Intel® Internet Exchange Architecture Software Development Kit

Software Framework Getting Started Guide

March 2004

Document Number: C53937-002

Contents

Revision History

Date	Revision	Description
March 2004	002	Intel® IXA SDK 3.51
November 2003	001	SDK 3.5. Added IPv6 Application Information

About This Publication 1

This publication, the *Intel® Internet Exchange Architecture Software Development Kit Software Framework Getting Started Guide*, provides an overview of the Intel® Internet Exchange Architecture (Intel® IXA) SDK Software Framework CD and guides you through the following tasks:

- running an example application on the Developer Workbench simulator
- running an IPv4 example application on the Intel® IXDP2400 Advanced Development Platform using core components
- running an IPv6 example application on the Intel® IXDP2400 Advanced Development Platform using core components
- running an IPv4/IPv6 forwarding example application on the Intel® IXDP2800 Advanced Development Platform
- running an example application on the Intel® IXDP2401 Advanced Development Platform
- debugging an example application on the Developer Workbench simulator
- creating a new core component
- porting an existing core component from VxWorks* to Linux
- writing and running a network application in Microengine C using the Intel® Internet Exchange Architecture Portability Framework
- adding a microblock to an application

1.1 Audience

This publication is intended for software developers who will design, develop, and deliver network applications that must process packets at high speed. It assumes that you are familiar with the following:

- C Programming
- realtime network applications
- Developer Workbench that is included with the Intel® IXA SDK Tools CD

1.2 How to Use This Publication

Refer to this publication after you have installed the Intel® Internet Exchange Architecture Software Development Kit Tools CD, the Intel IXA® SDK Firmware and Drivers CD (if you are working with an Advanced Development Platform) and the Intel IXA® SDK Software Framework CD.

The information in this publication is organized as follows:

- Chapter 2, “Overview” provides an overview of the Intel® Internet Exchange Architecture Portability Framework and a directory tour of the Intel IXA® SDK.
- Chapter 3, “Running Applications on the Developer Workbench Simulator” includes steps for running an example application on the Developer Workbench simulator.
- Chapter 4, “Running Applications on Hardware” contains steps for running example applications (IPv4 and IPv6) on an Intel® IXDP2400 Advanced Development Platform.
- Chapter 5, “Debugging Applications on the Developer Workbench Simulator” discusses debugging an example application using break points.
- Chapter 6, “Working With Core Components” contains a variety of information for working with core components (creating a core component, adding a core component, porting a core component from VxWorks* to Linux*, etc.) Information about configuring the system application is also included.
- Chapter 7, “Adding Microblocks to an Application” describes the steps required to change an existing application, by adding new building blocks, modifying dispatch loops, etc. A dispatch loop provides the "glue" that combines building blocks to create a meaningful application
- Chapter 9, “Routing Table and L2 Table” provides reference information for the routing table and the L2 table. Information about populating the routing table to conduct IPv4 and IPv6 ping tests between your development host and an Intel® IXDP2400 Advanced Development Platform is also included.
- Chapter 8, “Using Resource Manager for Linux” discusses how to build the Resource Manager loadable modules for the Linux kernel.
- Appendix A, “IXDP2401 Application Tutorial” provides procedures for running and debugging an example application on the Intel® IXDP 2401 Advanced Development Platform.

1.3 Other Sources of Information

This manual is part of the Intel® Internet Exchange Architecture Software Development Kit documentation set, which also includes the following documents:

Note: There are two *index.htm* files on the CD. One provides a description of the Intel® IXA SDK Software Framework documentation. The other provides a description of the CP-PDK documentation. Refer to both for a complete description of all of the documentation on the Intel® IXA SDK Software Framework CD.

- *Intel® IXA Portability Framework: Reference Manual*
- *Intel® IXA Portability Framework: Developer’s Manual*
- *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer’s Manual*
- *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Reference Manual*
- *Intel ® Internet Exchange Architecture (IXA) Software Building Blocks Applications Design Guide*
- *Intel® IXA SDK Software Framework Release Notes*

Overview 2

This chapter begins with a brief overview of the Intel® Internet Exchange Architecture (IXA) Portability Framework and provides a tour of certain key directories installed with the Intel® IXA SDK. This overview provides a foundation for understanding the following concepts:

- Applications written with the Intel® Internet Exchange Architecture Software Portability Framework
- Microblocks
- Combining microblocks to form an application

2.1 Portability Framework Overview

The Intel® IXA Portability Framework comprises a software infrastructure for writing modular and portable code for network applications for use in VxWorks and Linux configurations. The Intel® IXA Portability Framework provides the following advantages:

- Application development is accelerated due to the infrastructure libraries provided for commonly used functions
- Development of high performance applications is supported by including sample applications running at data rates ranging from OC-12 to OC-192
- Code re-use allows users to leverage their development effort over multiple implementations
- Defined structures provide portability across the IXP2XXX product line architecture

For more details on the Intel® IXA Portability Framework, refer to the *Intel® IXA Portability Framework: Developer's Manual.*

With the tools provided by the Tools CD of the SDK, you can start to develop applications. Performance-critical portions of applications run on the data plane, handling processing and forwarding of packets at high speed. The data plane consists of two kinds of processing:

- fast path processing running on the MEv2 microengines
- slow path processing running on the Intel XScale® core

This chapter focuses on coding for the fast path, using an approach that divides fast path processing into logical networking functions called microblocks. A microblock is a macro or Microengine C function written using low-level libraries and an infrastructure optimized for fast packet processing. Microblocks are different from generic macros, because they have a state associated with them and they operate in a coarse-grain fashion. The libraries and infrastructure enable you to write microblocks that are independent of each other. This independence improves reusability and enables you to combine microblocks in different ways to create many applications, each application precisely fulfilling a particular requirement.

2.2 SDK Software Framework in General

With the applications provided by the Software Framework CD of the SDK, you can start examining how microblocks are organized in an application. To start looking at the organization of an application, first consider how its files are organized into the **\src** directory of the overall SDK directory structure. The **\src** directory is located as follows:

- Windows 2000* or Windows XP* development host: **<install drive>: IXA_SDK_3.5\src**
- Linux Red Hat 7.3 development host: **/opt/ixa_sdk_3.5/src**

Table 2-1 describes the subdirectories of the **src** directory.

Table 2-1. src Subdirectory Descriptions

Directory	Description
applications	Contains application-specific files in subdirectories named for a data plane application. Also, some applications are written in both Microengine C and microcode. Where applicable, the subdirectory **wbench_c_project** contains files for a Microengine C application, and the subdirectory **wbench_project** contains files for a microcode application.
building_blocks	Contains subdirectories for the pre-written modules or microblocks that are shared across applications. Microengine C files are identified by the .c suffix, located in subdirectory named **microc**. Microcode files have a .uc suffix and are located in a subdirectory named **microcode**. For both types of microblock files, there are .h files organized under the specific application directories.
cp_pdk	Contains the Control Plane Platform Development Kit, which provides the tools necessary to connect core components through standardized interfaces. For more information, refer to the CP-PDK Document Index, \documentation\Software-Framework\CP-PDK\index.htm. **NOTE:** The CP-PDK supports the Intel® IXDP2400 Advanced Development Platform hardware only.
EXAMPLES	Consists of a list of example code written primarily to demonstrate programming concepts and new features in the hardware. These examples provide simple illustrations of these concepts and new features.
framework	Contains the files for the core component infrastructure, which are used by the components running on the Intel XScale® core.
include	Contains the common include files.
library	Contains the utility functions or macros commonly used by building blocks and applications—for example, functions for hash table access, CRC computation, endian swaps, and other low-level tasks. Also, this folder contains microblock libraries used for buffer and meta data management.
utilities	Provides command line functions to create route entries through the WindRiver* VxWorks shell.
workspace	Contains WindRiver* Tornado workspace files for building the entire system: a base project for each module to inherit, project files for each module, and several make-support files.

This chapter will focus on the **applications**, **building_blocks**, and **library** subdirectories of the **src** directory.

2.3 Understanding the Directories

This section will help you understand some of the directories installed by the Software Framework CD of the Intel® IXA SDK, including the directory structure, the content, and the rules and rationales.

The directory structure is organized in such a way that the application code is in project directories while the basic building blocks common to several projects are in a separate, parallel directory structure. The applications and building blocks are explained in the following subsections.

2.3.1 Applications Directory

The applications are in the directory **`IXA_SDK_3.5\src\applications`** with subdirectories identifying the application. For the IPV4 forwarder on POS media at OC-48 on an ingress processor example application, the path is as follows:

- Windows*: **`<install drive>:\ixa_sdk_3.5\src\applications\ipv4_forwarder\oc48_pos\ingress`**
- Linux Red Hat*: **`opt/ixa_sdk_3.5/src/applications/ipv4_forwarder/oc48_pos/ingress`**

For Windows*, there are two project-specific subdirectories: **`wbench_c_project`**, written in Microengine C, and **`wbench_project`**, written in microcode. In this chapter, we will concentrate on the Microengine C application. The contents of the **`wbench_c_project`** directory are:

Directory	Description
`dispatch_loop`	Source files that contain the dispatch_loop implementation. The dispatch loop combines one or more microblocks on a microengine and implements the data flow between them. It caches commonly used variables in registers or local memory.
`list`	Output of the build, mainly the .list files generated during assembly and compilation. When compiling Microengine C files, the list directory also contains .obj files.
`log`	Essentially the log of packets at receive and after transmit. These are useful to verify if the program is functioning correctly.
`scripts`	Script .ind files used in system set up, configuration, setting up route tables, and similar tasks
`streams`	Packet streams to send as input to the project
`oc48_pos_ipv4_ingress.dwp`	Workbench project file (Windows* only)

2.3.2 Building Blocks Directory

The building blocks (microblocks) are installed in separate directories under the **`IXA_SDK_3.5\src\building_blocks`** (for Windows*) or **`opt/ixa_sdk_3.5`)/src/building_blocks** (for Linux) directory. Microblocks are written in Microengine C or microcode. Each of the microblocks implements a specific set of functions. A typical application consists of more than one microblock combined together to form an application. Imagine microblocks, in the organization of the Intel® IXA Portability Framework, as a set of “building blocks” that are available for many applications.

The microblocks are modular and independent of other microblocks; under these conditions, they can be re-used to build different applications. For example, the ipv4 microblock used in the oc48_pos_diffserv_ingress application can also be used in the oc48_pos_ipv4_ingress application. Re-use in a number of applications is possible. For more details on building blocks, refer to the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

2.3.3 Library Directory

The Intel® IXA Portability framework provides an extensive set of low-level functions written in Microengine C or microcode. These functions are optimized for high performance and minimal code space utilization. You can use these library functions directly to create microblocks that are easy to read, understand, and maintain. The **`IXA_SDK_3.5\src\library`** (for Windows*) or **`opt/ixa_sdk_3.5)/src/library`** (for Linux) directory contains the following subdirectories:

Directory	Description
`dataplane_library`	Contains the low-level functions which can be used to perform operations such as byte_field decrement, creating buffer descriptor freelist, verifying ipv4 header checksum, and similar low-level tasks. All the functions written in Microengine C are prefixed with **`ixp_`**.
`microblocks_library`	Contains libraries specific to building microblocks, for example, buffer and metadata management.
`xscale`	Contains libraries to run core components running on the Intel XScale® core, for example, fragmentation, route table manager, and L2 table manager.

Running Applications on the Developer Workbench Simulator 3

This chapter provides procedural information for running an example application (4 Gigabit Ethernet IPv4 Forwarding Ingress) on the Developer Workbench simulator. The procedures included in this chapter are designed to clarify the following:

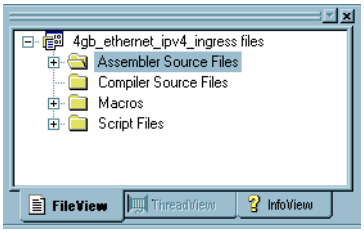
- where the sample applications are located after the Intel® IXA SDK Software Framework CD is installed
- which settings are required to run example applications in simulation mode
- some of the Developer Workbench features

Use the following procedure to run the 4 Gigabit Ethernet IPv4 Forwarding Ingress application on the Developer Workbench simulator:

1. Launch the Developer Workbench from the Windows* Start menu (**Start> Programs> IXA SDK 3.5> DevWorkbench**).
2. From the Developer Workbench Menu toolbar, select **File> Open Project...**
3. A Dialog box appears. Navigate to **<install drive>:\ IXA_SDK_3.5\src\applications\ipv4_forwarder\4gb_ethernet\ingress\wbench_project\ 4gb_ethernet_ipv4_ingress.dwp.**

Note: The .dwp file extension is used for Developer Workbench project files. The .dwp file contains contains project-specific information, including locations of the source/header/script files, compiler/assembler build settings, project level `# define` statements and assignments of blocks of microcode to microengines.

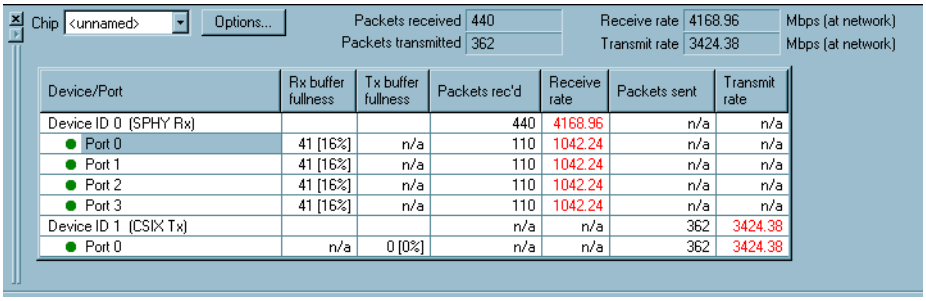
4. Click **Open** to open the Developer Workbench project.
5.
6. Since the 4 Gigabit Ethernet IPv4 Forwarding Ingress application is written in microcode (instead of Microengine C), all of the source code files can be found under the Assembler Source Files folder as shown in the figure below. Click on any one of the files within this folder to view the source code



7. After the project file has been opened, you must set the project build settings for simulation mode. From the Developer Workbench Menu toolbar, select **Build> Settings...**

8. The Build Settings window appears. Under the **General** tab, verify that the **Preprocessor definitions (common to all microengines)** text box contains IXP_SIMULATION. The IXP_SIMULATION flag is used in the source code to establish the correct settings for running the application on hardware versus in simulation mode (on the Developer Workbench transactor).
9. After making changes to the project, you can build the project by selecting **Build> Build** from the Developer Workbench Menu toolbar. Alternatively, you can press the F7 key. You can re-build the project completely by selecting **Build> Rebuild** from the Developer Workbench Menu toolbar (you can also use the Alt+F7 key combination).
10. After the project has been built without any errors, you must configure the simulation settings. To run the application in simulation mode, select **Debug> Simulation** from the Developer Workbench Menu toolbar.
11. You can then select **Simulation> Options...** from the Developer Workbench Menu toolbar to customize the project settings.

    For example, on the **History** tab, you can choose to collect thread, reference and queue histories during simulation. See the *Developer's Tools User Guide* for more information about these debugging features.
12. You must then click either the **Debug** button or select **Debug> Start Debugging** from the Developer Workbench Menu toolbar to initialize the simulator. The simulator can run a number of startup scripts at initialization, such as initializing areas of memory or tables. You can specify these startup scripts on the **Startup** tab that appears when you select **Simulation> Options...** from the Developer Workbench Menu toolbar. Each Developer Workbench application on the Intel® IXA SDK includes the appropriate startup scripts.
13. Click the **Go** button or select **Debug>Run Control>Go** from the Developer Workbench Menu toolbar to start the application in simulation mode.
14. To view the status of packets/cells that are transferred over the IXP2XXX media interface, either click the **Packet Simulation Status** button or select **View> Debug Windows> Packet Simulation Status** from the Developer Workbench Menu toolbar. The window that appears shows the data rates, packet received/transmitted counts etc:



15. Click the **Options** button shown in the Simulation Status (shown above) window to specify the rate at which the data within this window is updated. You can also determine the type of data that is displayed.
16. Either click the **Stop** button or select **Debug> Run Control> Stop** from the Developer Workbench Menu toolbar to stop the application.

17. Use the **Debug> Breakpoint** option from the Developer Workbench Menu toolbar to set breakpoints in the code. Alternatively, you can right-click on a line of code within any of the microengine threads to set up breakpoints.
18. You can then select **Simulation** from the Developer Workbench Menu toolbar to specify individual data streams for the simulator. Each Developer Workbench application includes a set of data streams to be fed into the simulator. However, these data streams can be modified or new ones added.

Running Applications on Hardware 4

This chapter provides procedural information for running example applications on the Intel® IXDP2400 Advanced Development Platform (with 4GbEth I/O option card) using core components. Procedural information is provided for both VxWorks and Linux systems.

Note: For information about running example applications on the hardware without using core components, refer to the `readme` files in sub-directories of the *IXA_SDK_3.5\src\EXAMPLES* directory.

4.1 Connecting to a Packet Generator

Before you run any of the example applications included in this chapter, you will need to use fiber optic cables to connect both the Rx and Tx ports of the development platform to a packet generator. Connect the packet generator's Rx port with the platform's Tx port. Connect the platform's Rx port with the packet generator's Tx port.

Note: If you are using the Advanced Development Platform's copper Ethernet ports, refer to section Section 4.4, "Using the Advanced Development Platform's Copper Ethernet Ports" on page 30.

4.2 Windows 2000/XP with VxWorks Systems

This section provides procedural information for running an IPv4 forwarding application on an Ethernet pipeline within a Windows 2000 system. Before following the procedures outlined in this section, you must use the procedures in the *Intel® Internet Exchange Architecture Software Development Kit Tools Installation Guide* to set up your system with the following:

- Tornado* 2.2 or later software
- running FTP server
- HyperTerminals for the Ingress and Egress NPUs
- target servers for both the Ingress and Egress NPUs

Note: Before following the procedures in this chapter, ensure that the Tornado 2.2.1 system variable is set correctly. This is done as follows:

a. In the WIND_BASE directory (e.g: c:\\Tornado2.2.1), run *torvars.bat* to set the path and environment variables.

b. Select My Computer--> Properties-->Advanced --> Environment Variables

c. At the top window, click New. A New User and System Variable window appears, enter the following settings:

Variable Name: Path

Variable Value: C:\\Tornado2.2.1\\host\\x86-win32\\bin

4.2.1 IPv4 Forwarding Application on Hardware Using Core Components

Use the following procedure to run the example IPv4 forwarding application on an Ethernet pipeline:

1. Connect the packet generator to Port 0 of the Intel® IXDP2400 Advanced Development Platform.
2. Launch the Developer's Workbench (**Start>Programs>IXA_ SDK 3.5>DevWorkbench**)
3. From the **File** menu, select **Open Project...**.
4. Open the Developer's Workbench project for the Egress NPU. It is located at **IXA_SDK_3.5\src\applications\ipv4_forwarder\4gb_ethernet\egress\wbench_project\4gb_ethernet_ipv4_egress.dwp**.
5. On a Windows machine, you will need to build uof files for the Egress and Ingress side. Open the Developers' Workbench and open the project **4gb_ethernet_ipv4_egress.dwp** under **IXA_SDK_3.5/src/applications/ipv4_forwarder/4gb_ethernet/egress/wbench_project** and build the uof file.

