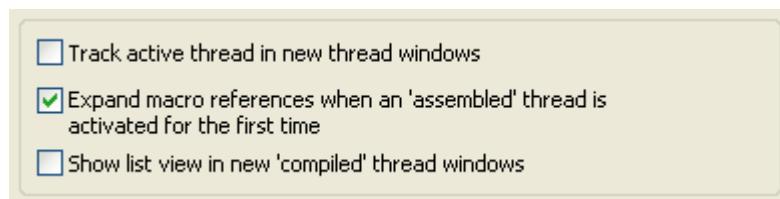
First Time Thread Activation

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

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

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

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



When you re-activate an assembled thread, Programmer Studio 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.

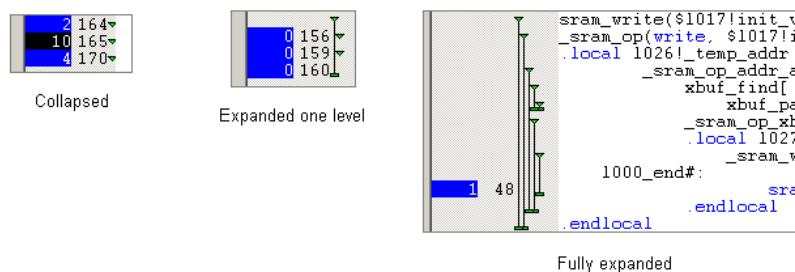


Figure 2.28. Expanding Macros

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**.

Programmer Studio:

- 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.

2.12.6.8 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.

2.12.6.9 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 2.2:

Table 2.2. 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 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.

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.

2.12.6.10 List and Source View Microaddress Markers

Programmer Studio provides markers to identify related microcode and source code in the List view and Source view windows. Microaddress markers in the List view are displayed in differing colors, and those in the source view are highlighted with colored symbols (see Table 2.3).

Where there is a direct relationship, microaddress markers in the List View are displayed in black and toggling between the two views moves the cursor to the related lines.

In views where microcode has been generated but the microaddress is not breakable in the source view, the microaddress markers are displayed in red. Toggling between views does not move the cursor to a related line.

In the List View, where microcode has been generated by the compiler, but does not map directly to a line in a C source file, the microaddress markers are displayed in dark green. As there is not a direct relationship to a line in a source file, toggling between the views opens a blank Source view window.

Table 2.3 displays sample List View and Source View markers.

Table 2.3. List View Markers

List View Symbol	Source View Symbol ^a	Function
0	{+} {*}	In the List view it indicates microcode that relates directly to a line of C code in the Source view. When you toggle between the views the cursor maps to the related line in each view. {+} indicates that it is breakable and not inlined. {*} indicates that it is breakable and inlined. These markers display in black.
1	{-}	In the List view it indicates microcode has been generated but the microaddress is not breakable in the Source view. When you toggle between the views, the cursor maps to a line near to where the code was generated. {-} indicates that it is not breakable. These markers display in red.
2	None	In the List view it indicates where microcode has been generated by the compiler, but does not map directly to a line in a C source file. When you toggle between the views, a blank Source view window is displayed. You can then use Source File Selection group box to select a Source file to view. This symbol displays in green.

^aIf the program counter maps to a statement with one of these symbols, the symbol is replaced with the address.

2.12.6.11 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, Programmer Studio 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 has 6 Pipe Stage markers, one on each of the 6 instructions. Each marker has a different arrowhead filled with the appropriate color. For example, if an instruction is executing in stage 3, then its marker has the stage 3 arrowhead filled with black with all other arrowheads unfilled. If an instruction is aborting in stage 2, then its marker has 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 has multiple arrowheads filled in, possibly with different colors. This also means that there are 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 shows 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 shows only a single arrowhead, corresponding to stage 4. The other thread marker shows 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.

2.12.6.12 Document and Thread Window History

Programmer Studio 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, select **Back** from the **Window** menu. To move forward through the history, select **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 you go **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, is deleted. For example, assume the history contains windows A, B, C, D, E and F. If you go back to C, then execute a **Go To Macro** command, windows D, E and F are deleted from the history.

2.12.6.13 Instruction Execution Tracing

The Instruction Execution Tracing feature lets you rapidly navigate backward and forward through thread/PC history from the current point in time (the debug “cycle of interest”) to some other point in time when a designated instruction was executed by the microengine or thread associated with the focused Thread Window.

When you right-click on a Thread Window line that is associated with an instruction address, the popup context menu contains an option to “Trace selected PC nn instruction execution”. When you select this item, a submenu appears with 8 options:

- Earliest by this microengine
- Earliest by this thread
- Previous by this thread
- Previous by this microengine
- Next by this microengine
- Next by this thread
- Latest by this thread
- Latest by this microengine.

Based on the item selected, the debug cycle of interest is changed backward or forward in time to the earliest, previous, next, or latest time when the specified instruction was executed in the thread or microengine. If the instruction was never executed, was not executed before/after the current debug cycle of interest, or if the instruction was executed before the current cut-off of history, then an error message appears indicating that the instruction execution time could not be located.

Using this feature, you can quickly establish whether and when certain key instructions have been executed (for example error handling code) and examine the historical state of registers and memory at that point in time.

2.12.6.14 Instruction Operand Tracing

The Instruction Operand Tracing feature provides the ability to move backward and forward through instruction execution history, following the dependencies between source and destination operands. This allows you to rapidly move to the instruction where a particular source operand was last written, allowing you to easily follow packet processing decisions.

Tracing is available through the Programmer Studio thread window list view while debugging in simulation mode. The following register types are supported by instruction operand tracing:

- GPRs
- Neighbor Registers
- Transfer Registers
- Local Memory.

The Instruction Operand Tracing feature is activated when **Collect reference history** is checked. This check box is under **History** tab when **Simulation->Options...** menu is selected as shown in Figure 2.29.

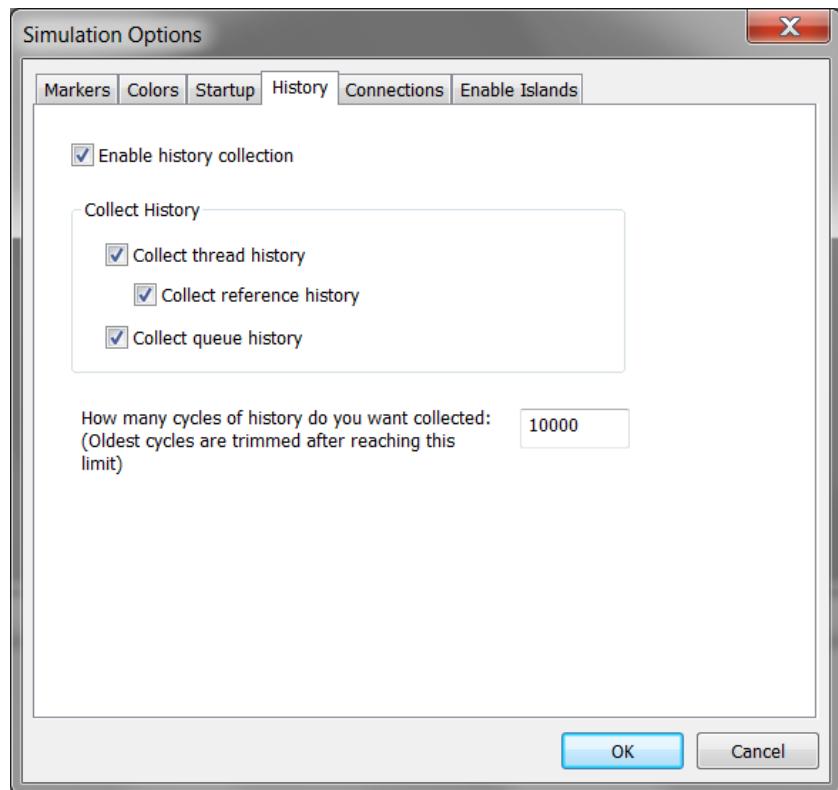


Figure 2.29. Simulation Options window

If you have stopped the simulation or if a breakpoint has been reached, then the thread window marks the next instruction to be executed. If you have already stepped back through the microengine history to a previous cycle, then the thread window marks the instruction that is about to be executed at that cycle. In either case, the marked instruction is considered the “instruction of interest” and the simulation cycle for the marked instruction is considered the “cycle of interest”.

Figure 2.30 presents an example of instruction operand “racing where the instruction of interest is “alu[var1, var2, +, var3]”. The value of source operand “var2” can be traced back to previous write. The value of destination operand “var1” can be traced forward to the next use.

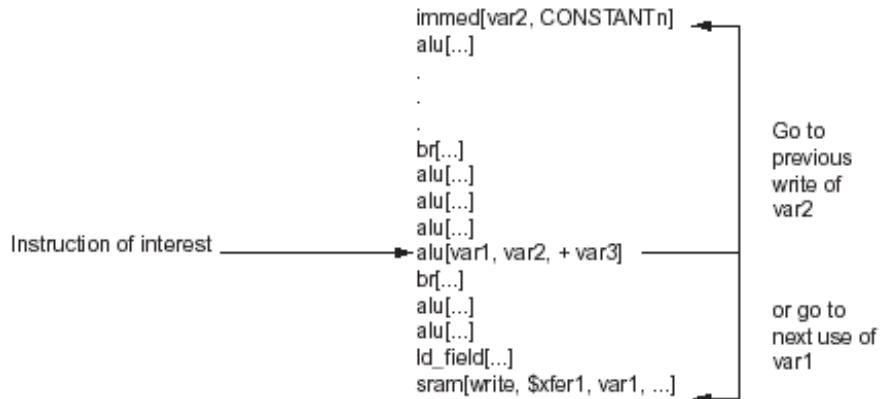


Figure 2.30. Instruction Operand Tracing

When you right-click in the thread window list view, a popup menu appears, as shown in Figure 2.31. The popup menu contains a new option called “Trace PC nn instruction operands” which produces a submenu containing some of the following options, depending on the instruction type being hovered over (I/O or not) and the number of registers of various types used or written by the instruction:

Go to previous write of source src1RegAddr (src1RegName)	
Go to next read of source src1RegAddr (src1RegName)	
Go to previous write of source src2RegAddr (src2RegName)	
Go to next read of destination destRegAddr (destRegName)	← not shown for I/O instructions
Go to previous write of write xfer reg xferRegAddr (xferRegName)	← shown for I/O write instr, iterated ref_cnt times
Go to next read of read xfer reg xferRegAddr (xferRegName)	← shown for I/O read instr, iterated ref_cnt times

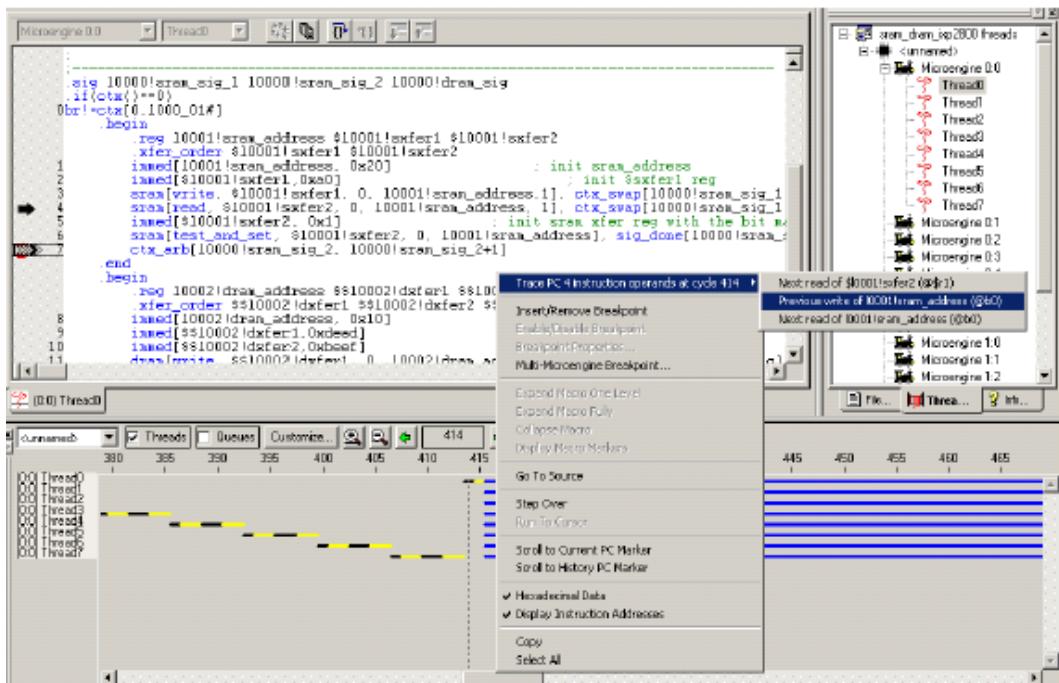


Figure 2.31. Thread window list

There can be zero or one destination registers shown and between zero and two source registers shown. For I/O instructions, the list of referenced transfer registers is shown. If a destination or source operand is not specified (e.g. destination was “--” in an alu), or is not meaningful for the instruction (e.g. immed instruction has no source), then the corresponding option is not shown in the submenu. For I/O instructions with reference count greater than one, multiple transfer register items are shown in the submenu.

Instruction operand tracing is not available at the current debug cycle since the marked instruction has not been executed yet. Sometimes when navigating forward through history, a read match for an operand cannot be found. This is usually caused because the operand has not been read before reaching the current debug cycle.

2.12.7 Run Control

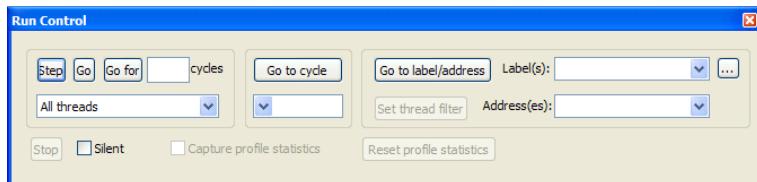
Run Control lets you govern execution of the Microengines. Different control operations are available from Programmer Studio depending on whether you are in Simulation or Hardware mode (see Table 2.1).

In Simulation mode, Programmer Studio 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.

2.12.7.1 Single Stepping

Single stepping has four variations:

- | | |
|-------------------------|---|
| Microengine Step | Performed on Microengines (see Section 2.12.7.2). |
| Step Into | Performed on a single thread in a compiled thread window only (see Section 2.12.7.4). |
| Step Over | Performed on one thread (see Section 2.12.7.3). |
| Step Out | Performed on a single thread in a compiled thread window only (see Section 2.12.7.5). |

2.12.7.2 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

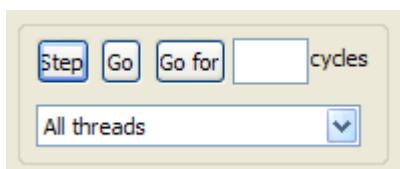
Press SHIFT+F10, or

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.

2.12.7.3 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 the window where 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.

2.12.7.4 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.

2.12.7.5 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

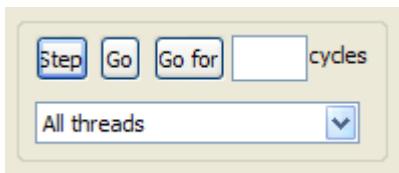
Stepping out is not supported when debugging hardware.

2.12.7.6 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**.



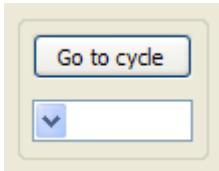
Note

Executing multiple cycles is not supported when debugging hardware.

2.12.7.7 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.

2.12.7.8 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.

2.12.7.9 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 2.12.7.10).

2.12.7.10 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 continues to run. Programmer Studio displays a message indicating which microengines did not stop.

2.12.7.11 Resetting the Simulation

Reset executes a Network Flow Simulator sim_reset command, which puts the simulation back to the state after the original initialization was done. After the reset, Programmer Studio re-executes the options specified by the Startup page of the **Simulation Options** dialog box. However, if you specified that Programmer Studio should initialize the model, it does not 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 Network Flow Simulator sim_reset command, which puts the simulation back to the state after the original init was done. After the reset, Programmer Studio re-executes the options specified by the **Simulation Startup** page of the **Simulation Options** dialog box. However, if you specified that Programmer Studio should initialize the model, it does not repeat the chip and memory definition commands and the init command since they are unnecessary.

2.12.8 About Breakpoints

The Breakpoints window is activated through the **View**→**Debug Windows**→**Breakpoints** path. This feature allows you to look at the breakpoints set in the currently open project. It allows you to modify the status of breakpoints, remove them, and sort them by a certain criteria to display a subset of breakpoints.

Figure 2.32 is a screenshot of the **Breakpoints** window.

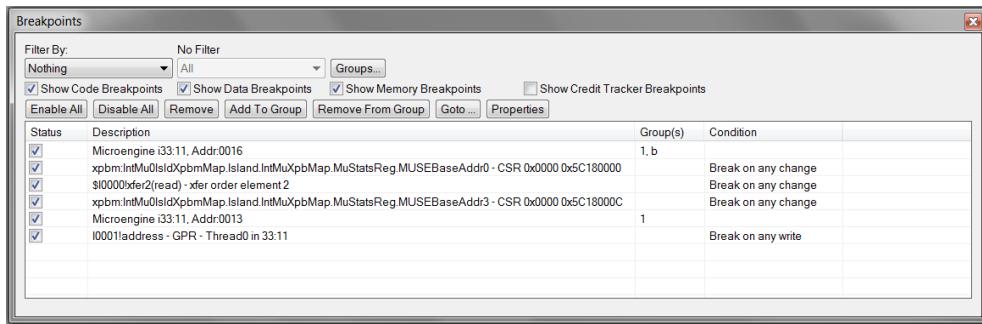


Figure 2.32. Breakpoints window

2.12.8.1 Filtering Breakpoints

You can select criteria to display breakpoints that match the criteria. Available breakpoints can be filtered using the following categories.

Filter Categories

1. **Filter By** dropdown menu option is the primary filter. It offers the following selections:
 - Group...
 - Chip - the available chips are listed by name.
 - Nothing.
2. The **SubFilter** dropdown menu option title changes according to the primary filter (Filter By) selection. This option is therefore the secondary filter. This option has three states:
 - If '**Nothing**' is chosen at the primary filter Filter By dropdown menu, the label of the secondary filter changes to 'No Filter' and it is disabled.
 - If '**Group...**' is chosen, the label changes to 'Select Group' and the list is populated with the names of the available breakpoint groups. Only the breakpoints in the selected group are displayed in the breakpoint list.
 - If '**Chip X**' is chosen, the label changes to 'Select List File/Module', and the dropdown menu is populated with the available list files or modules for the selected chip. Only the breakpoints in the selected list file or module are displayed in the breakpoint list. No memory breakpoints are displayed.
3. Checking the **Show Code Breakpoints** box, displays breakpoints that are code type breakpoints that match the filter criteria. In other words, those breakpoints that are placed inside the actual code in list view or source view are displayed.
4. Checking the **Show Data Breakpoints** box, displays breakpoints that are data type breakpoints that match the filter criteria. In other words, those breakpoints that are set in the Data Watch window are displayed.
5. Checking the **Show Memory Breakpoints** box, displays breakpoints that are memory type breakpoints that match the filter criteria. In other words, those breakpoints that are set in the memory watch window are displayed.

2.12.8.2 Breakpoint List and Operations

You can modify one or more breakpoints listed according to the filter criteria. Figure 2.33 displays the buttons provided for modifying breakpoints.



Figure 2.33. Breakpoint Buttons and Properties

2.12.8.2.1 Breakpoint Properties

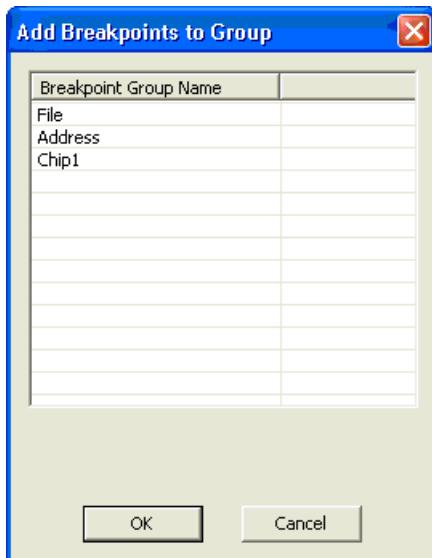
The properties according to which breakpoints are categorized in the listbox are:

1. Status: Indicates whether the breakpoint is enabled or disabled.
2. Description: Contains a description of the breakpoint.
3. Group(s): Displays the Group(s) that a particular breakpoint belongs to.
4. Condition: Displays the condition under which the breakpoint is activated, provided the breakpoint is conditional.

2.12.8.2.2 Breakpoint Operations

The possible operations that can be performed on the breakpoints are listed below:

1. **Enable All:** Click this button to enable all of the breakpoints visible in the listbox.
2. **Disable All:** Click this button to disable all of the breakpoints visible in the listbox.
3. Remove:
 - a. Select the breakpoints to be removed.
 - b. Click the **Remove** button to remove the selected breakpoint(s).
 - c. Alternatively, select the breakpoints and hit the **Delete** key on the keyboard.
4. Add To Group:
 - a. Select the breakpoints to be added to a group.
 - b. Click the **Add To Group** button to bring up the dialog box that lets you select or create a new breakpoint group.
 - c. Click **OK**. The breakpoints that were selected are added to the selected group.



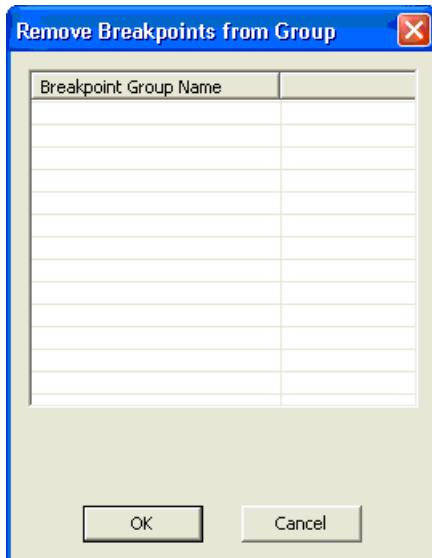
5. Remove From Group:

- Click the **Remove From Group** button to display a dialog box that lets you see the group(s) that the currently selected breakpoints are in.
- Click **OK** to remove the selected breakpoints from the group selected.



Note

The groups available for selection are limited to the groups common to the breakpoints that are selected.



6. **Go To Breakpoint:** Clicking this button takes you to the line of code where a selected breakpoint exists.

- a. If a code breakpoint is selected, the cursor goes to the line in the Thread Window where the breakpoint is set.
 - b. If a data breakpoint is selected, the Data Watch window opens and the line on which the breakpoint exists is highlighted.
 - c. If a memory breakpoint is selected, the Memory Watch window opens and the line on which the breakpoint exists is highlighted.
 - d. Alternatively, double-click on the breakpoint in the listbox to display the line in which the breakpoint is present.
7. **Properties:** Clicking this button displays the properties of a selected breakpoint. At a time, you can view the properties of only a single breakpoint.

Breakpoint Listbox Right-Click Menu

All of the above operations can be accessed by right-clicking on breakpoints in the listbox. The operations so accessed have the same restrictions and results as the ones listed above.



2.12.8.3 Group Management Operations

Alongside the filtering options there is a button labeled ‘Groups...’. Click this button to display the **Breakpoint Group Management** dialog. See Figure 2.34.

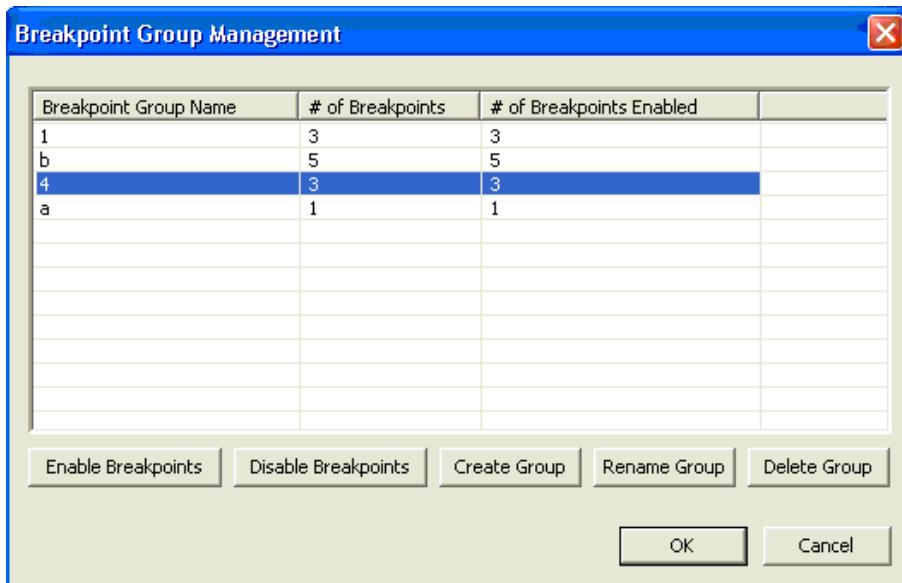


Figure 2.34. Breakpoint Group Management Dialog

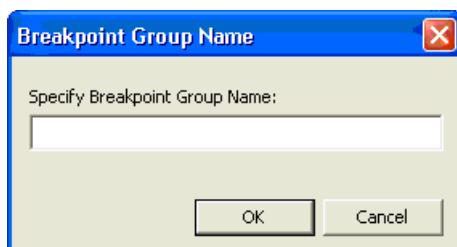
The **Breakpoint Group Management** dialog contains the list of breakpoints' groups, the number of breakpoints each group contains and the number of breakpoints enabled for each group:

- Breakpoint Group Name – is the column containing the names of the available breakpoint groups.
- # of Breakpoints – is the column containing the number of breakpoints in the corresponding group.
- # of Breakpoints Enabled – is the column containing the number of breakpoints enabled in the corresponding group.

Breakpoint Group Management Buttons

At the base of the list of breakpoint groups in the **Breakpoint Group Management** dialog are the buttons that allow you to manipulate the breakpoints within a group or across groups:

- Enable Breakpoints – Enables all the breakpoints that are contained in the selected group(s).
 - Disable Breakpoints – Disables all the breakpoints that are contained in the selected group(s).
 - Create Group – Creates a breakpoint group with a specified name.
1. Click the **Create Group** button. The **Breakpoint Group Name** dialog box is displayed.



2. Enter a name for the breakpoint group and click **OK**. The entered name is verified against current breakpoint group names to ensure that it is not duplicated.
 3. Click **Cancel** to end the operation without creating a new group name.
 4. To add breakpoints to the newly created group see Step 4 in Section 2.12.8.2.2.
- Rename Group – Renames a breakpoint group with a specified name.
 1. Click the **Rename Group** button. The **Breakpoint Group Name** dialog is displayed with the name of the breakpoint group name to be renamed.
 2. Modify the name of an existing group and click **OK**. The entered name is verified against current breakpoint group names to ensure that it is not duplicated.
 3. Click **Cancel** to end the operation without renaming a group.
 - Delete Group – Deletes a selected breakpoint group/s.
 1. Click the **Delete Group** button. The **Breakpoint Group Name** dialog is displayed with the name of the breakpoint group name to be deleted. Alternatively, enter the name of the breakpoint group to be deleted.
 2. Click **OK** to confirm deletion.



Note

Deleting breakpoint group(s) does not affect any of the breakpoints in the selected group(s). The action simply removes the breakpoints from the selected group because it no longer exists.

3. Click **Cancel** to end the operation without deleting a group.

2.12.9 About Code Breakpoints



Note

You cannot set a breakpoint while a simulation is running or when Microengines are running in hardware.

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 Code Breakpoint Hit** option.

A breakpoint can be set for all Microengine contexts or some contexts.

- **All context** breakpoints halts execution when any context in the Microengine reaches that instruction.

- **Some context** breakpoints halt execution when the context in the Microengine you have assigned to the breakpoint reaches that instruction.

Conditional breakpoints have an associated function that is executed when the breakpoint instruction is reached. The function result determines whether to pause the simulation or continue uninterrupted. Conditional breakpoints can be applied to all contexts or some contexts.

Programmer Studio also supports soft breakpoints, which are inserted into assembler code using the **ctx_arb[bpt]** instruction and into C code using the **_assert macro**, which in turn inserts a **ctx_arb[bpt]** instruction. When this instruction is executed, Programmer Studio is notified and displays a message box indicating where the breakpoint occurred, i.e., the chip, Microengine, thread, and instruction address. In simulation mode, Programmer Studio stops the simulation. In hardware mode, Programmer Studio stops all the other Microengines. In either case, you can resume execution by clicking **Go, Step Over**, etc.

2.12.9.1 P4 Source View Breakpoints. C Source View Breakpoints and List View Breakpoints

If the code in a Microengine is assembled, i.e., generated by the Assembler, then a thread window for that Microengine displays a list view, which is a flattened view of the source code after it has been assembled into a list file. If the code is compiled using Netronome C Compiler, then the thread window displays both a C source view and a list view. The C source view shows the C source files that were compiled to produce the list file. Code breakpoints can be set in either C source view or list view, or both. If the code is P4 source code and compiled with Netronome P4 Compiler, then the thread window displays a P4 source view, a C source view and a list view. The P4 source view shows the P4 source files that were compiled to produce the C source which compiled to produce the list file.

If you insert a breakpoint on a line in C source view, a C source view breakpoint is inserted on the line in the source file being displayed. The actual instruction on which the breakpoint gets set, is the first one generated by the source line that is also designated by the compiler as breakable. As the compiler can inline or clone a function so that there are multiple instances of it, setting a breakpoint on a source line in a function can result in multiple instances of the breakpoint being set, one in each instance of the function. However, it is still considered to be a single source view breakpoint and is persisted as such.

If you insert a breakpoint on a line in P4 source view, a P4 source view breakpoint is inserted on the line in the source file being displayed. Also a breakpoint is inserted on the generated line in the C source view. The actual instruction on which the breakpoint gets set, is the one generated by the source line that is also designated by the P4 compiler as breakable. Setting a breakpoint on a P4 source line results in two breakpoint being set on an instruction. However, it is still considered to be a single source view breakpoint and is persisted as such.

If you insert a breakpoint on a line in list view, a list view breakpoint is inserted on the instruction associated with that line.

It is possible to have multiple breakpoints set on the same instruction, but they must be of different origins. For example, a list view and a source view breakpoint can be set on the same instruction but two list view breakpoints cannot.

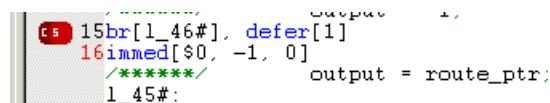
2.12.9.2 Code Breakpoint Markers

When the Source View is being displayed in a Thread window, markers for the Source View breakpoints are displayed as red with no letter. Markers for List View breakpoints are displayed as medium red with the letter L in the lower left.

When the List View is being displayed in a Thread window, markers for the List View breakpoints are displayed as red with no letter. Markers for the Source View breakpoints are displayed as medium red with the letter S in the lower left.

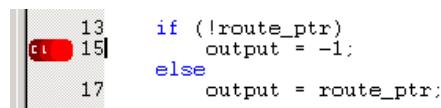
If multiple breakpoints are set on the same line, the markers are displayed as overlapped, with the leftmost marker being on top. In the Source View, a Source View marker is always on top. In the List View, a List View marker is always on top. A command line marker is always underneath, unless it is the only breakpoint on the line.

Three breakpoints on the same line in a List View would appear as:



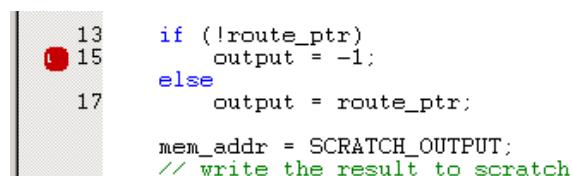
```
15br[1_46#], defer[1]
16immed[$0, -1, 0]
/***** output = route_ptr;
1_45#;
```

Three breakpoints on the same line in a Source View would appear as:



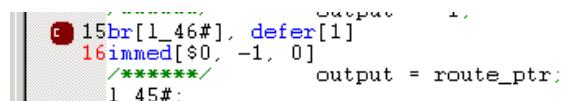
```
13
15 if (!route_ptr)
     output = -1;
else
    output = route_ptr;
17
```

A List View breakpoint on a line in Source View would appear as:



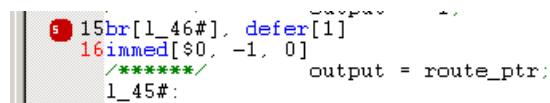
```
13
15 if (!route_ptr)
     output = -1;
else
    output = route_ptr;
mem_addr = SCRATCH_OUTPUT;
// write the result to scratch
17
```

A Command Line breakpoint on a line in Source View would appear as:



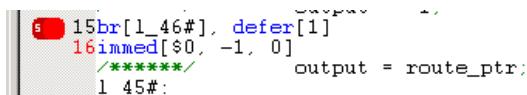
```
15br[1_46#], defer[1]
16immed[$0, -1, 0]
/***** output = route_ptr;
1_45#;
```

A Source View breakpoint on a line in List View would appear as:



```
15br[1_46#], defer[1]
16immed[$0, -1, 0]
/***** output = route_ptr;
1_45#;
```

A List View and a Source View breakpoint on a line in List View would appear as:

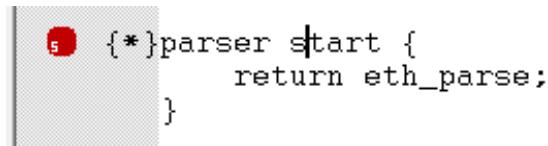


```

15 br[1_46#], defer[1]
16 immed[$0, -1, 0]
/*****/
    output = route_ptr;
1_45#:

```

A P4 Source View breakpoint on a line in P4 Source View would appear as:



```

5 {*}parser start {
    return eth_parse;
}

```

When a breakpoint is set, Programmer Studio 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 following lists the various breakpoint markers that appear:

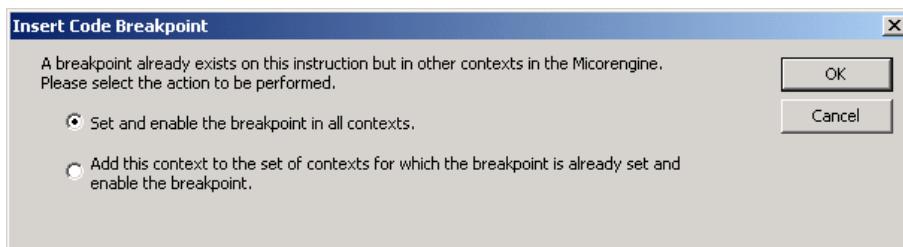
 Breakpoint	Enabled in the thread being viewed.
 Breakpoint	Disabled in the thread being viewed.
 List breakpoint	List view breakpoint on a line in a Source view.
 Source breakpoint	Source view breakpoint on a line in a List view
 Command Line breakpoint	Breakpoint set through the Command Line Interface using the ubreak command.
 Conditional breakpoint	Enabled.
 Conditional breakpoint	Disabled.
 List Conditional - enabled	List view conditional breakpoint on a line in a Source view.
 List Conditional - disabled	List view conditional breakpoint on a line in a Source view.
 Source Conditional - enabled	Source view conditional breakpoint on a line in a List view.
 Source Conditional - disabled	Source view conditional breakpoint on a line in a List view.
 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.

2.12.9.3 Code Breakpoint Behavior

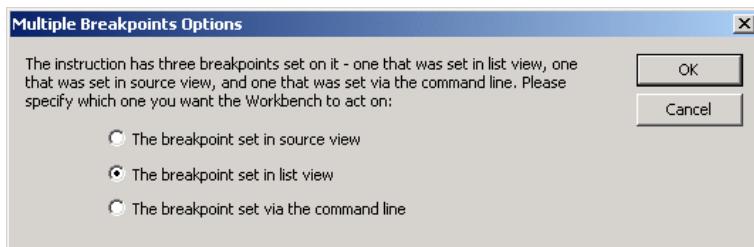
Some of the behaviors of code breakpoints are listed below:

- If you attempt to insert a breakpoint on a line that doesn't generate an instruction, Programmer Studio inserts the breakpoint on the next closest line that does generate an instruction.
- If the Microengines are running, in simulation or in hardware, all code breakpoint operations are disabled.
- If a breakpoint's properties are changed so that it is not set in all contexts in a Microengine, Programmer Studio does not display a breakpoint marker in thread windows in which the breakpoint is not set.
- If you insert a breakpoint in list view on a line that is a collapsed macro reference, Programmer Studio sets the breakpoint on the first instruction generated by the macro or its nested macros.

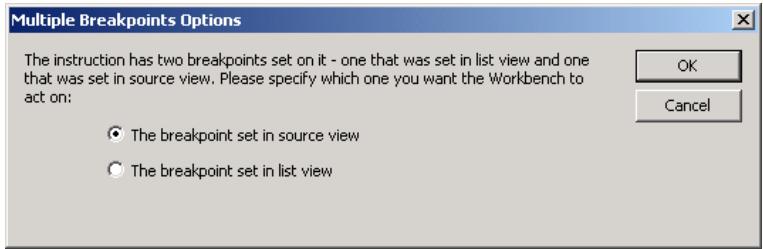
- In list view, if you act on a source view breakpoint that has been set in multiple instances of an inlined or cloned function, then the action affects all instances of the breakpoint.
- When a breakpoint is hit, Programmer Studio automatically switches the view (list or source) to the one in which the breakpoint was set.
- For a collapsed macro or a C source line, the marker for list view and command line breakpoints behaves as follows:
 - If all breakpoints on lines generated by the macro/source line are enabled, an enabled marker is displayed. If you do a Remove Breakpoint, then all the breakpoints are removed. If you do a Disable Breakpoint, then all breakpoints are disabled.
 - If all breakpoints on lines generated by the macro/source line are disabled, a disabled marker is displayed. If you do a Remove Breakpoint, then all the breakpoints are removed. If you do an Enable Breakpoint, then all breakpoints are enabled.
 - If some breakpoints on lines generated by the macro/source line are enabled and some are disabled, an enabled marker is displayed. If you do a Remove Breakpoint, then all the breakpoints are removed. If you do a Disable Breakpoint, then all enabled breakpoints are disabled.
- If you attempt to insert a breakpoint for a thread and there is already a breakpoint set on the line but in the other threads in the same engine, then the dialog box shown below is displayed.