   Open the ingress project (**4gb_ethernet_ipv4_ingress.dwp**) under **IXA_SDK_3.5/src/applications/ipv4_forwarder/4gb_ethernet/ingress/wbench_project** and build the uof file. You will need to copy these two uof files onto the Linux distribution directory, which is **/opt/xscale_be_test/linux_kernel/xscale_be/ixp2400** in case of release build and **/opt/xscale_be_test/linux_kernel/xscale_be/ixp2400/debug** in case of the debug build.
6. Go to the **Build** menu in the Developer's Workbench and select **Settings...**. Under **Preprocessor definitions**, remove IXP_SIMULATION and add USE_IMPORT_VAR.
7. Click **OK**.
8. Return to the **Build** menu and select **Rebuild**. This will create a microcode image file for the Egress.
9. Open the Developer's Workbench project (in the same Developer's Workbench instance) for Ingress NPU located at **IXA_SDK_3.5\src\applications\ipv4_forwarder\4gb_ethernet\ingress\wbench_project\4gb_ethernet_ipv4_ingress.dwp.**
10. Go to the **Build** menu in the Developer's Workbench and select **Settings...**. Under **Preprocessor definitions**, remove IXP_SIMULATION and add USE_IMPORT_VAR.
11. Click **OK**.
12. Return to the **Build** menu and select **Rebuild**. This will create a microcode image file for the Ingress.
13. Open the Tornado* workspace at **IXA_SDK_3.5\src\workspace\ixa_sdk_2.2.wsp**.
14. Click on the **Builds** tab in the workspace window. You should look at the *A_oc48_ethernet_egress* project. Make sure that the **Build type** selected for the *A_oc48_ethernet_egress* project is "XScalegnube". If you want to run the application in Debug mode, then right click and select the **Build type** as "XScalegnube_Debug". Debug mode will print more information to the screen.
15. Build the *A_oc48_ethernet_egress* project by right clicking and selecting **Rebuild All**.
16. In the Tornado* workspace, now look at *A_oc48_ethernet_ingress* project. Click on the **Builds** tab in the workspace window. Make sure that the **Build type** selected for the *A_oc48_ethernet_ingress* project. is "XScalegnube". If you want to run the application in

Debug mode, then right click and select the **Build type** as "XScalegnube_Debug". Debug mode will print more information to the screen.

17. Build the *A_oc48_ethernet _ingress* project by right clicking and selecting **Rebuild All**.
18. You can now download the Egress image. Make sure that the name of the Egress target server is selected in the Tornado* window's drop down list.
19. Right click on *A_oc48_ethernet_egress* and select **Download A_oc48_ethernet_egress.out**.
20. You can now download the Ingress image. Make sure that the name of the Ingress target server is selected in the Tornado* window's drop down list.
21. Right click on *A_oc48_ethernet_ingress* and select **Download A_oc48_ethernet_ingress.out**.
22. Start the system application on Egress by running "_ix_sa_entry 1" on the Egress shell. The Link LED on the platform should go on. Up to this point, the Error LED should have been lit.
23. The system application prints "Started all the microengines". Start the system application on the Ingress by running "_ix_sa_entry 0" on the Ingress shell.
24. The system application prints "Started all the microengines". Both the Ingress and Egress shells should no longer be usable.
25. Open two more shells, one Egress and one Ingress.
26. If you want to send packets through the system, then configure the packet generator to send packets out with a particular destination IPv4 address. For example, you can setup route table entries as shown below to send packets with a destination address of 32.0.0.1:

    On the Ingress shell, add these routes:

    ```
    addNextHop "20 1 1 0 9180 0 20.0.0.2 0"
    addRoute "32.0.0.1 255.255.255.255 20"
    ```

    Refer to Section 9.1, "Routing Table" on page 65 for more information about populating the Routing Table.
27. On the Egress shell, add L2 table entries by typing the following command:

    ```
    addV4EthEntry "1 20.0.0.2 oa:0b:0c:0d:0e:02 0a:0b:0c:0d:0e:01 DEFAULT"
    ```

    Refer to Section 9.2, "L2 Table" on page 68 for more information about populating the L2 Table.

    Refer to Section 9.2.1, "Layer 2 Table Manager" on page 68 for more information about L2 Table Manager commands.
28. Set up the packet generator to send IPv4 packets with destination address 32.0.0.1 (source address, source/destination MAC address can be anything).

    Note: To make it easy to view, make sure you do not send a burst, instead set the packet generator up to send one packet at a time.
29. Send the packets. The first packet usually doesn't go through, but subsequent packets should be received. When you send a 64-byte packet, it should go through a microblock pipeline and should be received at the packet generator.

4.3 Red Hat/Monta Vista Linux Systems

This section provides procedural information for running IPv4 and IPv6 forwarding applications on an Ethernet pipeline within a Linux system.

4.3.1 Setting up Linux Minicom for Egress and Ingress NPUs

This section provides procedural information for initiating a minicom session between the Linux development host and the Advanced Development Platform's Egress/Ingress NPUs.

1. Verify that the following Red Hat* 7.3 packages are installed on the development host:
   a. DHCP server package (dhcp-2.0pl5-8.i386.rpm). To check if the package is already installed, issue the following command from the shell prompt:
      `hostpc# rpm -qa |grep dhcp`

      If the DHCP server package is not installed, then insert Red Hat 7.3 CD #2, mount the CD-ROM drive and type the following command from the shell prompt:
      hostpc# rpm -i /mnt/cdrom/redhat/RPMS/dhcp-2.0p15-8.i386.rpm
   b. TFTP server package (tftp-server-0.28-2.i386.rpm). To check if the package is already installed, issue the following command from the shell prompt:
      hostpc# rpm -qa |grep tftp

      If the TFTP server package is not installed, then insert the Red Hat 7.3 CD #3, mount the CD-ROM drive and type the following command from the shell prompt:
      hostpc# rpm -i /mnt/cdrom/redhat/RPMS/tftp-server-0.28-2.i386.rpm

      After you install the TFTP server package, you must create a **/tftpboot** directory. Enter the following command:
      hostpc# mkdir /tftpboot
2. Install the Monta Vista* software with the latest patches. Refer to the Monta Vista documentation for installation information.
3. Add the following lines to **/etc/exports** to export the root file system:
   /opt/hardhat/devkit/arm/xscale_be/target*(rw,no_root_squash,no_all_squash)
   /opt/xscale_be_test/linux_kernel/xscale_be *(rw,no_root_squash,no_all_squash)
   a. Use the following command to restart NFS:
      hostpc# /etc/rc.d/init.d/nfs restart
   b. Enter the following command to verify that the root file system export is present:
      hostpc# exportfs
      /opt/hardhat/devkit/arm/xscale_be/target *
4. Copy the Linux kernel image (zImage) to the **/tftboot** directory with the following command:
   hostpc# cp zImage /tftpboot
5. Ensure that the tftp protocol is enabled in the **/etc/xinetd.d/tftp** file. If required, change **disable=yes** to **disable=no**.
6. Perform a soft reboot on the Advanced Development Platform and restart minicom for both Egress and Ingress. Type the following command at the Master Redboot prompt:
   Master-Redboot> cfg read -n 1
   Determine the IP addresses for both the Master and Slave Ethernet interfaces. Ensure that the development host and the target interfaces reside on the same subnet.
7. Set the correct MAC address in **/etc/dhcpd.conf** as follows:

```
subnet 10.3.31.0 netmask 255.255.255.0 {
   }
   subnet 10.10.10.0 netmask 255.255.255.0 {
      host master {
```

```
            hardware Ethernet 00:90:d7:00:11:1f;
            fixed-address 10.10.10.1;
            option root-path "/opt/hardhat/devkit/arm/xscale_be/target";
   }
   host slave {
            hardware Ethernet 00:90:d7:00:11:20;
            fixed-address 10.10.10.2;
            option root-path "/opt/hardhat/devkit/arm/xscale_be/target";
      }
}
```

8. Run the following command to start inetd on the development host:
   hostpc# /etc/rc.d/init.d/xinetd restart
9. Issue the following command on the Egress side:
   Master-Redboot> load -r -b 0x1c208000 zImage
10. Use the following command to boot Linux:
    Master-Redboot> go 0x1c208000
11. Linux should boot up and display the login prompt. Login as root, without any password.
12. Repeat steps 10 and 11 on the Ingress side.
13. To NFS mount a particular directory on the target, type the following command:
    mount -o vers=2 10.10.10.10:/opt/xscale_be_test/linux_kernel/xscale_be /mnt

Note: Once you have setup minicom for Egress and Ingress, you can use the automated scripts located in **`opt/ixa_sdk_3.5/src/utilities/linux_setup_minicom`** directory to launch minicom for the Egress and Ingress. Copy the following scripts from the **`opt/ixa_sdk_3.5/src/utilities/`** **linux_setup_minicom** directory to the **/minicom** directory:

- bootixp.egress
- bootixp.ingress
- start.egress
- start.ingress

You can customize the **start.egress** and **start.ingress** scripts based on the IP addresses for your development host and target.

After the scripts have been copied, you can launch minicom for egress by running **./bootixp.egress** and launch minicom for ingress by running **./bootixp.ingress**.

4.3.2 IPv4 Forwarding Application on an Intel® IXDP2400 Using Core Components

Use the following procedure to run the example IPv4 forwarding application on an Ethernet pipeline:

1. Set the following environment variables on the Linux host:
   export IXA_SDK_DEV=/opt/ixa_sdk_3.5
   export PATH=$PATH:/opt/hardhat/devkit/arm/xscale_be/bin
   export IXP2XXX_TOOLCHAIN_ROOT=/opt/hardhat/devkit/arm/xscale_be
   export IXP2400_KERNEL_SOURCE_ROOT=/opt/hardhat/devkit/lsp/intel-ixdp2400-arm_xscale_be/linux-2.4.18_mvl30

Refer to Chapter 8, "Using Resource Manager for Linux" for more information about Linux environment variables.

2. To build the Egress side application, go to **$(IXA_SDK_DEV)/src/applications/ipv4_forwarder/4gb_ethernet/egress** directory and type the following commands:
   make -f Makefile.linux_kernel clean
   make -f Makefile.linux_kernel
3. To build the Ingress side application, go to **$(IXA_SDK_DEV)/src/applications/ipv4_forwarder/4gb_Ethernet/ingress** directory and type the following commands:
   make -f Makefile.linux_kernel clean
   make -f Makefile.linux_kernel

   Note: The command issued in step 2 will also copy the relevant modules to the **/opt/xscale_be_test/linux_kernel/xscale_be** directory that has been mounted on the target as **/mnt**.
4. Copy **halMeDrv.o, halMev2_lib.o, libossl.o** and **uclo_lib.o** from **$(IXA_SDK_DEV)/me_tools/bin_linux_kernel_be** to the distribution directory (**/opt/xscale_be_test/linux_kernel/xscale_be/ixp2400/debug** in case of debug build and **/opt/xscale_be_test/linux_kernel/xscale_be/ixp2400** in case of release build).
5. Copy the following scripts from the **$(IXA_SDK_DEV)/src/utilities/run_app** directory to **/opt/xscale_be_test/linux_kernel/xscale_be/ixp2400/debug** in case of debug build and to **/opt/xscale_be_test/linux_kernel/xscale_be/ixp2400** directory in case of release build:

- startegress.sh
- startingress.sh
- stopegress.sh
- stopingress.sh

6. Launch minicom for Egress and Ingress. (Refer to section Section 4.3.1, "Setting up Linux Minicom for Egress and Ingress NPUs" on page 20 for more information.)
7. Type one of the following commands on the Egress minicom prompt:
   cd /mnt/ixp2400/debug (for debug build)
   cd /mnt/ixp2400 (for release build)
8. Run the following command, again at the Egress prompt:
   ./startegress.sh
   This loads all the required Linux kernel modules.
9. A message will appear that reminds you to create the following device nodes on the Egress side:
   mknod /dev/saUtil c 254 0 (this is required to run the system application)
   mknod /dev/L2Config c 252 0 (this is required to run the L2 configuration utility)
10. Run the following command to start the system application on the Egress side:
    ./sa start 1

    Note: After the "Started All microengines" message appears, this shell will not be usable. You must telnet into the Egress IP address to start another shell.
11. On the minicom for Ingress, enter the following command:
    cd /mnt/ixp2400/debug (for debug build)
    cd /mnt/ixp2400 (for release build)
12. Run the following command:
    ./startingress.sh
    This will load all the required modules on the Ingress side.

13. A message will remind you to create the following device nodes on the Egress side:
    mknod /dev/SaUtil c 254 0 (this is required by the system application)
    mknod /dev/RConfig c 253 0 (this is required to run the Route configuration utility)
14. Run the following command to start the system application on the Ingress side:
    ./sa start 0
15. Add the route to the routing table on the Ingress side by entering the following:
    ```
    ./rconfig addNextHop "9 1 1 0 1500 0 10.10.10.5 0"
    ./rconfig addRoute "32.0.0.1 255.255.255.255 9"
    ```
16. Add the L2 table entry on the Egress side as follows:
    ```
    ./l2config addV4EthEntry "1 10.10.10.5 01:02:03:04:05:06 0a:0b:0c:0d:0e:0f
    DEFAULT"
    ```

After you have connected port 0 of the IXP2400 to port 0 of the packet generator, send a packet with IP address 32.0.0.1. This will be forwarded and received by the packet generator.

4.3.3 IPv6 Forwarding Application on an Intel® IXDP2400 Using Core Components

This section provides procedural information for running an IPv6 forwarding, NATPT and IPv6-v4 tunneling application on an Ethernet pipeline. Before following the procedures in this section, you must use the procedures in the *Intel® Internet Exchange Architecture Software Development Kit Tools Installation Guide* to configure your system with the following:

- MontaVista Linux 3.0, Professional Edition* with latest patches
- start NFS by entering the following command:
  /etc/rc.d/init.d/nfs start
- start the xinetd daemon by issuing the following command:
  /etc/rc.d/init.d/xinetd start
- start the dhcpd daemon by issuing the following command:
  /etc/rc.d/init.d/dhcpd start
- start minicom sessions for the egress and ingress target NPUs

Note: If you are using the Advanced Development Platform's copper Ethernet ports, refer to Section 4.4, "Using the Advanced Development Platform's Copper Ethernet Ports" on page 30.

Use the following procedure to run an IPv6 forwarding, NATPT and IPv6-v4 tunneling application on an Ethernet pipeline:

1. Set the following environment variables on the Linux development host:
   export IXA_SDK_DEV=/opt/ixa_sdk_3.5
   export PATH=$PATH:/opt/hardhat/devkit/arm/xscale_be/bin
   export IXP2XXX_TOOLCHAIN_ROOT=/opt/hardhat/devkit/arm/xscale_be
   export IXP2400_KERNEL_SOURCE_ROOT=/opt/hardhat/devkit/lsp/intel-ixdp2400-arm_xscale_be/linux-2.4.18_mvl130
   Refer to Chapter 8, "Using Resource Manager for Linux" for information about Linux environment variables.
2. Connect a packet generator to Port 0 of the Intel® IXDP2400 Advanced Development Platform.

3. Open the Makefile for the ingress NPU located at **IXA_SDK_3.5\src\applications\ipv4_v6_forwarder\4gb_ethernet\ingress\wbench_project** and use the USE_IMPORT_VAR to change the IXP_SIMULATION flag.
4. Generate the microcode image file for the ingress NPU by running the following:

   make -f Makefile
5. Open the Makefile for the egress NPU located at **IXA_SDK_3.5\src\applications\ipv4_forwarder\4gb_ethernet\egress\wbench_project** and use the USE_IMPORT_VAR to change the IXP_SIMULATION flag.
6. Generate the microcode image file for the egress NPU by running the following:

   make -f Makefile
7. Build and generate all necessary modules required for the IPv6 ingress application by executing the following commands in the directory **/opt/IXA_SDK_3.5/src/applications/ipv4_v6_forwarder/4gb_ethernet/ingress**:

   make -f Makefile.linux_kernel clean

   make -f Makefile.linux_kernel
8. Build and generate all necessary moduled required for the IPv4 egress application by executing the following commands in the directory **/opt/IXA_SDK_3.5/src/applications/ipv4_forwarder/4gb_ethernet/egress**:

   make -f Makefile.linux_kernel clean

   make -f Makefile.linux_kernel
9. Mount the host distribution directory by entering the following command at the ingress and egress minicom prompts:

   mount -o vers=2 <hostIpAddress>:/opt/xscale_be_test/linux_kernel/xscale_be/ixp2400/debug /mnt

   opt/xscale_be_test/linux_kernel/xscale_be/ixp2400/debug is the distribution directory where modules and script files are available. The host IP address and distribution directory are user defined.
10. Insert all related modules at egress by running the **startegress.sh** script. This script inserts all related modules for the egress NPU.
11. A message appears that reminds you to create the following device nodes on the egress NPU:

    mknod /dev/saUtil c 254 0 (this is required to run the system application)

    mknod /dev/L2Config c 252 0 (this is required to run the L2 configuration utility)
12. Insert all related modules at ingress by executing the **start6ingress.sh** script. This script inserts all related modules for the ingress NPU.
13. A message appears that reminds you to create the following device nodes on the ingress side:

    mknod /dev/saUtil c 254 0 (this is required to run the system application)

    mknod /dev/RConfig c 253 0 (this is required to run the route configuration utility)

    mknod /dev/NatptConfig c 240 0 (this is required to run the NATPT configuration utility)

    mknod /dev/TunnelConfig c 241 0 (this is required to run the TUNNEL configuration utility)
14. Start the system application on the egress NPU by executing the following command at the egress prompt:

    ./sa start 1

    The link LED on the advanced development platform should go on. Until this point, the Error LED should have been lit.

15. The system application outputs the following message:

    Started all microengines.

    Add a default L2 table entry at the egress prompt. Use the reserved L2 Index 100 as follows:

    ./l2config addV6EthEntry “100 ff00::0 0a:0b:0c:0d:0e:02 0a:0b:0c:0d:0e:01

    This is required for auto address configuration to work properly. Refer to Chapter 9, “Routing Table and L2 Table” for more information about L2 table manager commands.

16. Start the system application on the ingress NPU by running the following command at the ingress prompt:

    ./sa start 2

17. To send packets through the system, configure the packet generator to send packets out with a particular IPv6 destination address. For example, you can setup route table entries as shown below to send packets with a destination address of 3ffe:0:100:f102::fe0d:0e0f:

    On the ingress shell, add these routes:

    ./rconfig addNextHopV6 “4 1 2 1 1500 0 0”
    ./rconfig addRouteV6 “3ffe:0:100:f102::fe0d:0e0f 64 4

    On the egress shell, add L2 table entries:

    /l2config addV6EthEntry “23ffe:0:100:f102::fe0d:0e0f 00:07:e9:ad:5c:e4 0b:0c:0d:0e:0f:0a DEFAULT”

18. Set up the packet generator to send IPv6 packets with destination address of **3ffe:0:100:f102::fe0d:0d0f**

Note: Source address, source/destination MAC address can be set to anything. To make viewing easier, do not send a burst. Instead, set the packet generator to send one packet at a time.

4.3.4 10x1GB IPv4/v6 Forwarding Application on Intel® IXDP2800 Using Core Components

This section provides procedural information for running an IPv4/v6 forwarding application on an Intel® IXDP2800 10x1GB Advanced Development platform.

Use the following procedure to run an IPv4/v6 10x1GB forwarding application on an Ethernet pipeline:

1. Set up the Linux development host. Refer to the steps described in section Section 4.3, “Red Hat/Monta Vista Linux Systems” on page 19 for the information about how to install, load and launch the Linux image. These include:
   - install MontaVista Linux 3.0 and any patches
   - install Intel® IXDP2800 Linux Support Package
   - re-build Linux kernel image (zImage)
   - Host IP Setup
   - TFTP Server (TFTPD) Setup
   - DHCP Server (DHCPD) Setup
   - NFS Server (NFSD) Setup
2. Setup the minicom console sessions:

To connect the serial ports from the Intel® IXDP2800 Advanced Development platform to your PC, use the serial connectors provided with the platform.

On the Linux host system, launch minicom and setup two console sessions: one for Ingress and one for Egress. Settings are as follows:

— Baud Rate to 9600

— 8 data bits

— No Parity

— 1 Stop bit

— Flow controls set to None.