- If you attempt to Enable Breakpoint, Disable Breakpoint, Remove Breakpoint, Breakpoint Properties, or Assign to a Breakpoint Group on a line that has three breakpoints set on it, a dialog box is displayed allowing you to select which breakpoint to act on. For example,



If the line has two breakpoints set on it, a similar dialog box is displayed. For example,



2.12.9.4 Inline-function Breakpoint Support

Programmer Studio supports inline-function breakpoints. Setting or clearing breakpoints in the Source view causes all instances of the inline function in the List view to handle the breakpoint (see Figure 2.35). All breakpoints enabled or disabled in the list view as a group.

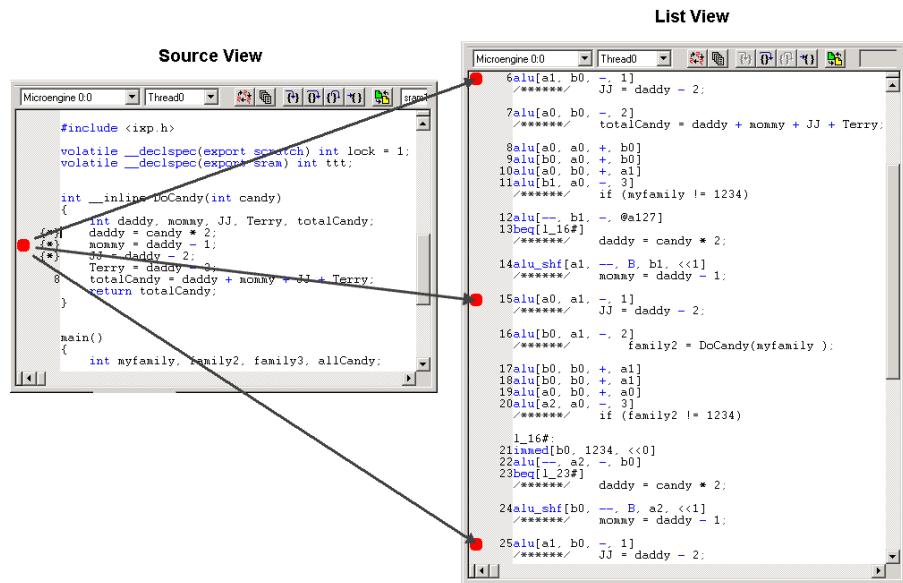


Figure 2.35. Inline Function Breakpoints in Source and List Views

2.12.9.5 Setting Breakpoints in Hardware Mode

You can set breakpoints in Hardware mode with the following restrictions:

- Each breakpoint you insert causes the Debug library in the Netronome ARM processor to place a breakpoint routine in unused Control Store space within the Microengines. Consequently, the number of breakpoints you can insert is limited by the size of your microcode image.
- You may be prevented from setting a breakpoint on certain instructions because processing the breakpoint adversely affects the thread's execution state or register contents. If this occurs, the following message will be displayed 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 continues to run. You should check the **Thread Status** window after a breakpoint has been reached to determine if any threads are still active.

Hardware Mode Restrictions

When debugging in Hardware mode, you cannot set breakpoints on instructions that:

- Are in defer shadows, or
- Are indirect branches.

2.12.9.6 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 **Code Breakpoint**, then click **Enable/Disable**, or

Click the  button on the **Debug** toolbar. (This button is not on the default **Debug** menu. To add this button, see Section 2.2.3.4), or

Press CTRL+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 **Code Breakpoint**, then click **Disable All**, or

Click the  button on the **Debug** toolbar. (This button is not on the default **Debug** menu. To add this button, see Section 2.2.3.4).

To enable all breakpoints:

- On the **Debug** menu, click **Code Breakpoint**, then click **Enable All**, or

Click the  button on the **Debug** toolbar. (This button is not on the default **Debug** menu. To add this button, see Section 2.2.3.4).

2.12.9.7 Removing All Breakpoints

To remove all breakpoints in all Microengines:

- On the **Debug** menu, click **Code Breakpoint**, then click **Remove All**, or

Click the  button on the **Debug** toolbar.

2.12.9.8 About Multi-Microengine Breakpoint Support

Support for the ability to manipulate a breakpoint in multiple Microengines simultaneously has been added to Programmer Studio. When you right-click on a code line in a thread window that has already breakpoint set, the context menu that gets displayed contains a new item labeled **Multi-Microengine Breakpoint**. If you select this item, Programmer Studio displays the dialog box shown in Figure 2.36.

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 (see Section 2.12.9.2).

Select one or more MEs from the list and click 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 you click **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.

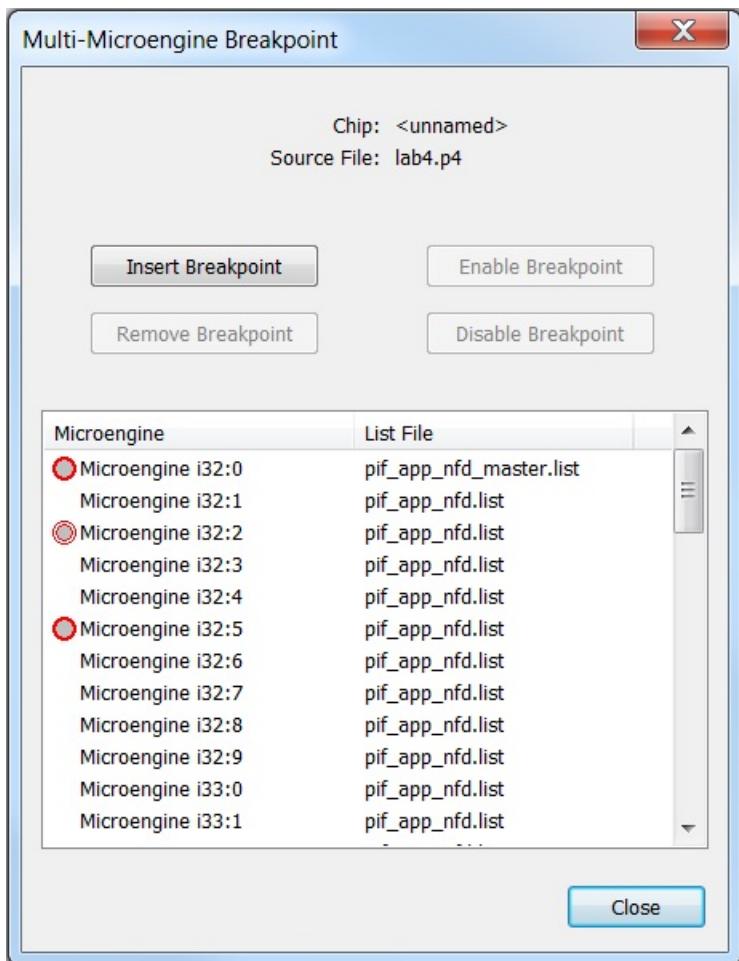


Figure 2.36. Multi-Microengine Breakpoint Dialog Box

2.12.9.9 Grouped Breakpoints

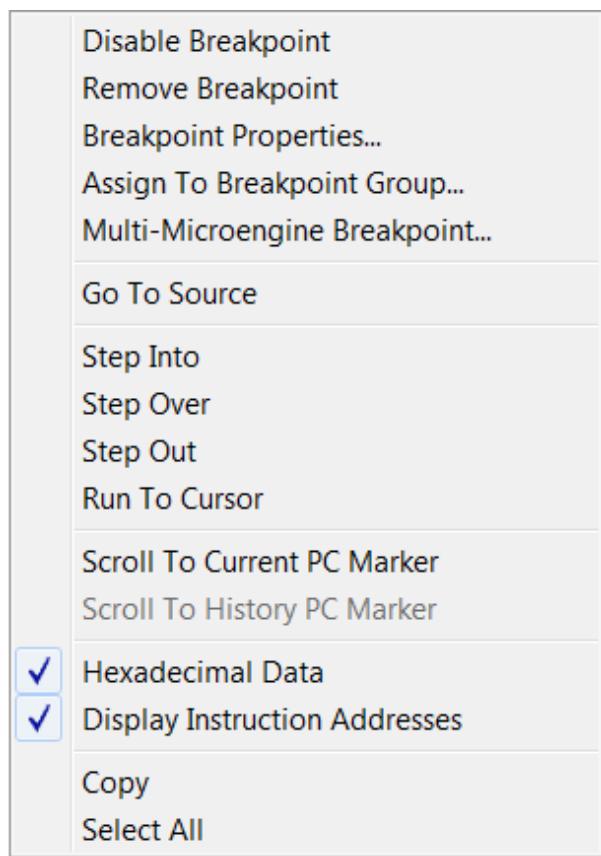
This feature allows you to define groups of Code Breakpoints that can be enabled or disabled. It is easier to switch between certain debug scenarios or processing stages if logical groups of breakpoints have been defined. For example, when setting up a simulation, it may be helpful to define a set of breakpoints for all the table lookup results. In a next step, it may be interesting to disable other breakpoint groups and define a set of breakpoints to check certain packet processing steps.

Viewing and managing these breakpoint groups is only possible while debugging.

Breakpoint group settings are saved in the project's options file.

Only Code Breakpoints can be assigned to a group. No Data Watch, Memory Watch, or Queue Breakpoints can be assigned to a group.

You can insert, remove, enable or disable breakpoints by right-clicking on the instruction in the Thread Window.



There is a new option to assign the breakpoint to a breakpoint group - **Assign To Breakpoint Group**.

Selecting **Assign To Breakpoint Group** will display the **Code Breakpoint Group Assignment** dialog shown in Figure 2.37.

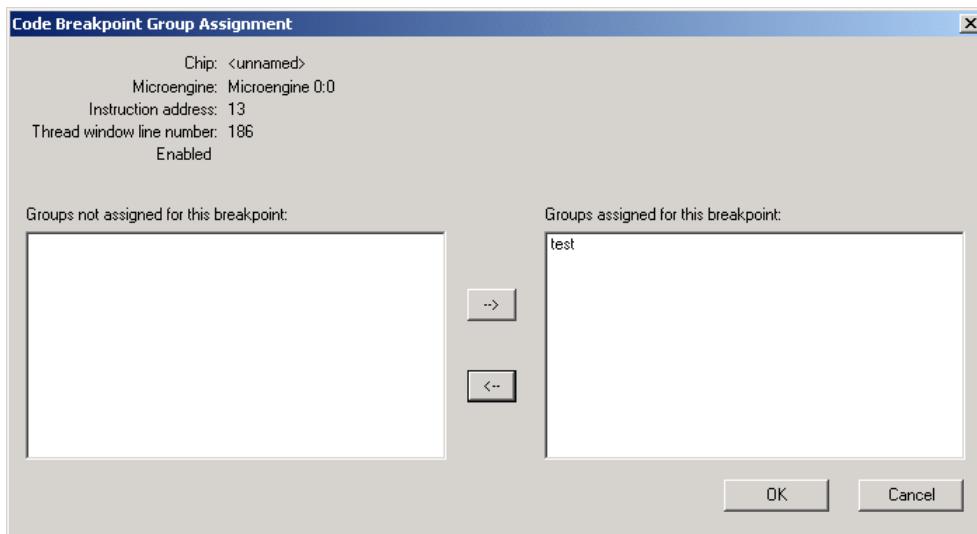


Figure 2.37. Code Breakpoints Assign to Group Dialog Box

The information for **Chip**, **Microengine**, **Instruction address**, **Thread window line number** and **Breakpoint State** (Enabled or Disabled) is displayed. This is the same information listed under the **Breakpoint Properties** dialog.

The **Groups not assigned for this breakpoint** list box displays the Breakpoint Groups that do not have the breakpoint as part of the group.

The **Groups assigned for this breakpoint** list box displays the Breakpoint Groups which have the breakpoint as part of the group.

The  and  buttons move groups between the lists.

The **OK** button makes the dialog changes permanent.

The **Cancel** button ignores any changes that have been made in the dialog.

Other Breakpoint Interactions

Removing breakpoints outside of the breakpoint group dialogs, also deletes the breakpoint from any breakpoint group it is assigned to. If the breakpoint is in an enabled breakpoint group, you will first be prompted with a message indicating that the breakpoint is in a breakpoint group. You have the option of continuing or canceling the operation.

Changing the enable/disable state of a breakpoint outside of the breakpoint group dialogs, and which is in an enabled breakpoint group, causes a message to be displayed indicating that the breakpoint is in an enabled breakpoint group. You have the option of continuing or canceling the operation.

2.12.10 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.

Programmer Studio displays the contents of the register assigned to that symbol as a pop-up window beneath the cursor.

```
273  ctx_arb[voluntary], defer[1]
274  alu_shf_ri_[r0, r0, +, 1, 0]
275  br[ctx3_43#] 0x00000001 (b,rel)
; Test Block #22
```

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, Programmer Studio records the register history. The values for all GPRs, transfer registers and neighbor registers in each Microengine are 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 is 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 displays the register’s value at the active cycle.

Programmer Studio allows users to hover over a register or C variable in a thread window. The current value of the register or C variable is displayed in a datatip. However, both the assembler and the C compilers use the concept of live range to restrict the range of instructions within which a register or C variable actually has a value associated with it. This means that you can hover over a register or C variable but Programmer Studio cannot display a value for it although it may be in scope. Instead Programmer Studio displays “out of live range”.

To lessen the impact of this behavior, the new datatip trace back feature added to Programmer Studio traces back in history to the most recent cycle at which that register or C variable was in live range. It then displays a datatip indicating that the variable is “out of live range” along with its most recent value, as shown below. It also displays the cycle and PC at which it was last live.

Inaccessible - Out of live range
[previous live value (at cycle 264, PC 3): count = 0x12345678 (in b1)]

If the register name or C variable was never in live range, Programmer Studio displays the datatip as shown below.

Inaccessible - Out of live range
[no previous live value available]

If you disable history collecting, Programmer Studio cannot perform the traceback so it displays the datatip shown below.

Inaccessible - Out of live range
[no previous live value: history is disabled]

Transfer registers comprise both the read and write registers. The datatip for a transfer register that is used for both reading and writing has a datatip that displays the value for both registers. If one of the registers is out of live range, the traced back value is displayed as shown below.

\$I0001Ix0 (read) = <out of live range>
[no previous live value available]
\$I0001Ix0 (write) = 0x33221100

If both registers are out of live range, the datatip looks like the one shown below.

```
$I0000!sram1 (read) = <out of live range>  
[no previous live value]  
$I0000!sram1 (write) = <out of live range>  
[previous live value (at cycle 987, PC 18): $I0000!sram1 (write) = 0x00005555]
```

Programmer Studio Behavior for Datatip Trace Back Feature

Programmer Studio behavior with respect to the datatip trace back feature is listed below.

- Neighbor registers are always in live range and thus do not require trace back.
- If you step back in history and hover over a register or C variable that is out of live range at that historical cycle, Programmer Studio traces back from that historical cycle to find a live value.
- If the most recent live value for a register or C variable that is out of live range was 500000 cycles or more in the past, then Programmer Studio can take a few seconds to display the datatip.

Changing 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 display the current register values. The history PC marker in all thread windows is hidden at this time.

When the active cycle is changed to a non-current cycle, data watches and datatips of non-register states and variables display an appropriate message to indicate that the historical value is not available.

2.12.11 Data Watches

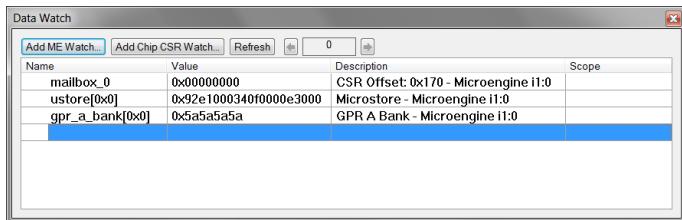
You can monitor the values of simulation states in Debug mode using the Data Watch window. Programmer Studio recognizes Control and Status Registers (CSR), Microengine Memory.

To open the **Data Watch** window:

- 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 two buttons, forward and back arrows, and a list with three columns of information about the data watch:



Add ME Watch

Button used to access the **Add ME Watch** dialog box.

Add Chip CSR Watch

Button used to access the **Add Chip CSR Watch** dialog box.

Refresh

Watch values are updated whenever microcode execution stops. To force an update of the values click the **Refresh** button.

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.

The data watch categories are:

- Microengine CSRs
- Microengine Memory
- NBI CSRs
- ILA CSRs
- XPB CSRs
- XPBm CSRs
- ARM CSRs.
- PCIe CSRs.
- CLS CSRs.

2.12.11.1 Adding Data Watches

There are two mechanisms available for adding either microengine or chip CSR watches. You can add a microengine or chip CSR watch from the **Add ME Watch** or **Add Chip CSR Watch** dialog box, or manually add a **New Data Watch** by typing the entry into the Data Watch window (see Section 2.12.11.2).

To add a microengine data watch from the **Add ME Watch** dialog:

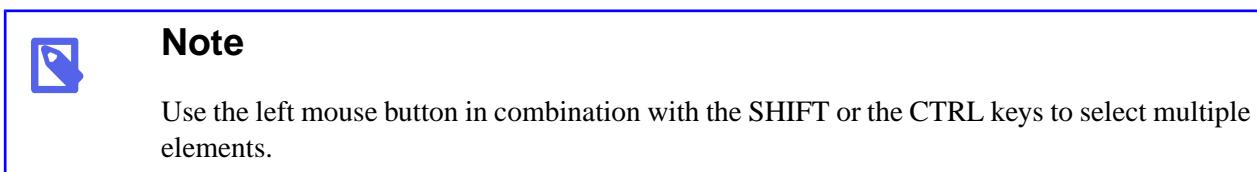
1. Click the **Add ME Watch** button.

The **Add ME 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.



4. Click **Add Watch** to have a watch added for each selected element. If you select from the list of Microengine CSRs, you will be prompted with a dialog box to select in which Microengine you want the watch to be done. If your project contains multiple chips, you will be prompted to select a chip.
5. When you have finished adding your watches, click **Close**.

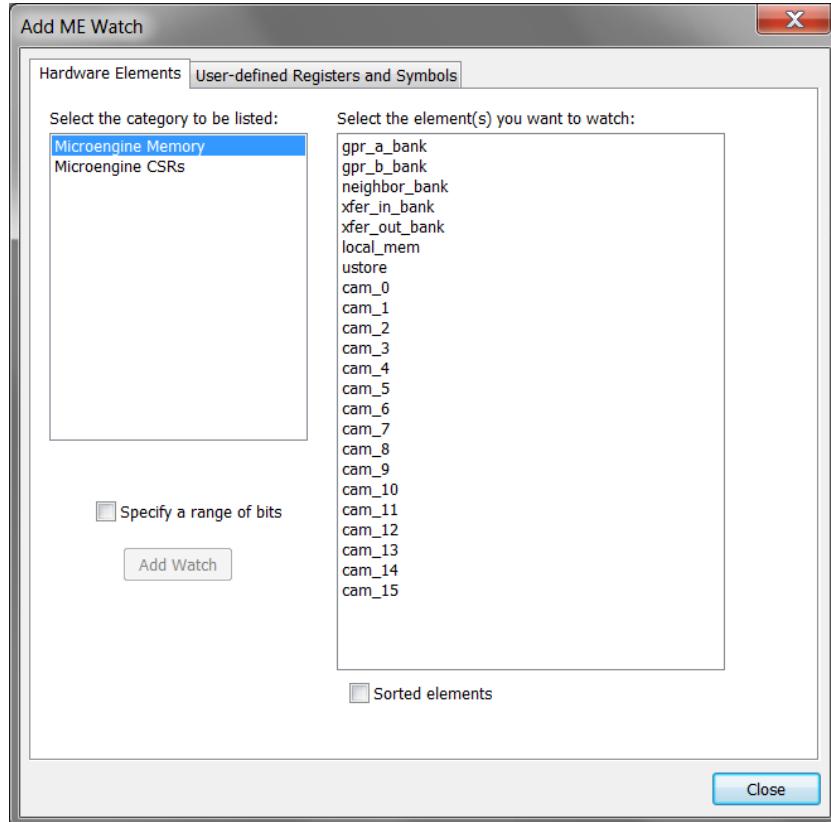


Figure 2.38. Add ME Watch Dialog Box

To add a chip CSR data watch from the **Add Chip CSR Watch** dialog:

1. Click the **Add Chip CSR Watch** button.

The **Add Chip CSR Watch** dialog box appears.

2. Select the component that you want listed by expanding the component.

3. In the list, select one or more elements you want to watch and click the button to add the CSR to list that appears on the right.
4. To unselect one or more elements click the button to remove the CSR from the list on the right.
5. Click **Add Watches** to have a watch added for each selected element. You will be prompted with a dialog box that how many watches had been added to **Data Watch** dialog. If your project contains multiple chips, you will be prompted to select a chip.
6. When you have finished adding your watches, click **Close**.

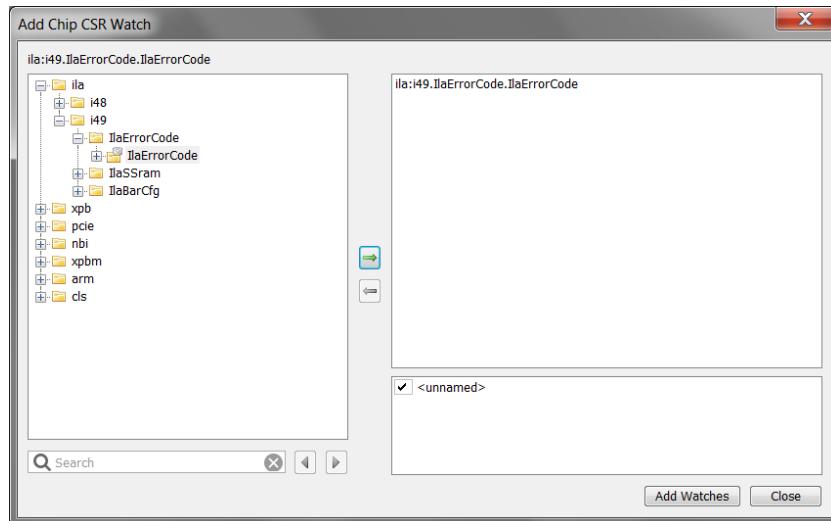


Figure 2.39. Add Chip CSR Watch Dialog Box

Array States

For states that are arrays, such as local memory, you must enter a range of array locations to be watched. The following describes the format for the range:

- [**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.

Inlined Functions

You can set a data watch on a variable within inlined functions and watch that variable in all instances of the inlined function.

Programmer Studio finds the instance of the variable within the current scope and updates the variable's value and the inlined line number in the "scope" field (see Figure 2.40).

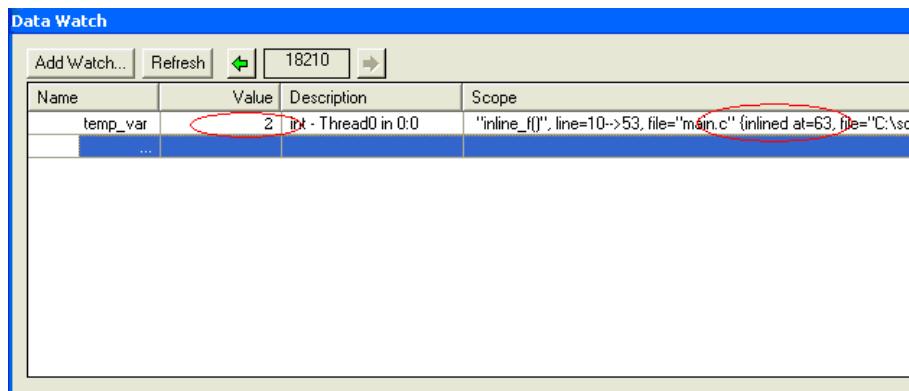


Figure 2.40. Data Watch for Inlined functions

Programmer Studio Behavior for Inlined functions' Data Watch

- Programmer Studio supports data watches for int, enum, struct, pointer, array type variables in inlined functions. The existing features such as conditional breakpoint and history tracking are also active on the inlined function data watch.
- Restarting debugging or reopening a project where a variable in an inlined function was added to data watch does not affect the data watch functionality.
- When opening an existing project with a data watch for a variable in an inlined function, the data watch is upgraded to reflect it automatically.

Save

Data watches are saved with project debug settings.

2.12.11.2 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 popup 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 into the entry box.

2.12.11.3 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>**.

- 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. Programmer Studio does not know that the text you select is a variable if it does not have any debug data. If this is the case, the **Set Data Watch** option on the shortcut menu is unavailable.

2.12.11.4 Watching Microengine Registers

To watch Microengine registers (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 ME Watch** or **Add Chip CSR Watch** or right-click in the **Data Watch** window and click **Add Watch** from the shortcut menu.

The **Add Data Watch** dialog box appears (see Figure 2.41).

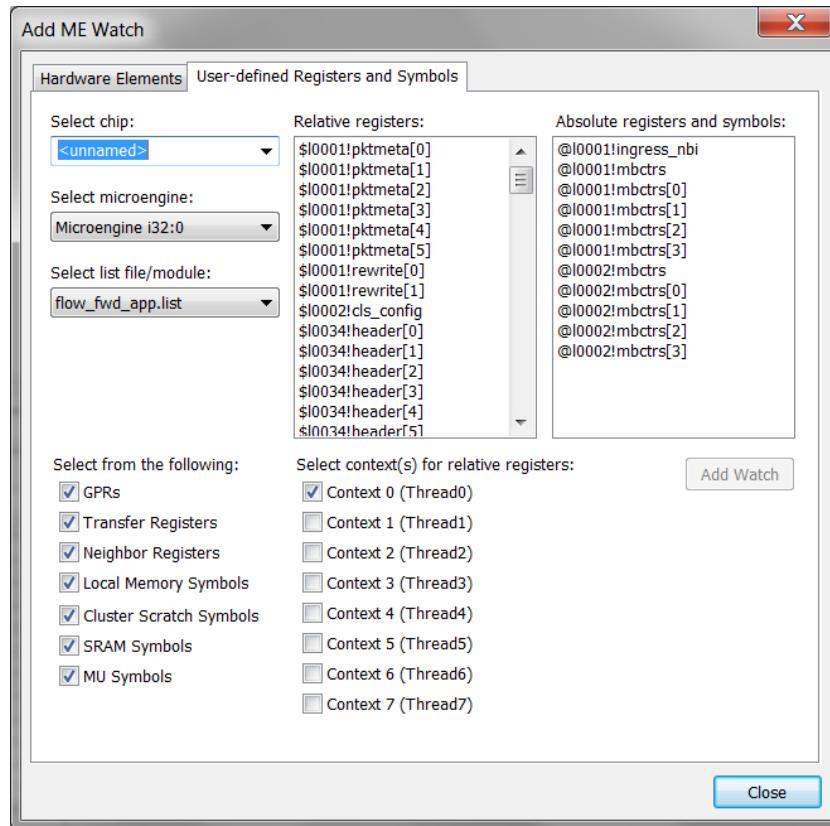


Figure 2.41. Add Data Watch Dialog Box - User-defined Registers and Symbols

2. Select the **User-defined Registers and Symbols** tab.
 - a. Select the chip and Microengine in which you want to watch registers.
 - b. Select whether you want **GPRs** listed by selecting or clearing the checkbox.
 - c. Select whether you want **Transfer Registers** listed by selecting or clearing the checkbox.
 - d. Select whether you want **Neighbor Registers** listed by selecting or clearing the checkbox.
 - e. Select whether you want **Local Memory Symbols** listed by selecting or clearing the checkbox.
 - f. Select whether you want **Cluster Scratch Symbols** listed by selecting or clearing the checkbox. This selection allows you to watch the assembly of Cluster Scratch symbolic constants.
 - g. Select whether you want **SRAM Symbols** listed by selecting or clearing the checkbox. This selection allows you to watch the assembly of SRAM symbolic constants.
 - h. Select whether you want **MU Symbols** listed by selecting or clearing the checkbox. This selection allows you to watch the assembly of DRAM symbolic constants.

Based on your checked selections, the relative registers are listed in the **Relative registers** list box and the absolute registers are listed in the **Absolute registers and Symbols** list box.

3. 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, you must specify which threads you want to watch by selecting or clearing the **Select context(s) relative registers** check boxes beneath the relative registers list.

- When you have finished adding your watches, click **Close**.



Note

Programmer Studio only displays 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, Programmer Studio displays 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” appears.

Aggregate Register Support

The Assembler supports the declaration and use of registers using array notation. Programmer Studio handles the square brackets in register names:

- You 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, its data watch value always shows “out of scope”.
- When you hover (using the 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 you hover 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 you right-click 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 the first bullet 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”). You 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 the first bullet 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.

2.12.11.5 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.

2.12.11.6 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 that you want to change the name of, then, on the **Debug menu**, click **Data Watch**, then click **Edit Name**, or

Double-click the name to be changed.

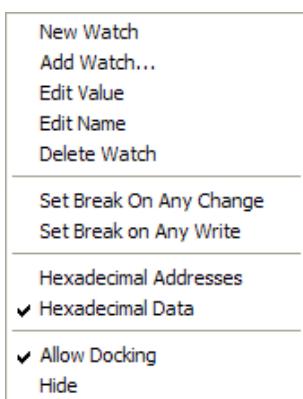
2. Type the state name.
3. Press ENTER.

2.12.11.7 Changing Data Watch Display Notation

Programmer Studio allows you the choice of data watch displays in decimal or hexadecimal notation for both data and addresses. The setting is toggled on the shortcut (right-click) menu in the data watch window. The hexadecimal address setting is global, so changing the setting for data watches also changes the setting for memory watches and vice versa.

To select whether watch values are displayed in decimal or hexadecimal:

1. Right-click anywhere in the **Data Watch** window. The shortcut window appears.



2. Click **Hexadecimal Addresses** on the shortcut menu to display addresses in hexadecimal format or clear it to display in decimal format.

3. Click **Hexadecimal Data** on the shortcut menu to display data in hexadecimal format or clear it to display in decimal format.

2.12.11.8 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 that you want to change the value of, 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.

2.12.11.9 Breaking on Data Changes and Writes

In a **Data Watch**, in simulation mode, you can halt microcode execution when:

- A state’s value changes (**Break on Any Change**).
- A state’s value is written (**Break on Any Write**).

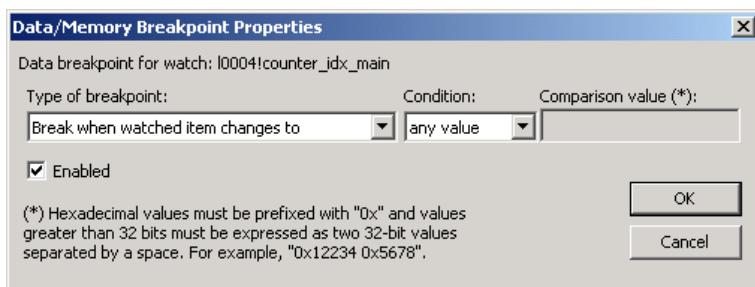


Note

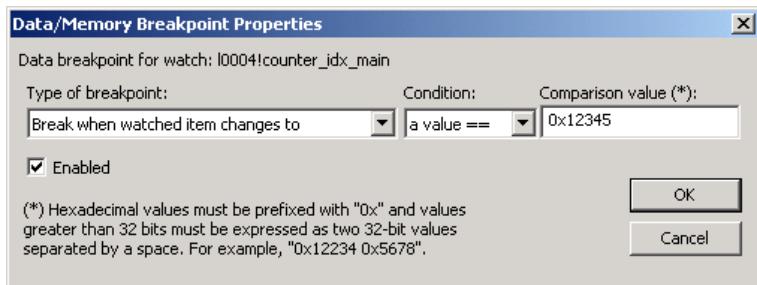
Breaking on data changes or writes is not supported in **Hardware** mode.

To break execution on a changed or written data value:

1. Create a data watch.
2. Right-click the name or value of the state and click **Set Break On Any Change**.
3. Right-click on the watch again and select **Breakpoint Properties** from the popup menu. Programmer Studio displays a **Data/Memory Breakpoint Properties** dialog.



In this dialog you change the type of breakpoint between break-on-change and break-on-write by selecting **Break when watched item changes to** or **Break when watched item is written with...**



You change the condition on which the break occurs by selecting one of the options from the combo box labeled **Condition**, and enter a the **Comparison value** for the condition in the associated text entry area.

The condition on which the break occurs are:

any value	The break occurs regardless of what the value is.
a value ==	The break occurs only if the value is equal to the comparison value.
a value >	The break occurs only if the value is greater than the comparison value.
a value <	The break occurs only if the value is less than the comparison value.
a value >=	The break occurs only if the value is greater than or equal to the comparison value.
a value <=	The break occurs only if the value is less than or equal to the comparison value.
a value !=	The break occurs only if the value is not equal to the comparison value.

If you select the condition **any value**, the breakpoint is considered to be unconditional. You do not specify a value and Programmer Studio disables the **Comparison value** entry field.

If you select any of the other conditions, the breakpoint is conditional. Programmer Studio enables the **Comparison value** edit control and you must enter a value which Programmer Studio compares against the value of the watched item to determine whether the break is to occur.

You can set and remove a **Break on Any Change**, or **Break on Write** on an aggregate state (an array) or on its individual elements. Setting or removing a data breakpoint on an aggregate state affects all its elements.

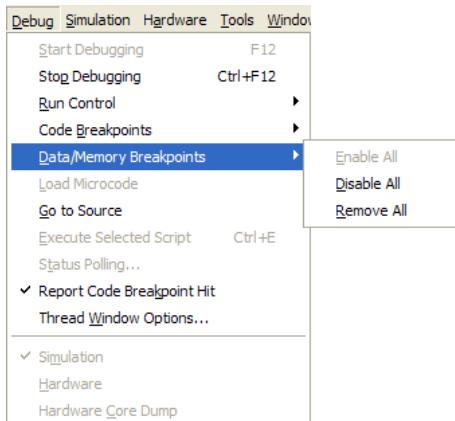
When the value changes for a state on which a break-on-change is set, microcode execution halts and a message box is displayed containing the state name along with its old and new values.

The different markers and their meanings are described below. If the item is an aggregate item, then the icon applies to all the sub-elements as well.

- **Break on Any Change** Indicates a break-on-change breakpoint for the watch item when set for any value.
- **Disabled Break on Any Change** Indicates a disabled break-on-change breakpoint for the watch item when set for any value.

- ② **Conditional Break on Any Change** Indicates a break-on-change breakpoint for the watch item when set with a Comparison value.
- ② **Disabled Conditional Break on Any Change** Indicates a disabled break-on-change breakpoint for the watch item when set with a Comparison value.
- **Break on Any Write** Indicates a break-on-write breakpoint for the watch item when set for any value.
- **Disabled Break on Any Write** Indicates a disabled break-on-write breakpoint for the watch item when set for any value.
- ② **Conditional Break on any Write** Indicates a break-on-write breakpoint for the watch item when set with a Comparison value.
- ② **Disabled Conditional Break on any Write** Indicates a disabled break-on-write breakpoint for the watch item when set with a Comparison value.
- ④ **Different Aggregate Setting** Indicates that a breakpoint is set on one or more of an aggregate's subelements, but that breakpoint settings differ between sub-elements. For example, **Break on Any Change** breakpoints might be enabled for all but one element.

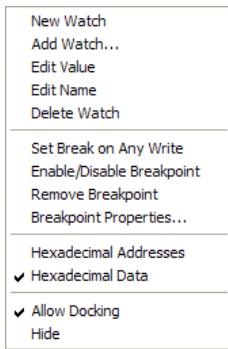
The **Data/Memory Breakpoints** menu item allows you to act on all the data/memory breakpoints simultaneously:



- **Enable All** enables all disabled data/memory breakpoints.
- **Disable All** disables all enabled data/memory breakpoints.
- **Remove All** removes all data/memory breakpoints.

To enable, disable or remove a specific break on any change or write:

1. Right-click the name or value of the state and a popup appears.

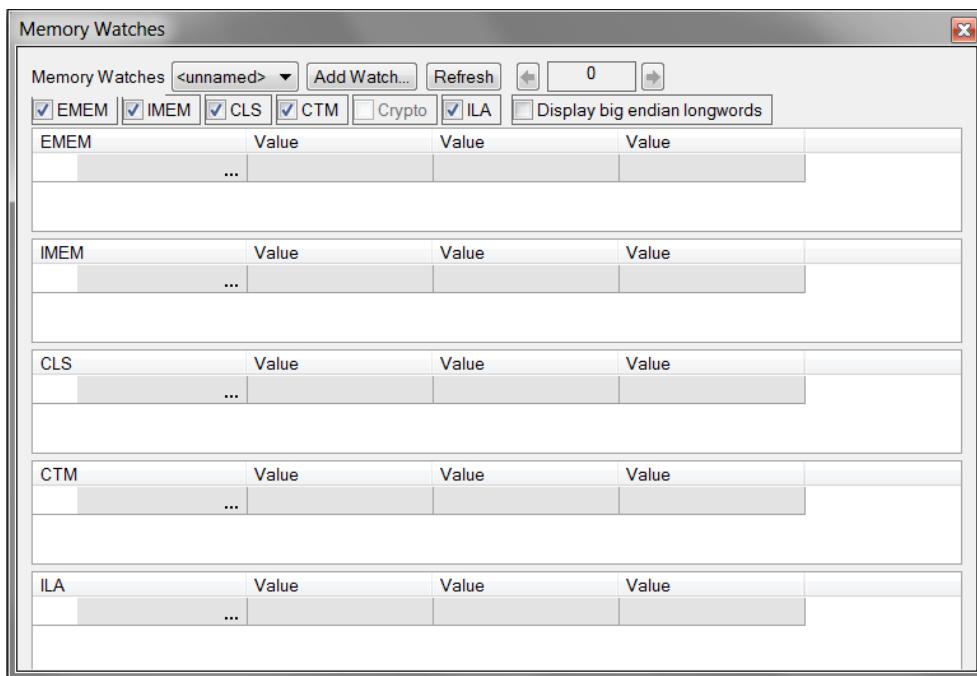


2. Click **Enable/Disable Breakpoint**, or **Remove Data Breakpoint**.

The enable/disable option acts as a toggle, switching between the current state of the selected breakpoint.

2.12.12 Memory Watch

In debug mode, you can monitor the values of EMEM, IMEM, Cluster Target Memory (CTM), Cluster Local Scratch (CLS) and ILA SRAM memory locations using the **Memory Watch** window.



The Network Processors address memory in bytes, thus the **Memory Watch** window interprets the address to be watched as a byte address. A memory byte address is rounded to the next lower quadword and the data is displayed in quadwords. For example, if you specify a watch of i29.imem[3:8], the watch is shown as i29.imem[0:15].

When you enter the address in the **Add Memory Watch** window and click 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

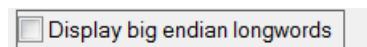
The **Memory Watch** window comprises three subwindows, one for each memory type. Each memory type has a check box at the top of the **Memory Watch** window to control visibility of the subwindow.



Each subwindow contains a multicolumn tree. The first column contains the range of the location(s) being watched, e.g. i29.imem[0:7]. 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 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.

[Refresh](#)

2.12.12.1 Entering a New Memory Watch

To enter a new memory watch:

1. Click on the **Add Watch...** button at the top of the Memory Watch window

[Add Watch...](#)

or right-click anywhere in the name or value column of the **Memory Watch** subwindow (either EMEM, IMEM, CTM, CLS or ILA SRAM) 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 following describes the format for the range:

[m]	To watch a single location. For example, i29.imem[1] watches location one in IMEM Island 29.
[m:n]	To watch locations m through n inclusive. For example, i24.emem[0:7] watches locations 0 through 7 in EMEM Island 24. And since a range can be specified in ascending or descending order, you could specify this watch as i24.emem[7:0].
[m:+n]	To watch location m plus the n locations following it. For example, i24.emem[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, i24.emem[0]<12> watches only bit 12 of the Island 24 EMEM location 0.
<m:n>	To watch bits m through n of a state, with m being greater than n. For example, i24.emem[0:7]<12:10> watches bits 12 through 10 of Island 24 EMEM locations 0 through 7.

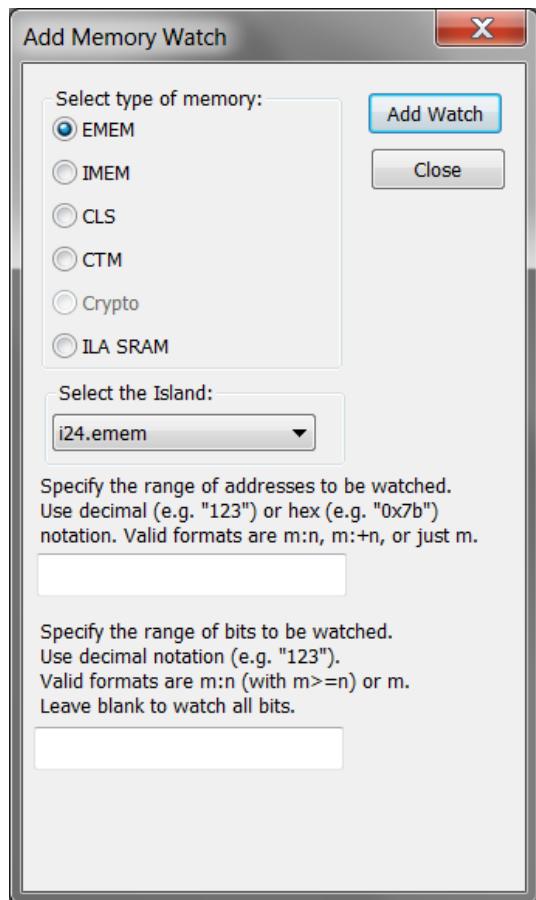
2.12.12.2 Adding a Memory Watch

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 EMEM, IMEM, CTM, CLS or ILA SRAM under **Select the type of memory**.
3. Select the Island for specific memory type under **Select the Island**.
4. 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.

5. Click **Add Watch**.
6. When you have finished adding your watches, click **Close**.

2.12.12.3 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.

2.12.12.4 Changing a Memory Watch

To change a memory watch address:

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.

To change a memory watch value:

1. Right-click the watch to be changed in the **Memory Watch** window and click **Edit Value** on the shortcut menu, or

Double-click the mouse button on the watch value to be changed.

2. Edit the new value.
3. Press ENTER.



Note

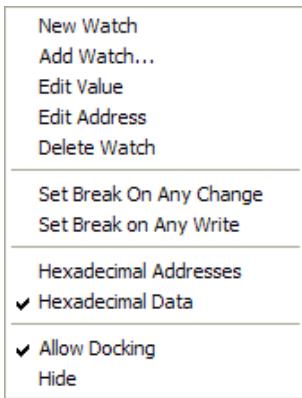
In **Hardware** mode, you can change only those entries that were added as a single memory location. You cannot change any value that is displayed as a result of adding a range of memory locations.

2.12.12.5 Changing the Memory Watch Display Notation

Programmer Studio allows you the choice of memory watch displays in decimal or hexadecimal notation for both data and addresses. The setting is toggled on the shortcut (right-click) menu in the memory watch window. The hexadecimal address setting is global, so changing the setting for memory watches also changes the setting for data watches and vice versa.

To select whether watch values are displayed in decimal or hexadecimal:

1. Right-click anywhere in the **Memory Watch** window. The shortcut window appears.



2. Click **Hexadecimal Addresses** on the shortcut menu to display addresses in hexadecimal format or clear it to display in decimal format.
3. Click **Hexadecimal Data** on the shortcut menu to display data in hexadecimal format or clear it to display in decimal format.

2.12.12.6 Breaking on Memory Changes or Writes

In a Memory Watch, in simulation mode, you can halt microcode execution when:

- A state's value changes (**Break on Any Change**).
- A state's value is written (**Break on Any Write**).



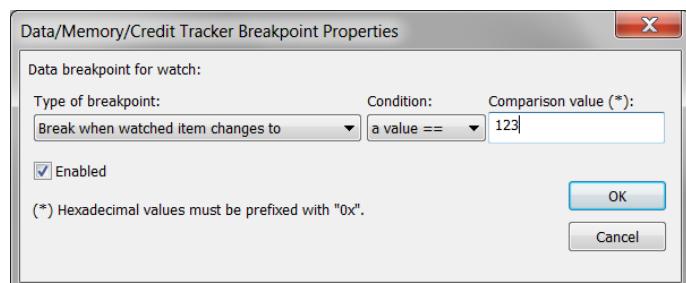
Note

Breaking on changes or writes is not supported in **Hardware** mode.

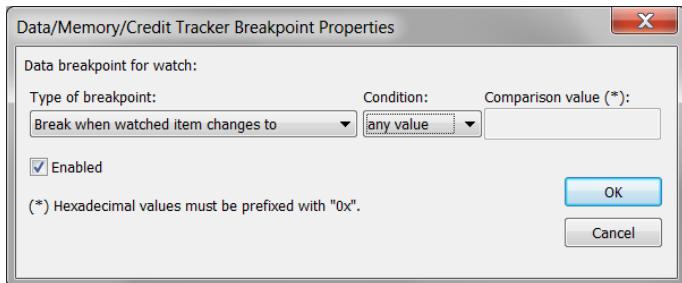
To break execution on a changed or written data value:

1. Create a memory watch.
2. Right-click the name or value of the state and click **Set Break On Any Change**.
3. Right-click on the watch again and select **Breakpoint Properties** from the popup menu.

Programmer Studio displays a **Data/Memory/Credit Tracker Breakpoint Properties** dialog.



In this dialog you change the type of breakpoint between break-on-change and break-on-write by selecting **Break when watched item changes to** or **Break when watched item is written with...**



You change the condition on which the break occurs by selecting one of the options from the combo box labeled **Condition**, and enter a the **Comparison value** for the condition in the associated text entry area.

The condition on which the break occurs are:

any value	The break occurs regardless of what the value is.
a value ==	The break occurs only if the value is equal to the comparison value.
a value >	The break occurs only if the value is greater than the comparison value.
a value <	The break occurs only if the value is less than the comparison value.
a value >=	The break occurs only if the value is greater than or equal to the comparison value.
a value <=	The break occurs only if the value is less than or equal to the comparison value.
a value !=	The break occurs only if the value is not equal to the comparison value.

If you select the condition **any value**, the breakpoint is considered to be unconditional. You do not specify a value and Programmer Studio disables the **Comparison value** entry field.

If you select any of the other conditions, the breakpoint is conditional. Programmer Studio enables the **Comparison value** edit control and you must enter a value which Programmer Studio compares against the value of the watched item to determine whether the break is to occur.

You can set and remove a **Break on Any Change**, or **Break on Write** on an aggregate state (an array) or on its individual elements. Setting or removing a data breakpoint on an aggregate state affects all its elements.

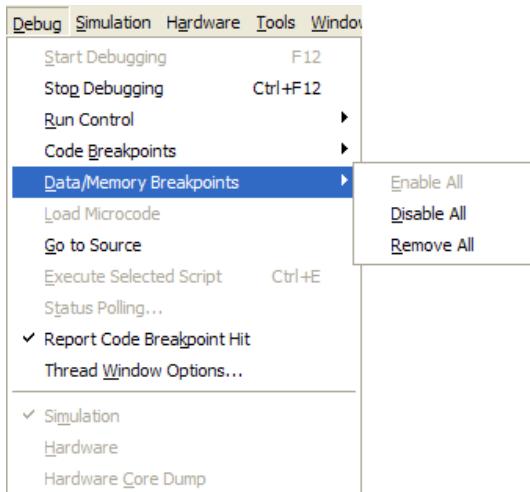
When the value changes for a state on which a break-on-change is set, microcode execution halts and a message box is displayed containing the state name along with its old and new values.

The different markers and their meanings are described below. If the item is an aggregate item, the icon applies to all the sub-elements as well.

- **Break on Any Change** Indicates a break-on-change breakpoint for the watch item when set for any value.
- **Break on Any Change** Indicates a disabled break-on-change breakpoint for the watch item when set for any value.

- ② **Conditional Break on Any Change** Indicates a break-on-change breakpoint for the watch item when set with a Comparison value.
- ② **Disabled Conditional Break on Any Change** Indicates a disabled break-on-change breakpoint for the watch item when set with a Comparison value.
- **Break on Any Write** Indicates a break-on-write breakpoint for the watch item when set for any value.
- **Disabled Break on Any Write** Indicates a disabled break-on-write breakpoint for the watch item when set for any value.
- ② **Conditional Break on any Write** Indicates a break-on-write breakpoint for the watch item when set with a Comparison value.
- ② **Disabled Conditional Break on any Write** Indicates a disabled break-on-write breakpoint for the watch item when set with a Comparison value.
- ④ **Different Aggregate Setting** Indicates that a breakpoint is set on one or more of an aggregate's subelements, but that breakpoint settings differ between sub-elements. For example, **Break on Any Change** breakpoints might be enabled for all but one element.

The **Data/Memory Breakpoints** menu item allows you to act on all the data/memory breakpoints simultaneously:

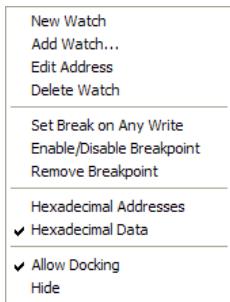


- **Enable All** enables all disabled data/memory/credit tracker breakpoints.
- **Disable All** disables all enabled data/memory/credit tracker breakpoints.
- **Remove All** removes all data/memory/credit tracker breakpoints.

To enable, disable or remove a specific break on any change or write:

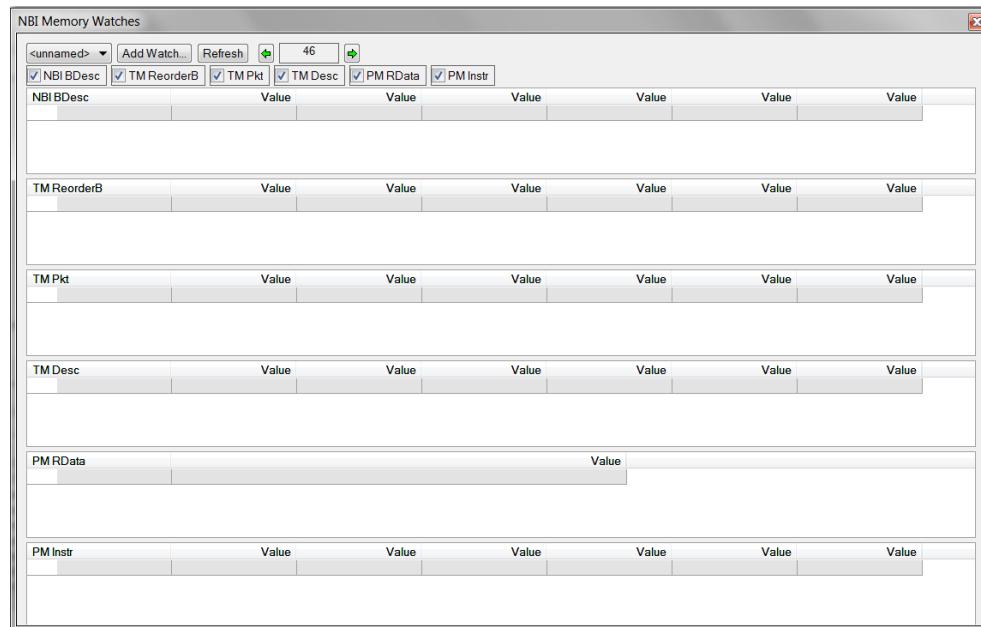
1. Right-click the name or value of the state and a popup appears.
2. Click **Enable/Disable Breakpoint**, or **Remove Data Breakpoint**.

The enable/disable option acts as a toggle, switching between the current state of the selected breakpoint.



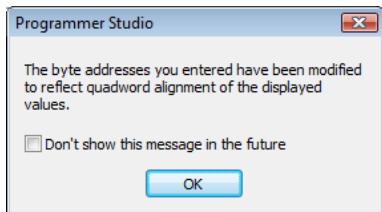
2.12.13 NBI Memory Watch

In debug mode, you can monitor the values of NBI Buffer Descriptor, Traffic Manager Packet SRAM, Traffic Manager Descriptor SRAM, Traffic Manager Recorder Buffer, Packet Manager Instructions and Replacement Data for each NBI Islands using the **NBI Memory Watch** window.



The Network Processors NBI address memory in quad-words, thus the **NBI Memory Watch** window interprets the address to be watched as a quadwords address. A NBI memory byte address is rounded to the next lower quadword and the data is displayed in quadwords. For example, if you specify a watch of [3:8], the watch is shown as [0:15].

When you enter the address in the **Add NBI Memory Watch** window and click 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 **NBI Memory Watch** window:

- On the **View menu**, click **Debug Windows**, then click **NBI 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 **NBI Memory Watch** window. Chip selection is synchronized with chip selection in the **History**, **Thread Status** and **Queue Status** windows.



Subwindows

The **NBI Memory Watch** window comprises six subwindows, one for each memory type. Each memory type has a check box at the top of the **NBI 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. i8.nbibusdesc[0:7]. The values of the locations are displayed in columns 1 through n. The number of value columns varies with the width of the **NBI Memory Watch** window, with a minimum of one value column.

Values Updates

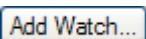
Watch values are updated whenever microcode execution stops. To force an update of the values at other times, click **Refresh** at the top of the **NBI Memory Watch** window.



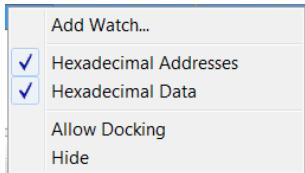
2.12.13.1 Entering a New NBI Memory Watch

To enter a new NBI memory watch:

1. Click on the **Add Watch...** button at the top of the NBI Memory Watch window



or right-click anywhere in the name or value column of the **NBI Memory Watch** subwindow and click **Add 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 following describes the format for the range:

[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, i8.nbibufdesc[0:7] watches locations 0 through 7 in buffer descriptor of NBI Island 8. And since a range can be specified in ascending or descending order, you could specify this watch as i8.nbibufdesc[7:0].
[m:+n]	To watch location m plus the n locations following it. For example, i8.nbibufdesc[5:+3] watches locations 0 through 15.

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, i8.nbibufdesc[0]<12> watches only bit 12 of the buffer descriptor location 0 of NBI Island 8.
<m:n>	To watch bits m through n of a state, with m being greater than n. For example, i8.nbibufdesc[0:3]<12:10> watches bits 12 through 10 of buffer descriptor locations 0 through 3 of NBI Island 8.

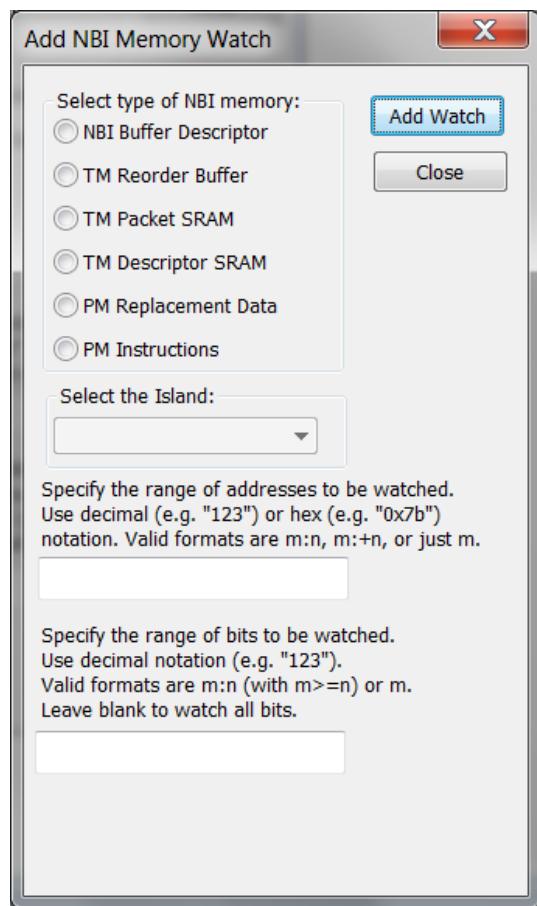
2.12.13.2 Adding a NBI Memory Watch

To add a NBI memory watch:

1. Click **Add Watch** at the top of the **NBI Memory Watch** window, or

Right-click in the name or value column window and click **Add Watch** on the shortcut menu.

The **Add NBI Memory Watch** dialog box appears.



2. Select NBI Buffer Descriptor, Traffic Manager Packet SRAM, Traffic Manager Descriptor SRAM, Traffic Manager Recorder Buffer, Packet Manager Instructions and Replacement Data under **Select the type of NBI memory**.
3. Select the NBI Island under **Select the Island**.
4. 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.
5. Click **Add Watch**.
6. When you have finished adding your watches, click **Close**.

2.12.13.3 Deleting a NBI Memory Watch

To delete a NBI memory watch:

1. Right-click the watch to be deleted in the **NBI 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 NBI Memory Watch

To change a NBI memory watch address:

1. Right-click the watch to be changed in the **NBI 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.

To change a memory watch value:

1. Right-click the watch to be changed in the **NBI Memory Watch** window and click **Edit Value** on the shortcut menu, or

Double-click the mouse button on the watch value to be changed.

2. Edit the new value.
3. Press ENTER.



Note

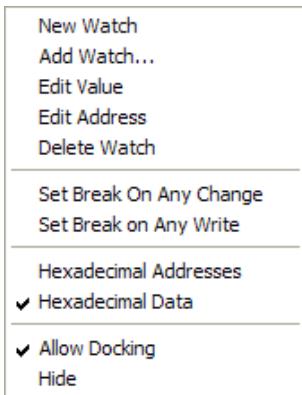
In **Hardware** mode, you can change only those entries that were added as a single memory location. You cannot change any value that is displayed as a result of adding a range of memory locations.

2.12.13.5 Changing the NBI Memory Watch Display Notation

Programmer Studio allows you the choice of NBI memory watch displays in decimal or hexadecimal notation for both data and addresses. The setting is toggled on the shortcut (right-click) menu in the NBI memory watch window. The hexadecimal address setting is global, so changing the setting for NBI memory watches also changes the setting of all other watches e.g. data watches and vice versa.

To select whether watch values are displayed in decimal or hexadecimal:

1. Right-click anywhere in the **NBI Memory Watch** window. The shortcut window appears.



2. Click **Hexadecimal Addresses** on the shortcut menu to display addresses in hexadecimal format or clear it to display in decimal format.
3. Click **Hexadecimal Data** on the shortcut menu to display data in hexadecimal format or clear it to display in decimal format.

2.12.13.6 Breaking on NBI Memory Changes or Writes

In a NBI Memory Watch, in simulation mode, you can halt microcode execution when:

- A state's value changes (**Break on Any Change**).
- A state's value is written (**Break on Any Write**).



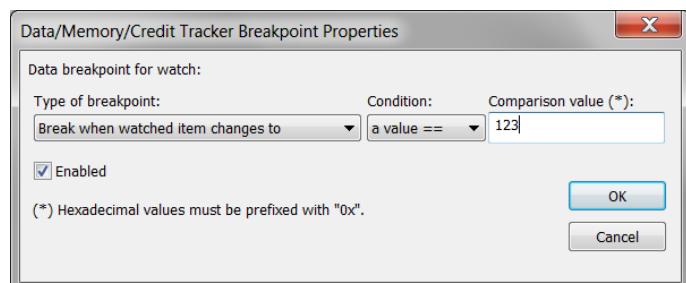
Note

Breaking on changes or writes is not supported in **Hardware** mode.

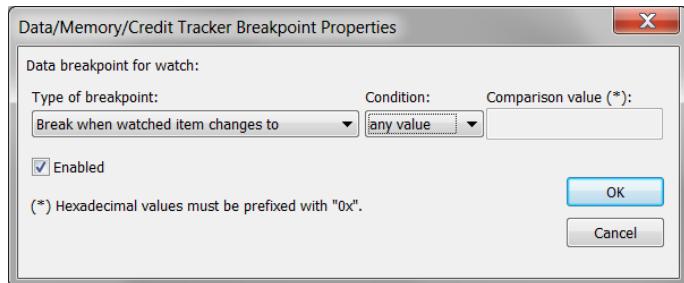
To break execution on a changed or written data value:

1. Create a memory watch.
2. Right-click the name or value of the state and click **Set Break On Any Change**.
3. Right-click on the watch again and select **Breakpoint Properties** from the popup menu.

Programmer Studio displays a **Data/Memory Breakpoint Properties** dialog.



In this dialog you change the type of breakpoint between break-on-change and break-on-write by selecting **Break when watched item changes to** or **Break when watched item is written with...**



You change the condition on which the break occurs by selecting one of the options from the combo box labeled **Condition**, and enter a the **Comparison value** for the condition in the associated text entry area.

The condition on which the break occurs are:

any value	The break occurs regardless of what the value is.
a value ==	The break occurs only if the value is equal to the comparison value.
a value >	The break occurs only if the value is greater than the comparison value.
a value <	The break occurs only if the value is less than the comparison value.
a value >=	The break occurs only if the value is greater than or equal to the comparison value.
a value <=	The break occurs only if the value is less than or equal to the comparison value.
a value !=	The break occurs only if the value is not equal to the comparison value.

If you select the condition **any value**, the breakpoint is considered to be unconditional. You do not specify a value and Programmer Studio disables the **Comparison value** entry field.

If you select any of the other conditions, the breakpoint is conditional. Programmer Studio enables the **Comparison value** edit control and you must enter a value which Programmer Studio compares against the value of the watched item to determine whether the break is to occur.

You can set and remove a **Break on Any Change**, or **Break on Write** on an aggregate state (an array) or on its individual elements. Setting or removing a data breakpoint on an aggregate state affects all its elements.

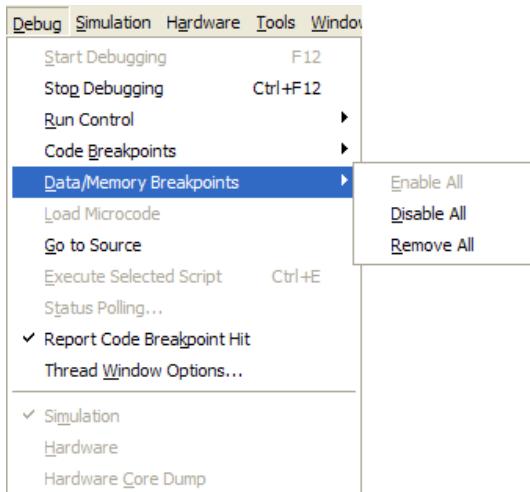
When the value changes for a state on which a break-on-change is set, microcode execution halts and a message box is displayed containing the state name along with its old and new values.

The different markers and their meanings are described below. If the item is an aggregate item, the icon applies to all the sub-elements as well.

- **Break on Any Change** Indicates a break-on-change breakpoint for the watch item when set for any value.
- **Break on Any Change** Indicates a disabled break-on-change breakpoint for the watch item when set for any value.

- ② **Conditional Break on Any Change** Indicates a break-on-change breakpoint for the watch item when set with a Comparison value.
- ② **Disabled Conditional Break on Any Change** Indicates a disabled break-on-change breakpoint for the watch item when set with a Comparison value.
- **Break on Any Write** Indicates a break-on-write breakpoint for the watch item when set for any value.
- **Disabled Break on Any Write** Indicates a disabled break-on-write breakpoint for the watch item when set for any value.
- ② **Conditional Break on any Write** Indicates a break-on-write breakpoint for the watch item when set with a Comparison value.
- ② **Disabled Conditional Break on any Write** Indicates a disabled break-on-write breakpoint for the watch item when set with a Comparison value.
- ④ **Different Aggregate Setting** Indicates that a breakpoint is set on one or more of an aggregate's subelements, but that breakpoint settings differ between sub-elements. For example, **Break on Any Change** breakpoints might be enabled for all but one element.

The **Data/Memory Breakpoints** menu item allows you to act on all the data/memory breakpoints simultaneously:

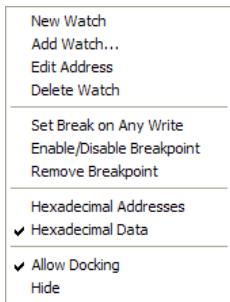


- **Enable All** enables all disabled data/memory breakpoints.
- **Disable All** disables all enabled data/memory breakpoints.
- **Remove All** removes all data/memory breakpoints.

To enable, disable or remove a specific break on any change or write:

1. Right-click the name or value of the state and a popup appears.
2. Click **Enable/Disable Breakpoint**, or **Remove Data Breakpoint**.

The enable/disable option acts as a toggle, switching between the current state of the selected breakpoint.



2.12.14 Watch Scripts

Watch scripts can be used to add additional functionality to your project. A watch script is simply a cscript that you add to your project, with the exception of having a special header definition. In this header you specify what the input parameters of your watch script is. Programmer Studio will autofill in some of the predefined input parameters. A dialog pops up when the watch script is executed where a user can change the input parameter's values.

2.12.14.1 Inserting A Watch Script To A Project

Use the same method to insert a watch script to a project as you would a normal script. Use the **Insert Script Files...** menu under **Project** on the menubar.

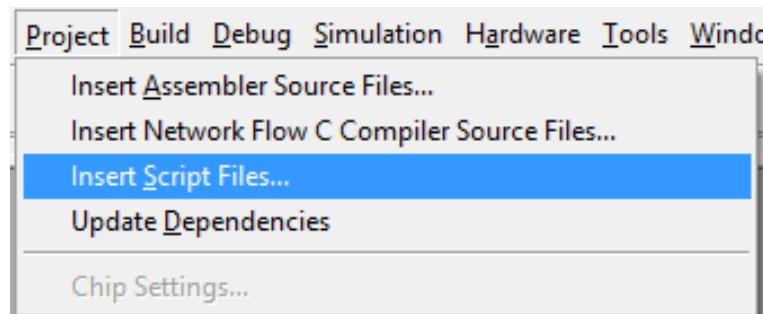


Figure 2.42. Insert Script Files

The script(s) you added should now appear under **Script Files** on the **FileView**.

2.12.14.2 Watch Script Format

A watch script is a cscript with a json header. The header is specified in comments with the following start and end tags:

```
/* cscript_header_start
   the json header will be here
cscript_header_end */
```

Figure 2.43. Watch Script Header Tags

The format of the watch script is as follows:

```
/* cscript_header_start
  "c_script": {
    "type": "watch",
    "input_params": {
      the script's input parameters will be here
    },
    "ps_params": {
      programmer studio's parameters will be here
    }
  }
cscript_header_end */
```

Figure 2.44. Watch Script Header Format

The **type** value simply tells Programmer Studio that this is a watch script. The **input_params** section is where you can define all possible input parameters that this script uses. The **ps_params** section is where you can specify which predefined Programmer Studio parameters this script can use. The **main** section is where you can specify all the main functions of the script, the input parameters for the main function, and the parameter order.

Each of these sections will now be discussed in more detail.

ps_params:

```
"ps_params": {
  "run_on_update": "Run Every SIM Update"
}.
```

Figure 2.45. Watch Script ps_params Section

Programmer Studio parameters are used to give certain behavior to a watch script. These are predefined parameters and are not used in the watch script. The parameter name is the key and the display string is the value.

The following are valid ps_params:

- **run_on_update** executes the watch script everytime you step or stop the simulator

input_params:

```
"input_params": {
    "dev_num": {
        "type": "int",
        "display_string": "NFP Device Number"
    },
    "mem_type": {
        "type": "const char*",
        "display_string": "Memory Type"
    },
    "address": {
        "type": "int",
        "display_string": "Address"
    },
    "length": {
        "type": "int",
        "display_string": "Length"
    }
}.
```

Figure 2.46. Watch Script input_params Section

Input parameters are the parameters the watch script takes as input in its main functions. The key for each parameter is the name of the input parameter. The type is the data type of the parameter. The display_string is the string that will be displayed to the user for that parameter.

The following data types are available:

- **int**
- **bool**
- **const char***
- **selection**

The **selection** type can be a selection of type const char* or int. The selection type and values can be specified as follows:

```
"output_format": {
    "type": "selection",
    "selection_type": "const char*",
    "selection_values": ["pdml", "psml", "ps", "text"],
    "display_string": "Output Format",
    "default": "text"
}.
```

Figure 2.47. Watch Script selection Type Input Parameter

There is also the optional **default** parameter which can be used to specify a default value for an input parameter.

Another optional parameter is **options**. This parameter can be used to specify additional predefined options. The following options are available:

- **multi-line** The multi-line parameter option can be used when the type of the input parameter is const char*. This will allow a user to input a multi-line string into the input parameter dialog.

These parameters can be anything you want for your script to work correctly. There are however a few predefined parameters that Programmer Studio will auto-fill for you. These parameters are: (Note that you only need to use the parameter name for it to be considered a predefined parameter. The display string can be anything you like.)

- **address**
- **length**
- **name**
- **dev_num**
- **mem_type**
- **mac_prepend**
- **ctm_offset**
- **split_length**
- **me_num**
- **reg_type**
- **context**
- **reg_index**

main:

```
/* cscript_header_start
{
    "c_script": {
        "type": "watch",
        "input_params": {
            "dev_num": {
                "type": "int",
                "display_string": "NFP Device Number"
            },
            "mem_type": {
                "type": "const char*",
                "display_string": "Memory Type"
            },
            "address": {
                "type": "int",
                "display_string": "Address"
            },
            "length": {
                "type": "int",
                "display_string": "Length"
            }
        },
        "ps_params": {
            "run_on_update": "Run Every SIM Update"
        },
        "main": {
            "all": [
                "dev_num", "mem_type", "address", "length"
            ]
        }
    }
} cscript_header_end */
```

Figure 2.48. Watch Script main Section

```
int example(int dev_num, const char* mem_type, int address, int length) {
    printf("You have executed example.c on watch %s with address %i and length %i.", mem_type, address, length);
    return 0;
}
```

Figure 2.49. Watch Script main Function

A watch script can have multiple main sections, each for a main function with different input parameters. The reason one would use different main functions is when different input parameters are used for different types of watches. The watch for which a main section can be executed is defined by the key of the main section. **all** is used to specify that this watch script can execute on all watch types. Other valid watch types are:

- **mereg**
- **ustore**
- **lmem**
- **meCSR**
- **xpBCSR**
- **cPCSR**
- **emem**
- **imem**
- **cls**
- **ctm**
- **sram**
- **emem_dc**
- **crypto**
- **ila**

You can specify a combination of any of these types in the key, which must be comma separated.

The order of the input parameters defined in the main section must match the order of the parameters in the main functions exactly.

2.12.14.3 Executing A Watch Script

You can see which watch scripts are available by right clicking on a watch, and selecting **Execute Script**.

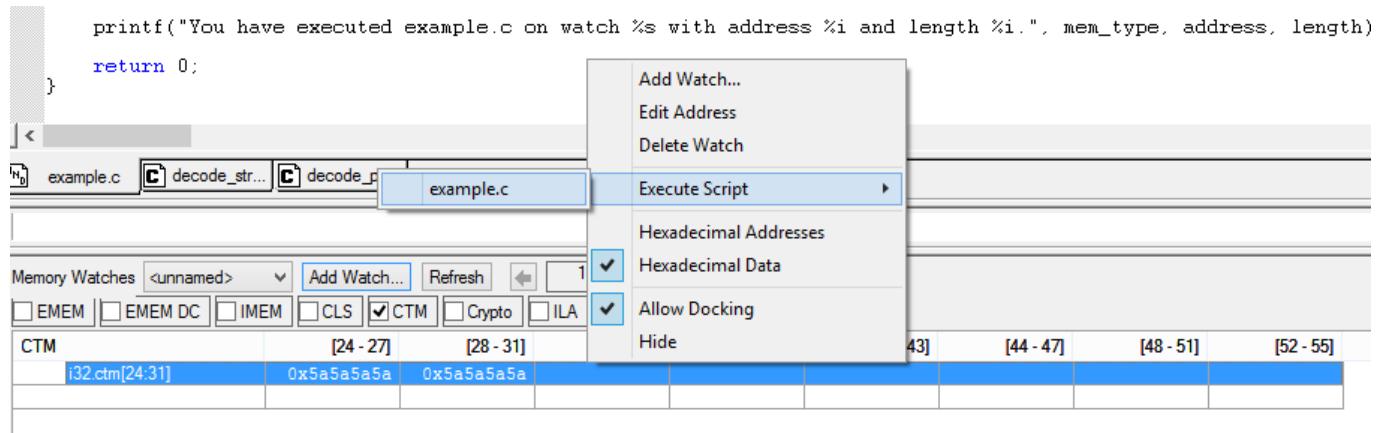


Figure 2.50. Execute Watch Script

When you click on the watch script you want to execute, a dialog pops up where you can fill out all the input parameters' values. Some of these values might already be autofilled by Programmer Studio.

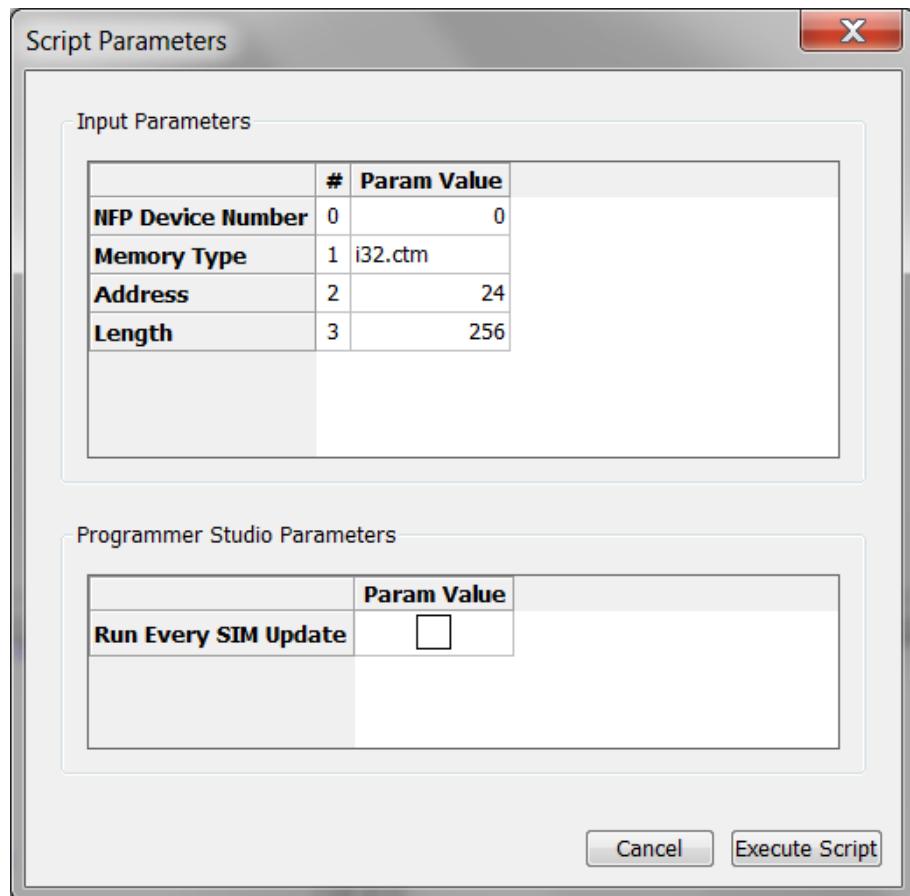


Figure 2.51. Watch Script Dialog

The watch script's output will display in the cling output window.

```
***** CLING *****
* Type C++ code and press enter to run it *
*           Type .q to exit *
*****
>>> .x C:\Users\Kobus de Bruyn\Desktop\example.c(0,"i32.ctm",24,8)
You have executed example.c on watch i32.ctm with address 24 and length 8.(int) 0
```

Figure 2.52. Watch Script Output

2.12.14.4 Example Watch Scripts

Programmer Studio has a couple of example watch scripts available. The example scripts can be found under bin/scripts/watch_script_examples in the SDK install directory. You can simply insert these scripts into your project and use them. The following example scripts are available:

decode_packet_tshark.c

The decode_packet_tshark.c script can be used to decode a packet that is currently in the CTM memory. This script has only one main function and can only execute on a CTM watch.