3. Re-build the Linux kernel with IPv6 support:

   To support IPv6, the Linux kernel image needs to be re-built with IPv6 enabled. To do this, follow the steps below after you have installed MVL 3.0 for the Intel® IXDP2800:

   a. Go to the MontaVista* web site, MVzone, to get the **ipv6_symbols.patch** (www.mvista.com).

   b. On your Linux host, cd to the IXDP2800 LSP directory and copy the patch to this directory:

   ```
   hostpc# cd /opt/hardhat/devkit/lsp/intel-ixdp2800-arm_xscale_be/linux-2.4.18_mvl30
   ```

   c. Export PATH for armtoolchain as follows:

   ```
   hostpc# export PATH=$PATH:/opt/hardhat/devkit/arm/xscale_be/bin:
   ```

   d. Apply the patch:

   ```
   hostpc# patch -p1 < ipv6_symbols.patch
   ```

   e. Enable the IPv6 in the LSP configuration file and re-build the kernel:

   ```
   hostpc# make distclean
   hostpc# make ixdp2800_config
   ```

   f. Run xconfig GUI software to enable IPv6 in the configuration file:

   ```
   hostpc# make xconfig
   ```

   g. A **Linux Kernel Configuration** window will appear. Select the **Networking Options** button.

   h. Scroll down and on the item called The **IPv6 protocol (EXPERIMENTAL)**.

   i. Check the **y** box.

   j. Click the **Main Menu** button.

   k. Click **Save and Exit** button.

   l. Continue with the following commands:

   ```
   hostpc# make oldconfig
   hostpc# make dep
   hostpc# make zImage
   ```

   m. When the build process finishes, a new compressed kernel image with IPv6 support is generated in **/opt/hardhat/devkit/lsp/intel-ixdp2800-arm_xscale_be/linux-2.4.18_mvl30/arch/arm/boot/zImage.** Copy the **zImage** to the **tftp** boot directory:

   ```
   hostpc# cp /opt/hardhat/devkit/lsp/intel-ixdp2800-arm_xscale_be/linux-2.4.18_mvl30/arch/arm/boot/zImage /tftpboot/
   ```

4. Launch the Intel® IXDP2800 Advanced Development Platform Ingress and Egress Processors:

a. Power up the Intel® IXDP2800 Advanced Development Platform. Type the following command in the Egress and Ingress minicom consoles:

```
>Egress> load \\<Host IP>\zImage 0x01008000
>Ingress> load \\<Host IP>\zImage 0x01008000
```

where <Host IP> is the IP address of the host system and zImage is the Linux kernel image.

b. Enter the following at the Ingress prompt:

```
>Ingress> launch 0x01008000
```

c. Enter the following at the Egress prompt:

```
>Egress> launch 0x01008000
```

Note: The launch command must be entered on the Ingress first so that the proper PCI initialization occurs.

d. Both Egress and Ingress processors should boot to the Monta Vista Linux 3.0 login prompt. You may now login to your target system under username root with no password.

5. Build the Ingress and Egress core components applications:

a. Ensure that the Intel® IXA SDK Tools CD, Intel® IXA SDK Firmware and Drivers CD and Intel® IXA SDK Software Framework CD are installed on your Linux development host.

b. Define the following environmental variables and path in your Linux development host:

```
export IXA_SDK_DEV=/opt/ixa_sdk_3.5
export IXP2XXX_TOOLCHAIN_ROOT=/opt/hardhat/devkit/arm/xscale_be
export IXP2800_KERNEL_SOURCE_ROOT=/opt/hardhat/devkit/lsp/intel-ixdp2800-
arm_xscale_be/linux-2.4.18_mvl30
export PATH=$PATH:/opt/hardhat/devkit/arm/xscale_be/bin
```

c. To build the Ingress core component application, change to the following directory:

```
/opt/ixa_sdk_3.5/src/applications/ipv4_v6_forwarder/10gb_ethernet/
10x1GbE_ingress
```

d. Enter the following commands:

```
$make -f Makefile.linux_kernel clean
$make -f Makefile.linux_kernel
```

e. To build the Egress core component application, change to the following directory:

```
/opt/ixa_sdk_3.5/src/applications/ipv4_v6_forwarder/10gb_ethernet/
10x1GbE_egress
```

f. Enter the following commands:

```
$make -f Makefile.linux_kernel clean
$make -f Makefile.linux_kernel
```

Note: The above Ingress and Egress build processes will install all the objects required by the applications to the distribution directory. This distribution directory is to be NFS-mounted from the target minicom console later in this procedure. This allows the Egress/Ingress processors to access the minicom sessions. The default distribution directory is defined as */opt/xscale_be_test/linux_kernel/scale_be/ixp2800/debug*.

6. Build the IPv4/V6 Ingress and Egress microengine images:

a. Install Intel® IXA SDK Tools CD and Intel® IXA SDK Firmware and Drivers CD on a Windows machine.

b. Launch the Developers Workbench.

c. Open an Ingress project from the Developers Workbench.
-- Select **File -> Open Project**
--Select *C:\IXA_SDK_3.5\src\applications\ipv4_v6_forwarder\10gb_ethernet\10x1GbE_ingress\wbench_project\10gb_ethernet_ingress.dwp*
--Under **Build->Settings**, add **USE_IMPORT_VAR** in the preprocessor definition box.
--Rebuild the image by selecting **Build->Rebuild**
--ftp the generated uof file (10gb_ethernet_ixp2800.uof) to the Linux host machine and copy it to the distribution directory (i.e. */opt/xscale_be_test/linux_kernel/scale_be/ixp2800/debug*).

d. Open the Egress project from the Developers Workbench
-- Select **File -> Open project**
-- Select *C:\IXA_SDK_3.5\src\applications\ipv4_v6_forwarder\10gb_ethernet\10x1GbE_egress\wbench_project\10x1gb_ethernet_egress.dwp*
--Under **Build->Settings**, add **USE_IMPORT_VAR** in the preprocessor definition box.
--Rebuild the image by selecting Build->Rebuild
--ftp the generated uof file (10x1gb_ethernet_egress.uof) to the Linux host machine and copy it to the distribution directory (i.e. */opt/xscale_be_test/linux_kernel/scale_be/ixp2800/debug)*.

7. Run the core component application:

a. On Ingress and Egress minicom windows, type the following command to mount the host distribution base directory:

```
mount -o vers=2 10.10.10.10:/opt/xscale_be_test/linux_kernel/xscale_be/mnt
```

(where 10.10.10.10 is the IP address of the Linux host machine)

b. Go to the distribution directory:

```
cd /mnt/ixp2800/debug
```

8. Load the Ingress and Egress application modules by typing the following script commands:

```
Ingress : ./startingress_2800
Egress  : ./startegress_2800
```

c. Enter the following commands in the order specified:

```
Ingress : ./sa imi1
Egress  : ./sa emi1
Ingress : ./sa imi2
Egress  : ./sa emi2
Ingress : ./sa imi3
Egress  : ./sa emi3a
Egress  : ./sa start 1
```

d. At this point, the Egress system app prints **Started all the microengines** and egress minicom console will freeze. Open a new command terminal from the Linux host machine and from there, telnet into the Egress target by typing "telnet <Egress IP>". Login to the Egress target system under username root with no password. Go to the distribution debug directory (i.e. cd `/mnt/ixp2800/debug`) and continue to execute the following steps:

```
Egress  : ./sa emi3b
Ingress : ./sa imi4
Egress  : ./sa emi4
```

Repeat the above steps starting from "./sa imi1" if there is a failure at any point. Strict order is to be maintained between imi1 and emi4 commands.

```
Ingress : ./sa start 1
```

e. At this point, the Ingress system app prints **Started all the microengines** and ingress minicom console will be freezed. Open a new command terminal from the Linux host machine and telnet into the Ingress target by typing "telnet <Ingress IP>". Login to the Ingress target system under username root with no password. Go to the distribution debug directory and continue with the next steps for adding route entries and l2 entries.

9. Set IPv4 and IPv6 route tables and L2 table for v4 forwarding or v6 forwarding:

   At this point, you may use the commands "./rconfig" and "./l2config" to set up the desired route table entries on Ingress and l2 entries on Egress for IPv4 or IPv6 data path tests. Please refer to Chapter 9, "Routing Table and L2 Table"for details on the usage of these utility commands.

   For IPv4 packets, refer to steps a-d. For IPv6 packets, refer to steps e-h.

   a. For example, to send IPv4 packets through the system, configure the packet generator to send packets out with a particular IPv4 destination address. You can setup IPv4 route table as shown below to send packets with a destination address 141.131.31.1:

   b. On the Ingress telnet window, type the following commands:

```
./rconfig addNextHop "8 2 1 1 1500 0 100.100.100.1 0"
./rconfig addRoute "141.131.31.1 255.255.255.255 8"
```

   c. On the Egress telnet window, type the following command:

```
./l2config addV4EthEntry "1 100.100.100.1 0a:0b:0c:0e:03:05
08:09:00:0a:0e:01 DEFAULT"
```

   d. setup your packet generator to send IPv4 packets with destination address of 141.131.31.1 to port 0 of the Intel® IXDP2800 platform. You should get the packet routed back by the system through port 1.

   e. To send IPv6 packets through the system, configure the packet generator to send packets out with a particular IPv6 destination address. You can setup IPv6 route table as shown below to send packets with a destination address of `3ffe:5555:6666:6666:7777:7777:8888:8888`

   f. On the Ingress telnet window, type the following commands:

```
./rconfig addNextHopV6 "16 2 7 2 1500 0 0"
./rconfig addRouteV6 "3ffe:5555:6666:6666:7777:7777:8888:8888 64 16"
```

   g. On the Egress telnet window, type the following command:

```
./l2config addV6EthEntry "7 3ffe:5555:6666:6666:7777:7777:8888:8888
0a:0b:0c:0e:03:05 0a:0b:0c:0d:0e:0f DEFAULT"
```

   h. Setup your packet generator to send IPv6 packets with destination address of 3ffe:5555:6666:6666:7777:7777:8888:8888 to port 0 of the IXDP2800 platform. You should get the packet routed back by the system through port 2.

Note: Source IP address, source/destination MAC addresses can be set to anything. To make viewing easier, you may set the packet generator to send one packet at a time.

10. Set IPv4 and IPv6 route tables and L2 table for v6/v4 tunneling:

    At this point, you may also test the tunneling feature of the system. Use the commands "./tunnelconfig", "./rconfig" and "./l2config" to enable the tunneling feature and add desired route entries on Ingress and l2 entries on Egress.

    For example, an IPv6 packet can be sent over IPv4 network by encapsulating it in an IPv4 packet. To enable v6/v4 encapsulation in tunneling, you can setup IPv4 and IPv6 route tables as shown below to send packets with a destination address 3ffe:0:100:f102::fe0d:0e0f

    a. On the Ingress telnet window, type the following commands:

```
./tunnelconfig addStartTunnel "1 2 0 1500 10.0.0.45 0.0.0.0 30 10"
./rconfig addNextHopV6 "3 2 2 2 1500 0 2"
```

```
./rconfig addRouteV6 "3ffe:0:100:f102::fe0d:0e0f 64 3"
./rconfig addNextHop "3 2 2 2 1500 0 10.0.0.45 0"
./rconfig addRoute "10.0.0.45 255.255.255.255 3"
```

b. On the Egress telnet window, type the following command:

```
./l2config addV4EthEntry "2 10.0.0.45 00:07:e9:ad:5d:e4 0a:0b:0c:0d:0e:0f
DEFAULT"
```

Setup your packet generator to send IPv6 packets with destination address of 3ffe:0:100:f102::fe0d:0e0f to port 0 of the Intel® IXDP2800 platform. You should get an IPv4 packet back at port 2 of the system. This IPv4 packet should have the original IPv6 packet encapsulated by IPv4 header with destination IP address of 10.0.0.45.

Decapsulation is used when an encapsulated IPv4 packet reaches the end of a tunnel. The process involves stripping the IPv4 header to get the IPv6 packet. This requires the setup of an end tunnel as described below.

c. On the Ingress telnet window, type the following commands:

```
./tunnelconfig addEndTunnel "2"
./tunnelconfig addAllowedSource "0x1 10.1.2.3 24"
./rconfig addNextHop "4 2 3 0 1500 0 10.0.0.1 0x2"
./rconfig addRoute "20.1.2.3 255.255.255.255 4"
./rconfig addNextHopV6 "4 2 3 0 1500 0 0"
./rconfig addRouteV6 "3ffe:0:0:3000::0 64 4"
```

d. On the Egress telnet window, type the following command:

```
./l2config addV6EthEntry "3 3ffe::0 00:07:e9:ad:5d:e4 0a:0b:0c:0d:0e:0f
DEFAULT"
```

e. Setup your packet generator to send IPv4/v6 tunneling packets with IPv4 destination address of 20.1.2.3 and source address of 10.1.2.3. The destination IPv6 address should be set to 3ffe:0:0:3000::0:303 and the IPv6 source address can be set to any valid v6 address. Send the packet to port 0 of the Intel® IXDP2800 system. You should get an IPv6 packet back at port 0 of the system. This IPv6 packet should have the destination IP address of 3ffe:0:0:3000::0:303.

4.4 Using the Advanced Development Platform's Copper Ethernet Ports

The default for the Intel® IXDP2400 Advanced Development Platform's Ethernet ports is fiber mode. If you want to use the copper ports on the system, this default setting will need to be changed in an external header file and the Egress application used by this platform (e.g. IXA_SDK_3.5\src\applications\ipv4_forwarder\4gb_ethernet_egress) will have to be re-built using the procedures in either Section 4.2, "Windows 2000/XP with VxWorks Systems" on page 17 or Section 4.3, "Red Hat/Monta Vista Linux Systems" on page 19.

To use the copper ports, edit the *ix_cc_eth_tx.h* external header file. It is located at **IXA_SDK_3.5\src\building_blocks\tx\core\ethernet_tx|include\cc**. Open the file and search for the IX_CC_ETH_TX_DRIVER_MODE_DEFAULT string. It is defined in the file as follows:

```
#define IX_CC_ETH_TX_DRIVER_MODE_DEFAULT(
(DEFAULT_MODE_BLOCK|DEFAULT_ETH_TX_MODE|DEFAULT_PARITY)
```

Change this entry to the following:

```
#define IX_CC_ETH_TX_DRIVER_MODE_DEFAULT
(DEFAULT_MODE_BLOCK|IX_CC_ETH_TX_DRIVER_MODE_GIGA_FULL_DUPLEX
|DEFAULT_PARITY)
```

All applicable modes are as follows:

Note: The first mode in the list below is for fiber, all others are copper modes for various link speeds.

```
#define IX_CC_ETH_TX_DRIVER_MODE_FIBER 0x0
#define IX_CC_ETH_TX_DRIVER_MODE_GIGA_HALF_DUPLEX 0x1
#define IX_CC_ETH_TX_DRIVER_MODE_GIGA_FULL_DUPLEX 0x2
#define IX_CC_ETH_TX_DRIVER_MODE_100_HALF_DUPLEX 0x3
#define IX_CC_ETH_TX_DRIVER_MODE_100_FULL_DUPLEX 0x4
#define IX_CC_ETH_TX_DRIVER_MODE_10_HALF_DUPLEX 0x5
#define IX_CC_ETH_TX_DRIVER_MODE_10_FULL_DUPLEX 0x6
```

Note: Copper ports are not supported by the Intel® IXDP2800 Advanced Development Platform. Only fiber ports are available. Editing the *ix_cc_eth_tx.h* as shown in this section will have no effect on Intel® IXDP2800 operability.

Debugging Applications on the Developer Workbench Simulator 5

This chapter contains an overview of certain application packet flow concepts and guides you in debugging the oc48_pos_ipv4_ingress application on the Developer Workbench simulator. As you step through the application code, you will see how the inter-microengine communication is implemented and how the packet information is exchanged from one microblock to the next.

5.1 Application Packet Flow Overview

The Intel® IXA Portability Framework uses certain types of structures which are unique to each packet and which specify the packet characteristics based on which microblocks make routing and processing decisions. Before debugging an application, it is important to understand the following packet flow concepts:

- packet metadata
- packet buffer
- dispatch loop variables

The *dl_system_ingress_default.h* file (found in **<install drive>:\IXA_SDK_3.5\src\library\microblocks_library\include**) contains the `# defines` for microblock IDs, packet metadata, and packet buffer.

5.1.1 Packet Metadata

Packet metadata is a set of variables which describe the characteristics of the packet, such as the buffer descriptors, packet length, header type, input port number, etc. By default, packet metadata is 8 long words in size and stored in SRAM. The packet metadata structure dl_meta_t is defined in dl_meta.h.

Some of the elements of packet metadata include:

- Amount of packet data in the buffer.
- DRAM offset where the packet begins. Each buffer in the DRAM is 2K or 2048 bytes long and the start of the packet is 128 bytes from the start of the buffer. Starting the packet at after 128 bytes comes in handy when a microblock has to prepend the header without moving the packet around in the DRAM. For example, the MPLS Marker microblock inserts an MPLS label before the IP header and adjusts the offset to 124 bytes.
- Packet length. If the packet length is more than 2K, the microblock learns that the packet is spread across a chain of buffers.
- Header type. Identifies whether it is a IPv4 packet or IPv6 packet.

5.1.2 Packet Buffer

This buffer contains the actual packet data received from the media interface. This is stored in DRAM and is 2K in size. If the total packet data is more than 2K in size, the microblock uses a chain of packet buffers. The packet buffer containing the SOP has a head room of 128 to 512 bytes. This allows room to prepend headers without having to move the packet within the DRAM.

5.1.3 Dispatch Loop Variables

Dispatch loop variables are exchanged from one microblock to another as the packet is passed from one microblock to the next. Dispatch loop variables include next block and buffer handles. Therefore, for each packet there will be a corresponding set of dispatch loop variables. After the current microblock has processed the packet, it sets the next block to the next microblock ID. The buffer handles uniquely identify where the packet metadata resides in the SRAM and where the actual packet resides in the DRAM. For instance, when the PacketRx microblock forwards the packet to the PPP-IPv4 microblock running on a different microengine, PacketRx sets the next block to BID_POS and writes the following dispatch loop variables to the scratch ring:

- Buffer handle containing start of packet (SOP). This is a 32-bit value where the lower 24 bits can be used to locate the packet metadata in the SRAM and the actual packet in the DRAM. The buffer handle structure buf_handle_t is defined in ixp_lib.h.
- Buffer handle containing end of packet (EOP). If the packet is longer than 2K (the default packet buffer size), this points to the location of the packet buffer which contains the end of the packet.

Similarly when the PPP-IPv4 microblock forwards the packet to the Queue Manager microblock, the IPv4 writes the buffer handles for start of packet, end of packet, and the port number. Depending on the functionality performed by the downstream microengine, the microblocks write the most relevant packet characteristics to the scratch ring. However, when the PPP microblock forwards the packet to IPv4Fwder, it sets the dl_next_block to BID_IPV4 and caches dispatch loop variables to local memory or GPRs.

5.2 Debugging oc48_pos_ipv4_ingress

This section explains how to do certain simple tasks using the Developer Workbench. For details on the full functionality provided by the Workbench, refer to the *IXP2400/IXP2800 Development Tools User's Guide*.

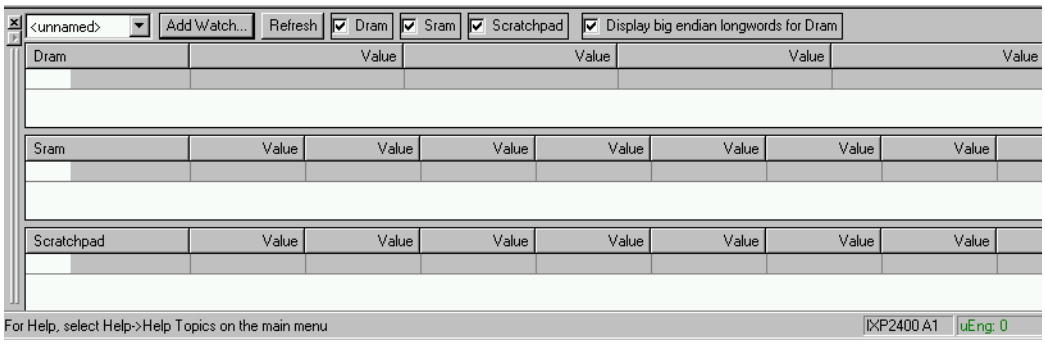
The exercises in this section will demonstrate:

- how packet information is exchanged from one microengine to another
- how packet characteristics and packet data is organized in SRAM and DRAM
- how the microblocks access and modify these characteristics and the packet header

Perform the following steps to debug the oc48_pos_ipv4_ingress application:

1. Launch the Developer Workbench and build the oc48_pos_ipv4_ingress application project.
2. Click the debug button. 
3. Click the Memory Watch button on the toolbar. 