The script dialog for this example script looks like this:

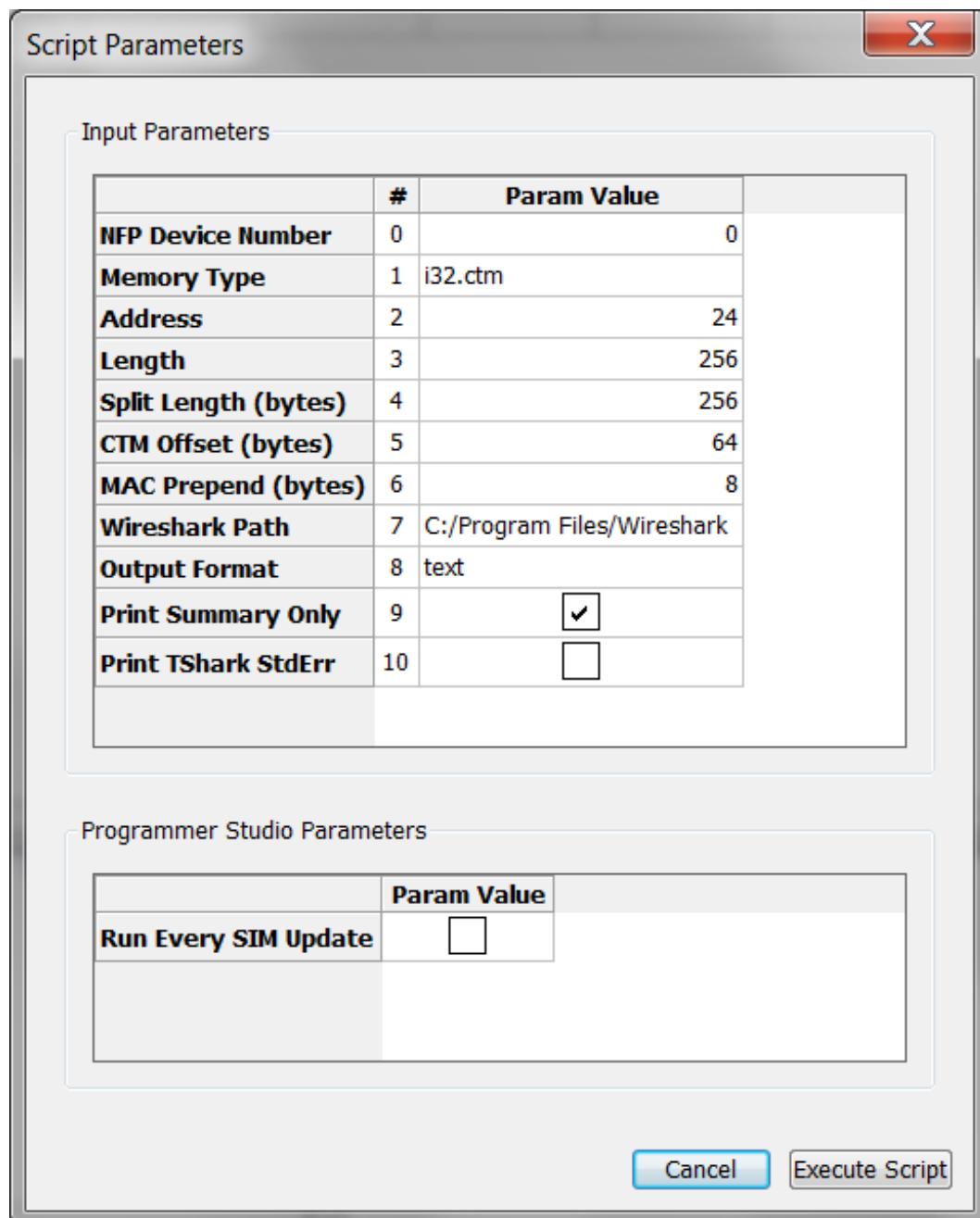


Figure 2.53. decode_packet_tshark.c Dialog

The resulting output when executing the script (with the options as in the above image) is as follows:

```
***** CLING *****
* Type C++ code and press enter to run it *
*           Type .q to exit
*****
>>> .x C:\Netro\Dev\Workspace\nfp_sdk_test_projects\beagle-temp.hg\decode_packet_tshark.o(0,"

0.000000 32.0.0.0 -> 16.0.0.0      UDP 60 Source port: systat Destination port: ssh
(int) 0
```

Figure 2.54. decode_packet_tshark.c Dialog

decode_struct.c

The decode_struct.c script can be used to decode a struct that you expect is in a certain memory or register.

The script dialog for this example script looks like this:

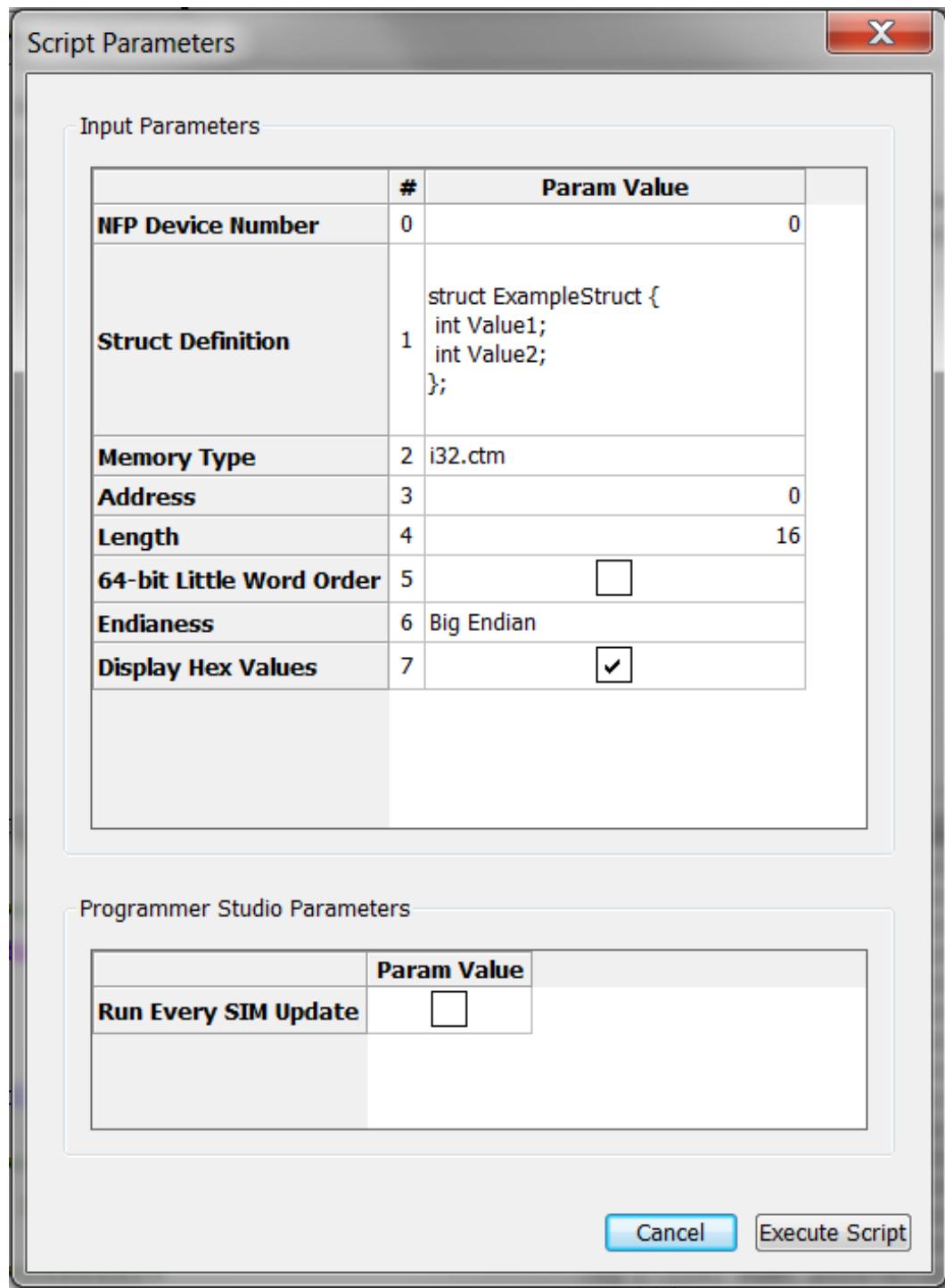


Figure 2.55. decode_struct.c Dialog

The resulting output when executing the script (with the options as in the above image) is as follows:

```
*****
* CLING *****
* Type C++ code and press enter to run it *
*           Type .q to exit
*****
>>> .x C:\Netro\Dev\Workspace\nfp_sdk_test_projects\beagle-temp.hg\decode_struct

struct ExampleStruct:
    Value1: -2147483580
    Value2: 67117568

(int) 0
```

Figure 2.56. decode_packet_tshark.c Dialog

2.12.15 NBI PM Modification Pipeline

In debug mode, you can monitor the NBI Packet Modifier Modification Pipeline using the **NBI PM Modification Pipeline** window.

Island 8	Stage 0	Stage 1	Stage 2	Stage 3	Stage 4
pkt_data_0x0	0x6000610062006300	0x6000610062006300	0x4000410042004300	0x2000210022002300	0x4c004d004e004f00
pkt_data_0x8	0x0000000000000000	0x0000000000000000	0x4000450046004700	0x0000000000000000	0x0400050006000700
pkt_data_0x10	0x0000000000000000	0x0000000000000000	0x480049004a004b00	0x280029002a002b00	0x0000000000000000
pkt_data_0x18	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x2c002d002e002f00	0x0000000000000000
pkt_data_0x20	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
pkt_data_0x28	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
pkt_data_0x30	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x6000610062006300
pkt_data_0x38	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x000001002000300	0x0000000000000000
pkt_data_0x40	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x080009000a000b00	0x0000000000000000
pkt_data_0x48	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
pkt_data_0x50	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000

The modification pipeline is tightly coupled with the Pipeline Control Block state machine. The modification pipeline works on 32-byte words. It is capable of performing one modification per stage. It is required to perform up to 4 modifications per 32-bytes and hence it is 4-stages deep. The total number of modifications allowed per packet is 8. The total replacement data is limited to be 52 bytes. Different fields of the packet header are present inside the pipeline stage on various clock cycles and each stage has to perform right operations based on the opcode and an offset. In the pipeline, individual operations only occur on data within a 32-byte boundary. Therefore, packet fields that span a word boundary require multiple operations. Data does not flow uniformly through the pipeline because of insertions and deletions. Pipeline stages communicate with each other through small buffers.

Visibility

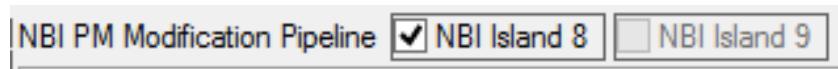
To toggle visibility of the **NBI PM Modification Pipeline** window:

- On the **View menu**, click **Debug Windows**, then click **NBI PM Modification Pipeline**, or

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

Subwindows

The **NBI PM Modification Pipeline** window comprises of up to two subwindows, one for each **NBI island**. Each **NBI island** has a check box at the top of the **NBI PM Modification Pipeline** window to control visibility of the subwindow.



Note

The **NBI island** check boxes are disabled for islands that do not contain NBI.

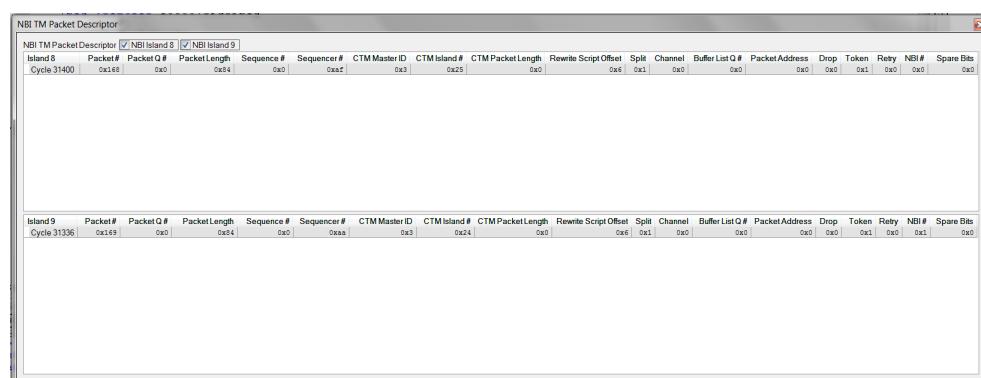
Each subwindow contains six columns. The first column contains the offset within the `pkt_data`, e.g. `pkt_data 0x28`. The values displayed in the remaining 5 columns correspond to the 5 stages of the modification pipeline. That is Stage 0, Stage 1, ..Stage 4.

Values Updates

Data values within each stage update automatically as and when they occur. There is no need to stop microcode execution.

2.12.16 NBI TM Packet Descriptor

In debug mode, you can monitor the NBI Packet Descriptors using the **NBI TM Packet Descriptor** window.



Packet descriptors are stored in-order in the descriptor memory and assigned to the appropriate queue based on queue number. Each descriptor is 128 bits which is displayed in **NBI TM Packet Descriptor** window as they are created. For more information on NBI Traffic Manager please see *NFP-6xxx-xC Databook*.

Visibility

To toggle visibility of the **NBI TM Packet Descriptor** window:

- On the **View menu**, click **Debug Windows**, then click **NBI TM Packet Descriptor Watch**, or

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

Subwindows

The **NBI TM Packet Descriptor** window comprises of two subwindows, one for each **NBI island**. Each **NBI island** has a check box at the top of the **NBI TM Packet Descriptor** window to control visibility of the subwindow.

NBI TM Packet Descriptor **NBI Island 8** **NBI Island 9**



Note

The **NBI island** check boxes are disabled for islands that do not contain NBI.

Values Updates

TM Packet Descriptor window for each NBI is update automatically as packet descriptors are created by the traffic manager. There is no need to stop microcode execution to update the window.



Note

The **NBI TM Packet Descriptor** window is only enabled when debugging on NFP SDK simulator.

2.12.17 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 maximum execution count among all generated instructions. For example, if a source line generates three instructions and the three instructions were executed 12, 12, and 10 times respectively, then the execution count displayed on that source line would be 12. For inlined functions, the count for a source line is the sum of the maximum counts for all instances. 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.

To display the **Execution Coverage** dialog box (see Figure 2.57):

1.  Stop simulation (if necessary).
2. On the **Simulation** menu, click **Execution Coverage**.

The **Execution Coverage** dialog box appears. *This dialog box is resizable.*

3. If your project contains multiple chips, select the chip that you wish to view the coverage data of from **Select a chip** list in the upper left corner of the dialog box.
4. Select a Microengine from the **Select a Microengine** list.
5. Select a list file/module from the 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.

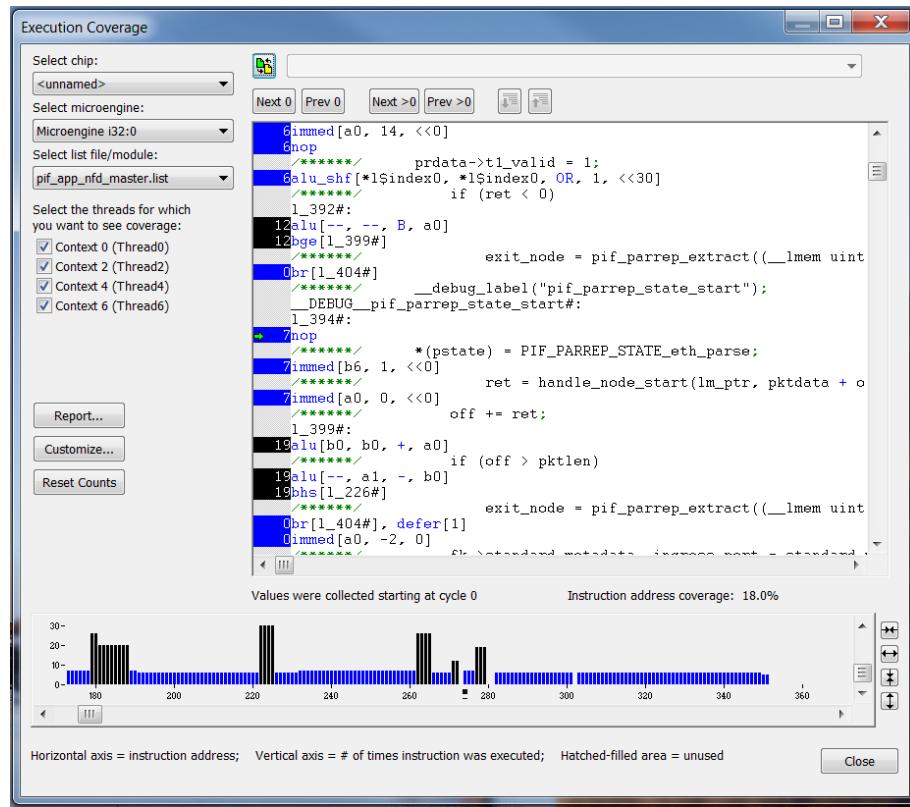


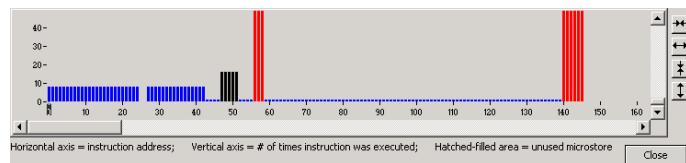
Figure 2.57. The Execution Coverage Window

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 2.12.17.1).

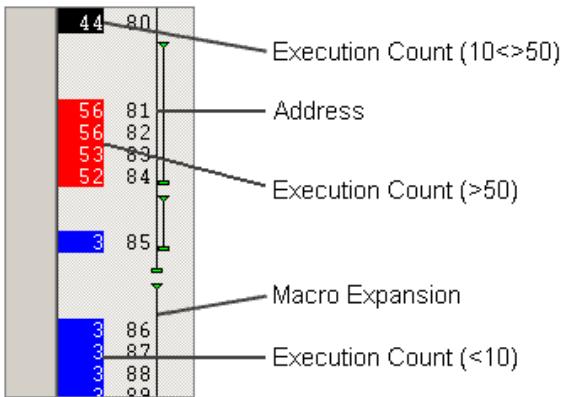
Bar Graph

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.



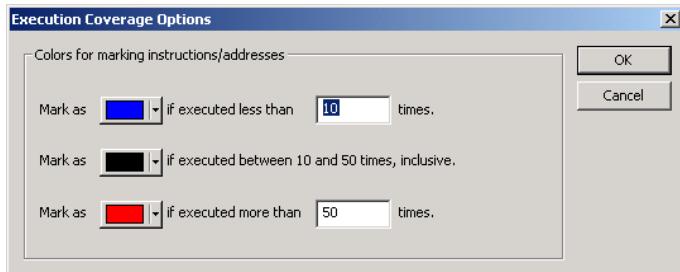
The execution count for a collapsed macro reference is the sum of the execution counts of all instructions generated by the macro.

2.12.17.1 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**, select a different color, and/or enter the values into the text, and Click **OK**.



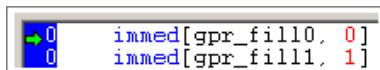
2.12.17.2 Displaying and Hiding Instruction Addresses

To toggle displaying and hiding instruction addresses in the code window:

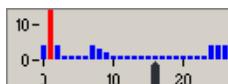
1. Right-click in the microcode window.
2. Select **Display Instruction Addresses** on the shortcut menu to display the addresses or clear to hide the addresses.

2.12.17.3 Instruction Markers

To synchronize viewing between the code window and the bar graph, Programmer Studio 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:

Next 0 Moves the marker to the next instruction that has an execution count of 0.

Prev 0 Moves the marker to the previous instruction that has an execution count of 0.

Next >0 Moves the marker to the next instruction that has an execution count that is greater than 0.

Prev >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.

2.12.17.4 Miscellaneous Controls

Other controls in the **Execution Coverage** dialog box are:

Expand macros (see Section 2.12.6.7).

Collapse macros (see Section 2.12.6.7).

Toggle between source and list view for the file selected.

Select the file to view.

2.12.17.5 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.

2.12.17.6 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**.

2.12.17.7 Execution Reports

When you click the **Report...** button, Programmer Studio displays the **Report Execution Coverage** dialog, as shown in Figure 2.57. You enter the output file to which the execution coverage data gets written - you can type the file name directly or you can browse to the file by clicking . You determine the type of report by selecting one of the three radio buttons.

The choices are:

1. Report executions counts for the Microengine and contexts selected in the **Execution Coverage** dialog (see Figure 2.58). Effectively, this reports the same executions counts that are being displayed in the Execution Coverage dialog. The report is by microstore address and includes only the instructions executed by the Microengine and contexts that are selected in the parent dialog.
2. Report executions counts by microstore address for all Microengines and in-use contexts. This reports the execution counts separately for all Microengines, summing the counts for each of the contexts that are in use in the Microengine.
3. Report execution counts by line for source file. Selecting this option enables the source file combo box, which lists all source files that are part of all the loaded Microengines (see Figure 2.59).

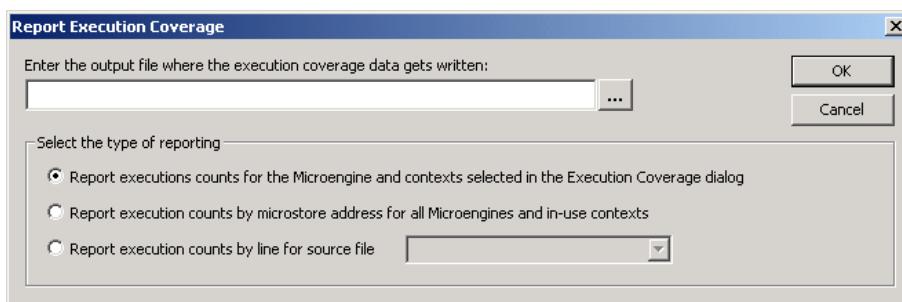


Figure 2.58. Report Execution Coverage for the Microengines

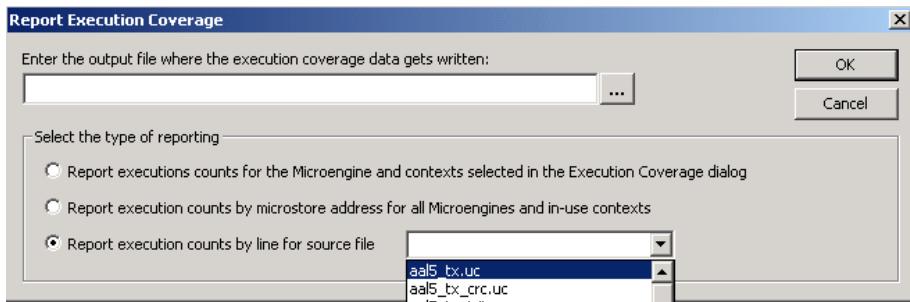


Figure 2.59. Report Execution Coverage by Line

The first two report types list each microstore address that has an instruction loaded into it along with the execution count for that address and the line of text that generated the instruction as contained in the list file. The percentage coverage represents the number of addresses that were executed at least once, divided by the number of addresses that have an instruction loaded in them.

An example is:

```
Chip: <unnamed>
Microengine 0:0
Contexts: 0, 1, 2, 3, 4, 5, 6, 7
List file = .\RefMiscTests_1.list
Coverage = 91.4%
Addr   Count     Instruction
      0       8   br=ctx[0, hash_test#]
      1       7   br=ctx[7, interthread#]
      2       6   ctx_arb[kill], any
      3       3   immed[$10000!hash_reg0, 0x1234]
      4       3   immed[$10000!hash_reg1, 0x2345]
      5       3   hash_48[$10000!hash_reg0, 1]
      6       3   hash_64[$10000!hash_reg0, 1]
      7       3   immed[$10000!hash_reg2, 0x3456]
      8       3   immed[$10000!hash_reg3, 0x4567]
      9       3   hash_128[$10000!hash_reg0, 1]
     10       3   hash_48[$10000!hash_reg0, 2]
     11       3   ctx_arb[10000!hash_sig, 10000!hash_sig+1]
```

The third type of report lists each line in the selected source file with the execution count next to each line. If a line does not generate any microwords, the count is blank. The percentage coverage represents the number of source lines that were executed at least once, divided by the number of source lines that generated at least one instruction.

An example is:

```
Execution counts for source file: RefMiscTests.uc
Chip: <unnamed>
Microengines: 0:0, 0:3, 1:7
Contexts: 0, 1, 2, 3, 4, 5, 6, 7
Coverage for lines that generated microwords = 91.7%
Count   Source line
          #include "IndirectRefFields.h"
          #ifdef FOUR_CTXS
```

```
.num_contexts 4
#endif

12 br=ctx[0, hash_test#]
#ifndef FOUR_CTXS
3 br=ctx[6, interthread#]
#else
7 br=ctx[7, interthread#]
#endif
8 ctx_arb[kill]
nop
nop

hash_test#:

.reg $hash_reg0,$hash_reg1,$hash_reg2,$hash_reg3,
.reg $hash_reg5,$hash_reg6,$hash_reg7,$hash_reg8,$hash_reg9
.reg $hash_reg10,$hash_reg11,$hash_reg12,$hash_reg13,
.xfer_order $hash_reg0,$hash_reg1,$hash_reg2,$hash_reg3,

.sig hash_sig
;
6 immed[$hash_reg0, 0x1234]
6 immed[$hash_reg1, 0x2345]
6 hash_48[$hash_reg0, 1]
```

Programmer Studio manages the execution counts as follows:

- When a project is opened, the counts are zeroed.
- When the user stops debugging or does a Reset while debugging, the counts are zeroed unless the user has unchecked Reset counts when simulation is reset.
- When a build is done resulting in a new list file for a Microengine, the counts for that Microengine are zeroed. The rationale for this is that a different list file means different instructions in the control store, making cumulative counts meaningless.
- When the **Reset Counts** button is clicked in the **Execution Coverage** dialog, the counts get zeroed.
- Whenever a microcode instruction is executed, the count associated with its address gets incremented.

2.12.18 ME Performance Statistics

Programmer Studio provides the ability to gather and display statistics on simulation performance. Statistics gathering is available only in **Simulation** mode.

2.12.18.1 Displaying Statistics

To display **ME Performance Statistics**:

1.  Stop debugging (if necessary).
2. On the **Simulation** menu, click **ME Performance Statistics**.

The **ME 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.

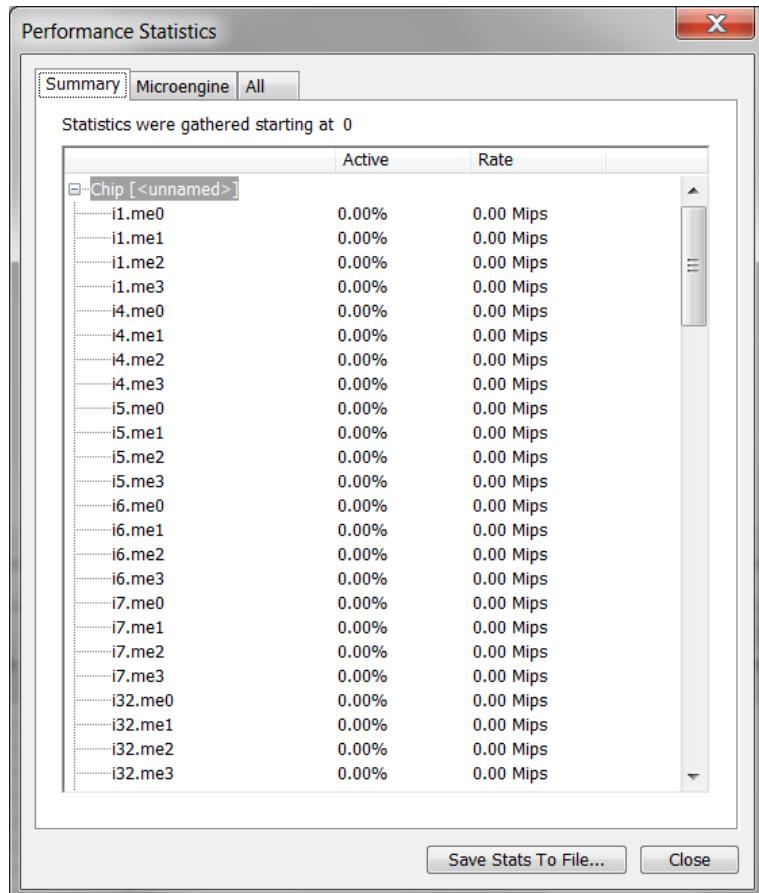


Figure 2.60. Performance Statistics - Summary Tab

4. To print the statistics to a .csv file, click the **Save Stats to File** button.

The **Performance Simulation Stats** dialog box displays (see Figure 2.61). Enter the filename and click **OK** to save the statistics. The data is output to a .csv file that can be read by a text editor or imported to Microsoft* Excel spreadsheet.

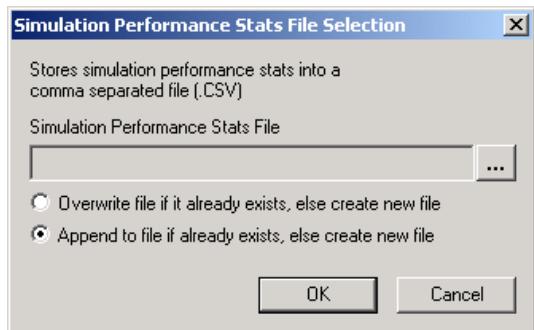


Figure 2.61. Save Performance Simulation Statistics to a File

5. 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.

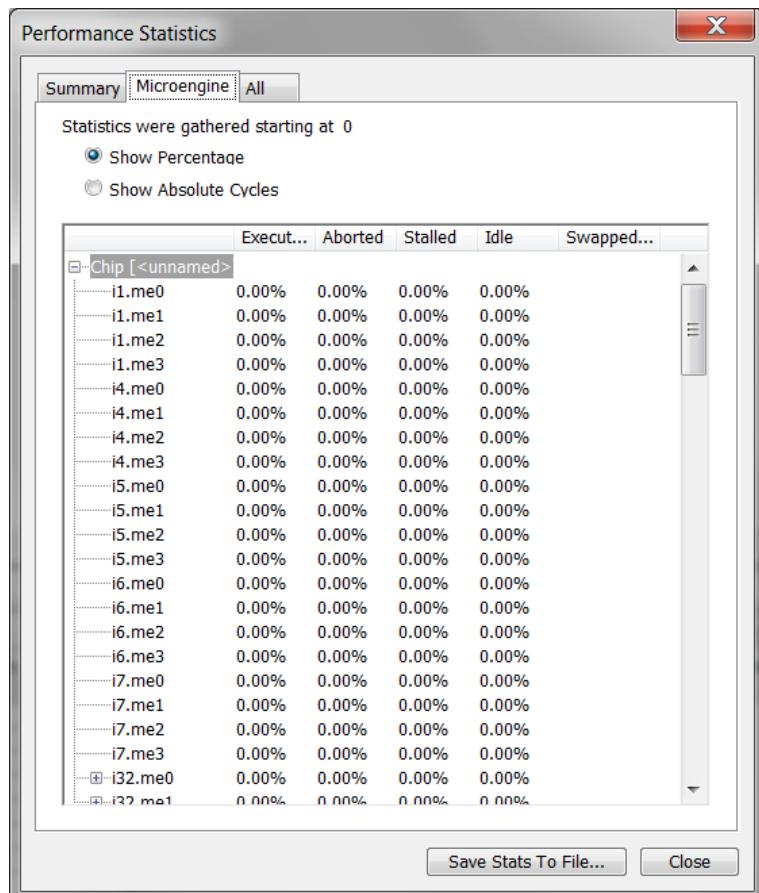


Figure 2.62. Performance Statistics - Microengine Tab

6. Click the **All** tab.

The **All** statistics page displays as shown in Figure 2.63.

The **All** statistics page allows you to look at all of the statistics gathered by the Network Flow Simulator. 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**.

Programmer Studio saves your customized list along with the project debug settings.

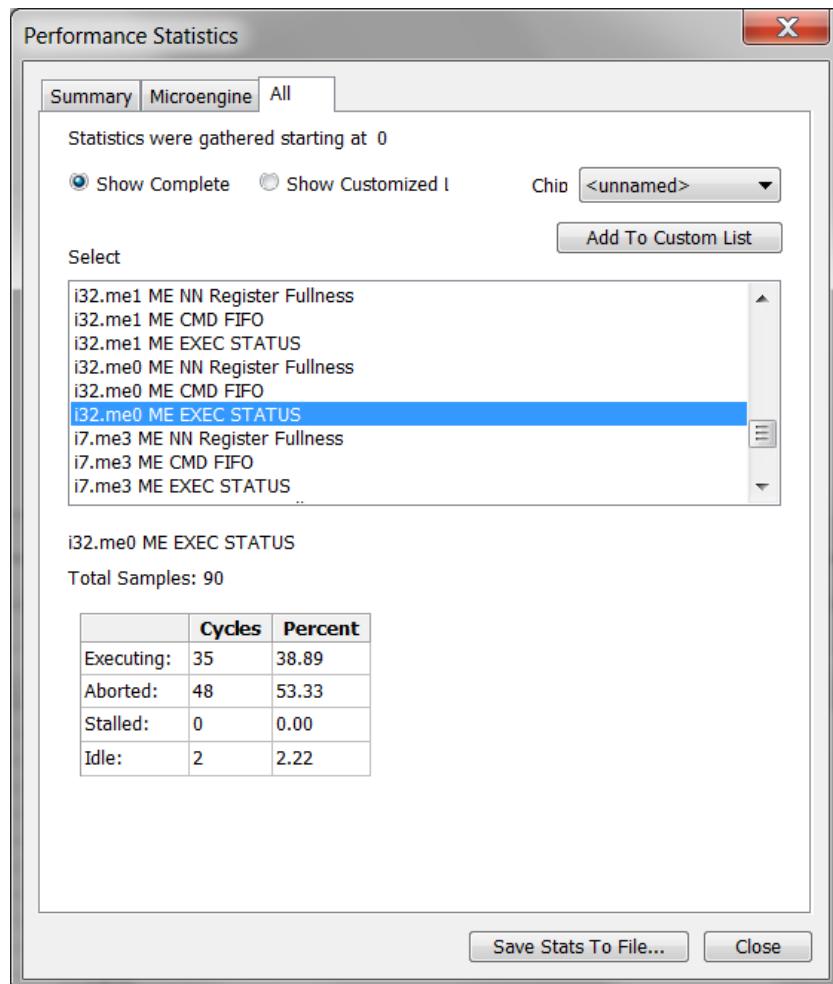


Figure 2.63. Performance Statistics - All Tab

2.12.18.2 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**.

2.12.19 Performance Statistics

Programmer Studio provides the ability to monitor DSF-CPP bus activity and provide overall chip performance and display statistics. Statistics gathering is available only in **Simulation** mode.

2.12.19.1 Displaying Statistics

To display **DSF/Mini Packet Performance Statistics**:

1.  Start debugging in Programmer Studio.
2. On the **View** menu, click **Debug Windows**, then click **Performance Statistics**. Or click .

The **DSP Performance Statistics Diagram** window appears with **DSF-CPP Bandwidth** tab selected.

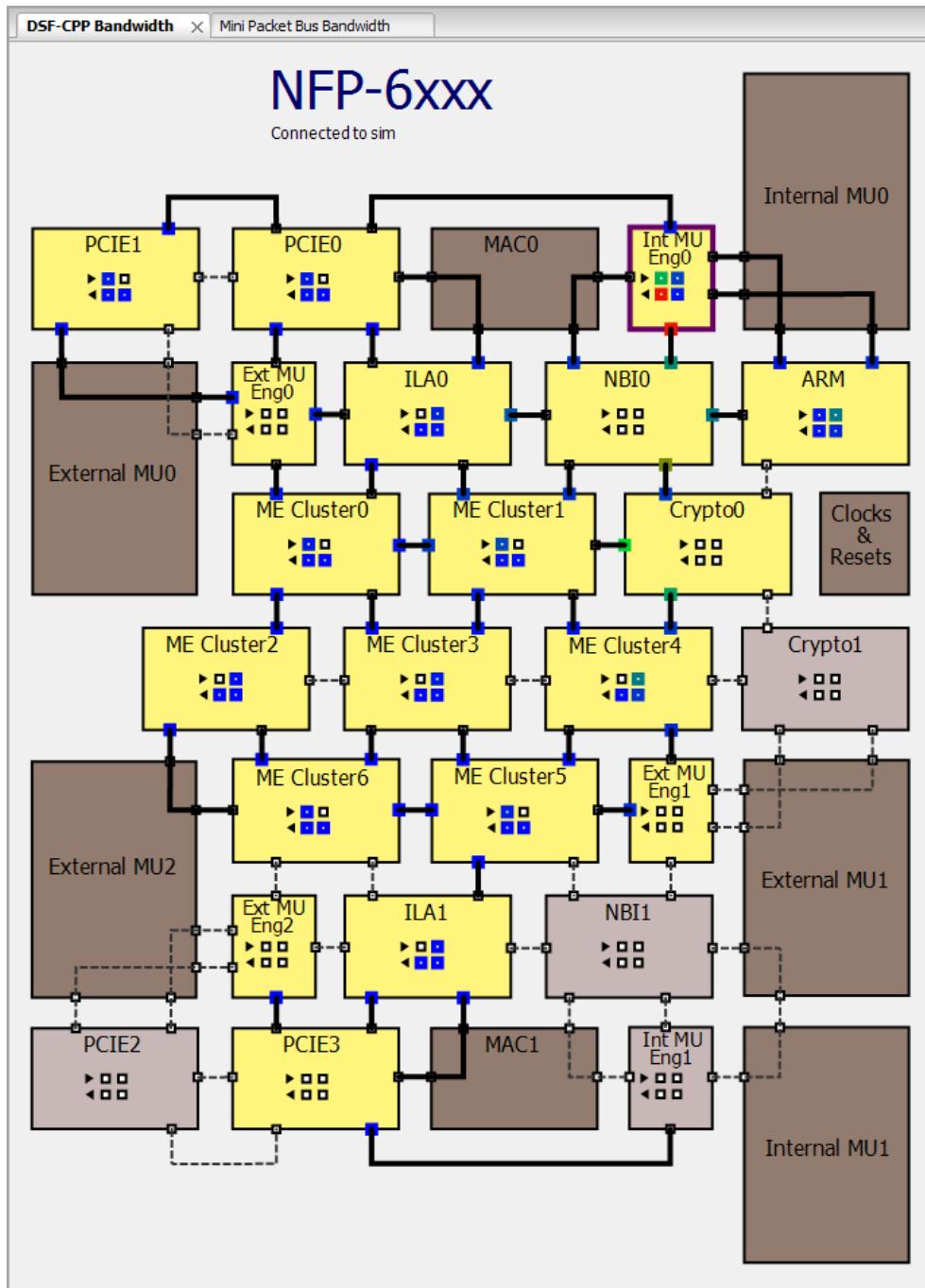


Figure 2.64. Performance Statistics - DSF-CPP Bandwidth Tab

The **DSF-CPP Bandwidth** page shows the DSF-CPP bandwidth and usage percentage statistics over a sliding cycle window. Statistics include CPP Commands, Data Ports 0 and 1, Pull IDs.

3. Click the **Mini Packet Bus Bandwidth** tab.

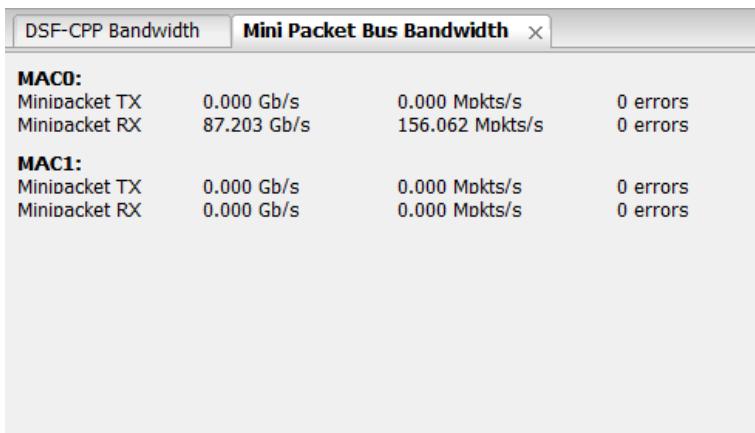


Figure 2.65. Performance Statistics - Mini Packet Bus Bandwidth Tab

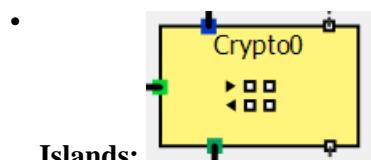
The **Mini Packet Bus Bandwidth** page shows the ingress and egress speed packet rate and error counts between **MAC** and **NBI** islands.

4. Click the Run toolbar button in either the diagram window or in the Programmer Studion main window.
5. Click the Stop toolbar button in either the diagram window or in the Programmer Studion main window to stop recording statistics.

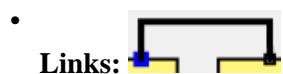
2.12.19.2 Diagram Elements



When a port is active its outline will be bold and coloured according to a heat scale from blue through green and ending in red. The colour represents the usage percentage of the port.



When an island is active it will be coloured yellow.



When a link between two ports is active it will be bold.

2.12.19.3 Diagram Controls

To interact with the diagram use the mouse. When the mouse cursor is hovered over an active element in the diagram it will have a purple colour style and if any statistics are available for it, they will be displayed on the **Selection** panel.

Scrolling or Panning can be done by either clicking and dragging anywhere on the diagram or by using the mouse wheel when the diagram has focus.

Zooming can be done by holding down the Ctrl key and using the mouse wheel when the diagram has focus.

2.12.19.4 Selection Panel

Selection		
Throughput for Island 28		
Statistic	Bandwidth	% Usage
Ingress Commands	0.30 Gops	15.04
Ingress Data	12.29 Gbps	4.80
Ingress Pull ID	0.00 Gops	0.00
Egress Commands	0.00 Gops	0.00
Egress Data	6.61 Gbps	2.58
Egress Pull ID	0.19 Gops	9.54

Figure 2.66. Performance Statistics - Selection Panel

If an Island or Port is selected in the diagram and there are statistics for it the data will be displayed in this panel.

- **Ports:** Shows bandwidth and percentage usage for ingress and egress ports.
- **Islands:** Shows total throughput of bandwidth and percentage usage for all ingress and egress ports on the island.

2.12.19.5 Options Panel

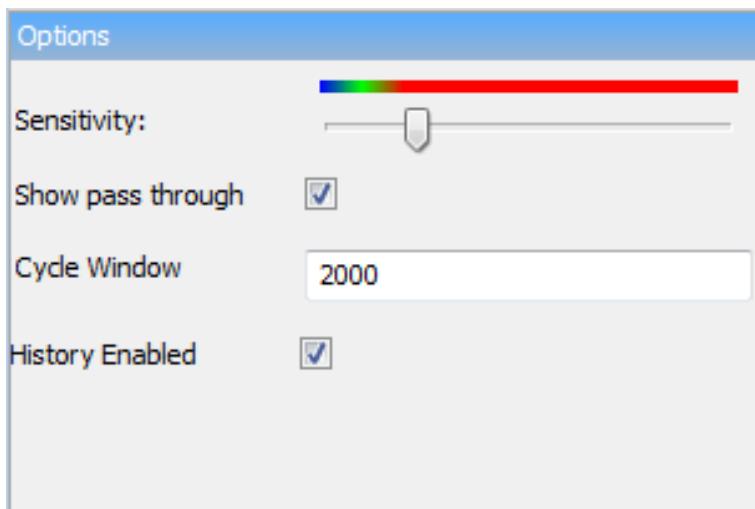


Figure 2.67. Performance Statistics - Options Panel

On the **Options** panel various settings can be changed that affects the diagram display and statistics recording.

- **Sensitivity:** A factor to multiply the heat value of the ports in the diagram with. This is useful to exaggerate the contrast between ports when usage numbers are low. Note the colour band above the slider as a reference to the scaling used.
- **Show pass through:** Show or Hide the components in the diagram that do not have statistics and are passed though.
- **Cycle Window:** The number of cycles in the sliding cycle window that statistics are collected for. The number can only be changed when the simulator is stopped. Changing it will reset the statistics.
- **History Enabled:** When this option is checked, statistics history will be recorded for both DSF and Mini Packets interfaces.

2.12.19.6 Statistics History

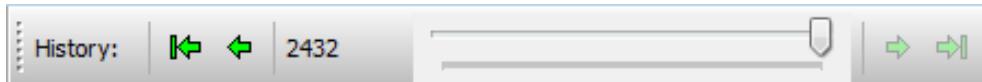


Figure 2.68. Performance Statistics - History Toolbar

If the **History Enabled** option is checked, statistics history will be recorded for both DSF and Mini Packets interfaces.

History can be navigated by the buttons and slider on the history toolbar. Note that only the cycles that were recorded while the Performance Statistics frame was open will be available.

The buttons on the history toolbar are (from left to right) First available cycle, Previous available cycle, Next available cycle and Last available cycle. The slider can be used to scroll through all available history.

The history position should be on the last cycle if you wish the diagram to update while the simulator is running.

2.12.20 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.

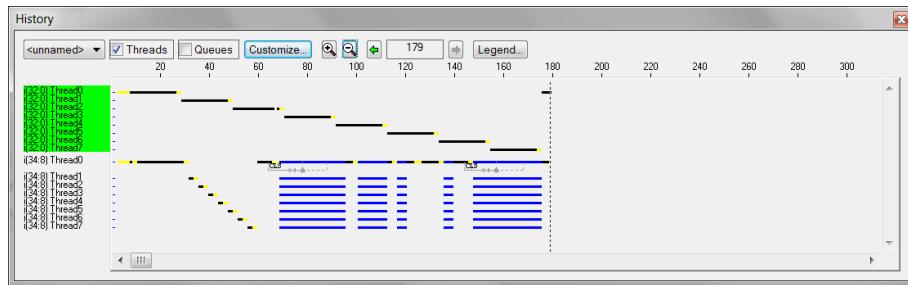


Figure 2.69. History Window

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 of which you want the threads 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.

2.12.20.1 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.

2.12.20.2 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 eight queue groups - EMEM, IMEM, Cluster Local Scratch (CLS), Cluster Target Memory (CTM), Microengines, NBI, ILA and PCIE - 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 you click on a grayed group box, it and all the contained queue's boxes become checked.

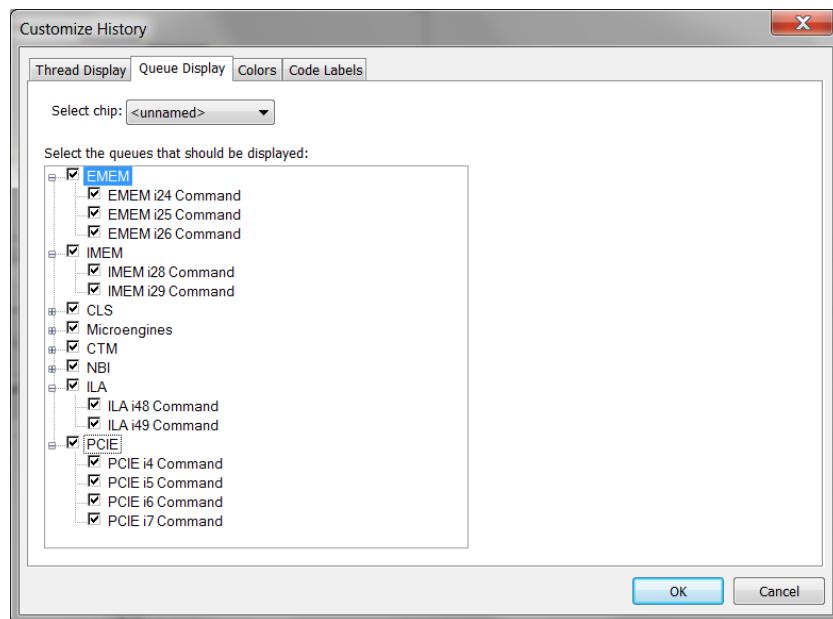


Figure 2.70. Queue Display Property Sheet

2.12.20.3 Hardware Debugging Restrictions

When debugging in the hardware configuration, thread and queue history is not supported.

2.12.20.4 Scaling the Display

To control the horizontal scale for the history display, use the zoom in  and zoom out  buttons.

2.12.20.5 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 indicates that item will be 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 you click on a grayed Microengine box, it and all the contained thread's boxes become checked.

Regardless of the state of the individual thread settings, you 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.

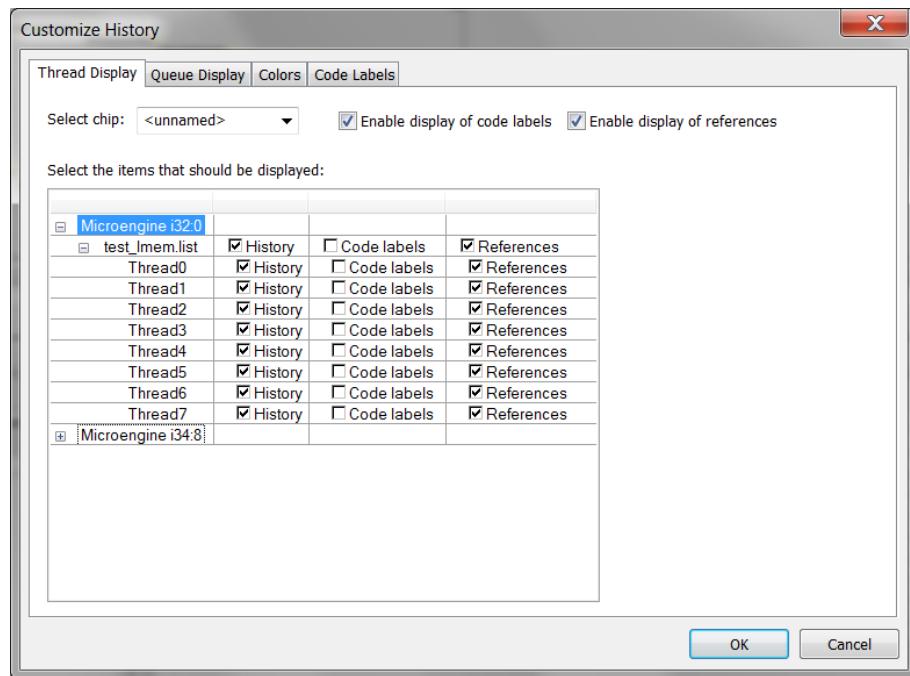
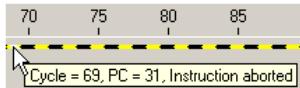


Figure 2.71. Display Threads Property Page

To get details about a thread's history, position the cursor over the history line and wait a moment. Programmer Studio displays the cycle count, PC, and the instruction state in a pop-up window beneath the cursor.



2.12.20.6 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 2.71).

- 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>**.

2.12.20.7 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:

- EMEM
- IMEM
- ARM
- CTM

- CLS
- PCIe
- NBI
- ILA

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 15, then the reference line under thread 63's history looks like:



And the reference line under thread 15's history looks like:



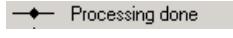
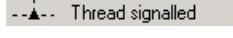
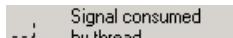
To get details about a reference:

- Position the cursor over the reference line and wait a moment.

Programmer Studio 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 Section 2.12.20.8.

On a reference line, Programmer Studio displays markers at the cycles when reference events occur.

 Put into queue	Displayed at the cycle when the reference is put into the queue of the unit being referenced.
 Removed from queue	Displayed at the cycle when the unit removes the reference from the queue.
 Processing done	Displayed at the cycle when the unit finishes processing the reference.
 Thread signalled	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.
 Signal consumed by thread	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.

You control the display of reference history by clicking the **Customize** button in the **History** window and then clicking the **Thread Display** tab. The property page shown in Figure 2.72 appears. Display of all references is enabled or disabled by checking or unchecking the **Enable display of references** check box. If this button is checked, 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, uncheck the box corresponding to that Microengine.

To display all references for a Microengine, check the box corresponding to that Microengine, as was done for Microengine 0:0 in Figure 2.72.

To hide references for a thread, uncheck the box corresponding to that thread. To display references for a thread, check the box corresponding to that thread. If some threads in a Microengine have references displayed and others do not, 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 threads 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, display them by right-clicking on the thread's history line and selecting **Display References for Threadn**. If a thread's references are displayed, hide them by right clicking on the thread's history line and selecting **Hide References for Threadn**.

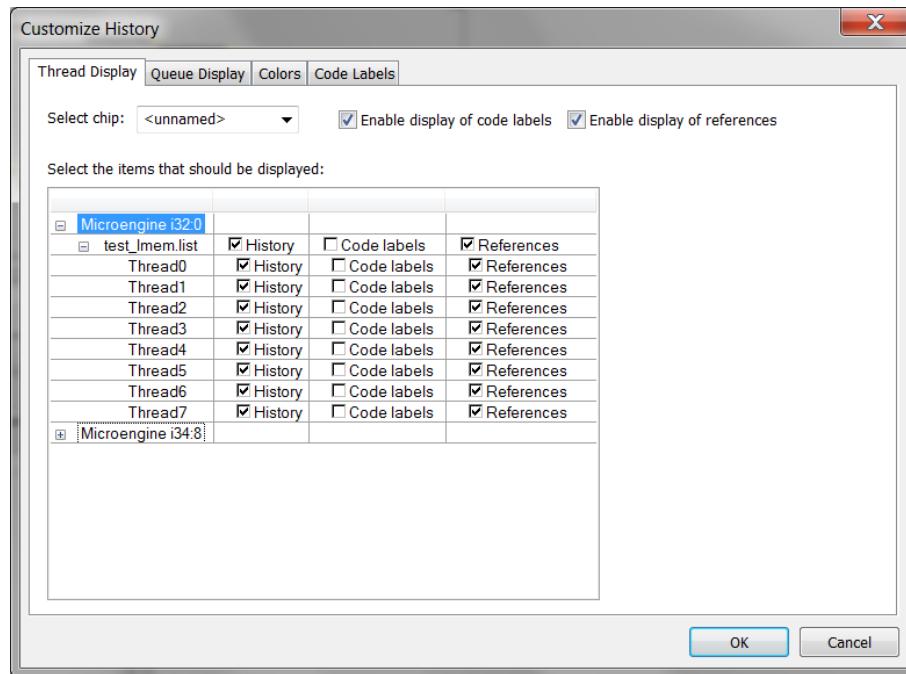


Figure 2.72. Customize History

2.12.20.8 Changing Thread History Colors

By default, Programmer Studio displays a thread history line in:

if an instruction is executing.

if an instruction is aborted.

if the thread is stalled.

if the microengine is idle.

By default, the reference line colors are:

EMEM black

IMEM red

CLS gray

ARM purple

NBI bright blue

CTM green

ILA yellow
PCI dark red

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 that you want to change the color of.
4. Select the new color.
5. Click **OK** after specifying the colors you want.

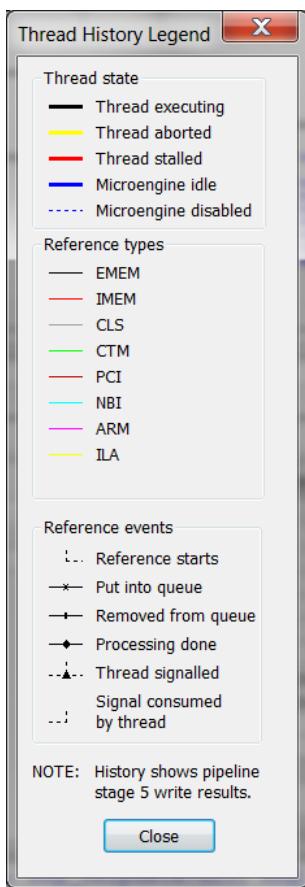


Note

For consistency, the execution state colors also apply to the instruction markers that are displayed in the thread windows.

2.12.20.9 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**

2.12.20.10 Tracing Instruction Execution

To view the instruction that a thread was executing at a given cycle count:

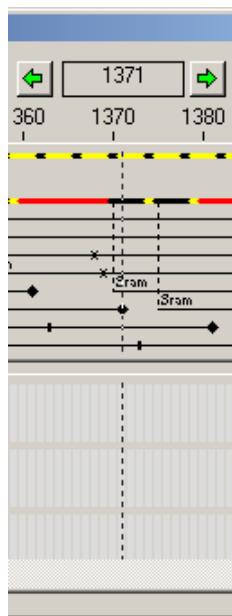
1. Right-click the thread's history line at the cycle count in which you are interested.
2. Click **Go To Instruction** on the shortcut menu.

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). 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.

2.12.20.11 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 a second.
2. Programmer Studio displays the number of entries in the queue and the size of the queue in a popup 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 2.12.21 for more information on the Queue Status window.)

2.12.21 Queue Status

The queue status window provides current and historical information on the contents of the eight queue groups: EMEM, IMEM, CLS, CTM, NBI, PCIe, ILA 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 2.73).

Queue	# Entries	Command	Address	Thread	PC	Cycle	# Lwords	Sig Done	FIFO Type
EMEM	0								
EMEM i24 Command	0								
EMEM i25 Command	0								
EMEM i26 Command	0								
IMEM	0								
IMEM i28 Command	1	[IMEM] cam128 lookup24 add increment	0x0000001F00	Thread4	216	1633	15	Yes	MEM
IMEM i29 Command	1	[IMEM] atomic write	0x0000001020	Thread0	346	1628	1	Yes	MEM
CLS	0								
Microengines	0								
Island 1	0								
Island 4	0								
Island 5	0								
Island 6	0								
Island 7	0								
Island 32	0								
Island 33	0								
Microengine i33.0 Command	0								
Microengine i33.1 Command	0								
Microengine i33.2 Command	0								
Microengine i33.3 Command	1	[IMEM] write	0x000001E800	Thread1	338	1667	4	Yes	MEM
Microengine i33.4 Command	0	[IMEM] subtract	0x0000001A40	Thread0	356	1665	1	Yes	MEM
Microengine i33.5 Command	0								
Microengine i33.6 Command	1								
Microengine i33.7 Command	0								
Microengine i33.8 Command	0								
Microengine i33.9 Command	0								
Microengine i33.10 Command	0								
Microengine i33.11 Command	0								
Island 34	0								
Island 35	0								
Island 36	0								
Island 37	0								
Island 38	0								
Island 48	0								
Island 49	0								
CTM	0								
PCI	0								
NBI	0								
NBI i8 Command	0								
NBI i9 Command	0								
ILA	0								
ILA i48 Command	0								
ILA i49 Command	0								

Figure 2.73. Queue Status Window

Select the chip of which the 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 EMEM, IMEM, CLS, CTM, ILA, PCI, NBI 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, Programmer Studio displays:

- The type of reference.
- The address being referenced.
- The thread that made the reference.
- The PC of the instruction that made the reference.
- The cycle count at which the reference was made.
- The number of longwords being referenced.
- Whether a signal is generated when the reference is completed.

Instruction Cross-reference

If you right-click a reference and click **Go To Instruction** on the shortcut menu, Programmer Studio 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.

2.12.21.1 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.

2.12.21.2 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.

2.12.22 Event List View

The **Event List** view shows a filtered list of memory reference and packet events sorted by cycle time. You can scroll backward and forward in time until an interesting event is identified. Right-clicking on the event and selecting the **Go To Event** option from the popup context menu causes the Programmer Studio **Thread History**, **Data Watch**, **Memory Watch**, and all **Thread Windows** to synchronize with the cycle of the selected event. You can then easily see what the microcode application was doing at the time of the event was logged. This may help you debug your microcode application.

You can display the **Event List** view by selecting the **View→Debug Windows** pulldown menu as shown in Figure 2.74. To enable display of memory reference and packet events, the **Collect reference history** and **Collect packet history** options must be checked on the **History** page of the **Simulation Options** dialog. Figure 2.75 shows a sample **Event List** View.

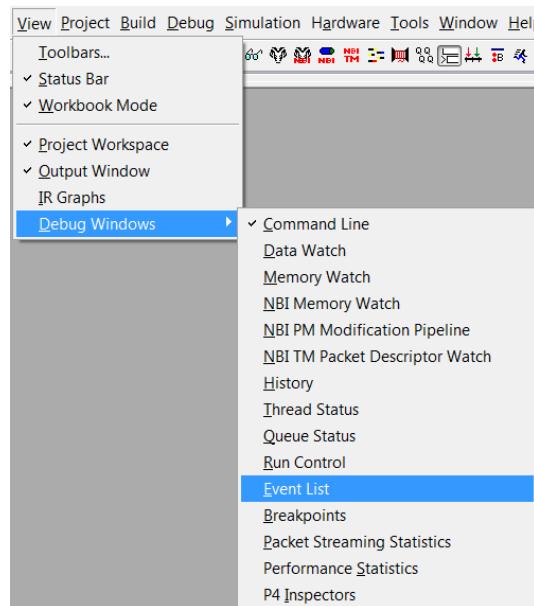


Figure 2.74. Enable Display Windows for Event List View

Event List					
Cycle	Thread	Type	SubType	Attributes	
1703	(34:1) Thread2	Reference	IMEM generic write	PC->340 Start Address=0x000000A060, End Address=0x000000A067, Longwords=1	
1704	(32:2) Thread5	Reference	IMEM write	PC->108 Start Address=0x0000150018, End Address=0x0000150027, Longwords=2	
1707	(32:4) Thread2	Reference	IMEM cam128 lookup24 add increment	PC->216 Start Address=0x0000001B40, End Address=0x0000001B78, Longwords=15	
1709	(32:5) Thread2	Reference	IMEM read32	PC->114 Start Address=0x000016002C, End Address=0x000016005B, Longwords=6	
1709	(33:1) Thread0	Reference	MEM PACKET ENGINE complete unicast	PC->365 Start Address=<unknown>, End Address=<unknown>, Longwords=6	
1710	(32:3) Thread5	Reference	IMEM read32	PC->114 Start Address=0x000015002C, End Address=0x000015005B, Longwords=6	
1712	(32:4) Thread0	Reference	IMEM clear	PC->361 Start Address=0x0000001EC0, End Address=0x00000001EC7, Longwords=1	
1712	(33:5) Thread2	Reference	IMEM write	PC->108 Start Address=0x0000160000, End Address=0x0000160077, Longwords=2	
1715	(33:1) Thread2	Reference	MEM PACKET ENGINE add thread	PC->216 Start Address=<unknown>, End Address=<unknown>, Longwords=6	
1716	(33:6) Thread0	Reference	IMEM cam128 lookup24 add increment	PC->216 Start Address=0x0000001A40, End Address=0x0000001AB8, Longwords=15	
1717	(32:3) Thread0	Reference	MEM PACKET ENGINE complete unicast	PC->365 Start Address=0x0000001FC0, End Address=0x00000000E7, Longwords=1	
1717	(34:3) Thread3	Reference	IMEM cam128 lookup24 add increment	PC->216 Start Address=0x0000000040, End Address=<unknown>, Longwords=6	
1718	(32:2) Thread1	Reference	IMEM clear	PC->361 Start Address=0x0000000040, End Address=0x000000007B, Longwords=15	
1718	(33:5) Thread2	Reference	IMEM read32	PC->114 Start Address=0x000016002C, End Address=0x000016005B, Longwords=6	
1719	(33:4) Thread1	Reference	IMEM write	PC->338 Start Address=0x000000C900, End Address=0x000000C31F, Longwords=4	
1722	(32:2) Thread2	Reference	IMEM clear	PC->361 Start Address=0x0000000180, End Address=0x0000000187, Longwords=1	
1722	(34:3) Thread0	Reference	IMEM clear	PC->361 Start Address=0x0000000E00, End Address=0x0000000E07, Longwords=1	
1723	(32:3) Thread0	Reference	MEM PACKET ENGINE add thread	PC->100 Start Address=<unknown>, End Address=<unknown>, Longwords=6	
1723	(33:4) Thread1	Reference	IMEM write8	PC->342 Start Address=0x000000C810, End Address=0x000000C811, Bytes=0	
1723	(34:1) Thread3	Reference	IMEM write	PC->338 Start Address=0x000000EC00, End Address=0x000000EC1F, Longwords=4	
1727	(32:2) Thread4	Reference	IMEM cam128 lookup24 add increment	PC->216 Start Address=0x0000001440, End Address=0x000000147B, Longwords=15	

Figure 2.75. Event List View

The memory reference events are logged automatically by Programmer Studio.

The slider is used to change the range of events shown in the view. The position of the slider relative to the start and end cycle counts establishes roughly the top and bottoms events shown. For example, if the start cycle is 100 and the end cycle is 200 and the slider is positioned in the middle, then an event near cycle count 150 is displayed at the top of the list. If there is no event at cycle count 150, then events near to 150 are displayed and the slider repositioned to represent that cycle count.

The up and down arrows are used to display a new page-worth of events.

2.12.22.1 Event List Filtering

You can cut down the number and type of events displayed by the **Event List** view using the **Filtering...** option (see Figure 2.76). You can choose to display almost any combination of reference events types. You can choose to show all types of reference events, or only those types that you select from the lists.

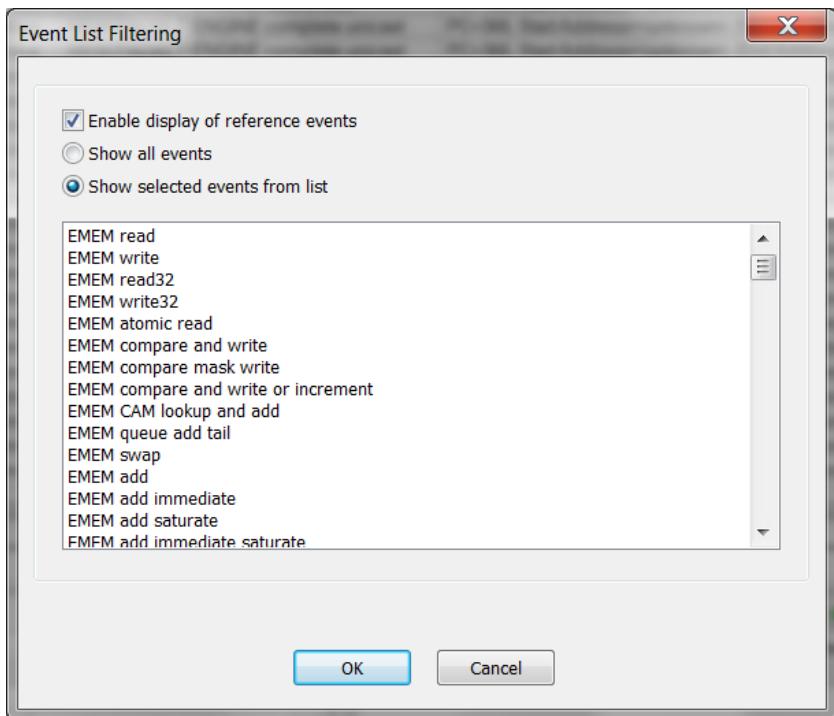


Figure 2.76. Event List Filtering

2.12.23 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 2.77), or

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

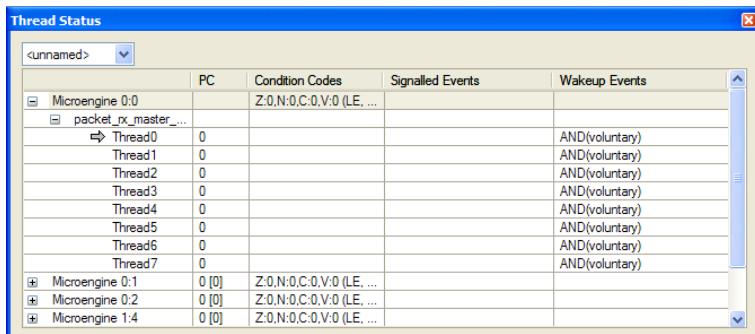


Figure 2.77. The Thread Status Window

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 the chip of which the status is displayed by 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 Programmer Studio 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.

2.12.24 P4 Inspectors

In debug mode, you can monitor the state of various P4 runtime values in the **P4 Inspectors** window.

2.12.24.1 Packet Page

The screenshot shows the 'Packet' tab of the P4 Inspectors window. The interface has a tree view on the left and a table view on the right. The table columns are 'Packet Structure', 'Value', and 'Width'. The tree view shows the following structure:

- Status: Valid
- Headers:
 - ethernet:
 - dstAddr: 0x0, Width: 48
 - srcAddr: 0x0, Width: 48
 - etherType: 0x800, Width: 16
 - ipv4: (dirty)
- Metadata:
 - routing_metadata:
 - nhop_ipv4: 0x0, Width: 32
 - standard_metadata:
 - packet_length: 60, Width: 32
 - egress_instance: 0, Width: 32
 - instance_type: normal, Width: 32
 - clone_spec: 0, Width: 32
 - egress_spec: p20, Width: 9
 - egress_port: p0, Width: 9
 - ingress_port: p0, Width: 9
 - test_metadata:
 - val0: 0xbeef, Width: 32
 - val1: 0x0, Width: 32

Refresh

ME:

i32:me0

CTX:

Thread 0

On the Packet Page a tree of the current packet parse representation is displayed for the selected Micro Engine and Context. The columns of the tree shows the header/field names, values and bit widths of the values. At the top of the tree is the status which can be **Valid** or **Incomplete/Invalid**. The next node contains the parsed packet **Headers** and below it the program **Metadata**.

2.12.24.2 Match Tables Page

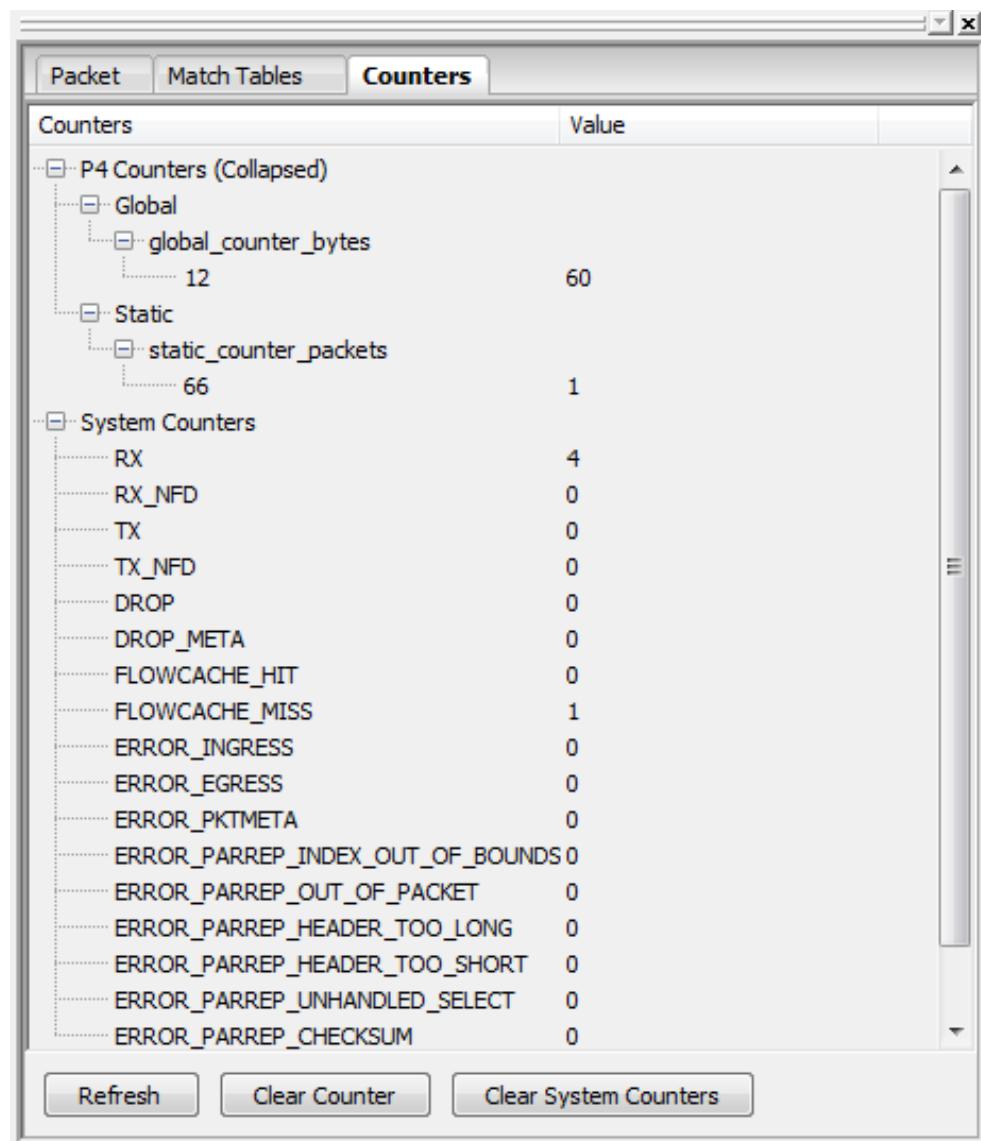
The screenshot shows a software interface titled "Match Tables". At the top, there are three tabs: "Packet", "Match Tables" (which is selected), and "Counters". Below the tabs is a table with five columns: "Tables and Rules", "Value", "Counter", "Counter: packets", and "Counter: bytes". The table displays the following data:

Tables and Rules	Value	Counter	Counter: packets	Counter: bytes
forward				
send_frame				
ipv4_lpm				
default: set_nhop	fwd_counter 1		60	
nhop_ipv4	0x0			
port	0x14			

At the bottom left of the main window is a "Refresh" button.

The Match Tables Page displays a tree of tables and under them the actions that have been matched to rules and action parameters. If a **Direct Counter** is associated with the table, its name and packets/bytes values will be displayed next to the action.

2.12.24.3 Counters Page



The Counters Page displays a tree of **P4 Counters** and **System Counters**. P4 counters include **Static Counters** and **Global Counters**. The type of the counter (packets/bytes) is included in the counter name. The counters are collapsed, this means that only non-zero counters are shown. To clear a P4 counter, select the node in the tree and click **Clear Counter**. All system counters can also be cleared by clicking **Clear System Counters**.

3. Assembler

This chapter provides information on running the Assembler. Background information on the Assembler functions appears in the *Netronome Network Flow Processor NFP-6000 Databook* for the network processor.

3.1 Assembly Process

This section describes how to invoke the Assembler and the steps that it takes when processing a microcode file.

3.1.1 Command Line Arguments

The Assembler is invoked from the command line:

```
nfas [options] microcode_file [microcode_file ...]
```

where the options consist of:

-verbose	Print more information on successful build, such as a register usage summary.
-chip <i>chip_id</i>	Select target chip. This is the preferred method of targeting chip types. A chip definition also includes a set of chip revisions. For example, -chip nfp-6xxx will target all revisions of nfp-6xxx and -chip nfp-6xxx-a0 will only include revision A0. It is possible to define custom chips with custom chip IDs in nfp_sdk_user_chips.json. If no -chip is specified, the environment variable NFP_CHIP_ID is checked. The default chip is nfp-6xxx
-chip_list	List all defined chips as found in chip data and user definition files.
-nfp <i>TYPE</i>	Set processor type, where <i>TYPE</i> is 6xxx or 6xxxc. Letter 'c' indicates the presence of the crypto unit. It is preferred to use -chip nfp-6xxx instead of -nfp6xxx and -chip nfp-6xxxc instead of -nfp6xxxc.
-ctx_start <i>n</i>	Only use contexts starting with <i>n</i> .
-ctx_end <i>n</i>	Only use contexts up to and including <i>n</i> .
-pml	Enables protect macro local feature (see Section 3.1.2.1)
-indirect_ref_format_nfp3200	Use NFP-32XX indirect reference format.
-indirect_ref_format_nfp6000	Use the NFP-6xxx indirect reference format.(Default)
-third_party_addressing_32_bit	Use 32-bit third party addressing mode.
-third_party_addressing_40_bit	Use 40-bit third party addressing mode (default).
-preproc32	Emulate 32 bit arithmetic in preprocessor expressions. It is best to use ELF32 for the output NFFW file with -preproc32 and ELF64 with -preproc64, with ELF32 then effectively being limited to 2GiB

	addressing space, because nfas expressions and constants are all signed integers.
-preproc64	Use 64 bit arithmetic in preprocessor expressions (default).
-preproc_c	Preprocess undefined macro to value 0.
-lm <i>start</i>	Define the <i>start</i> of local memory allocation in bytes. The starting and ending address of local memory must be aligned on 4-byte boundary.
-lm <i>start:size</i>	Define the <i>start</i> and <i>size</i> of local memory allocation in bytes. The starting and ending address of local memory must be aligned on 4-byte boundary.
-lr	Dumps the register liverange information into a file with a .lri extension. If the register allocation fails, the .uci file may not be created so the information will be available for viewing in the .lri file.
-o <i>file</i>	Use <i>file</i> 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

These two optimization options are independent of each other; in other words any combination can be specified. The default option, **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.
-spilling-no-code	Enables codeless spilling. That is spilling will not insert extra instructions to your code for use with spilling. See ???
-spill-def-ind IND	Specifies the default local memory index that is used with spilling. (See ???). Defaults to 1
-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.
-R	Register declarations are required (this is the default).
-Wn	Set Warning Level (n=0-4).
-w	Disable warnings (same as -W0).
-WX	Report error on any warning.
-C	Enables case sensitive assembly.
-help or --help	Displays help.
-version or --version	Displays current nfas version.
-t	Enable "terse" output.