The memory watch window shows the contents of scratchpad, SRAM, and DRAM.

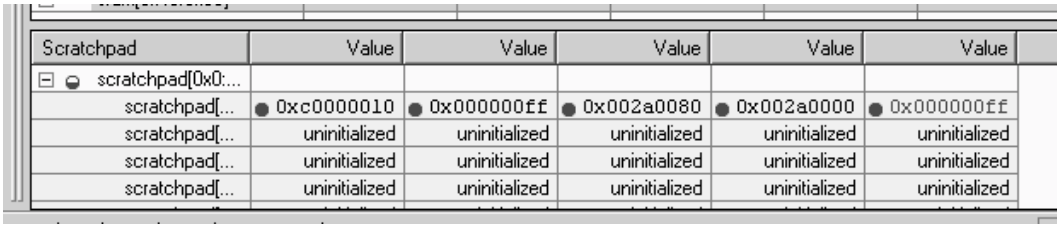


Set a breakpoint on change on scratchpad. This breakpoint will be hit when a microblock writes the dispatch loop variables to the next microblock. In this specific example, the breakpoint will be hit when the PacketRx microblock writes the dispatch loop variables to the scratch ring between PacketRx and PPP-IPv4. (Refer to DlSink in the *dl_source.c* file.)

4. Run the application by clicking the Go button on the toolbar. 

5. When the first break point is reached, open the file dl_source.c and go to **dl_sink( )**. The PacketRx microblock uses **dl_sink( )** to write 5 long words, (dlBufHandle, dlEopBufHandle, dram offset, etc) to the scratch ring.

6. Let the application run until the scratchpad memory gets initialized with the dispatch loop variables. The memory watch window will show 5 long words written to the scratchpad as shown below.



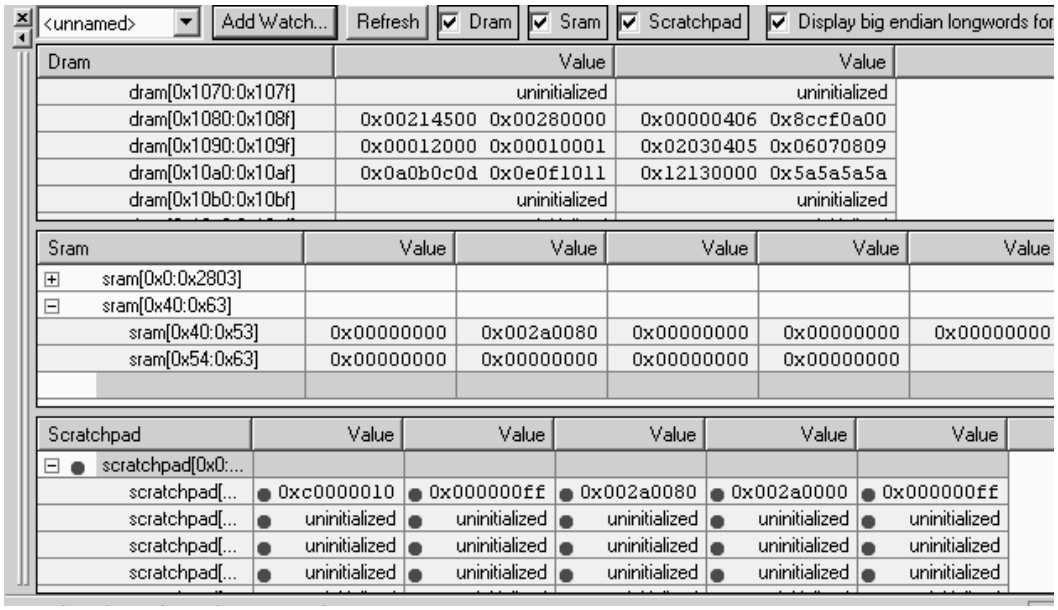
7. Using the dispatch loop variables, we can locate the packet metadata in SRAM and packet buffer in DRAM. The dlBufHandle variable tells that the packet buffer has both SOP and EOP, which implies that the packet length is less than 2K in size, therefore we can ignore the dlEopBufHandle. From the lower 24 bits we can derive the SRAM address for the packet metadata and DRAM address for the packet buffer as follows:
packet metadata address = (dlBufHandle.lw_offset << 2)
packet buffer address = (dlBufHandle.lw_offset << 8)

Note: In practice, the developer can use the library function Dl_BufGetDesc in dl_buf.c to obtain the packet metadata and the library function Dl_BufGetData in dl_buf.c to obtain the packet address.

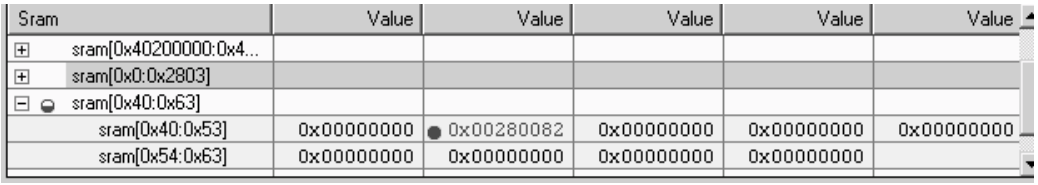
8. Given the dlBufHandle = 0xc0000010, the packet metadata resides at SRAM address 0x40. Add a SRAM watch point for address 0x40:+32. The second long word shows the packet size and buffer offset from where the packet data starts in the packet buffer.

9. Given the dlBufHandle = 0xc0000010, the packet buffer starts at 0x1000 and the actual packet data starts at 128 byte offset. Add a DRAM watch point for address 0x1080:+40. The contents show the PPP header, IP header and IP payload.

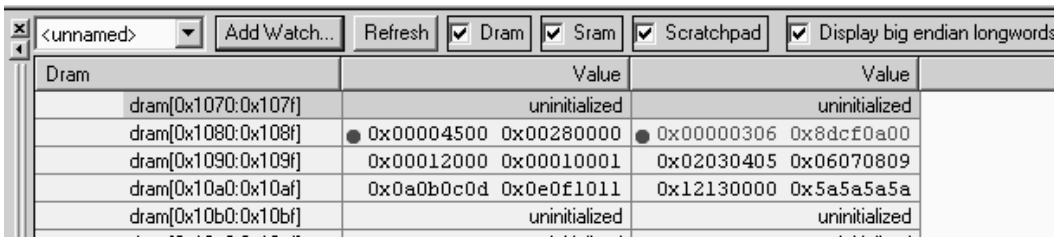
10. Set a break point on change on the SRAM address to see how the packet metadata gets updated.
11. Set a break point on change on the DRAM address to see how the PPP decapsulation and IPvFwder microblocks modify the IP header. The memory watch window will show DRAM, SRAM, and scratchpad as shown below.



12. When the break point on SRAM address is reached, the packet size and the buffer offset change. This implies that the PPP_Classify microblock has stripped off the PPP header and adjusted the packet length and buffer offset. (Note: See _ppp_decap in ppp.c). The ppp header still exists in the DRAM but from this point onwards, the IPv4Fwder microblock will operate as if the packet size is 40 bytes and packet starts at offset 130 instead of 128. Also, the PPP_Classify microblock updates the dlMeta.header_type in the packet metadata to PPP_IPV4_TYPE. (See _ppp_classify in ppp.c). The memory watch window will show SRAM as shown below.



13. When the break point on DRAM address is reached, it implies that the IPv4Fwder microblock has decremented the TTL and updated the checksum in the IP header. The memory watch window will show DRAM as shown below.



To gain a deeper perspective on the concepts and microblocks covered in this chapter, refer to the following documents:

- *Intel® IXA Portability Framework: Developer's Manual*: Dispatch Loop
- *Intel® IXA Portability Framework: Reference Manual*: Dispatch Loop

For details on the full functionality provided by the Developer Workbench, refer to the *IXP2400/ IXP2800 Development Tools User's Guide*.

Working With Core Components 6

Application development on the data plane consists of two kinds of processing:

- fast path processing running on the MEv2 microengines
- slow path processing running on the Intel XScale® core components

This chapter provides information about working with core components.

6.1 Handling of Exception Packets by Core Components

Intel XScale® core components are responsible for handling exception path packets. Some examples of exception packets are as follows:

1. In the case of an Ethernet pipeline, the Ethernet Rx core component handles ARP exception packets. For details, refer to the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manual*. When you send an ARP request packet from a packet generator to a Packet Rx microblock, then these packets are sent as exceptions to the Ethernet Rx core component. The Ethernet Rx core component sends it to the Ethernet Tx core component over the PCI bridge. The ARP reply will be generated by the Ethernet Tx core component and it should be received at the packet generator.
2. The IPV4 core component handles several types of exception packets. Some examples of exception packets handled by IPv4 core component are as follows:

   a. no route exception - the IPv4 core component generates an ICMP message. To generate this type of exception for VxWorks, you should set up the Ingress route table as follows:

   ```
   addNextHop "20 1 1 0 1500 0 20.0.0.2 0"
   addRoute "10.2.2.1 255.255.255.255 20"
   ```

   Note: If you are running the Ethernet pipeline, then make sure you set up the L2 table entry on the Egress as described in step 26 in Section 4.2.1, "IPv4 Forwarding Application on Hardware Using Core Components" on page 18.

   You can then send IPv4 packets with a destination IP address of 12.1.2.3 and source IP address of 10.2.2.1. You should be able to receive the ICMP packet at the packet generator

   b. packets requiring fragmentation- for VxWorks, setup a route table on Ingress as follows:

   ```
   addNextHop "20 1 1 0 1500 0 20.0.0.0.2 0"
   addRoute "32.0.0.1 255.255.255 20"
   ```

   Note: If you are running the Ethernet pipeline, then make sure you set up the L2 table entry on the Egress as described in step 26 in Section 4.2.1, "IPv4 Forwarding Application on Hardware Using Core Components" on page 18.

   You can then send an IPv4 packet of size 2996 bytes. You should be able to receive fragments back at the packet generator. Note that MTU setup in next hop database table is 1500. When 2996 bytes of packet is sent, it will undergo fragmentation in the IPv4 Forwarder core component.
3. Support for Ethernet MAC filtering of packets- build the Ingress workbench project after removing the DISABLE_MAC_FILTERING from the **Build->Settings** menu in Developer's Workbench. Build the Egress and Ingress Tornado* projects.

a. Download the code onto Ingress and Egress. Refer to Section 4.2.1, “IPv4 Forwarding Application on Hardware Using Core Components” on page 18 for an example procedure.

b. Start the system application on both Egress and Ingress.

c. Add the following MAC addresses in MAC filter table on Ingress shell:

```
addMac("0a:0b:0c:0d:0e:0f 0")
addMac("01:02:03:04:05:06 0")
addMac("01:07:06:04:05:06 0")
```

d. For Linux applications only, load the mac_config_util.o module by entering the following command :

```
insmod mac_config_util.o
```

e. For Linux applications only, on the ingress side, issue the following command:

```
mknod /dev/Mconfig c 250 0
```

f. To test that these MAC addresses are in the MAC filter table, you can call "ix_cc-eth_rx_dump_mac_filter_tbl" on the Ingress shell of a VxWorks system.

g. You can also try to lookup a particular port information for an entry by calling `lookupPort`, as follows:

```
lookupPort("01:02:03:04:05:06")
```

This should return a port value of 0.

h. Normally, all the ports are working in promiscuous mode. To get the ports working in unicast mode, remove the “DISABLE_MAC_FILTERINIG” defined from project GLOB_DEFINE (if you are working with VxWorks). If you are working with Linux, then remove it in Makefile.linux_kernel_common. Rebuild the project (VxWorks) or rebuild the application (Linux).

i. All ports are now set to unicast mode. When sending IPv4 packets through the platform with destination MAC address set to any of the three MAC addresses listed in step c, these packets would be forwarded. On the other hand, when sending any IPv4 packet with destination MAC address not in the table, these packets will be dropped.

Note: For Linux systems, you can send the same type of exception packets. The following examples show how to add routes and next hops in Linux:

```
rconfig addNextHop “20 1 1 0 1500 0 20.0.0.2 0”
rconfig addRoute “10.2.2.1 255.255.255.255 20”
```

6.2 Creating a Core Component

This section provides information about creating a new core component. The following files, all of which are located in **`opt/ixa_sdk_3.5/src/include`** can be modified to accommodate new core components:

- *ix_cc.h*- Any common definitions that will be shared between core components, should be added to this file. For example, the statistics information shared between the Ethernet Tx and Ethernet Rx core component is included in this file so that it can be accessed by either core component.
- *ix_cc_error.h*- This file defines the generic error codes for all core components. For example, IX_CC_ERROR_NULL.
- *ix_cc_macros.h*- This file contains definitions and macros that are shared across core components.

- *ix_cc_properties.h*- This file is for the interface property structure used by the core components when receiving property updates.
- *ix_cc_microengines_bindings.h*- This file contains all the commIds that are defined by the core components and needed by the microengines to send up the exception packets. Any microblock that sends packets to the packet Ids defined by the core components must include this file. This file is automatically included in the *bindings.h* file for core components. Each application has a specific bindings.h file that is located in the application's include directory (e.g. src/applications/ipv4_forwarder/4gb_ethernet/include).

Use the following procedures to create a new core component for your application (this example creates an Ethernet Rx example core component):

Note: Some of the procedural steps below are also explained in Section 6.4, "Adding a New Core Component" on page 47.

1. To create an Ethernet Rx core component, define an input ID for it in the *ix_cc_microengines_bindings.h* file as follows:

```
/* CommIds 64-74 are reserved for shared commIds between core and
micro-code*/
#define START_SHARED_COMM_IDS 64

/*IPv4 commIds are the same for high and low priority. One is added as
msg and the other as pkt hdlr*/
#define IX_CC_IPV4_PKT_MICROBLOCK_ID (START_SHARED_COMM_IDS + 1)

/* input id for packet handling */
/* Pos Rx communication ids  */
#define IX_CC_POS_RX_PKT_ID (IX_CC_IPV4_PKT_MICROBLOCK_ID + 1)

/* Ethernet Rx communications ids */
#define IX_CC_ETH_RX_PKT_ID (IX_CC_POS_RX_PKT_ID + 1)

/* Ethernet Tx communications ids */
#define IX_CC_ETH_TX_PKT_ID (IX_CC_ETH_RX_PKT_ID + 1)
```

2. The exception codes shared between a particular microblock and core component also has to be defined in *ix_cc_microengines_bindings.h*. For example, exception codes shared between the Ethernet Rx core component and microblock have to be defined as follows:

```
/* Exception codes for Ethernet Rx microblock and core component */
#define IX_ARP_PACKET 0x01
#define IX_NON_IP_PACKET 0x02
```

3. Define communication IDs for the core component in *bindings.h*. For the Ethernet Rx core component, it would be defined as follows:

```
enum  {
/* IPV4 communication ids */
IX_CC_IPV4_COMMON_ID = IX_CC_START_CORE_COMP_IDS,  /* input id for
common packet handling */
IX_CC_IPV4_PKT_STKDRV_ID, /* input id for stack driver packet handling
*/
/*Shared commIds moved to shared bindings file*/
/*IX_CC_IPV4_PKT_MICROBLOCK_HIGH_PRIORITY_ID,*/ /* input id for high
```

```
priority packet handling */
/*IX_CC_IPV4_PKT_MICROBLOCK_LOW_PRIORITY_ID,*/ /* input id for low
priority packet handling */
IX_CC_IPV4_MSG_ID, /* id for message handling */

/* Stack Driver communication ids */
IX_CC_STKDRV_COMMON_ID, /* communication input for common packet
handling */
IX_CC_STKDRV_LOW_PRIORITY_PKT_ID, /* communication input for low
priority packet handling */
IX_CC_STKDRV_HIGH_PRIORITY_PKT_ID, /* communication input for high
priority packet handling */
IX_CC_STKDRV_MSG_ID,   /* communication input for message handling */

/* Pos Rx communication ids  */
IX_CC_POS_RX_COMMON_ID,
IX_CC_POS_RX_MSG_ID,
/*Shared commIds moved to shared bindings file*/
/*IX_CC_POS_RX_PKT_ID,*/

/* FP Module communication ids  */
IX_CC_FP_MODULE_COMMON_ID,
IX_CC_FP_MODULE_MSG_ID,
IX_CC_FP_MODULE_PKT_ID,
IX_CC_FP_MODULE_EGRESS_ID,

/* Eth Rx communication ids  */
IX_CC_ETH_RX_COMMON_ID,
IX_CC_ETH_RX_MSG_ID,

/* AtmPosTx communication ids  */
IX_CC_ATM_POS_TX_COMMON_ID,
IX_CC_ATM_POS_TX_MSG_ID,
IX_CC_ATM_POS_TX_PROPERTY_MSG_ID,

/*  Ethernet Tx communication ids  */
IX_CC_ETH_TX_COMMON_ID,
IX_CC_ETH_TX_HIGH_PRIORITY_PKT_ID,
IX_CC_ETH_TX_MSG_ID,
IX_CC_ETH_TX_PROPERTY_MSG_ID,

/* Csix Rx communication ids  */
IX_CC_CSIX_RX_COMMON_ID,
IX_CC_CSIX_RX_MSG_ID,

/* CSIX Tx communication ids  */
IX_CC_CSIX_TX_COMMON_ID,
IX_CC_CSIX_TX_MSG_ID,

/* QM communication id */
IX_CC_QM_COMMON_ID,     /* communication input for common packet
handling */
IX_CC_QM_EGRESS_ID,     /* communication input for egress QM for
packets coming from ingress to egress */
IX_CC_QM_MSG_ID,        /* communication id for the message input */
```

```
IX_CC_QM_UBLOCK_ID,     /* communication id for the packet core to
ublocks input */

/* Message Support communication ids  */
IX_CC_MSUP_COMMON_ID,
IX_CC_MSUP_REPLY_EE_0_ID, /* reply input for EE 0 */
IX_CC_MSUP_REPLY_EE_1_ID, /* reply input for EE 1 */
IX_CC_MSUP_REPLY_EE_2_ID, /* reply input for EE 2 */
IX_CC_MSUP_REPLY_EE_3_ID, /* reply input for EE 3 */
IX_CC_MSUP_REPLY_EE_4_ID, /* reply input for EE 4 */

/* System App communication id */
IX_CC_SA_COMMON_ID,

/* WARNING - the values in this enumeration should no exceed
IX_COMM_FIRST_CORE_ID         + IX_COMM_LOCAL_ID_NUMBER */
IX_CC_COMMID_LAST
};
```

4. The input IDs and output IDs bindings have to be defined (as follows for the example Ethernet Rx core component):

```
/* START of Ethernet RX INPUT ID definitions */
#define IX_CC_ETH_RX_COMMON_INPUT
IX_RM_COMM_MAKE_LOCAL_ID(IX_CC_ETH_RX_COMMON_ID)
#define IX_CC_ETH_RX_MSG_INPUT
IX_RM_COMM_MAKE_LOCAL_ID(IX_CC_ETH_RX_MSG_ID)
#define IX_CC_ETH_MICROBLOCK_HIGH_PRIORITY_PKT_INPUT
IX_RM_COMM_MAKE_LOCAL_ID(IX_CC_ETH_RX_PKT_ID)
#define IX_CC_ETH_MICROBLOCK_LOW_PRIORITY_PKT_INPUT
IX_RM_COMM_MAKE_LOCAL_ID(IX_CC_ETH_RX_PKT_ID)
/* END of Ethernet RX INPUT ID definitions */

/*START of Ethernet Rx OUTPUT ID definitions */
#define IX_CC_ETH_RX_COMMON_PKT_OUTPUTIX_CC_PKT_DROP /*Non IP packets
are dropped right now*/
#define IX_CC_ETH_RX_ARP_PKT_OUTPUT
IX_RM_COMM_MAKE_ID(IX_CC_ETH_TX_ARP_PKT_INPUT, IX_PEER_SUBSYSTEM, 0)
/*END of Eth Rx OUTPUT ID definitions */
```

5. Define an identifier in the *bindings.h* file to be used by the system application:

```
#define IX_CC_ETH_RX            9
```

6. For clients interested in dynamic property updates, define them in *ix_cc_prop_clients.h*.
7. Add the core component that you are created to the list of core components started by System Application in a particular execution engine. Depending on whether the core component runs on ingress or egress processor, it has to be added to the *ix_sa_registry.xml* file.