-keep_unreachable_code Do not comment out unreachable code.

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 macros `-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=A2` 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`

<code>-REVISION=rev</code>	Targets assembly to chip version <code>rev</code> . This is equivalent to setting options ' <code>-REVISION_MIN=rev</code> ' and ' <code>-REVISION_MAX=rev</code> ' with the same value.
<code>-REVISION_MIN=rev</code>	Targets assembly to the minimum chip version <code>rev</code> . (The default is 0.)
<code>-REVISION_MAX=rev</code>	Targets assembly to the maximum chip version <code>rev</code> . (The default is 15, no limit.)

The following options are passed to the preprocessor:

<code>-P</code>	Preprocess only into a file: <code>microcode_file.ucp</code> .
<code>-E</code>	Preprocess only into stdout.
<code>-I<code>directory</code></code>	Add the <code>directory</code> to the end of the list of directories to search for included files.
<code>-D<code>name</code></code>	Define <code>name</code> as if the contained <code>"#define name 1"</code> .
<code>-D<code>name</code>=<code>def</code></code>	Define <code>name</code> as if the contained <code>"#define name def"</code> .
<code>-N</code>	Disable the preprocessor.

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 NFAS is invoked with no command line arguments, a usage summary is printed.

In Windows environments, the Assembler may also be invoked through Programmer Studio. In Linux environments, the Assembler is available as a command-line executable called `nfas` and has the same functionality and options as `NFAS.exe`.

3.1.2 Protect Macro Locals

Implementation of the Protect Macro Locals feature addresses a long-standing problem with the Assembler that is essentially caused by the fact that the preprocessor (like the C-preprocessor) is a text-based preprocessor and does not deal with semantically meaningful elements. The problem occurs when a register (or signal or label) is passed in to a macro as an argument, and that macro also declares a local register with the same name.

For example:

```
#macro test[arg]
.begin
.reg tmp
immed[tmp, 0x1234]
alu[arg, arg, +, tmp]
.end
#endm
.reg tmp
test[tmp]
```

This code is expanded by the preprocessor into:

```
.reg tmp
.begin
.reg tmp
immed[tmp, 0x1234]
alu[tmp, tmp, +, tmp] ; !!!! See comments below
.end
```

In the line `alu[tmp, tmp, +, tmp] ;` one of the ‘`tmp`’ registers refers to the local register, and another ‘`tmp`’ to the register that is passed in. However this distinction is lost during the expansion.

Previously, in order to address this issue, the Assembler generated a warning (Number 5155) when *there was a possibility that this condition would be encountered*. The Protect Macro Locals feature is intended to fix the problem by preventing it from occurring in the first place.



Note

The preprocessor is very general and can be used in a variety of ways. This feature only addresses the normal usage that accounts for the vast majority of cases.

When this feature is enabled, the preprocessor looks for register and signal declarations that are defined within a local scope (i.e. within a `.begin`/ `.end` pair) within the macro, and any labels defined within the macro. It does this using the original/unexpanded macro body text. This means that if there are constructs such as unmatched `.begin`/ `.end` directives due to `#if`, then the preprocessor’s *behavior gets unpredictable*.

If the preprocessor encounters such tokens that do not match with the name of a macro parameter, it appends a prefix to those tokens and references to them. This means that the above example expands to:

```
.reg tmp
.begin
.reg __M00000__tmp
immed[__M00000__tmp, 0x1234]
alu[tmp, tmp, +, __M00000__tmp]
.end
```

and the local and passed-in registers are distinct. The Assembler then filters out the prefix so that it appears in the list file as:

```
.reg 10000!tmp
.begin
.reg 10001!tmp
immed[10001!tmp, 0x1234]
alu[10000!tmp, 10000!tmp, +, 10001!tmp]
```

3.1.2.1 Enabling and Disabling the Protect Macro Local Feature

The preprocessor modifies the code and it has the potential to modify it incorrectly. For this reason, the Protect Macros Local feature is implemented with the option to be enabled or disabled on a **per-macro** basis. The enabling/disabling option is controlled by two directives:

```
#protect_macro_locals on

#protect_macro_locals off
```

The two directives increment and decrement the count of how many times the options are turned on. *At the time the macro is defined if the count is greater than 0 then the Protect Macro Local feature is enabled.*

A command-line argument “`-pml`” initializes the count to 1 (indicating that the feature is enabled). If this argument is missing, the count is initialized to 0 (indicating that the feature is disabled).

The presence of the command-line argument allows for several different usage models.

- The feature can be enabled initially (either through the directive or command-line argument), and the `#protect_macro_locals off` directive can be used to disable it for a specified macro.
- Alternately, the feature can be disabled initially, and the `#protect_macro_locals on` directive can be used to enable it for a specified macro.



Note

Since the directives are maintained as a count, the `#protect_macro_locals on/off` pair can nest; that is if there are two “on” directives in a row, then one would need two “off” directives to undo their effect.

The local registers for this feature are defined as registers declared within a `.begin/.end` block (where the `.begin/.end` are in the same macro body) without a global (or similar) keyword.

Identifiers and labels in the source code should avoid substrings of the form: “`__M#####_`”, where # represents a decimal digit. If such substrings appear and the Protect Macro Locals feature is enabled, then these identifiers and labels will not display properly in the list file.

3.1.3 Arithmetic Notation Feature

The Assembler supports the use of arithmetic notation for simple expressions that can be assembled into a single ALU instruction or into one or two IMMED instructions. The format is:

```
register = expression
```

The instruction must be on a single line, although you can always break it into multiple lines by using the line continuation character, for example:

```
x = y + \
z
```

If you prefer a C-language syntax, you can end the expression with a semicolon, although this is just a special case of a post-comment. The statement can be preceded by labels (either on the same line or on previous lines) and by comments on previous lines; for example:

```
; this is a comment
lab1#:
lab2#: x = 1 ; post-comment
```

A caveat is that there must be some white space between the register name and the '=' character; for example:

```
x=1 ; This is invalid as it is interpreted as the opcode "x=1"
```

The white space is necessary because the Assembler accepts some opcodes that contain the '=' characters. The '=' character may or may not be followed by white space.

In the case where the expression evaluates to a constant, it will be assembled into one or two IMMED instructions (actually it is an IMMED/IMMED_W1 pair). If the constant (taking into account shifts) can fit into a single instruction, only one IMMED instruction is used.

If a single instruction is not sufficient for the constant, and the destination is not a general purpose register (GPR) (for example: a transfer register or a neighbor register), then three instructions are generated—the first two to place the constant in a GPR and the third to copy the result to the real destination.

In the case where the expression is a constant expression that does not evaluate to a constant at the start of the assembly process (for example: if the expression includes an imported variable or references the address of a register or a signal), two instructions are used for safety (that is, all 32-bits of the register are set).

If two instructions are generated, a default level-3 warning (Number 5165) is generated. The warning is generated to alert the programmer that what appears to be a single instruction is actually two instructions. The warning might cause problems; for example, when someone codes:

```
br[lab#], defer[1]
x = 0x12345678
```

To avoid this warning, you can always use #PRAGMA WARNING to disable it. On the other hand, if you never want the Assembler to generate two instructions from one arithmetic instruction, you can use the same mechanism to turn this warning into an error.

If the expression does not evaluate to a constant, then the Assembler tries to assemble it into a single ALU or ALU_SHF instruction. Examples include:

```
x = y ; alu[x, --, b, y]
x = ~y ; alu[x, --, ~b, y]
x = ~(x << 2) | y ; alu[x, y, or, x, <<2]
```

The Assembler tries to resolve constant sub-expressions as early on in the expression tree as possible. Therefore, if the code reads:

```
x = x &~ 1 ; ERROR, this will not assemble
```

then it is not assembled into “alu[x, x, and~, 1]”. Rather it is assembled into “alu[x, x, and, (~1)]” which generates an error since the constant (~1) is too large.

If a shift is allowed but not used, and the constant is too big to fit into the ALU operand, but a shifted version will fit, then the Assembler inserts the appropriate shift. Because of this, “ $x = x | (1<<20)$ ” assembles correctly as “alu[x,x,or,1,<<20]”.

Note also that the unary “~” operator has a higher precedence than

“<<”, so that

```
x = x &~ y<<2 ; ERROR, this will not assemble
```

will not assemble since this is interpreted as $(x \& ((\sim y)<<2))$. In this case, the programmer probably intended:

```
x = x &~ (y<<2) ; This is the correct version.
```

Note that since this is being done by the Assembler itself rather than the preprocessor, it can distinguish between imported variables and registers. For example:

```
.reg x y
.import_var z
x = y ; generates an ALU to copy a register
x = z ; generates an IMMED/IMMED_W1 pair
```

If the expression cannot be converted to either an IMMED or ALU, an error is generated. Note that in some cases of invalid expressions, the expression may be converted to an IMMED or ALU, and then the IMMED/ALU generates the actual error.

3.1.4 Assembler Steps

As shown in Figure 3.1, invoking the Assembler results in a two-step process composed of preprocessing and assembly steps. The preprocessor step takes a .uc file and transforms it in memory by expanding macros, replacing #define and #define_eval literals, and processing include files. This preprocessed image is written out to a .ucp file. The next step transforms the preprocessed memory image into the .list file. At the end of assembly, an auxiliary file with the file name extension of .uci is also created. The .uci file contains error information and register allocation map which are useful for debugging purposes. During the assembly step the following functions are performed:

- Checks instruction restrictions.
- Resolves symbolic register names to physical locations.
- Optimizes the code.
- Resolves label addresses.
- Translates symbolic opcodes into binary patterns.

The preprocessor is invoked from within the Assembler. Command line options are available when invoking NFAS.exe (or NFAS.dll via the workbench).

```
nfas [options] microcode_file [microcode_file ...]
```

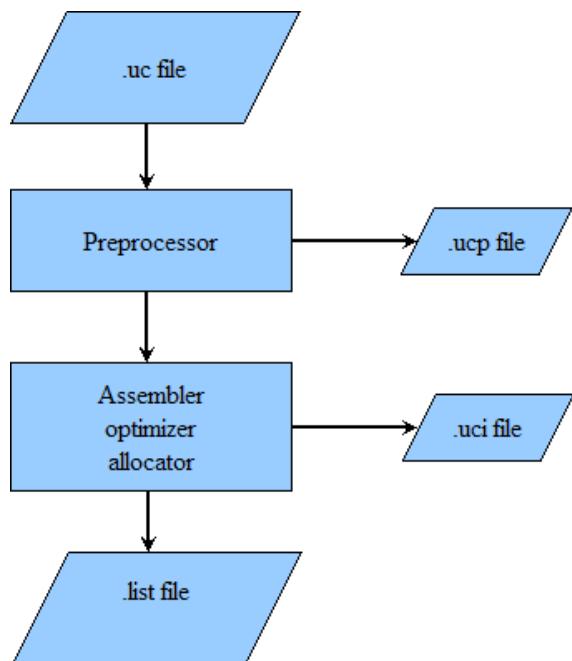


Figure 3.1. Assembly Process

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.

3.1.5 Case Sensitivity

The command line arguments are case sensitive. If the "-C" command line argument is not specified, the microcode file is case insensitive – all text in the file, with the exception of comments, is converted to lower case.

If case sensitivity is enabled (the "-C" argument was specified), the Assembler does not convert input file text to lower case. Predefined macros and predefined import variables, as well as user-defined registers, signals, and import

variables will all be case sensitive. For compatibility with earlier releases, Assembler keywords will be handled in a case insensitive manner (converted to lower case). These include the following:

- Instruction and directive names (e.g. alu and .reg)
- Operand Keywords (e.g. alu operators and CSR names)
- Built-in preprocessor function names.

3.1.6 Assembler Optimizations

The assembler optimizer performs the following optimizations:

- It will move instructions down to fill defer shadows for instructions that support the defer token.
- Unconditional branches to the next instruction are removed.
- It will move instructions down to fill defer shadows for instructions that support the defer token.
- It will remove unnecessary NOPs and preserve NOP_VOLATILEs.
- It will move instructions down to replace NOPs that cannot be removed, it will not replace NOP_VOLATILEs.

For more information, please see the *Databook* for the network processor.

3.1.7 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, you can specify a type and range of revisions for which you want your microcode assembled. This is done using the following command line options: **-chip <SKU>** (where SKU is a valid SKU show by the **-chip_list** option) and **-REVISION_MIN** and **-REVISION_MAX**. The legacy options **-nfp-6xxxc**, **-nfp3216**, **-nfp3216c**, **-nfp3240** and **nfp3240c** can also be used but the **-chip** is preferred. The **-REVISION_MIN** and **-REVISION_MAX** options are only needed when the SKU has multiple chip revisions.

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 macros will be defined according to type and revision.

For more information on the predefined macros and on writing version specific microcode, please see the *Netronome NFP-6XXX Databook* for the network processor.

4. C Compiler

The Netronome Flow Processor NFP-6000 supports microcode compiled from C language code to support the Microengines and their threads.

The C Compiler for Netronome Processors is available in **Explicit Partitioning Mode** that control the Netronome Flow Processor NFP-6000. This mode of the C Compiler is available on installing the Netronome NFP SDK Base/Encryption CD's Tools component that also installs Programmer Studio. With this Compiler mode, the programmer explicitly controls the actions of the various microengines on the Netronome Processor chip. For general information on how to use this Compiler refer to Section 2.7.

This chapter explains the subset of the C language supported by the C Compiler for Netronome Processors for Explicit mode and the extensions to the language to support the unique features of the Netronome Flow Processor NFP-6000.

The network processor supports microcode compiled from C language code to support the Microengines and their threads. You can create the C code using Programmer Studio or any suitable text editor, and then compile and link the code using Programmer Studio.

4.1 Explicit Mode Command Line

You can use the compiler command-line interface from a command prompt window on your system. To compile your source code with the command-line interface, do the following:

1. Open a command prompt window.
2. Go to the folder containing the C source files, typically: C:\<NFPSDK_6000>\bin where <NFPSDK_6000> is the user-defined installation directory.
3. Invoke the C compiler using this command:

```
nfcc -chip <chip_id> [options] filename...
```

The name that you assign to a source file is usually determined by your organization's naming conventions and policies. Generally, it's useful for a file name to be easily recognizable and intuitively connectable with the function of the file. For example, whereas a file name like AA345.c is relatively meaningless, file names like SwitchBostonNE4.c and string.h are more intuitive.



Note

C:\<NFPSDK_6000>\bin should be on your PATH. If you want to run the C compiler from the command line, the system path needs to be set up accordingly.

4.1.1 Explicit Mode Supported Compilations

Two kinds of compilations are supported:

- Compile one or more source files (*.c, *.i) into separate object (*.obj) files.
- Compile any combinations of source file (*.c, *.i) and/or object file (*.obj) into one list file (*.list).

In the first case, you must use the -c option in the command line in order to compile .c files into separate .obj files. You might want to use this method to compile .c files that don't change very often, for example, rtl.c, so that you don't have to recompile them every time you make a .list file.

Example:

```
nfcc -chip nfp-6xxx -c file1.c file2.i
```

In the second case, do not use the -c option. In the following example, two source files (.c and .i) and an object file (*.obj) are compiled to produce a .list file.

Example:

```
nfcc -chip nfp-6xxx file1.c file2.i rtl.obj
```

4.1.2 Explicit Mode Supported Compiler Options

Table 4.1 lists and defines all the supported C compiler command-line options. The Command-Line Interface (CLI) will ignore unknown options and issue warning messages. The CLI honors the last option if it conflicts with a previous one, for example,

```
nfcc -chip nfp-6xxx -c -O1 -O2 file.c
```

this generates the following warnings and proceeds:

```
nfcc: Command line warning: overriding '-O1' with '-O2'
```

If you enter other conflicting options such as -E and -EP, the last option entered always prevails.

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 4.1. Supported CLI Options

Option	Definition
-? -help --help	List the available options.
-c	Compile only: for each .c or .i file, produce a .obj file but do not produce the final .list file.

Option	Definition
-compat32	Use NFP-32xx compatible modes and settings.
-Dname[=value]	Specify a #define macro. The value, if omitted, is 1.
-E	Preprocess to stdout.
-EP	Preprocess to stdout, omitting #line directives.
-P	Preprocess to a .i file.
-Fo<file> -Fo<dir>	Specify name of object file, or object directory (for directory, include a trailing slash).
-Fe<file>	Set base name of executable. Defaults to the base name of the first source or object file specified on the command line followed by the extension (.list).
-Fi<file>	Override the base name of the .ind file.
-FI<file>	Force inclusion of a file.
-chip <chip_id> (where <chip_id> is a generic or specific chip SKU.)	Specify the target processor. Chip nfp-6xxx is the default. The compiler adds -D__NFP_IS_6000 or -D__NFP_IS_3200 as appropriate. To list the supported processors, use the -chip_list option.
-chip_list	Print a list of possible chip targets.
-I path[;path2...]	Path(s) to include files, prepended before path(s) specified in environment variable NFCC_INCLUDE.
-indirect_ref_format_nfp6000	Use NFP-6xxx compatible indirect reference format. This option defines __NFP INDIRECT_REF_FORMAT_NFP_6000.
-indirect_ref_format_nfp3200	Use NFP-32xx indirect reference format (default). This option defines __NFP INDIRECT_REF_FORMAT_NFP_3200.
-ng	Run alternative compiler (experimental).
-On	Optimize for: <ul style="list-style-type: none"> • n=1: size (default) • n=2: speed • n=d: debug (turns off optimizations and inlining, overriding -Obn below).
-Obn	Inlining control: <ul style="list-style-type: none"> • 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).
-Qapp_metadata=<string>	Inserts the specified string into the .list file.

Option	Definition
-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 <code>__declspec(sram)</code> . 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.
-Qliveinfo	Equivalent to -Qliveinfo=all
-Qliveinfo=gr,sr,...	<p>Print detailed liveness information for a given set of register classes:</p> <ul style="list-style-type: none"> • gr: general purpose registers • xr: SRAM/xfer read registers • xw: SRAM/xfer write registers • xrw: SRAM/xfer 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. Note: For Netronome® NFP-32XX network processors, setting -Qlm_start=1024 reserves all the local memory, and thus disables local memory usage by the compiler.
-Qlm_unsafe_addr	Disables the compiler's use of local memory auto increment addressing. Used when user code writes local memory pointers with invalid values. See ??? for more information.
-Qlmptr_reserve	Reserves local memory base pointer <code>1\$index1</code> for user inline assembly code.
-Qmapvr	Prints out pseudo-assembly code with annotations that map physical registers to user variables and compiler-generated temporary variables.
-Qnctx=<1, 2, 3, 4, 5, 6, 7, 8>	Specifies the number of contexts that will be made active in your program. Unused contexts will be made to execute the <code>ctx_arb[kill]</code> instruction and terminate. Compiler-allocated resources such as memory will not be allocated to unused threads. The underlying number of contexts supported by the hardware will not be changed, so hardware-managed resources such as registers will still be allocated to all threads. Defaults to the value of -Qnctx_mode (which defaults to 4). If -Qnctx is set greater than the value of -Qnctx_mode, -Qnctx_mode will be changed to the higher value.

Option	Definition
-Qnctx_mode=<4, 8>	Specify the number of contexts that the hardware should support. Changing this value from 4 to 8 halves the number of available context specific registers. Defaults to 4.
-Qnn_mode=<0, 1>	Sets NN_MODE in CTX_ENABLE for setting up next neighbor access mode. <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • -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 ('_').
-Qperfinfo=n	Prints performance information. It is possible to specify multiple options by repeating the -Qperfinfo option several times. <ul style="list-style-type: none"> • 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: <deprecated> • n=8: Function sizes • n=16: Local memory allocation • n=32: Live range conflicts causing IMEM 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 • n=1024: Information about the Boolean propagation optimization • n=2048: Register requirements report • n=4096: Information on switch statement optimizations • n=8192: Print information on I/O parallelization.
-Qrevision_min=n -Qrevision_max=m	The version arguments allow the compiler to generate code that works on a range of processor versions (steppings). <ul style="list-style-type: none"> • 0x00=A0 (default for -Qrevision_min) • 0x01=A1 • 0x10=B0

Option	Definition
	<ul style="list-style-type: none"> • 0x11=B1 <p>The default revision range is 0x00 to 0xff (all possible processor versions). The default for <code>-Qrevision_max</code> is 0xff. The compiler adds <code>-D__REVISION_MIN=n</code> and <code>-D__REVISION_MAX=m</code>. Note: The Netronome network processor program loader reports an error if a program compiled for a specific set of processors is loaded onto the wrong processor.</p>
<code>-Qspill=<n></code>	<p>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, CLS=Cluster Local Scratch)</p> <ul style="list-style-type: none"> • n=0: LM (most preferred) → NN → IMEM (least preferred) • n=1: NN→LM→IMEM • 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: IMEM only • n=7: No spill; halt if any spilling required • n=8: LM→IMEM • n=9: NN→LM→CLS→IMEM • n=10: LM→NN→CLS→IMEM • n=11: CLS only • n=12: LM→CLS <p>Default is n=0. You must set <code>-Qnn_mode=1</code> to use the NN registers as a spill region. If the NN registers are used by program code, NN spilling will be automatically disabled.</p>
<code>-Qspill_cls_limit=<n></code>	<p>In the event that spilling is enabled with cluster local scratch as possible spill region with IMEM following, the spill limit n will dictate how many bytes are allocated in cluster local scratch for spilling. Default is n=65536 and can be a value between 0 and 65536.</p>
<code>-third_party_addressing_32_bit</code>	<p>Use 32-bit third party addressing mode (default). This option defines <code>__NFP_THIRD_PARTY_ADDRESSING_32_BIT</code>.</p>
<code>-Wn where n=<0, 1, 2, 3, 4></code>	<p>Warning level: (default value is 1)</p> <ul style="list-style-type: none"> • 0: print only errors • 1, 2, 3: print only errors and warnings • 4: print errors, warnings, and remarks.
<code>-Zi</code>	<p>Produces debug information. The compiler generates a file with a .dbg extension for each source.</p>

4.1.3 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 or Loader 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 tailored according to its usage.
- Generates .list file for execution on single Microengine.

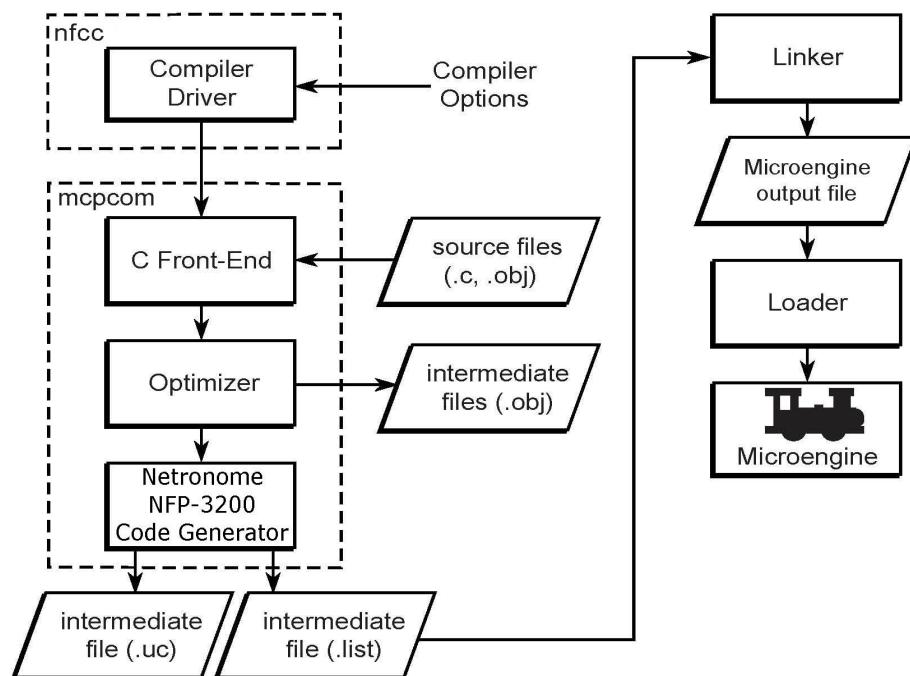


Figure 4.1. Compilation Steps

4.1.4 Case Sensitivity

The C language code as well as the command line switches are case sensitive.

5. P4 Compiler

The Netronome Flow Processor NFP-6000 supports microcode compiled from P4 language code to support the Microengines and their threads.

Netronome's P4 solution is divided into a front-end compiler and a NFP back-end compiler. The front-end compiler performs high level P4 source compilation and generates an Intermediate Representation (IR). The back-end takes the IR as an input and generates NFP specific codes as well as design information consumed by the P4 run-time tools. A final build stage compiles the generated codes into NFP firmware.

5.1 The P4 Front-end Compiler

Netronome uses a customized version of the official p4.org P4 front-end compiler, `p4-hlir`, to generate IR. The IR format used by the tools follows Open Networking Foundation (ONF) Protocol Independent Forwarding (PIF) group (An-IR) AIR format. AIR uses a simple YAML format. More information about PIF and the AIR format can be found from <https://github.com/OpenNetworkingFoundation/PIF-Open-Intermediate-Representation> and the P4.org frontend at www.p4.org.

5.1.1 Standard and Intrinsic Metadata

The P4 1.0 and 1.1 specifications defines a number of standard metadata fields. These fields have a special significance for the operation of the target. Table 5.1 includes information about widths and any NFP target related notes. Refer to the P4 Specifications for official definitions of the standard metadata fields.

Table 5.1. P4 Standard Metadata

Field	Width	Notes
<code>standard_metadata.packet_length</code>	14	Packet length in bytes.
<code>standard_metadata.ingress_port</code>	10	Opaque ingress port encoding
<code>standard_metadata.egress_spec</code>	16	Opaque egress specification encoding. It is possible to set to <code>standard_metadata.ingress_port</code> to return to source. Extra width due to channel encoding.
<code>standard_metadata.egress_port</code>	16	Opaque egress port encoding.
<code>standard_metadata.egress_instance</code>	10	Used to identify multicast instances
<code>standard_metadata.instance_type</code>	4	Contains encoding of packet recursion/duplication as follows: 0x0 - normal packet 0x1 - clone ingress to ingress 0x2 - clone egress to ingress

Field	Width	Notes
		0x3 - recirculate packet 0x4 - resubmit packet 0x8 - clone ingress to egress 0x9 - clone egress to egress 0xa - multicast packet
standard_metadata.clone_spec	32	Clone specification ID

All `standard_metadata` fields are automatically included in the P4 design. However, it is possible to include a custom header type and header instance for standard metadata. The layout of the custom definition must correspond to the definition in Table 5.1, any discrepancies will result in a warning and the incorrect field being ignored.

In addition to standard metadata, there is special handling for intrinsic metadata. This is metadata that has special behaviour that is not part of the standard pipeline. To use intrinsic metadata, the P4 application must define a header instance `intrinsic_metadata` with the type `intrinsic_metadata_t`. Every field that is defined in the header with the corresponding entries in Table 5.2 will automatically have those fields behave as described.

Table 5.2. P4 Intrinsic Metadata

Field	Width	Notes
<code>intrinsic_metadata.ingress_global_timestamp</code>	64-32	Time at start of parsing. Value is in ME cycles / 16. ME clock period can be determined from BSP HWINFO tool or API.

5.1.2 Target Specific P4 Pragmas

A number of NFP target specific pragmas are usable to achieve outcomes not describable in P4. In P4 pragmas are assigned to P4 constructs by declaring the pragma immediately before the P4 construct. Table 5.3 describes the pragmas in use.

Table 5.3. NFP Target P4 Pragmas

Pragma	Target	Notes
<code>netro reglocked</code>	Register	This pragma is used to indicate that all accesses to the given register should occur with mutual exclusion in place.
<code>netro no_lookup_caching</code>	Action	This pragma is used to disable lookup caching for a given action. This is often required when registers or meters affect lookups in a control flow. It is possible to just disable lookup caching for an egress flow and use the cached ingress flow. Disabling lookup caching will reduce performance.

Pragma	Target	Notes
netro_meter_drop_red	Meter	This will automatically drop a packet if a meter returns the colour RED. This may be used to avoid disabling lookup caching in some cases.
netro_ipv4_header	Header Instance	This enables the use of the MAC checksum verification metadata on NBI ingress for IPv4 packets. This should speed up IPv4 checksum verification for supported packets ingressing on the wire.
netro_udp_header	Header Instance	This enables the use of the MAC checksum verification metadata on NBI ingress for UDP packets. This should speed up UDP checksum verification for supported packets ingressing on the wire.
netro_tcp_header	Header Instance	This enables the use of the MAC checksum verification metadata on NBI ingress for TCP packets. This should speed up TCP checksum verification for supported packets ingressing on the wire.

5.1.3 Explicit P4 Front-end Compilations

The P4 Front-end Compiler is invoked implicitly by Programmer Studio when building from the GUI. It is possible to invoke The P4 Front-end Compiler explicitly from the command line.

You should find the P4 front-end tool `nfp4c` in a directory `p4/bin/` in the root of an SDK installation on both Windows and Linux. This tool can be run to generate IR from P4 sources.

On Windows the `nfp4term.bat` file should be run to setup environment variables and paths for the p4 and msys make tools.

The command line options for `nfp4c` are described in Table 5.4. Below is an example command line invocation:

Example:

```
nfp4c -D NO_MULTICAST -o simple_router.yml --source-info simple_router.p4
```

Table 5.4. Supported P4 Compiler CLI Options

Option	Definition
-? -help --help	List the available options.
-o <file>	The output IR filename.
--source-info	Embed P4 source info in IR output. Used for debugging.
-D [name[=value]] ...	Specify P4 #define macros.
-I <path> ...	Specify P4 include paths.
--p4-version <version>	Specify a P4 language version. Version may either be 1.0 or 1.1. Default is 1.0.

5.2 The P4 Back-end Compiler

The back-end compiler can be invoked as follows:

```
nfirc -o out_dir simple_router.yml
```

Table 5.5. Supported P4 Back-end Compiler CLI Options

Option	Definition
-? -help --help	List the available options.
-o <directory>	The output directory for the generated files.
-t <DEFAULT_TABLE_SIZE>	Default table size to apply if none is specified
--multicast_group_count <MULTICAST_GROUP_COUNT>	Number of multicast groups to support
--multicast_group_size <MULTICAST_GROUP_SIZE>	Maximum number of multicast ports per group
--parsegraph	Generate a parser graph
--globalgraph	Generate a global control graph

5.3 Building P4 Applications

P4 applications can be build with `nfp4build`. The following command performs all the steps to produce a `nffw` file that can be loaded on the NFP.

```
nfp4build -o simpler_router.nffw -4 simple_router.p4
```

Table 5.6. Supported CLI Options for building a `nffw` file

Option	Definition
-? -help --help	List the available options.
-o <NFFW_FILENAME>	The filepath where the firmware should be built to. The directory of this file is also used for generated and compiler output.
-p <directory>	The output directory for the generated PIF files.
-s <SKU>	Name of the SKU to build against.
-l <PLATFORM>	Name of the platform to build against. Options are bataan, hydrogen, starfighter1, lithium, beryllium
-4 <P4_SOURCE> ...	List of p4 source file names.
-c <SANDBOX_C> ...	List of sandbox c file names.
-r	Reduced thread usage. Use 4-context mode for microengines; this reduces memory usage by half but reduces performance.

Option	Definition
-i	Build firmware for simulator.
-e	Build with shared codestore support.
-n	Build without debug information.
-A <APPLICATION-ME-COUNT>	Number of application worker MEs to instantiate, default is maximum MEs.
--nfp4c_D [name[=value]] ...	Specify P4 #define macros.
--nfp4c_I <path> ...	Specify P4 include paths.
--nfirc_default_table_size <DEFAULT_TABLE_SIZE>	Default table size
--nfirc_multicast_group_count <MULTICAST_GROUP_COUNT>	Number of multicast groups to support
--nfirc_multicast_group_size <MULTICAST_GROUP_SIZE>	Maximum number of multicast ports per group
--nfp4c_graphs	Generate a parser and global control graph
--nfp4c_p4_version <version>	Specify a P4 language version. Version may either be 1.0 or 1.1. Default is 1.0.

6. Linker

The Linker is used to link microcode images. Microcode images are generated by the microcode compiler or Assembler, whereas application objects are generated by a Netronome ARM processor C/C++ Compiler or standard host compiler tools (like gcc). The method is C/C++ Compiler independent. Import variables can be used to share values between microcode and host/ARM applications.

This chapter describes how to use the microcode Linker, nfld. The task of nfld is to process one or more microcode Assembler, (nfas) output files, *.list, and create a NFFW file that can be loaded by the NFFW loader. nfas is described in Chapter 3.

6.1 About the Linker

Memory is shared between the Netronome ARM processor and the Microengines. A common mode of design will have the Netronome ARM processor 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 Netronome ARM processor applications to be written and compiled that can access the common address pointer.

Memory symbols (in nfas created with .global_mem or .local_mem) have addresses allocated and fixed at link time. Host/ARM applications can extract these addresses from the NFFW file or from an optional runtime symbol table in DRAM (see -rtsyms). Another option for more flexibility is to use import variables, which are specified and resolved at load time. This means host/ARM applications can control the import variable value at load time.

6.2 Microengine and Island IDs

Throughout the toolchain microengines are referenced using MEIDs and islands simply with the corresponding island ID. The island ID are also aliases with names that may be more convenient in some cases. The binary encoding of an MEID in an integer should not matter to users, but for simplicity it is currently encoded as a CPP Master ID for the microengine.

The NFP SDK provides a header file called nfp_resid.h which can be used to manipulate and build MEIDs. The C functions in nfp_resid allow parsing and building of MEID and island ID strings. The toolchain uses these to parse command line parameters and provide built-in assembler and compiler functions like __nfp_idstr2meid().

6.2.1 Island IDs

Whenever an island is identified by "iX" it can also be identified by an alias. The current aliases are listed below and closely match the island IDs in the NFP-6xxx Databook.

Table 6.1. NFP-6xxx Island ID Aliases

Island ID string	Alias ID string
i1	arm
i4	pcie0
i5	pcie1
i6	pcie2
i7	pcie3
i8	nbi0
i9	nbi1
i12	crypto0
i13	crypto1
i24	emem0
i25	emem1
i26	emem2
i28	imem0
i29	imem1
i32	mei0
i33	mei1
i34	mei2
i35	mei3
i36	mei4
i37	mei5
i38	mei6
i48	ila0
i49	ila1

6.2.2 MEIDs

Microengine ID strings use the format "<island_id>.meY" where "Y" is the 0 based microengine number within the island. The "island_id" can be either "iX" or an island alias. For example, "mei0.me0" refers to "i32.me0", the first microengine in island 32. These MEID strings can be passed to the built-in toolchain function `_nfp_idstr2meid()`, used on the command line when assigning list files or in most other places where a microengine is referenced.

6.3 Netronome Flow Firmware Linker (NFLD)

nfld is an executable that accepts a list of Microengine images (*.list) generated by the Assembler, (nfas), or by the Compiler (nfcc), and combines them into a single object that is loadable on hardware or in Programmer Studio for debugging.

6.3.1 Usage

```
nfld [options ...] list_file ...
```

6.3.2 Command Line Options

The following table lists the Linker command line options.

Table 6.2. Linker Command Line Options

Option	Definition
-h	Print a description of the nfld commands.
-res.<mem>.base byte_address	Define the base address from which the linker may allocate symbols for the resource specified by <mem> which can be iX.{emem,imem,ctm,cls}, where iX can also be an island alias.
-res.<mem>.size byte_size	Define the size of the resource from which the linker may allocate symbols for the resource specified by <mem> which can be iX.{emem,imem,ctm,cls}, where iX can also be an island alias.
-o outfile	Creates the output ELF64-format NFFW file. The default output file is the name of the first .list input file with a file type of .nffw.
-g	Include debugging information, to be used by Programmer Studio, in the output object file.
-seg file	Creates a 'C' header file defining the variables memory segments.
-u meid [meid...] [list_file]	Associates a list of microengines to subsequent list_file. The meids are of the form iX.meY, but iX can be an island alias. All microengines are assigned by default.
-codeshare meid1 meid2	Share codestore of microengines meid1 and meid2. meid1 must be an even numbered microengine and meid2 must be (meid1 + 1). The total number of available instructions doubles when sharing codestore.
-l	Used together with a preceding -u. The -l switch is used to disambiguate between microengine number and *.list file path. In earlier versions, if a *.list file name started with a number, that number would be parsed as a

Option	Definition
	microengine number rather than as part of the *.list file name. Using the -l switch permits *.list file names that begin with a number.
-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 NFFW file but with the extension .map. The .map file contains the symbols and their addresses.
-mip	Inserts a default MIP in the NFFW output image if none exists.
-rtsyms	Inserts a runtime symbol table in NFFW output image.
-noecc	Disable codestore ECC (correction/detection) in output image.
-nn_chain.<island> {A,B}	<p>Set an island's next neighbor chain mode:</p> <ul style="list-style-type: none"> • A - normal NN chaining • B - alternate NN chaining <p>Default is A. See linker help for more information.</p>

6.4 Generating a Microengine Application

On development system:

```
% nfas ueng_i32_me0.uc -o ueng_i32_me0.list
% nfas ueng_i32_me1_5.uc -o ueng_i32_me1_5.list
% nfld -u i32.me0 ueng_i32_me0.list -u i32.me1 i32.me2 i32.me3 i32.me4 \
i32.me5 ueng_i32_me1_5.list -o ueng.nffw
```

In some shells, like bash, it can be convenient to use the following syntax when assigning to multiple microengines at once:

```
% nfld -u i32.me0 ueng_i32_me0.list -u i32.me{1..5} \
ueng_i32_me1_5.list -o ueng.nffw
```

6.4.1 Using shared control store feature

If there is a need to link a large image containing more than 8K instructions, it is possible to use a shared control store feature of the Netronome Flow Processor. This can be done by sharing a large file between two paired micro engines.

For example, if you need to link a list file large.list, it is possible to do so by the following command:

```
nfld -u i32.me2 large.list -u i32.me3 large.list -codeshare i32.me2 i32.me3 -o large.nffw
```

This effectively will produce a link image large.nffw that is split between two micro engines (island 32, microengines 2 and 3 in this example). All even instructions will be loaded into microengine 2 and all odd instructions into microengine 3.

Another way to use shared control store is to concatenate two list files into one large list file, loading it onto two code-sharing microengines and setting the entry point for the odd microengine to the start of the second list file (the linker takes care of this automatically). In this mode (concatenated shared control store), the even microengine can be considered isolated from the odd microengine, whereas with normal shared control store the two microengines share memory symbols, code and microengine settings. The benefit of concatenated shared control store is that one list file can be small and the other larger than 8K and no changes would be required to use this mode. To do so, simply enable shared control store for the two microengines and then assign the first list file to the even microengine and the second list file to the odd microengine.

6.4.2 Using next neighbor configurations

The linker command line option -nn_chain.iX allows controlling the island's next neighbor configuration. "X" is the island ID. A value of A is the default and means normal linear next neighbor chaining is used. A value of B means alternative chaining mode is used (e.g. iX.me0,2,4,...(iX.last - 1),1,3,...(iX.last).).

This means correct link time interpretations of next neighbor register declarations and is useful for shared codestore usage. It will also result in a correct startup state when the NFFW is loaded to hardware or simulator, removing the need to perform ME initialisation code for this purpose.

6.5 Syntax Definitions

6.5.1 Import Variable and Memory Symbol Definition

This scheme provisions the sharing of configuration and address pointers between firmware and ARM/host software. The run-time symbol table provides the addresses and resources of memory symbols used in code, while import variables provide more generic, global, values which can typically be used for load-time configuration based on the target platform.

Symbols and import variables are closely related. They are used in nearly the same manner by the linker and loader, except that memory symbols are fully resolved at link-time while import variables are always moved to load-time. The exceptions to this are predefined import variables, such as __MEID, which are all fully resolved at link-time. Symbols and import variables both use the same import expression format in list files and the same evaluation mechanism in the linker. They can also be mixed in one expression.

An useful feature of using memory symbols is that the linker will report on any expression results that overflowed bit field which will be patched with the result of an import expression. With import variables, the linker can only do an initial evaluation of the expression, but the final value depends on the load-time value given to the import variable.

Import variables are declared using the .import_var keyword in the microcode source file (*.uc), prior to the variables being used. The assembler and compiler will create a list of microword addresses and the field bit positions within the microword where the variables are used and provide the information in its output file (*.list) in the format as described below. The linker will process the output files (*.list), possibly one for each Microengine, and store the import expressions as NFP specific ELF relocations. The loader has support for defining import variables at load-time during which it will evaluate the import expressions and apply the relocations (code patching). From the NFP SDK, nfiv can be used to set and resolve import variables in a NFFW file at any time. This may be useful for verification or pre-preparing specific versions of a firmware file for different targets. The modified NFFW file can then be loaded on hardware or in the simulator without the need for load-time import variable definitions.

Format for nfas source (*.uc) file:

```
.import_var variable_name variable_name ...
```

Format for nfcc source (*.c) file:

```
LoadTimeConstant( "variable_name" )
```

Format of output (*.list) files:

```
.%import_expr page_id uword_address
<msb:lsb:rshf, ...msb:lsb:rshf> fixup_expr
```

Where:

variable_name	String name of the external variable as declared in the source file.
page_id	String name of the page identifier. This is kept for historical purposes and no longer has any meaning.
uword_address	An integer number indicating the micro word address (instruction number or program counter).
<msb:lsb:rshf, ...msb:lsb:rshf>	A list of a maximum of four integer triads representing the bit fields most and least significant bit positions, and the number by which to right shift the result of the expression after it is evaluated. The NFFW file has a limited set of combinations that can be used for load-time expressions, while the linker can patch any msb, lsb and rshf field at link-time if the given expression is fully resolvable at link-time.
fixup_expr	Expression evaluated by the loader at load-time when all the symbols in the expression are resolved. The resulting value is used to modify the microcode instruction.

6.5.2 Microengine Assignment

Assigning list files to microengines generally map directly to loading of the code in those list files to the assigned microengines. When using concatenated shared code store, the code is loaded to the two code store sharing microengines. The -u option on the linker command line lists the MEID(s) of the microengines that the subsequent list file should be assigned to. 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.

The microengine ID (MEID) is a string of the form iX.meY. In previous SDK versions the -u parameter would take integer arguments for specifying microeninge numbers. This would be less readable and sometimes confusing when the microengine numbers would be printed in decimal format. The string based MEID clearly specifies a microengine within a specific island.

6.6 Examples

6.6.1 .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. This file is essentially the same as a symbol table printed by standard ELF tools (such as readelf), apart from the more readable memory region names.



Note

The address and size are 64-bit values for all regions.

```
Memory Map file: /tmp/o.map
Date: Tue Jan 28 13:06:58 2014

nfld version: 5.0.0.1-beta,  NFFW: /tmp/o.nffw

Address      Region      ByteSize      Symbol
=====
0x0000000000000000  i24.emem    64          ee1
0x0000000000000000  i25.emem    64          i32.me1.ee2
0x0000000000000000  i32.cls     32          i32.ccl
0x0000000000000040  i25.emem    64          i32.me2.ee2
0x0000000000000080  i25.emem    64          i33.me1.ee2
0x0000000000000000  i33.cls     32          i33.ccl
0x00000000000000c0  i25.emem    64          i33.me2.ee2

ImportVar           Uninitialized Value
=====
v                  0
```

6.7 Init-CSR

6.7.1 General Description

The NFP-6xxx NFFW file, linker and loader use a generic mechanism called init-csr to initialize the chip and the microengines prior to microcode execution as well as allowing the target to reserve specific CSRs where required so that NFFW files may not override them. The CSR database on the target and in the NFFW file are validated at load-time and any conflicts will result in the NFFW being rejected. An initialization conflict can occur by value or mode, where mode is a statement of intent by the firmware or target. For example, the firmware can state that it wants ownership of a CSR by marking it as "volatile" and if the target has already marked this same CSR as "const", the firmware is rejected. This conflict checking is explained in more detail later in this section. Note that this section will refer to a CSR as a whole, but in each instance a bit-field of a CSR can be used instead. Using bit fields in user code is recommended as it is more readable and allows the toolchain to perform some validation, such as reporting errors when trying to write to read-only fields.

The same init-csr sections are used to initialize both hardware and simulation via back-door/unclocked access. Microengine registers (GPRs, transfer registers and next neighbor registers) are also initialized using init-csr. While the init-csr mechanism does not provide any guarantee to users regarding initialization order, the init-csr sections do have an initialization order which is used by the linker to ensure a clean and simple load process.

The details of the format of these sections in the ELF file is provided in Section 6.8.

6.7.2 CSR Name Syntax and Scope

The CSR lookup string used in microcode .init_csr directives is built using documented names from the NFP-6xxx Databook with some minor tweaks. The lookup string starts with a top level target or domain with a colon separating it from the rest of the string, followed by possible island or microengine ID after which the documented map names are used up to bit fields within the CSR.

Format:

```
tgt:ID.map.map.reg.field
```

Table 6.3. Init-CSR Lookup Target Map Names

tgt	Map Name	ID
arm	ARMGasketMemoryMap	None
cls	ClusterScratchCppMap	iX
ila	IlaCppAddressMap	iX
nbi	NbiAddressMap	iX
pcie	PcieInternalTargetsCpp (same as ClusterScratchSSB, but with CPP offsets)	iX
xpb	ChipXPB	iX shorthand for real map
xpbm	ChipXPBM	iX shorthand for real map
mecsr	MeCsrCPP	iX.meY[.ctxN]

The "ID" part can be omitted only when used from microcode or microC when it is necessary to refer to the local island or local microengine. For example, "cls:i32.Rings" will always refer to island 32, but "cls:Rings" is only valid in a list file and refers to the island the list file is assigned to. The linker will resolve these local-scope names into absolute CSR names. The same is true for "mecsr:i32.me0.Mailbox0" and "mecsr:Mailbox0". Note that alternative names are available for microengine CSRs. These alternative names are the same as those used in code, for example INDIRECT_LM_ADDR_0 and IndLMAddr0.

The iX in "ID" can also be an island alias. See the Assembler User Guide for more information and examples.

6.7.3 CSR Database Conflict Rules

As described in the Assembler User Guide, there are "const", "required", "volatile" and "invalid" init-csr modes or value types. The linker handles "volatile" as two separate cases: "volatile-init" when the initialization also sets an initial value for the CSR and "volatile-noinit" when the initial value is not set. These mean more or less the same thing inside the linker and when comparing the NFFW CSR database with the target database during firmware loading the conflict rules for both volatile modes are again handled exactly the same way. To explain the process, one database will be referred to as the target database and the other as the external database.

6.7.3.1 List files and linker databases

In this scenario, all init-csr entries in all list files are gathered into a single external database and compared to the linker's target database. This means that conflict checking between list files are handled slightly differently and that is what will be explained here. The secondary conflict checking between list files and linker databases works the same way as between NFFW and load target. Some microengine CSRs are controlled by the linker, most of them are derived from command line parameters or other microcode directives, others are used to control execution entry points and other startup settings. Such 'reserved' CSRs cannot be initialized by the user and the linker will report an error in those cases.

The IMB CPP Address Translation CSRs will be set by the linker only if the user did not set them already, essentially providing the recommended defaults. The Assembler User Guide has examples of this, but generally users should not need to change the default IMB CPPAT CSRs. The default IMB CPPAT CSRs stored in the NFFW file are

only for those islands where the user assigned list files. The load target will, in turn, verify these. Some islands may allow user overrides and some may reject them. Currently, the IMB CPPAT CSRs on Class 1 islands (12 microengine islands) are considered user-overridable and the CSRs on the other islands are reserved. A user can still use 'required' init-csr entries to validate assumptions or requirements in code loaded to those islands.

The table below provides a comparison matrix of init-csr modes and values when a CSR is initialized from two different list files. For comparisons, valA is not equal to valB.

Table 6.4. List file Init-CSR conflict rules

Existing Entry	New const valA	New const valB
const valA	Same, ignored.	Value conflict, error.
const valB	Value conflict, error.	Same, ignored.
required valA	const ensures this, replace.	Value conflict, error.
required valB	Value conflict, error.	const ensures this, replace.
volatile-init valA	Mode conflict, error.	Mode conflict, error.
volatile-init valB	Mode conflict, error.	Mode conflict, error.
volatile-noinit	Mode conflict, error.	Mode conflict, error.
invalid valA	value conflict, error.	const ensure this, replace.
invalid valB	const ensure this, replace.	value conflict, error.
Existing Entry	New required valA	New required valB
const valA	const ensures this, ignored.	Value conflict, error.
const valB	Value conflict, error.	const ensures this, ignored.
required valA	Same, ignore.	Value conflict, error.
required valB	Value conflict, error.	Same, ignore.
volatile-init valA	Mode conflict, error.	Mode conflict, error.
volatile-init valB	Mode conflict, error.	Mode conflict, error.
volatile-noinit	Mode conflict, error.	Mode conflict, error.
invalid valA	value conflict, error.	required ensure this, replace.
invalid valB	required ensure this, replace.	value conflict, error.
Existing Entry	New volatile-init valA	New volatile-init valB
const valA	Mode conflict, error.	Mode conflict, error.
const valB	Mode conflict, error.	Mode conflict, error.
required valA	Mode conflict, error.	Mode conflict, error.
required valB	Mode conflict, error.	Mode conflict, error.
volatile-init valA	Same, ignored.	Value conflict, error.
volatile-init valB	Value conflict, error.	Same, ignored.
volatile-noinit	volatile-init ensures this, replace.	volatile-init ensures this, replace.
invalid valA	Mode conflict, error.	Mode conflict, error.

Existing Entry	New volatile-init valA	New volatile-init valB
invalid valB	Mode conflict, error.	Mode conflict, error.
Existing Entry	New volatile-noinit	
const valX	Mode conflict, error.	
required valX	Mode conflict, error.	
volatile-init valX	volatile-init ensures this, ignored.	
volatile-noinit	Same, ignored.	
invalid valX	Mode conflict, error.	
Existing Entry	New invalid valA	
const valA	Value conflict, error.	const ensure this, ignored.
const valB	const ensure this, ignored.	Value conflict, error.
required valA	Value conflict, error.	const ensure this, ignored.
required valB	const ensure this, ignored.	Value conflict, error.
volatile-init valA	Mode conflict, error.	Mode conflict, error.
volatile-init valB	Mode conflict, error.	Mode conflict, error.
volatile-noinit	Mode conflict, error.	Mode conflict, error.
invalid valA	Same, ignored.	No conflict, keep both.
invalid valB	No conflict, keep both.	Same, ignored.

6.7.3.2 NFFW and load target databases

In this scenario, we consider the more typical scenario where an external CSR database is compared to a target database, in which case the rules are more strict. Although it's unlikely that a target will use "required" values rather than "const" for fixed CSR values, we still show the comparisons for such cases. Also note that only one side may mark a CSR as "volatile". That side is then taking ownership of the CSR, stating that it may modify the CSR at any time. The next table shows existing targets entries compared to new entries coming from the external database.

Table 6.5. Target and external Init-CSR conflict rules

Target Entry	External const valA	External const valB
const valA	Same, ignored.	Value conflict, error.
const valB	Value conflict, error.	Same, ignored.
required valA	Mode conflict, error.	Mode conflict, error.
required valB	Mode conflict, error.	Mode conflict, error.
volatile-init valA	Mode conflict, error.	Mode conflict, error.
volatile-init valB	Mode conflict, error.	Mode conflict, error.
volatile-noinit	Mode conflict, error.	Mode conflict, error.
invalid valA	value conflict, error.	const ensure this, keep.
invalid valB	const ensure this, keep.	value conflict, error.

Existing Entry	New required valA	New required valB
const valA	const ensures this, ignored.	Value conflict, error.
const valB	Value conflict, error.	const ensures this, ignored.
required valA	Same, ignore.	Value conflict, error.
required valB	Value conflict, error.	Same, ignore.
volatile-init valA	Mode conflict, error.	Mode conflict, error.
volatile-init valB	Mode conflict, error.	Mode conflict, error.
volatile-noinit	Mode conflict, error.	Mode conflict, error.
invalid valA	value conflict, error.	const ensure this, keep.
invalid valB	const ensure this, keep.	value conflict, error.
Existing Entry	New volatile-init valA	New volatile-init valB
const valA	Mode conflict, error.	Mode conflict, error.
const valB	Mode conflict, error.	Mode conflict, error.
required valA	Mode conflict, error.	Mode conflict, error.
required valB	Mode conflict, error.	Mode conflict, error.
volatile-init valA	Mode conflict, error.	Mode conflict, error.
volatile-init valB	Mode conflict, error.	Mode conflict, error.
volatile-noinit	Mode conflict, error.	Mode conflict, error.
invalid valA	Mode conflict, error.	Mode conflict, error.
invalid valB	Mode conflict, error.	Mode conflict, error.
Existing Entry	New volatile-noinit	
const valX	Mode conflict, error.	
required valX	Mode conflict, error.	
volatile-init valX	Mode conflict, error.	
volatile-noinit	Mode conflict, error.	
invalid valX	Mode conflict, error.	
Existing Entry	New invalid valA	New invalid valB
const valA	Value conflict, error.	const ensure this, ignored.
const valB	const ensure this, ignored.	Value conflict, error.
required valA	Value conflict, error.	const ensure this, ignored.
required valB	const ensure this, ignored.	Value conflict, error.
volatile-init valA	Mode conflict, error.	Mode conflict, error.
volatile-init valB	Mode conflict, error.	Mode conflict, error.
volatile-noinit	Mode conflict, error.	Mode conflict, error.
invalid valA	Same, ignored.	No conflict, keep.
invalid valB	No conflict, keep.	Same, ignored.

6.8 Netronome Flow Firmware File Format

The Netronome Flow Firmware file format used in the NFP SDK and on host/ARM is standard ELF, but with processor specific types and content. Standard tools like readelf will work on the NFFW ELF file. The symtab (symbol table) contains the same symbols as in the map file, but also import expressions used to resolve import variable patch-ups in microcode (relocation).

6.8.1 NFP ELF file header (`ElfN_Ehdr`)

The ELF file contains no program headers and uses ELF relocation to resolve import variable expressions and patch up microcode.

6.8.1.1 `ElfN_Ehdr.e_ident`

All entries up to `EI_VERSION` have standard meanings. The ELF file is always little-endian, so `e_ident[EI_DATA]` is always `ELFDATA2LSB`.

6.8.1.2 `ElfN_Ehdr.e_type`

Value	Description
<code>ET_EXEC</code>	The file is ready to be loaded and contains no undefined import variables or unresolved import expressions.
<code>ET_REL</code>	The file still contains one or more undefined import variables or unresolved import expressions. The firmware loader will reject the file unless import variables are defined at load time.
<code>ET_NFP_PARTIAL_REL</code> (<code>ET_LOPROC + ET_REL</code>)	This is the same as <code>ET_REL</code> , except that partial files are intended to be linked into a complete NFFW file, rather than loaded directly. Partial firmware files are currently used for packaging NFP-6xxx picoengine firmware and passed to the linker.
<code>ET_NFP_PARTIAL_EXEC</code> (<code>ET_LOPROC + ET_EXEC</code>)	Same as <code>ET_EXEC</code> , but for partial firmware files.

6.8.1.3 `ElfN_Ehdr.e_machine`

Has the value `0x6000` for NFP-6xxx NFFW files.

6.8.1.4 `ElfN_Ehdr.e_flags`

Currently 0. For NFP-32xx this would hold configuration for the single memory unit. For NFP-6xxx, each MU island is configured via Init-CSRs.

6.8.2 NFP ELF section headers (**ElfN_Shdr**)

6.8.2.1 ElfN_Shdr.sh_type

Name	Value	Description
SHT_NFP_INITREG	SHT_LOPROC + 2	A register initialization section. Primarily used for Init-CSRs, but is also used for microengine registers. An NFFW file can have many of these sections.
SHT_UOF_DEBUG	SHT_LOUSER	The debug information. This is still the same content as was found in the UOF DBG_OBJS chunk, with updates for larger symbol addresses and sizes and additional information for firmware initialization such as initial register values.

6.8.2.2 ElfN_Shdr.sh_flags

The standard flags have standard meanings. It is worth noting that memory sections which do not have SHF_ALLOC set are ignored by the loader. Sections without SHF_ALLOC are used for "reserved" memory symbols as allocated with .alloc_mem and the "reserved" attribute. Below are NFP specific flags.

Name	Value	Description
SHF_NFP_INIT_SECTION	0x80000000	This discerns an application microcode section from an initialization microcode section. Sections with this flag are loaded and executed prior to application sections. These sections are not currently used for NFP-6xxx NFFW files.

6.8.2.3 ElfN_Shdr.sh_info

The meaning of this field depends on the section type. For SHT_REL sections, this is the section number of the section to apply the relocations to. For SHT_NOBITS and SHT_PROGBITS see the table below.

Bits (msb:lsb)	Description
15:0	Memory type.
31:16	Domain. A number depending on the memory type.

The table below shows the memory types, values and domain meanings. EMU_CACHE is used primarily to provide a memory region in the upper part of the EMEM cache which can be used as Direct Access memory without affecting the EMEM cache operation. The EMU_CACHE section base offset is still 0, so using the upper part of the cache memory requires a correct offset.

Memory type	Value	Domain meaning
USTORE (ME codestore)	1	Microengine MEID.
LMMEM (ME local memory)	2	Microengine MEID.
CLS (Cluster local scratch)	3	Island ID.
MU (EMEM, IMEM, CTM)	4	Island ID
SRAM (All VQDR)	5	N/A (Channel number is address bits <31:30>
PPC_LMEM (Picoengine local-memory)	7	Island ID
PPC_SMEM (Picoengine shared-memory)	8	Island ID
EMU_CACHE (EMEM Direct Access)	9	Island ID

For SHT_NFP_INITREG sections, the sh_info field encodes several values used to execute the initialization. These are listed in the table below. Note that the ORDER field is only under linker control and is used to ensure a clean and consistent load sequence. It does not provide any guarantee to user Init-CSRs entries. SHT_NFP_INITREG sections that have SHF_NFP_INIT_SECTION set are loaded before any .text sections that have SHF_NFP_INIT_SECTION set, if any. This generally means that any initialization microcode that the linker may add will execute in the same chip configuration as the user microcode, but the linker may still change the sequence sections are loaded in as required.

Bits (msb:lsb)	Description
31: 26	Island ID.
25: 22	CPP Target ID.
21: 17	Read CPP Action.
16: 15	Read CPP Token.
14: 10	Write CPP Action.
9: 8	Write CPP Token.
7: 0	Order.

6.8.3 NFP ELF symbols (ElfN_Sym)

6.8.3.1 ElfN_Sym.st_info

Normal symbols have a type of STT_OBJECT, import variables are STT_NOTYPE and import expressions are STT_NFP_EXPR. Import expressions are only used for relocation of import variables and can be ignored otherwise. Import expressions are always evaluated in the signed 64-bit domain. STT_SECTION sections are used in partial firmware files to mark memory sections that need to be linked into the NFFW file.

Symbol type	Value	Description
STT_NFP_EXPR	STT_LOPROC	Import expressions used in relocation. An expression can only be evaluated if all its import variables are defined.

6.8.3.2 ElfN_Sym.st_shndx

For normal symbols this points to the section containing the symbol. For import variables this is SHN_UNDEF for undefined import variables and SHN_ABS for defined import variables.

7. Command Line Tools

7.1 nfiv - Network Flow Import Variable Tool

This tool is used to manipulate import variables in NFFW files after linking and before loading. Import variables are assigned values on the command line or by specifying an IVD (see Section 7.1.4) or a KV (see Section 7.1.3) file. This tool does not provide any loading functionality.

A loadable NFFW file will not have any undefined import variables. Using this tool, each import variable can be defined by assigning a value to it. When modifying import variables, messages will be printed out if values are out of range or any undefined import variables remained. These messages can be controlled by command line options. In the NFFW file, import variables are considered truly global and as such it is not necessary to specify a value for each microengine as with UOF and IVD. For more information on using import variables for UOF in the simulator, refer to Section 2.12.3.4.

7.1.1 Usage

```
nfiv [options] command [command options] FILE
```

7.1.2 Command Line Options

The following table lists the primary `nfiv` command line options.

Table 7.1. `nfiv` Primary Command Line Options

Option	Definition
<code>-h</code> , <code>--help</code>	Print a description of the <code>nfiv</code> commands and parameters.
<code>--version</code>	Print version and build information.
<code>-q</code>	More quiet output. Add more to suppress more output. Errors and warnings are shown by default. One level down shows only errors. One level more shows only fatal errors. There are certain messages that will always be shown.
<code>-v</code>	More verbose output. Add more to get more output. Information messages are hidden by default. One level up shows information messages. One level more shows debugging.

The following table lists the `nfiv` commands.

Table 7.2. `nfiv` Commands

Command	Definition
<code>list</code>	List all import variables in the NFFW file, if any, and their current values.

Command	Definition
modify	Modify one or more import variables in the NFFW file. Refer to Table 7.3 for options. If no option is specified to indicate an output file or in-place modification, the modifications will not be written to any file. Note that, using the various options, it is possible to apply more than one IVD or KV file and also directly assign values on the command line. If an import variable is assigned a different value once previously defined, an information message will be printed about the change. Enable a more verbose output level (-v) to see these messages.

The following table lists the `nfiv modify` command options.

Table 7.3. `nfiv modify` Options

Option	Definition
<code>-o, --output FILE</code>	Write the modified firmware to FILE. This can be the same as the input file. If this option is not specified, the modified firmware will not be written to any file.
<code>--inplace</code>	This can be used in place of <code>--output</code> and will write the output to the input file.
<code><name>=<value></code>	Assign a value to an import variable. <code><value></code> can be any format.
<code>-U<name>, -u<name></code>	Undefine an import variable. If it was previously defined, its value will be retained in microcode, but the file will be deemed unloadable.
<code>--ivd FILE</code>	Assign import variable values from an IVD file. This option is available for compatibility. Use of Key Value files are recommended instead (<code>--kv</code>). Refer to Section 7.1.4 for the file format.
<code>--kv FILE</code>	Assign import variable values from a KV file. Refer to Section 7.1.3 for the file format.

7.1.3 Key Value Import Variable Data File

This is the recommended file format to use when defining many import variable values. It has a very simple format as shown below. Variable names and values are separated by any number of tabs or spaces.

```
// Comment line
# Comment line
Variable1      Value
Variable2      Value
```

7.1.4 UOF Import Variable Data File

`nfiv` support the UOF related IVD file format for compatibility. It interprets is slightly differently due to the way the import variables are implemented in the NFFW file. The first two columns are ignored and for each image (or microengine), the same value must be given to an import variable if specified for more than one image. The file format is shown below.

```
// Comment line
"Chip Name" "Image Name1" Variable1 Value1
ChipName     ImageName2      Variable2 Value2
" "          ImageName3      Variable2 Value2
```

7.2 cling - C interpreter

Cling is the C interpreter included in the SDK. It is available from the Windows Command Prompt as well as the Programmer Studio graphical user interface.

The C interpreter is a standards compliant interpreter based on the LLVM and Clang libraries. It provides a command line prompt for interactive use and supports meta commands to allow the execution of C script files. More information can be obtained at <http://root.cern.ch/drupal/content/cling>.

Cling is described in the following sections:

- Invoking the C interpreter (See Section 7.2.1)
- Command line options (See Section 7.2.2)
- Meta commands (See Section 7.2.3)
- Programming Interfaces (See Section 7.2.4)

7.2.1 Invoking the C interpreter

The C interpreter is started by entering the command `cling`. This will start the cling executable with the commandline options for specifying include paths as well as libraries to load. On a default install the command will be:

```
C:\>NFP_SDK_5.x.y\bin\cling
```

```
***** CLING *****
* Type C++ code and press enter to run it *
*           Type .q to exit
*****
[cling]$
```

Commands can then be entered interactively and are executed line by line.

```
[cling]$ #include <stdio.h>
[cling]$ printf("Hello world\r\n");
Hello world
[cling]$
```

When the simulator is running, the following shows how a ME GPR can be written and read back in an interactive session:

```
***** CLING *****
* Type C++ code and press enter to run it *
*           Type .q to exit
*****
[cling]$ #include <nfp.h>
[cling]$
[cling]$ struct nfp_device *nfp;
[cling]$ nfp = nfp_device_open(0);
[cling]$
[cling]$ #include <nfp_resid.h>
[cling]$ int meid = nfp_idstr2meid(NFP_CHIP_FAMILY_NFP6000, "i32.me5", NULL);
[cling]$ meid
(int) 521
[cling]$
[cling]$ int ret
(int) 0
[cling]$ ret = nfp_me_register_write(nfp, meid, nfp_mereg_gprA(0), 12345678);
[cling]$ uint32_t value;
[cling]$ ret = nfp_me_register_read(nfp, meid, nfp_mereg_gprA(0), &value);
[cling]$ value
(uint32_t) 12345678
[cling]$
[cling]$ nfp_device_close(nfp);
[cling]$
```

7.2.2 Command Line Options

The following command line options are supported:

Table 7.4. cling Command Line Options

Option	Definition
-h, --help	Print a description of the options.
--metastr	Set the meta command tag, default '!'.
--nologo	Do not show startup-banner
-L <directory>	Add directory to library search path. It can be specified multiple times
-l <library>	Load a library at startup. It can be specified multiple times
-v	Enable verbose output



Note

Cling is a 32-bit process and only 32-bit libraries are supported.

7.2.3 Meta Commands

The following meta commands are supported:

Table 7.5. `cling` Meta Commands

Option	Definition
<code>.help</code>	List the meta commands
<code>.q</code>	Exit <code>cling</code>
<code>.L <filename></code>	Load file or library
<code>.(x x) <filename>[args]</code>	Same as <code>.L</code> and runs a function with signature <code>ret_type filename(args)</code>
<code>.I [path]</code>	Shows the include path. If a path is given adds the path to the include paths
<code>.@</code>	Cancels and ignores the multiline input
<code>.rawInput [0 1]</code>	Toggle wrapping and printing the execution results of the input

7.2.4 Programming Interface

The programming interface is documented in the *NFP-6xxx Simulator API Reference Manual*.

7.3 Tools common to SDK and BSP

A number of NFP command line tools are available in both the BSP and the SDK. The BSP tools interact directly with hardware while the SDK tools use the SDK's libnfp* library set which connect to the simulator or a remote hardware debug server. These tools usually have a close mapping to an API that is available in both BSP and SDK libraries. As the APIs build on top of each other, these tools also build on each other to provide more convenience for common tasks with better error and safety checking.

For some tools there are extra options available on either BSP or SDK. For example, the SDK adds an option to let a user decide if a command should be back-doored (execute without clocking) or front-doored (inject real CPP command and clock). Such an option is irrelevant on hardware. This section will cover the common tools and specifically point out any differences between BSP and SDK. For each tool, the command line parameters are explained in the help message (`--help`).

7.3.1 nfp-cpp and nfp-xpb

The `nfp-cpp` tool is a very low level tool that lets you send almost any custom CPP command. The CPP API does have some sanity checks to prevent the worst of typos from causing problems, but use with caution. The `nfp-xpb`

tool uses the `nfp_xpb_*` API which is a more convenient API to access XPB targets rather than having to craft the CPP command each time. The `nfp-xpb` command line options are simple and there are no SDK specific options.

The address used with the `nfp-cpp` tool is translated if the `target` included an island ID. If no island ID was included the address is sent on the CPP bus unchanged. The translation depends on the IMB CPP Address Translation CSRs of the island where the command enters the NFP. For the BSP this island depends on the hardware setup, but typically a PCIe island or the ARM island. The IMB CPP Address Translation CSRs for these island (driver islands) are reserved and all have the same values. The simulator uses these same values for back or front-door commands. For example, `nfp-cpp 25@mu:0` is the same as `nfp-cpp mu:0x2800000000` and `nfp-cpp 35@cls:0` is the same as `nfp-cpp cls:0x8c00000000`.

Table 7.6. NFP SDK `nfp-cpp` Command Line Options

Option	Description
<code>-f, --front</code>	By default, all simulator access is back-doored - reads and writes directly affect the registers, memories or other elements it needs to just as if a normal CPP command was issued. This happens without having to clock the simulator. Sometimes it is useful to simulate a normal clocked CPP transaction instead or perhaps the particular command cannot be back-doored. This option lets a user specifically use clocked CPP transactions. The <code>nfp-cpp</code> call will block until the command finishes.

7.3.2 `nfp-power`

The `nfp-power` tool provides a way to control which parts of the chip is enabled and clocked. When used with the simulator, the tool writes XPB CSRs using back-door access, but the effect of the change can only propagate properly by clocking the simulator (50 cycles recommended).

7.3.3 `nfp-mem`

The `nfp-mem` tool reads and writes memory resources on an NFP. It uses memory access APIs (e.g. `nfp_emem_write`) rather than the CPP API directly. The primary difference between SDK and BSP for this tool is that the SDK `nfp-mem` can access simulator memories of microengines while they are active, whereas on hardware MEs must be idle for safe access to lmem and ustore memories.

7.3.4 `nfp-reg`

The `nfp-reg` tool reads and writes all registers and provides details about register offset, bit fields and more. It uses a lookup string of the format `target[:map[.reg[.field]]]` to identify registers using names matching those in the Databook and the string can also be used with Init-CSR directives in the toolchain. It can also list entry details without accessing any registers, making it easier to locate specific CSRs. Writing to a bit field of a register will result in a read-modify-write action while writing to a whole register will only result in a write action. If a

register contains bits that will be affected by accessing other bits, nfp-reg will print an error message and exit. For such registers, only whole register access is allowed.

Since not all CSRs are back-door accessible, the `--access` option provides a choice between front-door, back-door and first back-door then front-door as a fallback. Front-door will then result in simulator clocking to access the CSR. Back-door access is the default and front-door can be used on all registers. Verbose listing will indicate which registers can be back-door accessed.

Please see the `nfp-reg --help` for more details as well as Section 6.7.

7.3.5 nfp-mereg

The `nfp-mereg` tool provides access to ME registers using the same API as `nfp-reg mereg:iX.meY.reg`, but has a different command line interface and output format. This tool existed before `nfp-reg` and is sometimes a more convenient way of debugging ME register contents. To reduce duplication, this tool's interface and output format will be merged into `nfp-reg` in future.

7.3.6 nfp-nffw

The `nfp-nffw` tool loads, unloads, modifies and inspects firmware as well as provides status and reset commands to view loaded firmware details and reset any microengines if required. It also allows loading firmware without enabling microengine contexts, which can then be started later using the `nfp-nffw start` command. This is useful when firmware needs to be loaded for additional system configuration to take place without microengines running and interfering or reacting to any of the configuration. For simulator firmware loading this allows such configuration to clock the simulator if required.

Table 7.7. NFP SDK nfp-nffw Command Line Options

Option	Description
<code>load -r, --reset</code>	Only applies to the simulator. This will reset the simulator to a BSP level before loading firmware.
<code>load -R, --no-reset</code>	Do not reset the simulator before loading. Note that any microengines that will be loaded must be fully idle and the loader assumes that this is the case. This is the default.
<code>load -s, --start</code>	This behaves exactly the same as on hardware - the loaded microengines' CtxEnables are written to enable contexts. The difference for the SDK simulator is that this does not start any clocks. This is the default.
<code>load -S, --no-start</code>	Don't write CtxEnables after loading. Firmware can then be started later using the <code>start</code> command.
<code>load -c, --clock-start</code>	Specifically provided to allow a load command to start the microengines and also start the simulator clock. When the tool exits the simulator is running freely.
<code>load -C, --clock-start</code>	Specifically provided to allow a load command to start the microengines and also start the simulator clock. When the tool exits the simulator is running freely.

Option	Description
load -C, --no-clock-start	Do not run the simulator after loading. This is the default.
start -c, --clock-start	Write CtxEnables to start firmware and also run the simulator.
reset	This command behaves the same in the SDK as in the BSP, but the simulator must be clocked for this to take effect (similar to nfp-power, 50 cycles recommended).
unload	This command write 0 to CtxEnables.CtxEnables to disable microengine contexts, but it does not wait until the microengine goes fully idle. The simulator must be run until all microengines go idle. This can take a varying number of cycles depending on the firmware. If the microengine is executing a loop without swapping contexts it cannot be stopped and must be reset.
unload -c, --clock-stop	After disabling microengine contexts, stop the simulator if it is running. Note that the simulator would still need to be clocked until all microengines go idle.
unload -C, --no-clock-stop	Do not stop the simulator if it is running after unloading. This is the default.

7.3.7 nfp-rtsym

The nfp-rtsym tool is based on the nfp-mem, but instead of resource names it uses runtime symbol names. This requires the loaded firmware to be built with a runtime symbol included. See nfp-rtsym --help for more details.

7.3.8 nfp-hwinfo

The NFP SDK nfp-hwinfo tool provides a very limited subset of information compared to the BSP nfp-hwinfo.

7.3.9 nfp-tcache

The NFP SDK nfp-tcache tool displays statistics on external memory unit tcache (tag cache) usage, such as hits and evictions.

7.3.10 nfp-tminit, nfp-macinit

These tools provide the same functionality when used with the NFP SDK simulator as with the BSP, except that any PHY configuration is ignored (does not apply to the simulator).

8. Simulator

This chapter describes the Cycle Accurate Simulator and its use. The Programmer Studio graphical user interface to the Simulator is described in Chapter 2.

8.1 Overview

The NFP-6xxx simulator is a cycle-based (as opposed to event-driven) logic simulator. It demonstrates the functional behavior and performance characteristics of a system design based on the network processors without relying on the hardware.

The simulator is designed to run as a stand-alone server application. Simulator users, or clients, interact with the server over TCP sockets. This design means the simulator can be run locally, on the machine that the client applications are run, or on any machine within a connected TCP/IP network. Note that high network latency will have a negative affect on client RPC performance and will lead to a degraded experience. For this reason it is recommended that client connect to the simulator over a wired LAN.

8.2 Simulator Limitations

The simulator does not simulate logic for the following design blocks: ARM IP, PCIe IP, PLL, MAC IP and DDR3 Controller IP. However, the simulator has the following functional models, which emulate the behaviour of the following blocks:

- DDR3 Memory controller and DRAM memory
- MAC Ethernet and Interlaken
- DSF/CPP bus master (attached to DSF port 4 on the ARM island)
- PCIe Controller

As the ARM IP is not simulated or modelled, certain CPP and XPB transactions may not work. Notably, CPP transactions using the ARM targets are likely to be affected.

There are many MAC CSRs within MacInterlaken and MacEthernet maps which may not be accessible. However, some are modelled to achieve necessary functionality. There are also some Ethernet statistics missing, with only a small selection implemented.

8.3 Networking

The Interlaken and Hydra (Ethernet) blocks within the MAC are modelled by the SDK simulator. The modelled blocks interact with buses connected to the MAC Gasket to inject and retrieve frames. A set of networking APIs are used to retrieve egress frames and inject ingress frames.

Both the Interlaken and Hydra blocks perform rate limiting so that the port rate never exceeds the configured port rate. Rate limiting is based on the MAC clock period and the configured port rate and takes wire protocol overhead into consideration.

The Ethernet model supports both simple Ethernet Flow Control and Priority Flow Control. If a pause frame is injected using the networking APIs, the Ethernet model will adjust and maintain pause timers, passing the XOFF information to the Gasket. On egress the Ethernet model has the ability to issue pause frames, should the Gasket request them. This modelled behaviour is sufficient to simulate flow control between multiple NFP devices or external network infrastructure models.