   The Ethernet Rx core component has to be added to *IXA SDK 3.5/src/applications/ipv4_forwarder/4gb_ethernet/ingress/oc48_ethernet_ingress_config/ix_sa_registry.xml* as follows:

```
<property name="SA_EXEC_ENGINES">
<property name="SA_EE_01" type="uint32" value="1">
```

```
<property name="CC_LIST" type="string" value=~IX_CC_STKDRV IX_CC_IPV4
IX_CC_CSIX_TX IX_CC_QM IX_CC_SCHEDULER IX_CC_ETH_RX~/>
</property>
</property>
```

8. Create the infrastructure APIs required for the core components. For the Ethernet Rx core component, these are the required ones based on the functionality needed:

   a. `ix_error ix_cc_eth_rx_init(ix_cc_handle arg_hCc, void **arg_ppContext)`
   This function initializes the Ethernet Rx core component. This primitive will be called and returned successfully before requesting any service from the Ethernet Rx core component. This primitive should be called only once to initialize the Ethernet Rx core component. This function performs allocation of memory for symbols to be patched, creation of 64-bit counters, registration of packet and message handlers, and allocation and initialization of internal data structures

   b. `ix_error ix_cc_eth_rx_fini(ix_cc_handle arg_hCc, void *arg_pContext)`
   This function terminates the Ethernet Rx core component.

   c. `ix_error ix_cc_eth_rx_msg_handler(ix_buffer_handle arg_Msg, ix_uint32 arg_UserData, void *arg_pContext)`
   This function is the common message handler routine for Ethernet Rx core component. The Ethernet Rx core component receives messages from other core components through this message handler function and it internally calls appropriate library function to process the message. This message handler will be used to update dynamic properties.

   d. `ix_error ix_cc_eth_rx_low_priority_pkt_handler(ix_buffer_handle arg_hDataBuffer, ix_uint32 arg_ExceptionCode, void* arg_pContext)`
   This function is the low priority packet handler routine of the Ethernet Rx core component to handle non-ARP packets coming from microblock and other core components.

   e. `ix_error ix_cc_eth_rx_high_priority_pkt_handler(ix_buffer_handle arg_hDataBuffer, ix_uint32 arg_ExceptionCode, void* arg_pContext)`
   This function is the high priority packet handler routine of the Ethernet Rx core component for handling ARP packets.

9. Decide which APIs to expose to clients. The Ethernet Rx core component exposes the following messaging APIs:

```
a.ix_error ix_cc_eth_rx_async_get_statistics_info
  (ix_cc_eth_rx_statistics_info_context *arg_pMgInfoCtx,
  ix_cc_eth_rx_cb_get_statistics_info arg_Callback)

b.ix_error ix_cc_eth_rx_async_get_interface_state
  (ix_cc_eth_rx_if_state_context *arg_pStateContext,
  ix_cc_eth_rx_cb_get_interface_state arg_Callback)

c.ix_error ix_cc_eth_rx_async_add_mac_addr(ix_uint8 *arg_destMac,
  ix_uint32arg_portNum, ix_cc_eth_rx_cb_mac_addr_op arg_Callback, void
  *arg_pUserContext)

d.ix_error ix_cc_eth_rx_async_delete_mac_addr(ix_uint8 *arg_destMac,
  ix_cc_eth_rx_cb_mac_addr_op arg_Callback, void *arg_pUserContext)

e.ix_error ix_cc_eth_rx_async_lookup_port(ix_uint8 *arg_destMac,
  ix_cc_eth_rx_cb_lookup_port arg_Callback, void *arg_pUserContext)
```

10. Implement internal callback functions for these asynchronous function calls. For the Ethernet Rx core component, internal callback functions are as follows:

```
a.static ix_error _ix_cc_eth_rx_icb_get_statistics_info
  (ix_error arg_Result, void *arg_pContext, void *arg_Msg,
  ix_uint32 arg_MsgLen)
```

```
b.ix_error _ix_cc_eth_rx_icb_get_interface_state(ix_error arg_Result,
  void *arg_pContext, void *arg_Msg,ix_uint32 arg_MsgLen)

c.static ix_error _ix_cc_eth_rx_icb_general(ix_error arg_Result,
  void *arg_pContext, void *arg_pMsg, ix_uint32 arg_MsgLength)

d.static ix_error _ix_cc_eth_rx_icb_lookup_port
  (ix_error arg_Result, void *arg_pContext,void *arg_pMsg,
  ix_uint32 arg_MsgLength)
```

11. Implement library APIs to execute the operations requested in asynchronous calls. For Ethernet Rx core component, these are as follows:

```
a.ix_errorix_cc_eth_rx_get_statistics_info
  (ix_cc_eth_rx_statistics_info arg_Entity, ix_uint32 arg_Index,
  ix_cc_eth_rx_statistics_info_data *arg_pBuffer, void *arg_pContext)

b.ix_error ix_cc_eth_rx_get_interface_state(ix_uint32 arg_PortId,
  ix_cc_eth_rx_if_state *arg_pIfState, void *arg_pContext)

c.ix_error ix_cc_eth_rx_add_mac_addr(ix_uint8 *arg_pMacAddr, ix_uint32
  arg_PortId, void *arg_pContext)

d.ix_error ix_cc_eth_rx_del_mac_addr(ix_uint8 *arg_pMacAddr,
  void *arg_pContext)

e.ix_error ix_cc_eth_rx_lookup_port(ix_uint8 *arg_pMacAddr,
  ix_uint32 *arg_pPortNum, void *arg_pContext)
```

12. For dynamic property updates, you have to implement a set_property library API in each core component to get the property update from Master core component (stack driver) for properties this Core Component is interested in. For example, the Ethernet Rx Core Component is interested in the following interface status property:

```
ix_error ix_cc_eth_rx_set_property(ix_uint32 arg_PropId,
ix_cc_properties *arg_pProperty, void *arg_pContext)
```

13. To compile your core component with the build system, you will have to add your core component to the main project workspace. To add the Ethernet Rx core component to the main build system, follow the steps listed below:

    a. Launch Tornado 2.2 and open *ixa_sdk_2.2.wsp* located in *IXA_SDK_3.5\src\workspace*.

    b. Right click on *ixa_sdk.wsp* and then select **Add project**. Browse to **IXA_SDK_3.5\src\building_blocks\rx\core\ethernet_rx\ethernet_rx.wp***j*.

    c. Since the Ethernet Rx core component will be running on the ingress side of the Ethernet pipeline, you have to add it to *A_oc48_ethernet_ingress* project as SUB_PROJS. Select the **Build** tab.

    d. Right click and select macro as SUB_PROJS.
    Add **IXA_SDK_3.5\src\building_blocks\rx\core\ethernet_rx** as another SUB_PROJS.

    e. Right click on *A_oc48_ethernet_ingress* and select **Rebuild All**.

6.3 Porting Core Components from VxWorks to Linux

The layered framework of the Intel® IXA SDK allows you to easily port core components from one operating system to another. The Intel® IXA SDK framework has three layers:

- Operating System Services (OSSL)
- Resource Manager (RM)
- Core Component Infrastructure (CCI)

Core Components are built on top of Core Component Infrastructure layer. OSSL, RM and CCI hide core component from the operating system. Core Components do not use any system calls directly - even **malloc()** is invoked through the OSSL.

There is one part of the stack driver Core Component that is operating system dependent. It is the Virtual Interface Device Driver (VIDD) part of the Stack driver Core Component. The VIDD is the part that takes care of the communication with the local TCP/IP stack. This means that this part of the stack driver has to be written for each new operating system you want to run Core Components on.

The primary concern when porting other Core Components from VxWorks to Linux is that there is no concept of a kernel in VxWorks. In Linux, all the Core Components run as kernel modules.

6.3.1 Porting Guidelines

Keep the following points in mind when porting Core Components from VxWorks to Linux:

- build system for Linux Core Components - a Makefile must be created for building each Core Component as a Linux kernel module. VxWorks uses Tornado Project for the build system. In Linux, you must write the Makefiles for each of the components to build each component as a kernel module.
- user space to kernel space communication - The system application plays the role of the system designer and is responsible for launching the application. To launch the system application, there is a need for a user utility to make a call into the system application kernel module. This is also the case with the route configuration utility and l2 table configuration utility.
- loading the kernel modules - Here is a brief list of all the points to remember when loading the kernel modules:
  - There is no floating point support in the kernel
  - If you need any byte swapping macros to take care of endianness, you would need to implement these.
  - You cannot use **printf( )** or **fprintf( )** for kernel modules. Use **ix_ossl_message_log** instead.
  - Since **fprintf( )** can't be used in kernel mode, the **ix_error_dump( )** function in **src/library/xscale_utilities/source/error.c** has been ported to use **ix_ossl_message_log()**. As a result of this, 0 needs to be used for the first argument when calling **ix_error_dump( )**.
  - There is no support for mathematical functions like **atoi( )**. You will need to implement any mathematical functions.
- The Stack driver VIDD portion needs to be written from scratch to enable communication with Linux TCP/IP stack.
- Ethernet Tx core component needs to use new media driver running in kernel mode for the configuration of the media interface card. In Linux, usually a kernel mode driver can be used by applications running in user mode through the system calls to **open( )**, **close( )**, and **ioctl( )**. However since the Ethernet Tx Core Component is also running in kernel mode, direct function calls into the kernel driver are required. In the Intel® IXA SDK, this is achieved by writing a set of wrapper functions in the Ethernet Tx core component through the use of an internal driver API. This wrapper provides the set of API functions similar to the ones provided by VxWorks. These APIs are currently placed in *ix_cc_eth_tx_drv_wrapper.c*.

- To be OS independent, any component C file that earlier included *stdlib.h*, *string.h*, *errno.h* now has been changed to include *ix_ossl.h*.

6.4 Adding a New Core Component

When a new core component needs to be started by the system application, take the following steps:

1. In the application of your choosing - for example, oc48_pos_ingress- find at the end of the application-specific *bindings.h* (each application has a unique *bindings.h* file that is located in **\src\applications\<*application_name*>\include** directory) a list of CCIDs defined (like: `#define IX_CC_IPV4  0`).

   a. Add an CCID for the new core component before `IX_CC_ENUM_LAST`.

   b. Take the next number and make sure that `IX_CC_ENUM_LAST` has the value of your new CCID+1.

2. In the application of your choosing—for example, oc48_pos_ingress—open **IXA_SDK_3.5\src\applications\ipv4_forwarder\oc48_pos\ingress\oc48_pos_ingress_config\ix_sa_cc_list.c**.

   a. In the function `ix_sa_cc_list`, register the `init` and `fini` function for your new core component. The new function call should look like this:

   ```
   _ix_sa_set_cc(arg_pCC_List,
                 IX_CC_XXXX, your_init_func, your_fini_func);
   ```

   where *`IX_CC_XXXX`* is the new CC ID you added to `bindings.h`.

Note: You will need to use a `#include` directive to include the header that specifies your `init` and `fini` functions.

3. In the configuration of your choosing—for example, oc48_pos_ingress—open **IXA_SDK_3.5\src\applications\ipv4_forwarder\oc48_pos\ingress\oc48_pos_ingress_config\ix_sa_registry_data.xml**.

   a. Find the execution engine you want to add your CC to. For example,

   ```
   <property name="SA_EXEC_ENGINES">
     <property name="SA_EE_01" type="uint32" value="1">
     <property name="CC_LIST" type="string" value=~IX_CC_XXXX~/>
     </property>
   </property>
   ```

   b. Add your CC ID, as you did in *src\include\bindings.h,* to the CC_LIST property. If there are other core components specified in this execution engine, just separate the names specified with a space. Make sure the value is enclosed in tildes: *`~value~`*.

During startup, the system application will call `ix_sa_cc_list` function for the image configuration. Your newly added registration call will associate your `init` and `fini` functions with the CC ID. The system application will then create the execution engine which in turn will create your core component.

6.5 Adding Top Level Projects

The Intel® IXA SDK provides a number of top-level projects. Eventually, you will want to go beyond that set of projects:

- To create your own project
- To create an image configuration that the system application will use for your project

Take the following steps to accomplish these two tasks.

1. Create a new directory for your top-level project.
2. Inside the new directory, create another directory of the same name with `_config` attached. For example, if you were to have a new top level project directory named `test`, then you would create a subdirectory named `test_config`.
3. In the new config directory, create the directory tree *include\sa\internal*—for example, **test\test_config\include\sa\internal**.
4. Copy from the directory **IXA_SDK_3.5\src\applications\ipv4_forwarder\oc48_pos\ingress\oc48_pos_ingress_config** directory, the following files:
   - `ix_sa_cc_list.c`
   - `ix_sa_symbols.c`
   - `ix_sa_registry_data.xml`

   You will use these files as a starting point.
5. Modify the following code in `ix_sa_cc_list.c` and `ix_sa_symbols.c` to reflect your new image configuration name. For example, if your new top level project were named `test123`, then you would change the code in the following way:

   From:

   ```
   #if !defined(IX_CONFIGURATION_oc48_pos_ingress)
   #error IX_CONFIGURATION_oc48_pos_ingress not defined! Wrong
   configuration header.
   #endif
   ```

   To:

   ```
   #if !defined(IX_CONFIGURATION_test123)
   #error IX_CONFIGURATION_test123 not defined! Wrong configuration
   header.
   #endif
   ```

   Configure the rest of these files as needed for your image configuration.
6. Modify the `ix_sa_registry_data.xml` file for your image configuration.
   - Make sure to check the properties `Target_Config` and `INGRESS_EGRESS`.
   - If you do not have microcode, then remove the line:

     ```
     #include "../wbench_project/dispatch_loop/dl_system.h"
     ```
7. In Tornado*, create a new downloadable project.
   - Make sure the project name and directory name match.
   - Base this project on the `A_oc48_pos_ingress` project in **IXA_SDK_3.5\src\applications\ipv4_forwarder\oc48_pos\ingress**.

8. Select one of the builds for your new project, and open the properties window.
   a. Create a new rule named *projectname*.out where *projectname* is the exact same name as the current project.
   b. Select each of the other builds and set this new rule as the default.
9. Select the macros tab and, for each build, change the GLOG_DEFINES macro.
   — Change -DIX_CONFIGURATION_oc48_pos_ingress to -DIX_CONFIGURATION_*projectname* where *projectname* is the name of this project.
10. In Tornado, create a new downloadable project in the *projectname*_config directory.
    — Make sure the project name and directory name match.
    — Base this project on the oc48_pos_ingress_config project.
11. The new project will have files in it from oc48_pos_ingress_config. Remove ix_sa_cc_list.c and ix_sa_symbols.c and add these files from your *projectname*_config directory. Regenerate dependencies and save.
12. Select one of the builds from the config project, and open the properties dialog.
    a. Select the rule *sa\internal\internal_registry_data.h* and click **new/edit**.

       Tornado currently forces you to specify the path to the .xml file here (using the $(PRJ_DIR) variable does not work).
    b. Change the path to the .xml file to the correct path. For example, if the top level project were named test and the config project named test_config, then you would change the path to the xml file to opt/ixa_sdk_3.5\src\apps\test\test_config\ix_sa_registry_data.xml. Leave the rest of the command alone.
13. For each build, set *projectname*.out as the default rule. It will already exist from step 8.
14. Go back into the top level project.
    — For each of the builds, setup the SUB_PROJS macro to include the modules you need.
    — Most importantly, make sure SUB_PROJS does *not* include **IXA_SDK_3.5\src\applications\ipv4_forwarder\oc48_pos\ingress\oc48_pos_ingress_config**, and make sure SUB_PROJS *does* include the path to your new image config project—for example, **src\apps\test\test_config.**

6.6 Configuring the System Application

The system application configures the system and provides services. For general information about the system application, refer to the *Internet Exchange Architecture (IXA) Software Building Blocks Developer's Manua*l.

The system application code is broken in two parts: sysapp_common and a variety of image configurations. Each image configuration has a configuration project, sysapp_common uses the configuration project to specify all of the custom information for a given image. This sub-section shows what needs to be done with the configurations and how to add a core component under this arrangement.

6.6.1 Image Configurations

The system application configuration consists of three files for each image configuration. Each image configuration has a project called *image**image*_config, where *image* is the name of the configuration. For example, say image is oc48_pos_ingress, so the directory **IXA_SDK_3.5\src\applications\ipv4_forwarder\oc48_pos\ingress\oc48_pos_ingress** contains a directory *oc48_pos_ingress_config* with a configuration project oc48_pos_ingress_config.wpj.

The configuration files are as follows:

- ix_sa_registry_data.xml

  This file contains the properties that will be added to the system registry during system application startup. The properties in this file contain configuration information for the system application as well as all of the core components.

  The format is explained in Section 6.6.1.1, "Format of ix_sa_registry_data.xml" on page 50.

- ix_sa_symbols.c

  This file contains the function _ix_sa_patch_symbols which is invoked by the common system application code. This function patches any microcode symbols that the system application needs to patch for this image configuration.

- ix_sa_cc_list.c

  This file contains the function _ix_sa_cc_list which is invoked by the common system application code. Inside this function are registered the init and fini functions for each core component. Additionally, the global variable g_sa_uofFile is defined. This variable contains the full path to the uof microcode file for this image configuration.

The image configuration project performs the following tasks:

1. Builds ix_sa_symbols.c and ix_sa_cc_list.c.
2. Processes ix_sa_registry_data.xml.

   The processing of the xml file generates a header file called internal_registry_data.h. This file is located in the **image_config\include\sa\internal** directory and is included by the common system application code.

6.6.1.1 Format of ix_sa_registry_data.xml

There are three primary sections to the XML file: the header, the C include block, and the IxaSdkConfig block.

Note: Core components running on Linux do not use the *ix_sa_registry_data.xml* file to generate *internal_registry_data.h*. System and particular application properties are edited manually in the corresponding *internal_registry_data_linux.h* file.

Header

The first block in the file is the XML header which looks like this:

```
<?xml version="1.0"?>
<!DOCTYPE IxaSdkConfig [
  <!ELEMENT IxaSdkConfig (property*)>
  <!ELEMENT property (property*)>
  <!ATTLIST property name CDATA #REQUIRED type CDATA #IMPLIED value CDATA #IMPLIED>
```

```
]>
```

The header should never be modified. When interpreted by various XML editors, they can indicate syntax errors while editing this file.

C Include Block

Before processing the XML file, we run it through the C preprocessor. This allows us to use various shared `#define` directives instead of using hard coded values. This section contains an XML comment tag which looks like this:

```
<!-- START HEADERS
    #include "bindings.h"
        ...
 END HEADERS-->
```

All C preprocessor directives *must* be specified within this comment block.