8.4 PCIe Simulation

The PCIe support works by passing TCP socket data between a connected PCIe client and the PCIe controller busses within the simulator. The simulator can optionally listen on a user-defined port for client connections. To enable the PCIe socket feature, the user needs to add the `ix.pcie.sockport` key to the simulator config file. See Section 8.8 for further details.

The PCIe simulation feature exists with the intention of it being used with QEMU to provide a platform for developing PCIe host code without the need for hardware. Refer to the application note titled [PCIe Simulation in NFP-6xxx](#) for details on how to setup and run PCIe simulations.

8.5 Hardware Debug

The simulator supports a number of hardware debug features through the `nfp_hwdbg.h` NFHAL API. This API provides an interface for run control and managing software and code breakpoint that is compatible with the hardware debug implementations. A client could use the hardware debug API on both the simulator and NFP hardware via a hardware debug server.

Handling software breakpoints requires the simulator to perform CSR writes that affect the execution on ME code. This is a unique case where the simulator is actively manipulating the chip state. Note also that running an ME using the hardware debug API may set `CtxEnables.CtxEnables`. This behaviour will only be enabled when a hardware debug handle is opened by the client. The simulator implementation of hardware debug code breakpoints is passive and does not manipulate chip state.

8.6 State Access

On hardware, a host typically accesses chip state via the PCIe bus. The PCIe bus, in turn, provides access to the DSF/CPP bus (as well as PCIe resources). On hardware, the NFHAL APIs primarily map to DSF/CPP bus transactions.

The simulator provides two possibilities for issuing clocked CPP commands. Firstly through the PCIe socket interface and secondly through the DSF/CPP BFM that is attached to the ARM. It is possible to issue CPP transaction

via the BFM using the `nfp_cpp.h` API. However, one needs to mark the CPP handle as clocked/frontdoored using the `nfp_sal.h` function `nfp_sal_cpp_mode()` ..

While there are two clocked methods for accessing simulator state, it is preferable to access state without issuing any clocks. This makes performing configuration and retrieving status considerably faster as clocked CPP access can take several hundred simulation cycles. By default all NFHAL and NFSAL APIs will attempt to perform unclocked state access. There are several CSRs that cannot be implemented without issuing clocks which will require clocked access. The NFHAL function for accessing these CSRs will return the standard error code `ENOENT`.

8.7 Invoking the Simulator

The simulator is invoked using the `nfsim` shell script within the `bin` directory on Linux, or using the `nfsim.bat` script on Windows. It is possible to configure the simulator through a number of command line options and environment variables.

8.7.1 Command Line Options

<code>--daemonize, -d</code>	Launch the simulator as a background process, or daemon. This option is only supported on Linux. Note that syslog will be used for logging, unless logging to file is enabled.
<code>--cfgfile filename, -f filename</code>	Specify a simulator configuration file to use.
<code>--port port, -p port</code>	Specify the TCP port to use for client RPC.
<code>--saddr ip, -s ip</code>	Specify the source IP to be used for RPC sockets.
<code>--paddir</code>	Specify the directory to store the PID file. The pid file always takes the name <code>nfsim-[port].pid</code> and contains the Process ID (PID) of the simulation server. If this argument isn't provided the PID file will be stored in <code>/tmp</code> on Linux or, on Windows, in which ever of the following is present first: <code>\$TMPDIR</code> , <code>\$TEMPDIR</code> , <code>c:\tmp</code> , <code>c:\temp</code> .
<code>--threads count, -t count</code>	Specify the number of hardware worker threads to create. By default, the number will be set to the number of CPU cores present. Note that a value of 0 will create no hardware threads, which will lead to a single threaded logic simulation.
<code>--nobspreset</code>	By default the simulator will perform a BSP reset at startup, which involves performing a number of configuration steps not performed on a true chip reset. If this flag is provided, the additional configuration will not take place.
<code>--startrunning</code>	Run the simulator on startup; if not specified the simulator will be stopped.
<code>--logfile LOGFILE</code>	Write log messages to a file instead of to <code>stdout/stderr</code> or <code>syslog</code> .
<code>--forcestdout</code>	Log to <code>stdout/stderr</code> even if logging to file is enabled.
<code>--debug</code>	Start the simulator with the log level set to <code>DEBUG</code> .
<code>--extra</code>	Start the simulator with the log level set to <code>EXTRA</code> .

--heavy	Start the simulator with the log level set to HEAVY.
--version, -v	Display the simulator version information.

8.7.2 Environment Variables

NFP_RPC_VERSION_CHECK This environment variable controls how the simulator handles RPC API version mismatches. If set to -1, the server will report a single warning when a version mismatch occurs, the client will receive no indication of a mismatch. If set to 0, a single warning will occur as before, but the client will be notified of the mismatch. If set to 1, a single error will be logged and the version-mismatched RPC call will fail. The default value is -1.

8.8 Simulator Configuration File

There are a number of simulator options that are controlled via entries in the simulator configuration file. This file uses the JSON format.

By default, the simulator uses a configuration file called `nfp6000sim.cfg` and its default location is in the `etc` directory within the installation hierarchy. It is possible to override the filename with the `cfgfile` command line parameter.

There is a class of configuration entries that are applied per island. They have the following format:
`ix.subsystem.key`, where `x` refers to the island number.

The following table describes the available simulator configuration options:

port	A number presenting the TCP port that the simulation server will use for communicating with clients. A default of 20606 will apply if the configuration entry is not present. Note that this will be overridden if a port is provided from the command line.
SKU	A string containing the SKU of the chip to simulate. A default of nfp-6xxx will apply if the configuration entry is not present.
configurator	A string containing the filename of the configurator binary to use. The configurator binary should match the SKU that is provided. A default configurator image will be used if no entry is present.
fT_CLK	A floating point number representing the frequency to use for T_CLK within the simulator in megahertz. T_CLK is used to clock the MicroEngines. A default of 1200.0 will apply if the configuration entry is not present.
fP_CLK	A floating point number representing the frequency in megahertz to use for P_CLK within the simulator. P_CLK is used to clock various chip infrastructure. A default of 1000.0 will apply if the configuration entry is not present.

ILA0_CLK 	A floating point number representing the frequency in megahertz to use for ILA_CLKs within the simulator. ILA0_CLK is used to clock MAC island 0 and ILA1_CLK MAC island 1. A default of 800.0 will apply if the configuration entry is not present.
FD0_CLK FD1_CLK FD2_CLK	A floating point number representing the frequency in megahertz to use for dram clocks within the simulator. D0_CLK is used to clock External Memory Island 0, D1_CLK External Memory Island 1 and D2_CLK External Memory Island 2. A default of 533.33 will apply if the configuration entry is not present.
ILK0SERDESCLK 	A floating point number representing the frequency to use for the SERDES IO within the simulator in gigahertz. This will affect the rate which Interlaken network interfaces will operate at. A default of 10.3125 will apply if the configuration entry is not present
ILK1SERDESCLK	
ix_PCIE.sockport	Specify the integer port(s) to listen on for PCIe simulation. Note that this needs to be specified per PCIe island. The PCIe worker thread will not be created if no PCIe islands are assigned a port.
ix_emem.size 	Specify the memory size as either "4G" or "8G" and the memory channels as either "1" or "2". Note that only "8G" is supported in dual channel mode.
ix_emem.channels	
ix_emem.leveling	Set to "on" to enable leveling and "off" to disable leveling.

8.9 Simulator API

The NFP-6xxx simulator supports two distinct sets of APIs for interacting with the simulated device. These are NFHAL and NFSAL:

The NFHAL API provides a hardware abstraction layer consisting of calls that are the same as those provided by the standard Netronome BSP libraries. This makes it possible to write code that can execute on both simulated and real hardware with little or no modification.

The NFSAL API provides access to features only available to the simulator. These include run control, break-on-change, network access and history.

Both of these APIs are described fully in the NFHAL/SAL Reference Manual.

9. Technical Support

To obtain additional information, or to provide feedback, please email <support@netronome.com> or contact the nearest **Netronome** technical support representative.

Appendix A. Programmer Studio Shortcuts

In Programmer Studio there are at least three ways to initiate an action:

- A menu command
- A keyboard shortcut
- A toolbar button

The following tables list most of the actions supported by Programmer Studio with the shortcuts for each action.

Table A.1. Programmer Studio 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
	CTRL+SHIFT+B		Moves back to previous window.	
	CTRL+SHIFT+F		Moves forward one window.	

Table A.2. Programmer Studio 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 2.3.5
	ALT+P+A	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+D	Project, Update Dependencies	Update project dependencies.	Section 2.6.1
	ALT+S+C	Simulation, System Configuration	Specify system configuration.	Section 2.10

Table A.3. Programmer Studio Shortcuts — Edit

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
	ALT+F		Find next.	Section 2.5.10
	ALT+F	Find Previous	Same as Find Next only you must click Up in the Direction area first.	
	ALT+E+I	Edit, Find in Files	Find in text files.	Section 2.5.12

Button	Keyboard	Menu	Action	Reference
	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	To search for text in the active file, type the text in the Search box and press ENTER. Programmer Studio 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 A.4. Programmer Studio 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 A.5. Programmer Studio Shortcuts — Builds

Button	Keyboard	Menu	Action	Reference
	CTRL+F7	Build, Assemble	Assemble	Section 2.6.3.1
	CTRL+SHIFT+F7	Build, Compile	Compile	Section 2.7.3
	F7	Build, Build	Link	Section 2.9.2
	ALT+F7	Build, Rebuild	Rebuild	Section 2.9.2
	F4		Go to next error/tag	Section 2.5.11, Section 2.5.12, Section 2.6.3.2, Section 2.7.4.
	SHIFT+F4		Go to previous error/tag	Section 2.5.11, Section 2.6.3.2, Section 2.7.4.

Table A.6. Programmer Studio 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
			Start and stop profiling	Section 2.12.2
			Reset Compiler profiling statistics.	Section 2.12.2
	CTRL+1		Execute custom command script 1.	???
	CTRL+2		Execute custom command script 2.	???
	CTRL+3		Execute custom command script 3.	???
	CTRL+4		Execute custom command script 4.	???
	CTRL+5		Execute custom command script 5.	???
	CTRL+6		Execute custom command script 6.	???
	CTRL+7		Execute custom command script 7.	???
	CTRL+8		Execute custom command script 8.	???
	CTRL+9		Execute custom command script 9.	???
	CTRL+0		Execute custom command script 10.	???

Table A.7. Programmer Studio Shortcuts — Run Control

Button	Keyboard	Menu	Action	Reference
	F5	Debug, Run Control, Go	Start simulation.	Section 2.12.7.10
	SHIFT+F5	Debug, Run Control, Stop	Stop simulation.	Section 2.12.7.10
	F10	Debug, Run Control, Step Over	Step over.	Section 2.12.7.3
	F11	Debug, Run Control, Step Into	Step into (Compiler thread only).	Section 2.12.7.4
	SHIFT+F11	Debug, Run Control, Step Out	Step out (Compiler thread only).	Section 2.12.7.5
	CTRL+F10	Debug, Run Control, Run To Cursor	Run to cursor.	Section 2.12.6.4

Button	Keyboard	Menu	Action	Reference
	SHIFT+F10	Debug, Run Control, Step Microengines	Step Microengines	Section 2.12.7.2
	CTRL+SHIFT+F12	Debug, Run Control, Reset	Reset.	Section 2.12.7.11
	None	Debug, Run Control	Toggle View	Section 2.12.6.5

Table A.8. Programmer Studio 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+P	View, Project Workspace	Toggle visibility of the project workspace	Section 2.4
	ALT+V+D+C	View, Debug Windows, Command Line	Toggle visibility of Command Line window.	Section 7.2
	ALT+V+D+D	View, Debug Windows, Data Watch	Toggle visibility of Data Watch window	Section 2.12.11
	ALT+V+D+M	View, Debug Windows, Memory Watch	Toggle visibility of Memory Watch window.	Section 2.12.12
	ALT+V+D+H	View, Debug Windows, History	Toggle visibility of History window.	Section 2.12.20
	ALT+V+D+T	View, Debug Windows, Thread Status	Toggle visibility of Thread Status window.	Section 2.12.23
	ALT+V+D+Q	View, Debug Windows, Queue Status	Toggle visibility of Queue Status window.	Section 2.12.21
	ALT+V+D+R	View, Debug Windows, Run Control	Toggle visibility of Run Control window.	Section 2.12.7
	ALT+V+D+P	View, Debug Windows, Packet Streaming Statistics	Toggle visibility of Packet Streaming Statistics window.	Section 2.11.3

Appendix B. P4 C Sandbox

Within the NFP SDK, it is possible to call into C code from a P4 program. Over and above the access to the NPE and flowenv libraries, the C code has access to P4 headers and P4 metadata. This allows the expressiveness of C to be combined with the P4 pipeline.

The NFP SDK IDE allows you to add C Sandbox code to a P4 project. This code is callable from a P4 defined action as the result of a table lookup. Within your P4 source you must declare the external C function to be called as a P4 primitive action. An example of the declaration with a function name *my_function* follows below:

```
primitive_action my_function();
```

The declared primitive action may then be called within a P4 defined compound action in the same way as any other primitive action, e.g. `modify_field`. Note that at this time no arguments may be passed to the function.

The implementation of the custom primitive action resides in C Sandbox code. The C sandbox code would declare a function matching the above custom *my_function* primitive action as follows:

```
int pif_plugin_my_function(EXTRACTED_HEADERS_T *headers, ACTION_DATA_T *action_data)
```

The return code of the P4 plugin function should be `PIF_PLUGIN_RETURN_DROP` to indicate the packet should be dropped with further processing; or `PIF_PLUGIN_RETURN_FORWARD` to allow continued processing. The *headers* argument is an opaque handle to the P4 headers and P4 metadata. This information may be accessed by through various functions generated as part of the P4 build process. These functions are described in the section *Access to P4 header data*. The argument *action_data* is an opaque handle which points to P4 action parameters. Parameters are currently unsupported, and this argument is unused.

The NPE library may then be used within the P4 C Sandbox code as within any MicroC code.

B.1 Access to P4 data

As part of the build process a number of C header files are generated which allow the C Sandbox code to access the P4 headers and P4 metadata. To use the generated accessor code include *pif_plugin.h* in your sandbox code.

B.1.1 Headers

For every P4 header instance a set of functions and macros are generated in a file *pif_plugin_HDR.h* where *HDR* is the name of the instantiated header. In this file a number of standard function are generated which allow access to the extracted headers. These are as follows:

```
int pif_plugin_hdr_HDR_present(EXTRACTED_HEADERS_T *headers);
```

Returns 1 if *HDR* is present, 0 if not.

```
PIF_PLUGIN_HDR_T *pif_plugin_hdr_get_HDR(EXTRACTED_HEADERS_T *headers);
```

Returns a pointer to the extracted header. The pointer references a regular C structure. The structure definition is located in the generated C header file for the P4 header instance.

```
int pif_plugin_hdr_HDR_add(EXTRACTED_HEADERS_T *headers);
```

Marks the header as added. Returns 0 on success, -1 on failure. May fail if header can not be added.

```
int pif_plugin_hdr_HDR_remove(EXTRACTED_HEADERS_T *headers);
```

Marks the header as removed. Returns 0 on success, -1 on failure. May fail if header can not be removed.

B.1.2 Metadata

A C header file *pif_plugin_metadata.h* is generated which contains functions for reading and writing P4 metadata values. Two functions are generated per metadata field entry; these take the form:

```
uint32_t pif_plugin_meta_get__MDHDR__MDFLD(EXTRACTED_HEADERS_T *headers);  
void pif_plugin_meta_set__MDHDR__MDFLD(EXTRACTED_HEADERS_T *headers, uint32_t val);
```

For both of these function *MDHDR* indicates the metadata header and *MDFLD* indicates the metadata field.

Appendix C. Managed C Applications

Managed C application development is an experimental feature in Programmer Studio. When a new project is created the **Managed C** checkbox needs to be checked. User code can then be added by inserting a new C file into the project and selecting it from the Build Settings, which will then be built with the dataplane provided by the SDK.

The `mc_main` function in the users application is invoked for each ingress packet handing processing over to the users C code. If per ME setup or one time global setup is required the **Call managed c init functions** checkbox should be set from **Build Settings** and `mc_init` and `mc_init_master` functions should be provided in your managed c file. Alternatively, when building with the `nfp4build` tool, a **-D MC_INIT** flag should be used to enable this.

The number of instances of users code executing concurrently is controlled from **Build Settings** and selecting the **Number of worker MEs**.

Interaction with the dataplane is via the packet metadata structure show below:

```
struct pkt_meta {
    union {
        struct {
            /* first word mirrors nbi_meta_pkt_info[0] */
            unsigned int isl:6;
            unsigned int pkt_num:10;
            unsigned int bls:2;
            unsigned int pkt_len:14;

            /* second word mirrors nbi_meta_pkt_info[1] */
            unsigned int split:1;
            unsigned int resv0:2;
            unsigned int muptr:29;

            /* third word is ... */
            unsigned int ctm_size:2;
            unsigned int ctm_isl:7;
            unsigned int seqr:5;
            unsigned int unseq:1;
            unsigned int seq:16;

            /* fourth word is ingress info */
            struct {
                unsigned int type:2;
                unsigned int nbi:2;
                unsigned int port:8;

                unsigned int resv0:20;
            } ig_port;

            /* fifth word is egress info */
            struct {
                unsigned int type:2;
                unsigned int nbi:2;
                unsigned int port:8;
            } eg_port;
        };
    };
};
```

```
        unsigned int resv0:20;
    } eg_port;

    /* 6th + 7th word is the ctm buffer address */
    __addr40 void *pkt_buf;

    /* 8th word */
    unsigned int trunc_len:14;
    unsigned int resv1:18;

};

uint32_t __raw[8];
};

};

}
```

Packet egress is specified in the `eg_port` field of the metadata with the `type` indicating whether the packet is to be sent from a physical port or via PCIe to the host.

```
/* from/to physical port */
#define PKT_META_SRC_NBI 0
#define PKT_META_DST_NBI PKT_META_SRC_NBI
/* from/to host/pcie */
#define PKT_META_SRC_PCIE 1
#define PKT_META_DST_PCIE PKT_META_SRC_PCIE
```

The user controls whether a packet is dropped or forwarded using the value returned from `mc_main()`.

```
#define MC_DROP      -1
#define MC_FORWARD  0
```

An example for modifying and forwarding a packet is shown below.

```
#include <stdint.h>
#include <nfp.h>

/* Netronome FlowEnv MicroEngine library */
#include <nfp/me.h>
/* Netronome FlowEnv bulk memory library */
#include <nfp/mem_bulk.h>

#include "pkt_meta.h"

*****
* Defines
***** */

/* ethernet (14B) */
struct hdr_ethernet {
    /* dstAddr [32:16] */
    unsigned int dstAddr_0:32;
    /* dstAddr [16:0] */
    unsigned int dstAddr_1:16;
    /* srcAddr [16:32] */
    unsigned int srcAddr_0:16;
    /* srcAddr [32:16] */
    unsigned int srcAddr_1:32;
};
```

```
unsigned int srcAddr_0:16;
/* srcAddr [32:0] */
unsigned int srcAddr_1:32;
unsigned int etherType:16;
};

/********************* General ********************/
/* modify the ethernet destination address
 * illustrates how to access packet data
 */
static void modify_daddr(__mem uint8_t *pktdata)
{
    __xread struct hdr_ethernet rd_eth;
    struct hdr_ethernet buf_eth;
    __xwrite struct hdr_ethernet wr_eth;

    /* copy the packet data into a read xfer register */
    mem_read8(&rd_eth, pktdata, sizeof(struct hdr_ethernet));

    /* copy into a buffer register so we can do subword assignments that
     * aren't available in write xfers
     */
    buf_eth = rd_eth;

    /* set dstAddr to fixed value */
    buf_eth.dstAddr_0 = 0x00010203;
    buf_eth.dstAddr_1 = 0x0405;

    wr_eth = buf_eth;

    mem_write8(&wr_eth, pktdata, sizeof(struct hdr_ethernet));
}

/********************* Managed C entry point ********************/
int mc_main(void)
{
    PKT_META_TYPE struct pkt_meta *pmeta = PKT_META_PTR;

    /* reflect back */
    pmeta->eg_port.type = pmeta->ig_port.type;
    pmeta->eg_port.port = pmeta->ig_port.port;

    /* do simple packet modification */
    modify_daddr((__mem uint8_t *) pmeta->pkt_buf);

    return MC_FORWARD;
}
```

Appendix D. Run Time Environment (RTE)

This appendix describes the use of Run Time Environment (RTE) on Linux OS. RTE must be started and running in order to load P4 applications into Netronome NFP card. If you are using simulator and running on Windows OS using PS, there is no need to start the RTE. PS will start and manage RTE automatically.

To install the required packages specify `install` option when executing `sdk6_rte_install.sh`; Hardware Debug Server, SDK6 RTE will be installed, the currently installed version of BSP will be checked and the user prompted for an upgrade if older version is detected. It is highly recommended to upgrade BSP when prompted to do so.

To force install (i.e. to downgrade) to the packaged BSP version use `install_force_bsp` option is specified when executing `sdk6_rte_install.sh`.



Note

The firmware version is checked and upgraded if necessary on all supported NFP devices that is installed on the system during the BSP upgrade process, if new device is installed it is recommended to reinstall using `install_force_bsp`

The RTE can be started in NORMAL MODE and stopped by calling `start nfp-sdk6-rte` and `stop nfp-sdk6-rte` respectively in OS using Upstart and `systemctl start nfp-sdk6-rte` and `systemctl stop nfp-sdk6-rte` respectively in OS using systemd or by specifying the load/unload script as `pif_ctl_nfd.sh` and the use of zlib by using `-z -s /opt/nfp-pif/scripts/pif_ctl_nfd.sh` when manually starting the RTE.

The RTE can be started in DEBUG MODE and stopped by calling `start nfp-sdk6-rte-debug` and `stop nfp-sdk6-rte-debug` respectively in OS using Upstart and `systemctl start nfp-sdk6-rte` and `systemctl stop nfp-sdk6-rte` respectively in OS using systemd or by specifying the load/unload script as `pif_ctl_nfd.sh`, the use of zlib and loading in SDK debug mode by using `-z --sdk_debug -s /opt/nfp-pif/scripts/pif_ctl_nfd.sh` when manually starting the RTE.

The RTE can be started in SIM MODE to enable it to be used with the SDK6 simulator in Linux. It is started and stopped by calling `start nfp-sdk6-rte-sim` and `stop nfp-sdk6-rte-sim` respectively in OS using Upstart and `systemctl start nfp-sdk6-rte-sim` and `systemctl stop nfp-sdk6-rte` respectively in OS using systemd or by specifying the load/unload script as `pif_ctl_nfd_sim.sh` and the use of zlib by using `-z -s /opt/nfp-pif/scripts/pif_ctl_sim.sh` when manually starting the RTE. The SDK6 simulator is not included in the install tarball and needs to be downloaded and installed separately.

Start or stop the Hardware Debug Server by calling `start nfp-hwdbg-srv` and `stop nfp-hwdbg-srv` respectively. When full performance is required and if the user is not debugging applications through breakpoints and code stepping via the NFP SDK6 debugging interface start RTE in NORMAL MODE. If the user is debugging application through code stepping and breakpoints run the RTE in DEBUG MODE, this causes the firmware to run on a single ME to ease debugging but will seriously limit performance.

If firmware is loaded via the NFP SDK6 interface the Hardware Debug Server also needs to be started with the RTE, if firmware is loaded via an alternative interface or via the RTE CLI the Hardware Debug Server needs to be stopped.

The following input argument options are available for the RTE:

Table D.1. SDk6 RTE Input Arguments and Descriptions

Argument	Description
-h,--help	Print Help dialog, basically this table
-z,--zlib	If set zlib compression will be used in the RPC transport layer, this flag must be set when Programmer Studio or the SDK6 CLI clients are used
-I, --no_stdin	When set the stdin terminal interface will be disabled
-p, --port <port>	Specify port that will be used for RPC (default: 20206)
-s, --load_script <ctlscript>	Specify the path to the load/unload control script to be used, this is a required input argument
-n, --nfp <nfp>	Specify the target NFP device via its device number. Use <code>nfp-hwinfo -n <nfp></code> to get more information on device with specified number (default: 0)
-c, --config_file <configjson>	Specify path the JSON configuration file (aka rules file) that must be loaded on start up, design and firmware files must be specified as well
-d, --design_file <designjson>	Specify path the JSON P4 design file that must be loaded on start up, must be specified with compatible firmware file
-f, --fw_file <nfpfw>	Specify path the NFP firmware file that must be loaded on start up, must be specified with compatible design file
-v, --log_level <lvl>	Set the SDK 6 RTE log level, currently supported levels in descending verbosity order are HEAVY, EXTRA, DEBUG, INFO
-l, --log_file <logfile>	Specify the path to which the output must be logged
-i, --pid_file <pidfile>	Specify where the PID file should be saved
--daemonize	Detach this instance of SDK6 RTE server and run as a daemon
--load-last-set	Load the last loaded firmware, design and if it was loaded configuration file. These files are stored in /var/tmp/ so if left unused for couple of days the OS might remove files and this command will no longer work until new set is loaded
--skip_fw_load	Skip the firmware loading step, this enables the loading of a design file and configuration file at startup without a compatible firmware file. This is meant to be used as debug tool and speed up development by skipping the lengthy firmware loading process; a compatible firmware file should already be loaded on the NFP before using this command otherwise incorrect system behaviour and errors will be encountered
--sdk-debug	Run the SDK6 RTE in DEBUG MODE, used if the user wants to debug application through code stepping and breakpoints, this causes the firmware to run on a single ME to ease debugging but will seriously limit performance
--version	Display the version information of the currently installed SDK6 RTE

Please refer to the file README included in the RTE tarball for more information on installing the RTE.

D.1 Virtual and Physical Port Setup

- SR-IOV VFs are created with names like vf0, vf1 etc. Port names to use in the *User Config File* are v0.0, v0.1 etc. Physical port names are p0 for the first port and p4 for the second port (on dual port 40G Agilio Intelligent Server Adapters). When using the Agilio Intelligent Server Adapters with 10G, the first port name is p0 and are sequentially numbered up to p7 depending on the ISA.



Note

Port and VF naming may change in future to align with Agilio OVS.

Refer to the *Agilio OVS User's Guide* for configuring and using the SR-IOV VFs with VMs or as netdevs in Linux.

- VFs receiving packets should be drained as they will buffer approximately 256 packets before packet processing will be suspended. Processing will resume once space becomes available again.
- SDK RTE supports up to 32 VFs. The default number of VFs are 4 and can be changed in the /etc/init/nfp-sdk6-rte* startup files.



Note

A host with an Agilio-LX or CX Intelligent Server Adapter must support ARI (Alternative Routing ID) and Intel's Virtualization Technology (VT-d, IOMMU) for SR-IOV VFs. If the host system does not support ARI, then set env DISABLE_NFD=yes in the /etc/init/nfp-sdk6-rte* startup files, and use the physical ports on the ISA connected to a suitable 10G/40G NIC for packet ingress/egress.

D.2 Loading a NFP Device at System Startup

Copy the desired firmware, P4 design and rules configuration files to the host system.

Depending the OS being used edit either the Upstart configuration file (found in /etc/init/) or the systemd service file (found in /usr/lib/systemd/system/) to specify the paths to the files that should be loaded and add them to the pif_rte execution command input arguments. A commented example to do this has been included in all the init files and just be uncommented and the applicable details filled in. The number of VF's that should be setup for the devices is set with the NUM_VFS variable.

To start the programs at system ready uncomment the startup line in nfp-sdk6-rte.conf, nfp-sdk6-rte-debug.conf or nfp-hwdbg-srv.conf in /etc/init/ for OS using Upstart and run the systemctl enable command for the specified service (ex. `systemctl enable nfp-sdk6-rte.service`) for OS using systemd

An example of the nfp-sdk6-rte.conf on Ubuntu 14.04 setup to load and start the NFP device at system start with 6 VF's:

```
#Uncomment line to start on system ready
start on runlevel [02]

env CTL_SCRIPT=/opt/nfp_pif/scripts/pif_ctl_nfd.sh
#To load RTE at startup set FW_FILE, DESIGN_FILE and CFG_FILE and set in exec
Environment=FW_FILE=/path/to/firmware/file/fwfile.nffw
Environment=DESIGN_FILE=/path/to/design/file/pif_design.json
Environment=CFG_FILE=/path/to/config/file/rulesfile.json
env LOAD_NETDEV=1
env NUM_VFS=6
#For custom DMA and MAC setting set NBI_DMA8_JSON and NBI_MAC8_JSON and set DETECT_MAC=no
#env NBI_DMA8_JSON=/opt/nfp_pif/etc/configs/platform_dma8_config.json
#env NBI_MAC8_JSON=/opt/nfp_pif/etc/configs/platform_mac8_config.json
#env NBI_TM_JSON=nfp_nbi_tm_12x10GE.json
env NFPSHUTILS=/opt/nfp_pif/scripts/shared/nfp-shutils
env DISABLE_NFD=no
env DETECT_MAC=yes
env LD_LIBRARY_PATH=/opt/netronome/lib:/opt/nfp_pif/lib:$LD_LIBRARY_PATH

pre-start script
#pre start work comes here
end script

script
echo "[${date}] Starting Netronome Systems SDK6 Run Time Environment Server"
#exec /opt/nfp_pif/bin/pif_rte -n 0 -p 20206 -I -z -s $CTL_SCRIPT
#                                         --log_file /var/log/nfp-sdk6-rte.log
exec /opt/nfp_pif/bin/pif_rte -n 0 -p 20206 -I -z -s $CTL_SCRIPT -f $FW_FILE
#                                         -d $DESIGN_FILE -c $CFG_FILE --log_file /var/log/nfp-sdk6-rte.log
end script
```

An example of the nfp-sdk6-rte.service on CentOS 7 setup to load and start the NFP device at system start with 2 VF's, remember to run `systemctl enable nfp-sdk6-rte.service`:

```
[Unit]
Description=Netronome SDK6 Run Time Environment NORMAL MODE
After=network.target

[Service]
Environment=CTL_SCRIPT=/opt/nfp_pif/scripts/pif_ctl_nfd.sh
#To load RTE at startup set FW_FILE, DESIGN_FILE and CFG_FILE and set in ExecStart
Environment=FW_FILE=/path/to/firmware/file/fwfile.nffw
Environment=DESIGN_FILE=/path/to/design/file/pif_design.json
Environment=CFG_FILE=/path/to/config/file/rulesfile.json
Environment=LOAD_NETDEV=1
Environment=NUM_VFS=2
#For custom DMA and MAC setting set NBI_DMA8_JSON and NBI_MAC8_JSON and set DETECT_MAC=no
#Environment=NBI_DMA8_JSON=/opt/nfp_pif/etc/configs/platform_dma8_config.json
#Environment=NBI_MAC8_JSON=/opt/nfp_pif/etc/configs/platform_mac8_config.json
#Environment=NBI_TM_JSON=nfp_nbi_tm_12x10GE.json
Environment=NFPSHUTILS=/opt/nfp_pif/scripts/shared/nfp-shutils
Environment=DISABLE_NFD=no
Environment=DETECT_MAC=yes
Environment=LD_LIBRARY_PATH=/opt/netronome/lib:/opt/nfp_pif/lib:$LD_LIBRARY_PATH
#ExecStart=/opt/nfp_pif/bin/pif_rte -n 0 -p 20206 -I -z -s $CTL_SCRIPT
#                                         --log_file /var/log/nfp-sdk6-rte.log
```

```
ExecStart=/opt/nfp_pif/bin/pif_rte -n 0 -p 20206 -I -z -s $CTL_SCRIPT -f $FW_FILE
          -d $DESIGN_FILE -c $CFG_FILE --log_file /var/log/nfp-sdk6-rte.log

[Install]
WantedBy=multi-user.target
```

D.3 Setting Up Multiple NFP Devices on a Single Host

Follow standard hardware installation instructions, please be aware that available host resources might limit the number of devices that can be run at the same time and the total number of VF's that can be enabled.

For each device an instance of the RTE in NORMAL_MODE and if debugging an instance for each device of the RTE in DEBUG_MODE, the Hardware Debug Server and Programmer Studio. For Hardware Debug Server and RTE the target device number should be specified and each instance should have an unused port number specified, these settings should be set in each instance of Programmer Studio (see Hardware Debugging)

For example debugging 2 devices on a single host

```
/opt/nfp_pif/bin/pif_rte -n 0 -p 20206 -I -z --sdk-debug \
-s /opt/nfp_pif/scripts/pif_ctl_nfd.sh

/opt/netronome/nfp-sdk-hwdbgsrv/server/nfp-sdk-hwdbgsrv -n 0 -p 20406

/opt/nfp_pif/bin/pif_rte -n 1 -p 20207 -I -z --sdk-debug \
-s /opt/nfp_pif/scripts/pif_ctl_nfd.sh

/opt/netronome/nfp-sdk-hwdbgsrv/server/nfp-sdk-hwdbgsrv -n 1 -p 20407
```

Appendix E. P4 CLI Tools Example

This chapter describes an end-to-end example for using Netronome's P4 command line tools for building and run-time interaction on Linux. It assumes the SDK toolchain package is installed and an environment variable `SDK_DIR` is defined to be the root of the SDK installation path.

Executing this example is also possible on Windows and the `nfp4term.bat` script is provided for setting up the environment. Its available in the `SDK_DIR\p4\bin` directory.

The Graphviz (<http://www.graphviz.org>) tools are used to generate graphs. On Windows these tools are bundled with the SDK installation but on Linux they need to be installed manually with your package manager of choice.

E.1 P4 Program

The P4 program used for this example is a pseudo-L3 router that forwards packets to ports based on IPv4 address. The P4 code is as follows:

```
/* Header definitions */
header_type eth_hdr {
    fields {
        dst : 48;
        src : 48;
        etype : 16;
    }
}

#define IPV4_ETYPE 0x0800

header_type ipv4_hdr {
    fields {
        ver : 4;
        ihl : 4;
        tos : 8;
        len : 16;
        id : 16;
        frag : 16;
        ttl : 8;
        proto : 8;
        csum : 16;
        src : 32;
        dst : 32;
    }
}

/* Header instances */

header eth_hdr eth;
header ipv4_hdr ipv4;

/* Parser */
parser start {
```

```
    return eth_parse;
}

parser eth_parse {
    extract(eth);
    return select(eth.ethertype) {
        IPV4_ETYPE: ipv4_parse;
        /* NOTE: no default case so non-ipv4 will be dropped in parser */
    }
}

parser ipv4_parse {
    extract(ipv4);
    return ingress;
}

/* Ingress */
action fwd_act(port)
{
    modify_field(standard_metadata.egress_spec, port);
}

action drop_act()
{
    drop();
}

table fwd_tbl {
    reads {
        ipv4.dst : lpm;
    }
    actions {
        fwd_act;
        drop_act;
    }
}

control ingress {
    apply(fwd_tbl);
}

/* No egress */
```

Save this file to 13minifwd.p4.

E.2 Building the P4 Program

The simplest way to build a P4 program using command line tools is to use the `nfp4build` tool. The following command will build the example for the HYDROGEN platform:

```
$SDK_DIR/p4/bin/nfp4build -p out/ -o 13minifwd.nffw -l hydrogen -4 13minifwd.p4
```

The build process will create two file that will be used further. These are the firmware file `l3minifwd.nffw` and the design description file `out/pif_design.json`.

E.3 Setting up a static P4 Configuration

For this example we will create a `user_config.json` file which will contain rules for the table in the design, `fwd_tbl`. The contents of the file follows:

```
{  
    "doc": "Table configurations for l3minifwd",  
    "tables": {  
        "fwd_tbl": {  
            "default_rule": {  
                "name": "default",  
                "action": { "type": "drop_act" }  
            },  
            "rules": [  
                {  
                    "name": "rule_0",  
                    "match": {  
                        "ipv4.dst": { "value": "10.0.0.0/16" }  
                    },  
                    "action": {  
                        "type": "fwd_act",  
                        "data": {  
                            "port": { "value": "v0.0" }  
                        }  
                    }  
                },  
                {  
                    "name": "rule_1",  
                    "match": {  
                        "ipv4.dst": { "value": "10.1.1.1" }  
                    },  
                    "action": {  
                        "type": "fwd_act",  
                        "data": {  
                            "port": { "value": "p0" }  
                        }  
                    }  
                }  
            ]  
        }  
    }  
}
```

This configuration contains three rules for the table `fwd_tbl`.

1. A default rule to apply if no matches are found. The action to be executed is set to `drop_act` which will perform an explicit drop.

2. A rule named `rule_0` which will apply if the field `ipv4.dst` falls within the provided address and prefix. Note that we have used dotted notation with prefix in the match value entry. We could also have entered `0x0a000000/24`. The action to execute on match is set as `fwd_act`. The action has a single action data entry associated with it, `port`, which we set to `v0.0`. This is a magic value which will set `egress_spec` in a way that the packet will be forwarded to the SRIOV Virtual Function number 0.
3. A rule named `rule_1` which will apply if the field `ipv4.dst` exactly matches `10.0.0.1`. Note that as the table match type is LPM this rule will take precedence over `rule_0` as the implied prefix, `/32`, is longer than the previous rule's. In this case `port` is set to `p0`. This indicates the packet will be sent to physical port 0.

E.4 Loading the Design onto the NFP

At this point you should have the NFP P4 Runtime Server running, before you attempt to load. **IMPORTANT:** you MUST NOT run the hardware debug server when using the command line tools. Doing so will lead to failures during load.

The following command will load the design, including the user configuration, onto the NFP:

```
$SDK_DIR/p4/bin/rtecli design-load -f l3minifwd.nffw -p out/pif_design.json -c user_config.json
```

The user configuration is optional. Note that it is possible to run the `rtecli` command on a machine remote from the P4 runtime server. Use the `-r` option to achieve this as follows:

```
$SDK_DIR/p4/bin/rtecli -r myremotemachine design-load -f l3minifwd.nffw -p out/pif_design.json -c user_config.json
```

If you wish to update the rules without reloading firmware, this is possible using the `rtecli` tool as follows:

```
$SDK_DIR/p4/bin/rtecli config-reload -c user_config.json
```

It is safe to reload configuration while traffic is passing.

E.5 Some useful commands

Get all system counters:

```
$SDK_DIR/p4/bin/rtecli counters list-system
```