Due to the fact that the C preprocessor ignores text in quotes, and the XML specification requires values to be specified in quotes, we use a tilde instead of quotes— ~ instead of " —when surrounding a value that contains a preprocessor macro or constant. For example, consider the following XML:

```
<property name="ID" type="uint32" value=~QM_TO_CSIX_TX_SCR_RING~/>
```

In this example, the preprocessor will replace QM_TO_CSIX_TX_SCR_RING with its defined value. We then replace the tildes with quotes and run the file through the XML parser. If we were to enclose the above value in quotes instead, the preprocessor will not replace the symbol with its value and the quoted text would become the value.

Note: This is limited to preprocessor constants. C expressions will not work. For example:

```
#define MY_VALUE 1+1
```

The above defined `MY_VALUE` would expand to the text `1+1` in the XML file. The math will not be performed. There is no C compiler used here.

IxaSdkConfig Block

This XML file uses two different tags. The first, `IxaSdkConfig`, opens the block containing properties to load into the registry at startup. The second, `property`, defines one of these properties.

The `property` tag takes one required option, `name`, and an optional pair of options, `type` and `value`. `type` and `value` must be specified together and can only be included or omitted as a pair.

The system registry defines a property as an object—that is, it can have a value, it can be a container to other properties, or both. If the pair `type` and `value` are omitted, the property simply becomes a container for other properties but has no value of its own. If the pair `type` and `value` are specified, the property is assigned a value but can still be used to contain other properties. The outcome of this arrangement is a tree of properties.

A tag and its options are opened and closed as follows.

```
<IxaSdkConfig>
    ....
</IxaSdkConfig>
```

Notice the forward slash— / —when the tag is closed, just like HTML.

Properties are specified in the same way. If other properties are defined inside of a property, they become its children in the registry tree. For example, consider the following definition:

```
<property name="Prop1">
    <property name="Prop2">
    </property>
</property>
```

In the above example, `Prop1` will be created and `Prop2` will be created as a child of `Prop1`.

Often there will be properties which do not have any children. These can be closed more easily by adding a forward slash— / —at the end of the tag. For example:

```
<property name="Prop1">
    <property name="Prop2"/>
</property>
```

Notice how `Prop2` is closed with the trailing forward slash and so does not require a separate `</property>` closing tag.

Note: Property tags are only valid inside the `IxaSdkConfig` tag.

There are two types of values that can be assigned to a property: `uint32` and `string`. These directives simply tell the system application what type of property to create. For example,

```
<IxaSdkConfig>
   <property name="MyAppConfig">
      <property name="IPADDR" type="string" value="10.0.0.1"/>
      <property name="MTU" type="uint32" value="1500"/>
   </property>
</IxaSdkConfig>
```

6.6.2 Properties Used by the System Application

The System Application properties are all contained as children of the `/SystemApp` property. Figure 1, “System Application Property Tree” on page 53 shows the System Application property tree where the forward slash— / —refers to properties as if they were a directory tree. Table 1 describes each of the properties in Figure 1.

Figure 1. System Application Property Tree

```
/SystemApp/
    FREELISTS/
        FL_MSG/
            ELEMENT_COUNT
            META_SIZE
            DATA_SIZE
        FL_01/
            ELEMENT_COUNT
            SRAM_SIZE
            SRAM_CHAN
            DRAM_SIZE
            DRAM_CHAN
    SCRATCH_RINGS/
        SR_01/
            ID
            CHAN
            ELEMENT_SIZE
        SR_02/  ...
        ... (*any number of scratch rings)
    SA_EXEC_ENGINES/
        SA_EE_01/
            CC_LIST
        SA_EE_02/ ...
        ... (*any number of execution engines)
    MICROENGINES/
        ME_01/
            CONTEXT_MASK
        ME_02/  ...
        ... (*an entry for each ME to start)
```

Table 1. System Application Property Descriptions

Property	Description
FREELISTS	Contains the subproperties that define the message and packet freelists for the system. The system application will create these freelists at startup.
FL_MSG	Contains the properties for the systems message freelist
ELEMENT_COUNT	Number of buffers to put on the msg freelist
META_SIZE	Size of the metadata for the buffers in the message freelist
DATA_SIZE	Size of the data for the buffers in the message freelist
FL_01	Contains the properties for the systems packet freelist
ELEMENT_COUNT	Number of buffers to put on the packet freelist
SRAM_SIZE	Size of buffer SRAM area
SRAM_CHAN	SRAM Channel

Table 1. System Application Property Descriptions

Property	Description
DRAM_SIZE	Size of buffer DRAM area
DRAM_CHAN	DRAM Channel
SCRATCH_RINGS	Contains scratch ring definitions The system application will create these scratch rings at startup.
SR_XX	Contains config properties for a given scratch ring XX is a two digit number. Any number of scratch rings can be specified where XX is incremented starting with 01: SR_01, SR_02 and so on.
ID	Id of the scratch ring This should match microcode.
CHAN	Scratch channel.
ELEMENT_SIZE	Size of the scratch rings elements 0 = 128 bytes 1 = 256 bytes 2 = 512 bytes 3 = 1024 bytes
SA_EXEC_ENGINES	Contains execution engine definitions Each definition is created by the system application during startup. When started, these execution engines proceed to create the core components specified in their CC_LIST property.
SA_EE_XX	Defines an execution engine XX is the execution engine ID starting with 01. This property must also have a uint32 value that is equal to XX.
CC_LIST	String containing space-separated integers Each integer is a CC ID to start. The system application creates a CC using the init and fini functions registered for that CC ID in ix_sa_cc_list.c.
MICROENGINES	Contains properties that specify which microengines (MEs) to start and the context to start them.
ME_XX	Specifies an ME to start XX is the index of the ME to start: 00 to 07 for the Intel® IXP2400, 00-15 for the Intel® IXP2800. This property must also have a uint value that contains the same ME number as XX.
CONTEXT_MASK	The context mask to pass to ix_rm_ueng_start when creating this ME.

Adding Microblocks to an Application 7

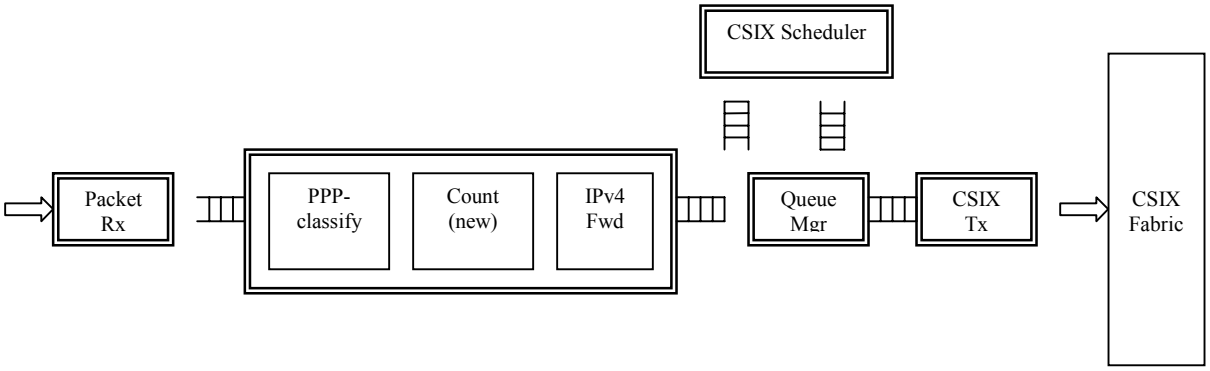
This chapter describes the steps involved to modify the existing oc48_pos_ipv4_ingress application by creating a new microblock, modifying the application's dispatch loop, rebuilding the project with new source files, and running the debugger. The information provided in this chapter assumes you are working with a Windows 2000* development host.

7.1 Changing the Application

This chapter provides procedures for modifying the existing oc48_pos_ipv4_ingress application and add a new microblock between the PPP_classify and IPV4Fwder microblocks. For the sake of simplicity, we will add a microblock which receives packets from PPP_Classify, increments a counter, and forwards the packet to IPv4Fwder. The new microblock is called Count and the new application is illustrated in Figure 7-1.

Note: In a real life application, the developer may want to make a more complex modification, such as adding an MPLS Marker microblock to the oc48_pos_ipv4_ingress application.

Figure 7-1. Block Diagram: Modified oc48_pos_ipv4_ingress Application



7.2 Creating a New Application

Use an existing application as a baseline for the new application using the following procedure:

1. Create a folder called wbench_c_practice in the directory **\IXA_SDK_3.5\src\applications\ipv4_forwarder\oc48_pos\ingress**
2. Copy all the folders from **\IXA_SDK_3.5\src\applications\ipv4_forwarder\oc48_pos\ingress\wbench_c_project** to the new folder.
3. Now open the project file oc48_pos_ipv4_ingress.dwp in the new folder.
4. The Workbench will prompt that a rebuild is required. Do a rebuild. It should compile with no errors and warnings.

7.3 Modifying Source Files

Use the following procedure to create source files for the new microblock and to modify existing application files to use the new microblock:

1. Under the folder **\IXA_SDK_3.5\src\building_blocks**, create a folder called count. Within the count directory, create subdirectories called microc and include. All *.c source files will be placed in microc and *.h files in the include subdirectory.
2. Create the **count.c** file under **\IXA_SDK_3.5\src\building_blocks\count\microc** as shown below:

```
/************************
* File: count.c
************************/

INLINE void count (void)
{

    // standard check to see if this packet is for us
    if (dlNextBlock != BID_COUNT)
        return ;

    sram_incr((volatile void __declspec(sram) *)(COUNT_SRAM_ADDR));

    // send to next block
    dlNextBlock = BID_IPV4;
    return;
}
```

3. The dl_system_ingress_default.h file includes the microblock IDs for PPP_Classify, IPv4Fwder, and others. We want to add a new ID for count. We see that the last microblock ID is BID_MPLSILM, which is defined as 0x29. In the dl_system.h we will add for the count microblock.

```
/* PracticeExcersize_begin */
#define BID_COUNT                   0x2A
/* PracticeExcersize_END */
```

4. The dl_system_ingress_default.h file includes the SRAM addresses for the various counters maintained by the microblocks. For the count microblock, specify the SRAM address where the microblock will increment the packet count. We see that CSIX_TX_COUNTERS_SRAM_BASE is set to 0x40300000. We will use the 0x40300200 for the Count microblock. Define the following in dl_system.h.

```
/* PracticeExcercise_begin */
#define COUNT_SRAM_ADDR             0x40300200
/* PracticeExcersize_end */
```

5. In the original project, the PPP_Classify sent packets to IPv4Fwd. Now it should send to the Count microblock. Modify the ppp.c file as shown below:

```
INLINE void _ppp_classify(void *p_pkt,UINT in_offset)
{
    ...

    if (type == PPP_IPV4)
```

```
    {
        dlMeta.headerType = PPP_IPV4_TYPE;

//      dlNextBlock = BID_IPV4;
/* PracticeExcersize_begin */
        dlNextBlock = BID_COUNT;
/* PracticeExcersize_begin */

    }
...
return;

}
```

Note: Remember to undo this modification after you complete the procedures in this chapter.

6. Modify the dispatch loop file pos_ipv4.c file to call **count( )** before calling IPv4Fwder as shown below:

```
/*-------------------------------------------------------------------------
 * The main function of the dispatch loop
 *-------------------------------------------------------------------------
 */
int main()
{

    /*
     * initialize the microblocks
     */

    dl_init();        //initialize the dispatch loop

--------
     */
    while(1)
    {

        SIGNAL     scratch_put;// signal in scratch write
        SIGNAL_MASKsig_mask = 0x0;// mask of signals to wait on

        ppp_decap_classify(pkt_hdr,0x0);
/* PracticeExcersize_begin */
        count();
/* PracticeExcersize_end */
        Ipv4Fwder(pkt_hdr,IP_HDR_OFFSET,pkt_hdr,IP_HDR_OFFSET);

        ------

    } // end while

    return 0;
}
```

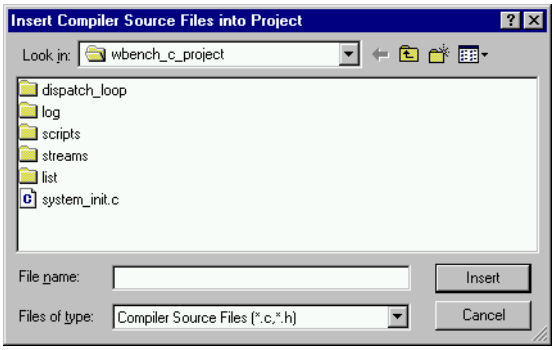
7. Include the **count.c** file in pos_ipv4.c.

```
#include "dl_source.c"
#include "ppp.c"
#include "ipv4_fwder.c"
/* PracticeExcersize_begin */
#include "count.c"
/* PracticeExcersize_end */
```

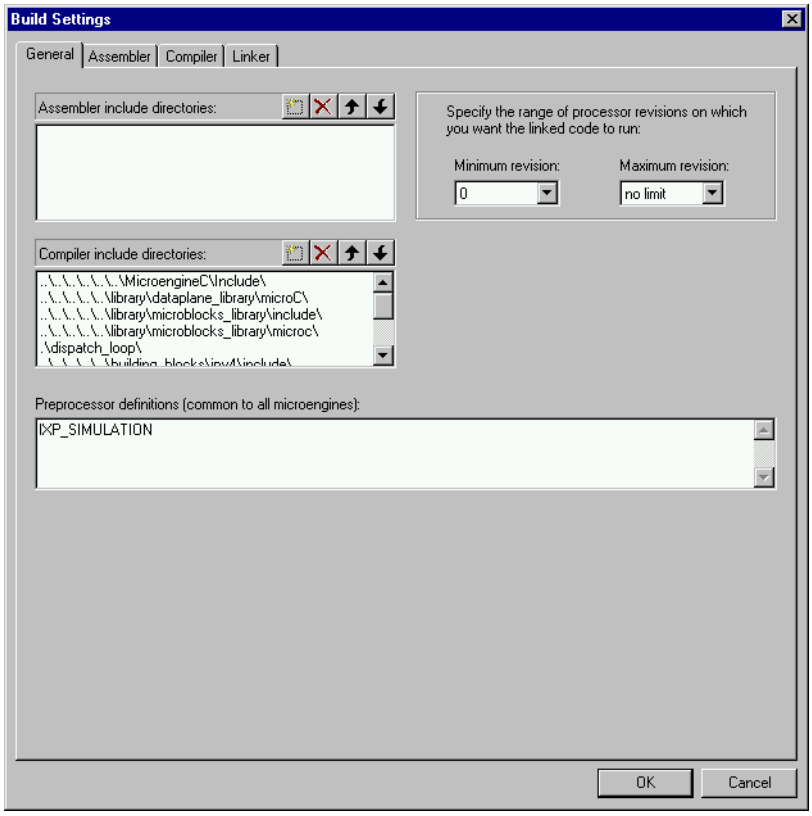
7.4 Building a New Project

Perform the following steps to build the new application and run the debugger to verify your work:

1. Add the c source file to the project by clicking **Project > Insert Compiler Source Files...** on the Developer Workbench Menu toolbar.



2. Add the include patch for **count.c** in the **Build > Settings...** dialog's **General** tab.



3. Modify the system_setup.ind file by adding the following statement just before ps_start_packet_receive()
   init_sram(0x0, 0x40300200, 0x40300203);
   This initializes the SRAM address for the count microblock.
4. Build the project. The project should compile with no errors and warnings.
5. Start debugging. Set a watch point at SRAM address 0x40300200 to see the count being incremented.

Using Resource Manager for Linux 8

This section provides information about how to build IXP2XXX product line Linux kernel libraries and applications on a Red Hat 7.3 Linux host and how to run applications on IXP2XXX product line embedded Linux OS. The Resource Manager (RM), Operating System Service Layer (OSSL) and all applications are implemented as loadable modules for the Linux kernel. The libraries as RM and OSSL are just modules that provide clients with a set of APIs to call. During `init_module( )` and `cleanup_module( )` they do not perform any action. It is the responsibility of the calling application to perform any required initialization.

For complete information about the Resource Manager and Operating System Service Layer, refer to *Intel® Internet Exchange Architecture (IXA) Portability Framework Developer's Manual.*

8.1 Building the Libraries

Separate makefiles have been created for all libraries and applications. They are uniformly named "Makefile.linux_kernel". The structure of these makefiles is similar between libraries (RM, OSSL) and applications.

The following variables are relevant to all makefiles:

- IXA_SDK_DEV - should be defined as an environment variable and should point to the top of the development source tree. This must be set to **opt/ixa_sdk_3.5/IXA_SDK_3.5**.
- IXP2XXX_TOOLCHAIN_ROOT - should be defined as an environment variable and should point to the directory that contains the tool chain binaries.

  Note: The remainder of this section assumes that the IXP2XXX_TOOLCHAIN_ROOT variable points to **/opt/hardhat/devkit/arm/xscale_be.**
- IXP2400_KERNEL_SOURCE_ROOT - should be defined as an environment variable and should point to the directory of the target kernel source for the IXP2400 LSP.

  Note: The remainder of this section assumes that the IXP2400_KERNEL_SOURCE_ROOT variable points to **/opt/hardhat/devkit/lsp/intel-ixdp2400-arm_xscale_be/linux-2.4.18_mv130**.
- IXP2800_KERNEL_SOURCE_ROOT - should be defined as an environment variable and should point to the directory of the target kernel source for the IXP2800 LSP.

  Note: The remainder of this section assumes that the IXP2800_KERNEL_SOURCE_ROOT variable points to **/opt/hardhat/devkit/lsp/intel-ixdp2800-arm_xscale_be/linux-2.4.18_mv130**.
- CONFIG - this can have two values: NATIVE or XSCALE_BE. It defines the build configuration type. The NATIVE build type is for the host machine, and is intended for debugging purposes. The XSCALE_BE build type is intended for the target hardware. For the current release, the default setting is XSCALE_BE.
- IXOS - defines the target OS and should always be set to LINUX.
- BUILD_TYPE - can be set to RELEASE or DEBUG, based on which type of build is desired. DEBUG will provide the RM, CCI and the applications with verbose, debug output information. For the current release, the default setting is DEBUG.

- BUILD_MODE - can set to HARDWARE or SIMULATION. HARDWARE is the default value. However, if the CONFIG variable is set to NATIVE then this value is changed to SIMULATION.
- HARDWARE_TYPE - can be set to IXP2400 or IXP2800. It defines the target hardware for which the build is intended. The default value is IXP2400.
- KERNEL_DIR - represents the base directory where the source for the target kernel resides. It is set to **/usr/src/linux-2.4** when the CONFIG variable is set to NATIVE, and will be set to the value of **opt/ai_target/linux** when the CONFIG variable is set to XSCALE_BE. While the first value will rarely change, the second is based on the base directory for the target kernel installation on the host.
- LINUX_ARCH_DIR - defines the kernel architecture and is currently set to "i386" for NATIVE and "arm" for XSCALE_BE. In most cases, this value can remain at the default.
- DSTDIR_BASE - represents the base distribution directory. Its default setting is to **/opt/xscale_be_test**. This is the base directory where the built modules will be installed.
- SOURCE - represents the list of .c files to be included in the build.
- TARGET - specifies the name of the target module.

The correct command to build for the target is as follows:

```
make -f Makefile.linux_kernel install
```

The order for building libraries and modules must be done as follows:

6. `rm_lib.o` in *$IXA_SDK_DEV/src/framework/rm*
7. `cci_lib.o` in *$IXA_SDK_DEV/src/framework/cci*

In the **$(IXA_SDK_DEV)/src/framework** directory the `make -f Makefile.linux_kernel install` command would build all the above modules.

8.2 Running the Resource Manager

In order to run any kernel applications, given that the directories and variables are set as shown in Section 8.1, “Building the Libraries,” on page 61 and your target/host configuration is setup according to the information contained in the *Intel® Internet Exchange Architecture Software Development Kit Tools Installation Guide*, follow these steps:

1. On the host system:
   a. Edit the file **/etc/exports** and add a new exported entry **/opt/xscale_be_test/linux_kernel/xscale_be**
   b. Run `/usr/sbin/exportfs -a`
   c. Run `/etc/rc.d/init.d/nfs stop`
   d. Run `/etc/rc.d/init.d/nfs start`
   That is to export the **/opt/xscale_be_test/linux_kernel/xscale_be** directory and make it visible to the target system.
2. On the target system:
   a. Boot Linux*.

b. Run `mount <host_ip>(host name):/opt /mnt` (e.g. `mount 10.3.31.224: /opt/xscale_be_test/linux kernel/xscale_be /mnt`)

c. Run `cd /mnt/ixp2400/debug`

d. Copy the *start.sh* and *stop.sh* scripts from **$(IXA_SDK_DEV)/test/unit/framework/linux_kernel_scripts** into the above directory

e. Copy **$(IXA_SDK_DEV)/test/unit/framework/rm/uof to /mnt/uof** (actually **/opt/xscale_be_test/linux_kernel/xscale_be/uof**) directory.

f. Run `./start.sh`

g. Run `insmod test_app_mod.o` (e.g. `insmod system_api_test_mod.o`)

h. Run `rmmod test_app_mod` (e.g. `rmmod system_api_test_mod.o`)

i. Run `./stop.sh`

***Note*:** For some applications the following command is required `insmod cci_lib.o`.

Routing Table and L2 Table 9

This chapter provides information about the Routing table and the L2 table.

9.1 Routing Table

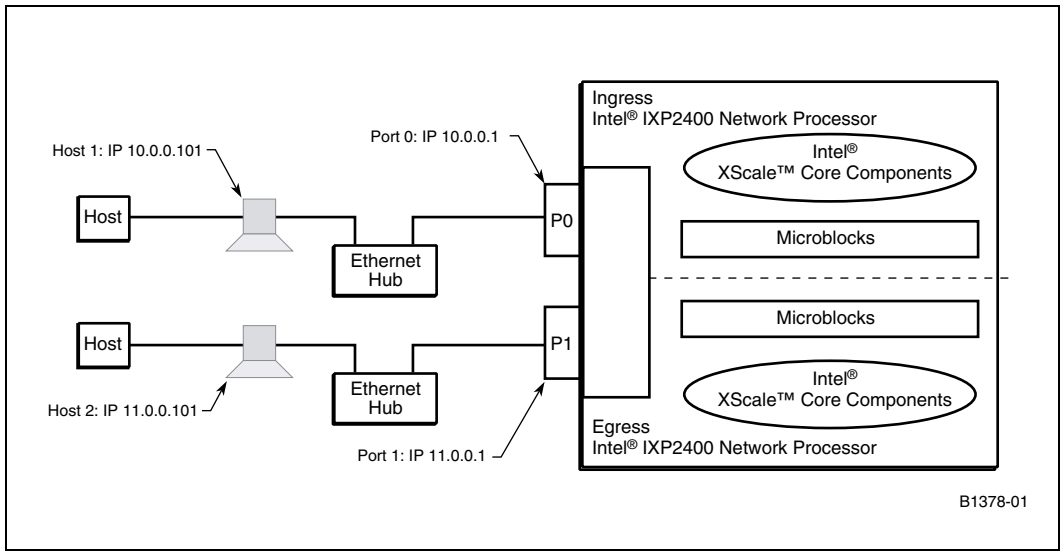
On the Ingress shell for VxWorks, type `route_config`. If you're using Linux, type `rconfig` on the Ingress shell. This shows all the supported Routing Table commands with usage syntax and examples. Some pertinent commands are:

- `addNextHop` explains the command to add next hop to the system for IPv4.
- `addRoute` explains the command to add a route to the system for IPv4.
- `addNextHopV6` shows how to use the command to add a next hop for IPv6.
- `addRouteV6` shows how to use the command to add a route for IPv6.

9.1.1 Populating the Routing Table for IPv4 Ping Tests

This section provides information about populating the route table to conduct IPv4 ping tests between the external hosts and the development platform (Intel® IXDP2400 Advanced Development Platform). The procedural information in this section assumes that two hosts (Host 1 and Host 2) are connected to the development platform's Ethernet data ports via an Ethernet hub, as shown in Figure 1:

Figure 1. Example System Setup for IPv4 Ping Tests



If you want to ping Port 0 from Host 1, follow these steps:

1. On the Ingress shell, add the following routes:
   ```
   addNextHop "0xb 2 25 0 1500 1 10.0.0.1 0"
   addRoute "10.0.0.1 255.255.255.255 0xb"
   addNextHop "0xc 2 25 0 1500 0 10.0.0.101 0"
   addRoute "10.0.0.101 255.255.255.255 0xc"
   ```
2. On the Egress shell, add the following L2 table entry:
   ```
   addV4EthEntry "25 10.0.0.101 00:03:47:bd:f9:dc 0a:0b:0c:0d:0e:0f
   DEFAULT"
   ```
   where `00:03:47:bd:f9:dc` is the MAC address of Host 1 and `0a:0b:0c:0d:0e:0f` is the default MAC address for Port 0 of the platform. Usually the MAC address of a host running Windows 2000* can be found by typing `route print` from the command window.
3. On host 1, add the following static ARP entry for Port 0 of the platform:
   ```
   C:\>arp -s 10.0.0.1 0a-0b-0c-0d-0e-0f 10.0.0.101
   ```
4. Type the ping command from Host 1as follows and you should get ping replies:
   ```
   C:\>ping 10.0.0.1
   ```

Note: Instead of using static ARP entries, it is possible to discover MAC addresses of both sides by using ARP. For this method, after you have done step 1 above, use the following command for step 2:

```
addV4L3Info "25 10.0.0.101 DEFAULT"
```

You can then skip step 3 and perform step 4 to get the ping replies.

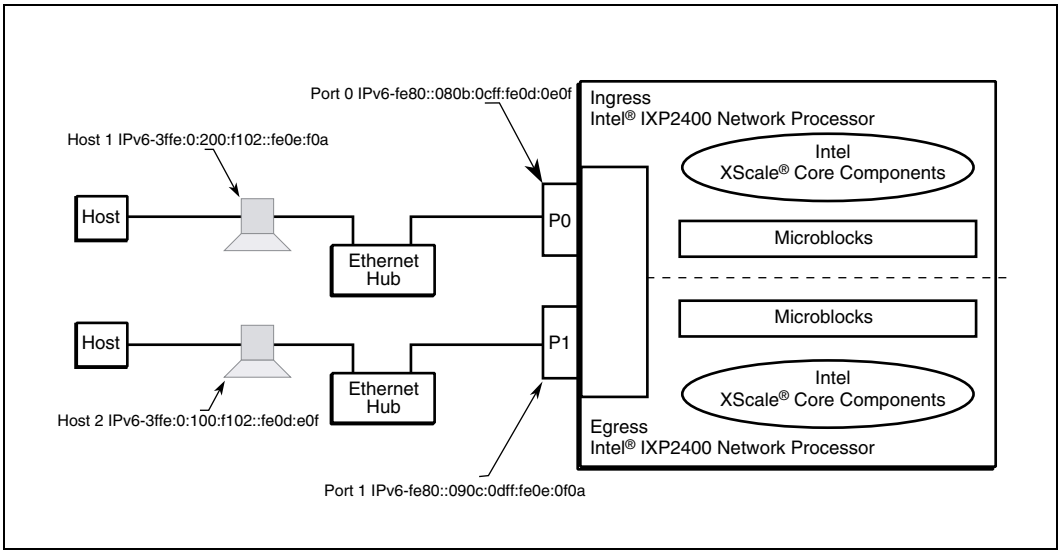
If you want to ping Host 2 from Host 1 or vice versa, conduct steps 1-4 above for adding routes for Host 1 and perform the following additional steps for adding routes to Host 2:

5. On the Ingress shell, add the following routes:
   ```
   addNextHop "0xd 2 26 1 1500 1 11.0.0.1 0"
   addRoute "11.0.0.1 255.255.255.255 0xd"
   addNextHop "0xe 2 26 1 1500 0 11.0.0.101 0"
   addRoute "11.0.0.101 255.255.255.255 0xe"
   ```
6. On the Egress shell, add the following L2 table entry:
   ```
   addV4EthEntry "26 11.0.0.1 00:02:b3:1c:f7:97 0b:0c:0d:0e:0f:0a
   DEFAULT"
   ```
   where 00:02:b3:1c:f7:97 is the MAC address of host 2 and 0b:0c:0d:0e:0f:0a is the default MAC address for Port 1 of the platform. Usually the MAC address of a host running Windows 2000* can be found by typing `route print` from the command window.
7. On host 2, add the following static ARP entry for Port 1of the platform:
   ```
   C:\>arp -s 11.0.0.1 0b-0c-0d-0e-0f-0a 11.0.0.101
   ```
8. Type the following command from Host 1:
   ```
   C:\>route add 11.0.0.101 mask 255.255.255.255 10.0.0.1
   ```
9. Type the following command from Host 2:
   ```
   C:\>route add 10.0.0.101 mask 255.255.255.255 11.0.0.1
   ```
10. Type the ping command from Host 1 as follows and you should get the ping replies from Host 2:
    ```
    C:\>ping 11.0.0.101
    ```
11. Type the ping command from Host 2as follows and you should get the ping replies from Host 1:
    ```
    C:\>ping 10.0.0.101
    ```

9.1.2 Populating the Routing Table for IPv6 Ping Tests

This section provides information about populating the route table to conduct IPv6 ping tests between the external hosts (either Windows 2000* or Windows XP* with IPv6 patch installed) and target development platform (Intel® IXDP2400 Advanced Development Platform with Monta Vista Linux*). The procedural information in this section assumes that two hosts (Host 1 and Host 2) are connected to the development platform's Ethernet data ports via an Ethernet hub, as shown in Figure 2:

Figure 2. Example System Setup for IPv6 Ping Tests



The ingress application assigns the link local address to the Intel® IXDP2400 Advanced Development Platform according to RFC 2464 and RFC 2461. The link local IPv6 address and the MAC address for each port is shown in Table 2:

Table 2. IPv6 Addresses and MAC Addresses for IXP2400 Ports

Port Number	IPv6 Address	MAC Address
0	fe80::080b:0cff:fe0d:0e0f	0a:0b:0c:0d:0e:0f
1	fe80::090c:0dff:fe0e:0f0a	0b:0c:0d:0e:0f:0a
2	fe80::0e0d:0eff:fe0f:0a0b	0c:0d:0e:0f:0a:0b
3	fe80::0f0e:0fff:fe0a:0b0c	0d:0e:0f:0a:0b:0c

If you want to ping Host 1 from Host 2, follow these steps:

1. At the ingress prompt, add the following routes:
   ./rconfig addNextHopV6 "3 1 1 0 1500 0 0"
   ./rconfig addRouteV6 "3ffe:0:200:f102::fe0e:0f0a 64 3"

./rconfig addNextHopV6 "6 1 2 1 1500 0 0"
./rconfig addRouteV6 "3ffe:0:100:f102::fe0d:0e0f64 6

2. At the egress prompt, add the following L2 table entry:

   ./l2config addV6EthEntry "1 3ffe:0:200:f102::fe0e:0f0a 00:07:e9:ad:55:5b 0a:0b:0c:0d:0e:0f DEFAULT"

   where 00:0f:e9:ad:55:5b is the MAC address of Host 1 and 0a:0b:0c:0d:0e:0f is the default MAC address for Port 0 of the Advanced Development Platform.

   ./l2config addV6EthEntry "2 3ffe:0:100:f102::fe0d:0e0f 00:07:e9:ad:58:12 0b:0c:0d:0e:0f:0a DEFAULT"

   where 00:07:e9:ad:58:12 is the MAC address of Host 2 and 0b:0c:0d:0e:0f:0a is the default MAC address for Port 1 of the Advanced Development Platform.

3. Add the following route at Host 1:

   C:\>ipv6 rtu 3ffe:0:100:f102::/64 4/fe80::080b:0cff:fe0d:0e0f

4. Add the following route to Host 2:

   C:\> ipv6 rtu 3ffe:0:200:f102::/64 4/fe80::090c:0dff:fe0e:0f0a

5. Enter the following ping command from Host 1:

   C:\> ping6 3ffe:0:100:f102::fe0d:0e0f

   You will get ping replies from Host 2.

6. Enter the following ping command from Host 2:

   C:\> ping6 3ffe:0:200:f102::fe0e:0f0a

   You will get ping replies from Host 1.

9.2 L2 Table

On the Egress shell, type `l2_config`. `l2_config` shows all the supported commands with usage syntax and examples. Some pertinent commands are as follows:

- `addV4L3Info` explains the command to add a Next Hop IP address to the system
- `addV4EthEntry` explains the command to add an entry containing both an IP address and Ethernet MAC addresses to the system
- `addV6EthEntry` shows how to use the command to add an L2 entry, containing both the IPv6 and the Ethernet MAC addresses to the system.

Refer to Section 9.2.1, “Layer 2 Table Manager,” on page 68 for complete information about the commands available in the Layer 2 Table Manager.

9.2.1 Layer 2 Table Manager

The L2 Table contains entries that are copied into packets before being sent out the ETH_TX or ATM_TX interfaces. The L2 Table is typically populated by an application, such as the CP-PDK, but may be manually controlled via commands issued from the Tornado shell.

This section describes how to modify the contents of the L2 Table and provides a brief overview of the L2 Table. It is assumed the appropriate Ingress and Egress images have been downloaded to the platform.

L2 Encapsulation microblocks obtain a 16-bit value from each packet's meta data (nexthop_id). This value serves as an index into the L2 Table. When adding or removing entries, the l2Index value indicates which entry in the L2 Table is affected. The nexthop_id value in packet metadata is populated by the IPv4 microblock on the Ingress side, according to the contents of the Route Table. See the Route Table Manager section *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Reference Manual* for more information.

The following procedure contains information about controlling the Layer 2 Table Manager:

1. Start a Tornado* shell for the Egress processor.
2. To see a list of commands, type `l2_config` at the prompt.
3. The following commands are listed:
   - addV4AtmEntry
   - addV4EthEntry
   - addV6EthEntry
   - addV4L3Info
   - deleteL2Entry
   - clearL2Info
   - purgeL2tm
   - dumpL2tm
4. Type the name of any command to get usage information.

Note: For Linux systems, you must prefix each command with L2config. For example, you would enter `l2config addV4EthEntry` to use the **addV4EthEntry** command.

The following subsections contain information about the various Layer 2 Table Manager commands.

Note: These commands are a simplification of the L2 Table Manager's API. See the *ix_cc_l2tm.h* header file located at **IXA_SDK_3.5\src\library\xscale\l2_table_mgr\include\cc** or the L2 Table Manager section of the *Intel® Internet Exchange Architecture (IXA) Software Building Blocks Reference Manua*l for more information.

9.2.1.1 Adding a Complete Entry for Ethernet/IPv4

Ethernet/IPv4 entries require the following information, which can be set using a single command as in the **Example**:

- l2Index (must match a value provided to addNextHop on Ingress)
- IPv4 address for the next hop
- Destination MAC address for the next hop
- Source MAC address for the outgoing port

Example:

```
-> addV4EthEntry "1 10.3.19.10 0a:0c:14:37:20:a0 cf:30:25:d6:4e:2f"
```

9.2.1.2 Adding a Complete Entry for Ethernet/IPv6

Ethernet/IPv6 entries require the following information, which can be set using a single command as in the **Example**:

- l2Index (must match a value provided to addNextHopV6 on Ingress)
- IPv6 address for the next hop
- Destination MAC address for the next hop
- Source MAC address for the outgoing port

Example:

```
-> addV6EthEntry "1 3ffe:0:0:1000::101 0a:0b:0c:0d:0e:02
0a:0b:0c:0d:0e:01"
```

9.2.1.3 Adding L3 information

It is possible to add an IP address and let ARP discover the MAC address. When a packet is sent to the specified IP address, the microblocks will forward the packet to the core because the MAC addresses are unknown. An ARP request is sent, and when the reply arrives the L2 Table is updated with the new information. The `addV4L3Info` command requires the following information:

- l2Index
- IPv4 Address

Example:

```
-> addV4L3Info "2 10.3.19.11"
```

9.2.1.4 Clearing L2 information

To remove Ethernet information from a complete entry (for example to force another ARP request), use `clearL2Info`. This leaves the entry in the same state as if it had been completely removed (`deleteL2Entry`) and re-added (`addV4L3Info`):

Example:

```
-> clearL2Info "3"
```

9.2.1.5 Removing an L2 Entry

To completely remove an L2 entry, use `deleteL2Entry`. Once an entry has been removed, its index may be used to add a new entry with different information:

Example:

```
-> deleteL2Entry "2"
```

9.2.1.6 Purging the L2 Table

All contents of the L2 Table may be removed with a single command as follows:

```
-> purgeL2tm
```

9.2.1.7 Printing the contents of the L2 Table

To see a listing of the contents of the L2Table, use the following command:

```
-> dumpL2tm
```

Note: For Linux systems, enter `l2config dumpL2tm`.

IXDP2401 Application Tutorial A

The information in this appendix provides procedures for running and debugging a Windows 2000 example application on the Intel® IXDP2401 Advanced Development Platform. The appendix is organized as follows:

- "Using the oc12_pos_gbeth_2401 Application"contains descriptions of a sample application, its data flow, and the files it uses. This section also includes exercises that allow you to perform tasks such as building the application and enabling logging.
- "Debugging the oc12_pos_gbeth_2401 Application" discusses debugging the application using break points.

A.1 Using the oc12_pos_gbeth_2401 Application

This section discusses a sample application that implements IPv4 forwarding running on POS and media at the OC-12 rate and on Gigabit Ethernet. The application is written using the Intel® IXA Portability Framework. This chapter contains the following topics:

- A high-level description of the oc12_pos_gbeth_2401 application, including the directories and the files used
- A set of exercises to help you become familiar with the contents and structure of the oc12_pos_gbeth_2401 application.

A.1.1 Overview of the oc12_pos_gbeth_2401 Application

The oc12_pos_gbeth_2401 application receives POS or Ethernet frames that carry IPv4 datagrams. The frames are assembled into IPv4 packets and the Layer-2 (Ethernet or PPP) headers are removed. Data comes in through the MSF into RBUFs. The data in the RBUFs is reassembled into PPP or Ethernet frames. For each frame, the Layer-2 header (PPP or Ethernet) is removed to yield an IP datagram.

The IPv4 microblock performs an RFC 1812 header check. Next a Longest Prefix Match (LPM) lookup is performed and the packets are segmented and transmitted over the appropriate port. The result of the LPM lookup determines which port is used to transmit the packet.

A.1.1.1 Building Blocks Relationship

The following figure shows the relationship of the blocks in a single chip configuration and the destination of the packets.

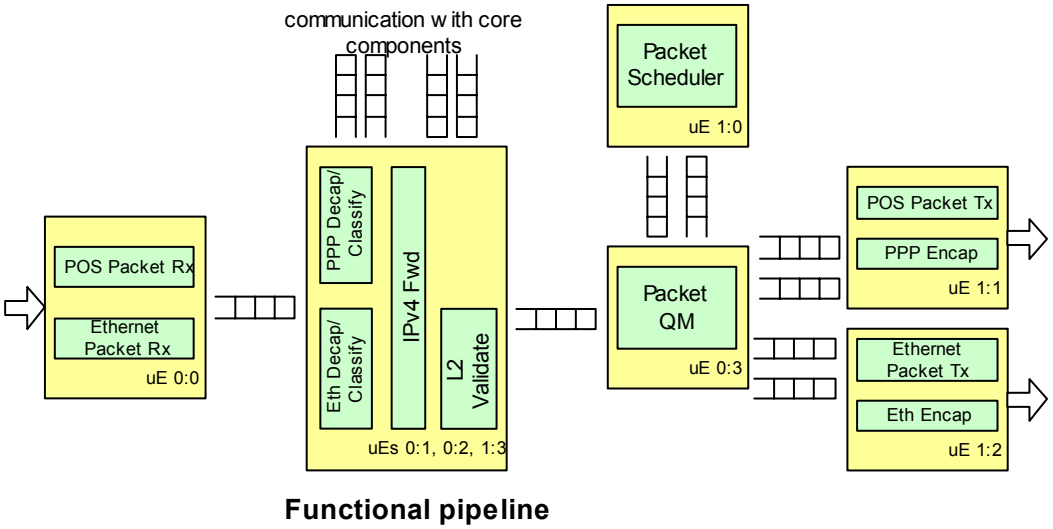


Figure 3. Block Diagram: oc12_pos_gbeth_2401 Application

The oc12_pos_gbeth_2401 project uses the following microblocks:

Microblock	Path and Description
packet_rx	C:\IXA_SDK_3.5\src\building_blocks\rx\microengine\packet_rx\ Packet receive operations
ppp_decap	C:\IXA_SDK_3.5\src\building_blocks\rx\microengine\l2_decap\ppp_decap\ PPP decapsulation operations
ethernet_decap	C:\IXA_SDK_3.5\src\building_blocks\rx\microengine\l2_decap\ethernet_decap Ethernet decapsulation operations
ipv4_fwder	C:\IXA_SDK_3.5\src\building_blocks\ipv4\ IPV4 forwarder
L2_validate	--
qm	C:\IXA_SDK_3.5\src\building_blocks\queue_manager\qm_packet\ Queue manager
scheduler	C:\IXA_SDK_3.5\src\building_blocks\scheduler\scheduler_packet\ Packet scheduler
ppp_encap	C:\IXA_SDK_3.5\src\building_blocks\tx\microengine\l2_encap\ppp_encap\ PPP encapsulation operations

ethernet_encap	C:\IXA_SDK_3.5\src\building_blocks\tx\microengine\l2_encap\ethernet_encap Ethernet encapsulation operations
packet_tx	C:\IXA_SDK_3.5\src\building_blocks\tx\microengine\packet_tx\ Packet transmission operations

A.1.1.2 Application-Specific Files

The directory `C:\IXA_SDK_3.5\src\applications\ipv4_forwarder\0c12_pos_gbeth_2401\wbench_project` contains the files specific to this application. The following subsections examine the contents of this directory.

A.1.1.2.1 dispatch_loop Subdirectory

The building blocks can be combined into many applications, using a dispatch loop which implements the packet data flow for an application. Consider the dispatch loop as the application-specific "glue code" which integrates the building blocks into one application.

The following related files are maintained under the `\dispatch_loop` directory:

File	Description
dispatch_loop.h	Contains global configurable parameters for the dispatch loop.
dl_system.h	Contains all the compile time defines specific to this application, for example: Flags specific to running this application in simulation or hardware Sizes of queue arrays Base addresses for the lookup and statistics tables Block IDs for the microblocks included in this application dl_system.h also sets the source and the destination microblocks.
dl_source.uc	The actual implementation of the packet data flow. Contains macros for dl_sink and dl_source for POS and IPv4 Blocks. These macros ensure that the microblocks remain independent from the other microblocks in the application. The functions also hide application-specific inter-microblock communications from the individual microblocks. For example, whether the microblocks communicate through next neighbor registers or through a scratch ring.
pos_eth_ipv4.uc	Contains code, which binds the L2 Classifier and IPv4 Fwd into an application such that the L2 Classifier receives packets from Packet_Rx through dl_source. After removing POS or Eth header, the packet undergoes IP lookup and header update, and the updated packet is sent to Queue Manager through dl_qm_sink.
system_init.uc	Contains code, which does all the initialization to get the system up and running. Typically it initializes all the various rings, creates the buffer free list, initializes the various cache (meta, ip header etc).
pkt_header_cache.uc	IP header caching macros. The macros support PPP (POS) and Ethernet encapsulation.

A.1.1.2.2 list Subdirectory

This folder is initially empty. After you build an application using the Developer Workbench, this folder contains all the list files that are created.

A.1.1.2.3 log Subdirectory

This folder is initially empty. After you run the application on the Transactor, this folder contains all the log files created for the packets received and transmitted. These log files may be analyzed by scripts to verify that the output is correct.

A.1.1.2.4 scripts Subdirectory

This folder contains all the files used to initialize the various tables in SRAM and DRAM prior to running the application on the Transactor—for example, creating the route tables, initializing the statistics counters, and similar tasks.

A.1.1.2.5 streams Subdirectory

This folder contains the different types of packet streams created to simulate traffic while running the application on the Transactor.

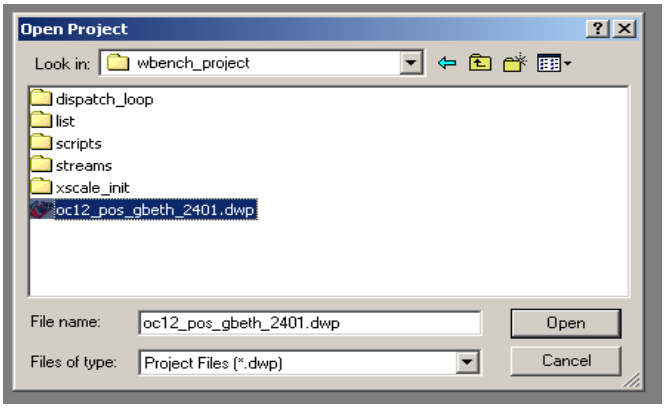
A.1.2 Working with the oc12_pos_gbeth_2401 Application

The following procedure is an exercise where you will perform some action, then get a mini-assignment about the file(s) you've opened:

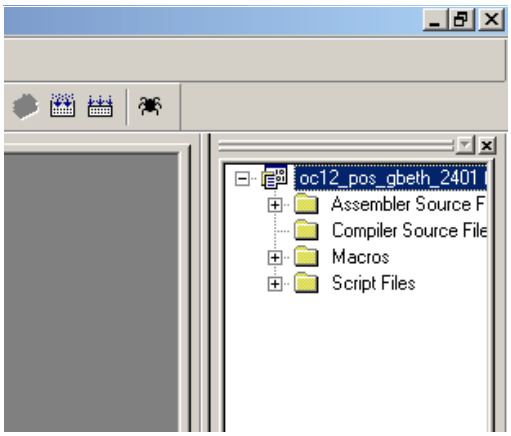
1. Open Workbench by clicking **Start > Programs > IXA SDK 3.5 > DevWorkbench**
2. Click on **File > Open Projec**t.
3. The oc12_pos_gbeth_2401 application is in the following directory:

   `CC:\IXA_SDK_3.5\src\applications\ipv4_forwarder\0c12_pos_gbeth_2401\wbench_project`.

   Select the `oc12_pos_gbeth_2401.dwp` file.



4. On the right side of the Workbench window, select the File View tab to display the files for this project. You can expand the list of Compiler Source Files to see the list of the `*.h` and `*.uc` files in the application. Double-click on any file to open it in the Workbench.



Exercises:

a. Open the file `dl_system.h`, search for `POS_RX_NEXT1`. What is this set to?

b. Open the file `packet_rx.uc`. Where is this file located? Search for `dl_sink`.

c. Open the file `dl_source.uc`. Where is this file located?

d. Open the file pos_eth_ipv4.uc. Where is the dispatch loop for `PPP_Classify` and `IPv4Fwd`? (Hint: search for `dl_source`.)

e. Open the file ppp.uc. Where is this file located? Search for `ppp_classify`.

f. Open the file `ipv4_fwder.uc`. Where is this file located? What does the sub routine `Ipv4Fwder` do?

g. In the `ipv4_fwder.uc` file, search for `IPV4_NEXT1`. What is `IPV4_NEXT1` set to? (Search in `dl_system.h`).

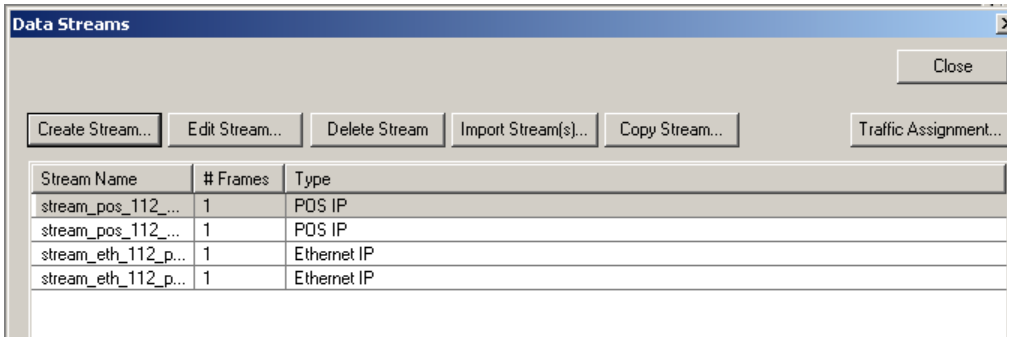
h. Open the file qm_packet_code.uc. What does this file contain?

Now we know how to search for the microblocks used to build an application and examine the dispatch loop for the application.

5. To build the application, click the **Build** button on the toolbar.
   The application should compile with 0 errors and warnings. The list files can be examined in the subdirectory

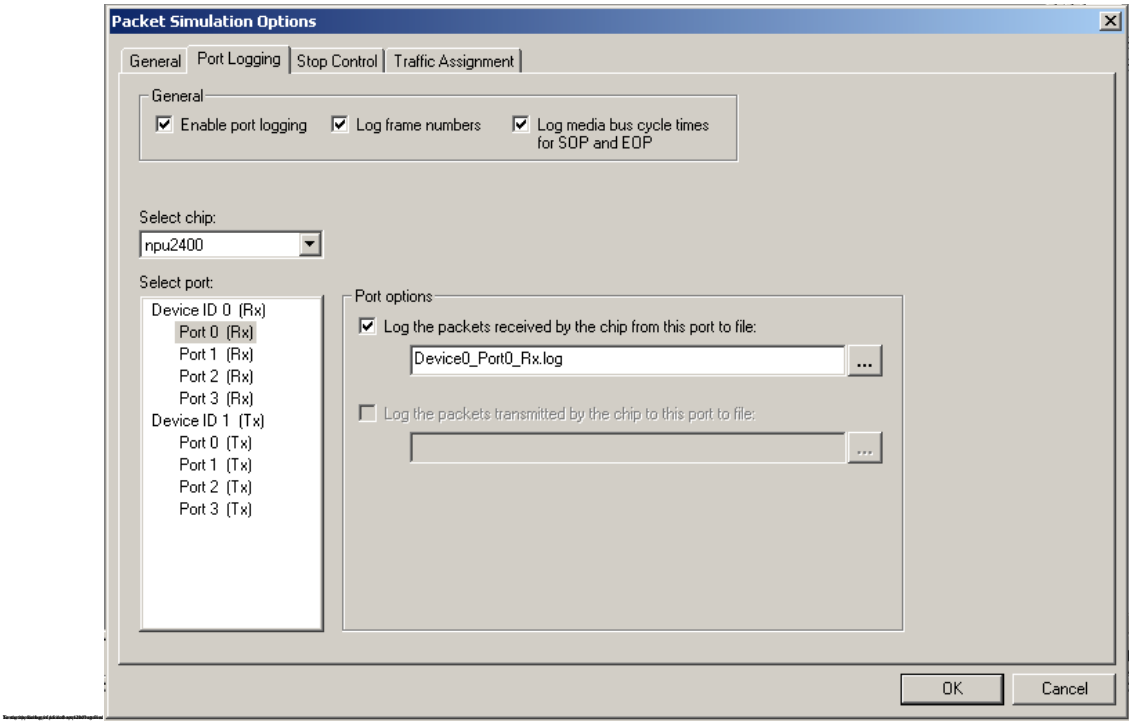
   ```
   :\IXA_SDK_3.5\src\apps\oc12_pos_gbeth_2401\wbench_project\list
   ```

6. Before running the application, let's check the data stream the application will receive. Click **Simulation > Data Streams** on the menu bar. We see that a stream has been selected.



We can see the contents of the packet by clicking on **Edit Stream->Edit Frame**. Where is the corresponding file for this stream?

7. To enable logging for the packets received by the application, click **Simulation > Packet Simulation Options > Port Logging**. Verify that the Enable Logging and Log frame numbers options are enabled. Check the path for the log file.



8. .To start the debugger, click the **Debug** button on the tool bar. 
   Let the application run for about 12000 cycles.
9. .To see the number of packets sent and received, click the **Packet Simulation Status** button on the tool bar.

10. Now go to the log folder and open the log files. What do you see?

A.2 Debugging the oc12_pos_gbeth_2401 Application

This chapter contains an overview of certain application packet flow concepts and guidelines for debugging the oc12_pos_gbeth_2401 application on the Transactor. Stepping through the application code, shows how the inter-microengine communication is implemented and how the packet information is exchanged from one microblock to the next.

A.2.1 Application Packet Flow Overview

The Intel® IXA Portability Framework uses certain types of structures which are unique to each packet and which specify the packet characteristics based on which microblocks make routing and processing decisions. Before debugging an application, it is important to understand the following packet flow concepts:

- dispatch loop variables
- packet metadata
- packet buffer

The `dl_ingress_system_default.h` file (found in `C:\IXA_SDK_3.5\src\library\microblocks_library\include`) contains the # defines for microblock IDs, packet metadata, and packet buffer.

A.2.1.1 Dispatch Loop Variables

Dispatch loop variables are exchanged from one microblock to another as the packet is passed from one microblock to the next. Dispatch loop variables include next block and buffer handles, for example. Therefore, for each packet there will be a corresponding set of dispatch loop variables. After the current microblock has processed the packet, it sets the next block to the next microblock ID. The other variables uniquely identify where the packet metadata resides in the SRAM and where the actual packet resides in the DRAM. For instance, when the PacketRx microblock forwards the packet to the PPP-IPv4 microblock running on a different microengine, PacketRx sets the next block to BID_POS and writes the following dispatch loop variables to the scratch ring:

- Buffer handle containing start of packet (SOP). This is a 32-bit value where the lower 24 bits can be used to locate the packet metadata in the SRAM and the actual packet in the DRAM.
  The buffer handle structure `buf_handle_t` is defined in `ixp_lib.h`.
- Buffer handle containing end of packet (EOP). If the packet is longer than 2K, this points to the location of the packet buffer which contains the end of the packet.

Similarly, when the PPP-IPv4 microblock forwards the packet to the Queue Manager microblock, the IPv4 writes the buffer handles for start of packet, end of packet, and the port number. Depending on the functionality performed by the downstream microengine, the microblocks write the most relevant packet characteristics to the scratch ring. However, when the PPP microblock forwards the packet to IPv4Fwder, it sets the `dl_next_block` to `BID_IPV4` and caches dispatch loop variables to local memory or GPRs.

A.2.1.2 Packet Metadata

Packet metadata is a set of variables that describes the characteristics of the packet, such as the buffer descriptors, packet length, header type, input port number, etc. By default, packet metadata is 8 long words in size and is stored in SRAM.

Some of the elements of packet metadata include:

- Amount of packet data in the buffer.
- DRAM offset where the packet begins. Each buffer in the DRAM is 2K or 2048 bytes long and the start of the packet is 128 bytes from the start of the buffer. Starting the packet at a 128 byte offset comes in handy when a microblock has to prepend the header without moving the packet around in the DRAM. For example, the MPLS Marker microblock inserts an MPLS label before the IP header and adjusts the offset to 124 bytes.
- Packet length. If the packet length is more than 2K, the microblock learns that the packet is spread across a chain of buffers.
- Header type. Identifies whether it is a IPv4 packet or a IPv6 packet.

A.2.1.3 Packet Buffer

This buffer contains the actual packet data received from the media interface. This is stored in DRAM and is 2K in size. If the total packet data is more than 2K in size, the microblock uses a chain of packet buffers. The packet buffer containing the SOP has headroom of 128 to 512 bytes. This allows room to prepend headers without having to move the packet within the DRAM.

A.2.2 Debugging oc12_pos_gbeth_2401

This section explains how to do certain simple tasks using the Developer Workbench. For details on the full functionality provided by the Workbench, refer to the *Development Tools User's Guide* on the Intel® IXA SDK Tools CD.

The exercises in this section will demonstrate:

- how packet information is exchanged from one microengine to another
- how packet characteristics and packet data is organized in SRAM and DRAM
- how the microblocks access and modify these characteristics and the packet header

Perform the following steps to debug the oc12_pos_gbeth_2401 application:

1. Build the oc12_pos_gbeth_2401 application project and click the **Debug** button.
2. Click the **Memory Watch** button on the toolbar:
3. The Memory Watch window shows the contents of scratchpad, SRAM, and DRAM. Set a breakpoint on change on scratchpad. This breakpoint will be hit when a microblock writes the dispatch loop variables to the next microblock. In this specific example, the breakpoint will be hit when the PacketRx microblock writes the

dispatch loop variables to the scratch ring between PacketRx and PPP-IPv4. (Refer to `dl_sink` in the `dl_source.uc` file.)

4. Run the application by clicking on the **Go** button on the toolbar.
5. When the first break point is reached, open the file `dl_source.uc` and go to `dl_sink`. The PacketRx microblock uses dl_sink) to write 5 long words, (`dlBufHandle`, `dlEopBufHandle`, `dram offset`, etc) to the scratch ring.
6. Let the application run until the scratchpad memory gets initialized with the dispatch loop variables.
7. Using the dispatch loop variables, locate the packet metadata in SRAM and packet buffer in DRAM. The dlBufHandle variable tells that the packet buffer has both SOP and EOP, which implies that the packet length is less than 2K in size, therefore we can ignore the dlEopBufHandle. From the lower 24 bits we can derive the SRAM address for the packet metadata and DRAM address for the packet buffer as follows:
   packet metadata address = (dlBufHandle.lw_offset << 2)
   packet buffer address = (dlBufHandle.lw_offset << 8)

   Note: In practice, use the library function Dl_`BufGetDesc` in dl_`buf.uc` to obtain the packet metadata and the library function D`l_BufGetData` in `dl_buf.uc` to obtain the packet address.
8. Given the `dlBufHandle` = 0xc0000010, the packet metadata resides at SRAM address 0x40. Add a SRAM watch point for address 0x40:+32. The second long word shows the packet size and buffer offset from where the packet data starts in the packet buffer.
9. Given the `dlBufHandle` = 0xc0000010, the packet buffer starts at 0x1000 and the actual packet data starts at 128 byte offset. Add a DRAM watch point for address 0x1080:+40. The contents show the PPP header, IP header, and IP payload.
10. Set a break point on change on the SRAM address to see how the packet metadata gets updated.
11. Set a break point on change on the DRAM address to see how the PPP decapsulation and IPvFwder microblocks modify the IP header.
12. When the break point on the SRAM address is reached, the packet size and the buffer offset change. This implies that the PPP_Classify microblock has stripped off the PPP header and adjusted the packet length and buffer offset. (Note: See `_ppp_decap` in `ppp.uc`). The ppp header still exists in the DRAM but from this point onwards, the IPv4Fwder microblock will operate as if the packet size is 40 bytes and packet starts at offset 130 instead of 128. Also, the PPP_Classify microblock updates the `dlMeta.header_type` in the packet metadata to `PPP_IPV4_TYPE`. (See `_ppp_classify` in `ppp.uc`).
13. When the break point on the DRAM address is reached, it implies that the IPv4Fwder microblock has decremented the TTL and updated the checksum in the IP header.

To gain a deeper perspective on the concepts and microblocks covered in this debugging exercise, refer to the following documents on the Intel® IXA SDK Software Framework CD:

- *Intel® Internet Exchange Architecture (IXA) Portability Framework Developer's Manual*
- *Intel® Internet Exchange Architecture (IXA) Portability Framework Reference Manual*

For details on the full functionality provided by the Developer Workbench, refer to the *Development Tools User's Guide* on the Intel® IXA SDK Tools CD.