# Intel ® Internet Exchange Architecture (IXA) Portability Framework

## Tutorial

*April 2003*

Order Number: 278665-002



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

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

# Contents



## Figures



## Tables

# Revision History

| Date | Revision | Description |
|---|---|---|
| February 2003 | 001 | IXA SDK 3.0 Pre-Release 6 |
| April 2003 | 002 | IXA SDK 3.0 PR6 FCS<br>Change bars show locations of minor edits and where new screen shots were added. |

# About This Publication 1

This publication, the Intel® Internet Exchange Architecture (IXA) Portability Framework Tutorial, introduces you to the Intel® Internet Exchange Architecture Software Development Kit (IXA SDK) and guides you through the following tasks:

- Learning the central files and structures that comprise the IXA Portability Framework
- Writing and running a network application in Microengine C using the IXA Portability Framework

The chapters in this Tutorial will teach you to:

- Identify the various modules within an application
- Run and debug an application
- Modify an existing 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

## 1.2 How to Use This Publication

Refer to this publication after you have installed the Intel® Exchange Architecture Software Development Kit (IXA SDK) Tools and Applications CDs.

The information in this publication is organized as follows:

- Chapter 2, "Framework Overview & Directory Tour," contains an overview of the IXA Portability Framework and the directory structure of the IXA SDK.
- Chapter 3, "Using the oc48_pos_ipv4_ingress Application," contains descriptions of an application, its data flow, and the files it uses. This chapter also includes exercises which allow you to perform tasks such as building an application and enabling logging.
- Chapter 4, "Debugging the oc48_pos_ipv4_ingress Application," discusses debugging the application using break points.
- Chapter 5, "Modifying the oc48_pos_ipv4_ingress Application," describes the steps required to change an existing application, by adding new building blocks, modifying dispatch loops, etc.

## 1.3 Other Sources of Information

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

- *Intel® Internet Exchange Architecture (IXA) Portability Framework Reference Manual*
- *Intel® Internet Exchange Architecture (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® IXP2400 Network Processor and Intel® IXP2800 Network Processor Getting Started*
- *Intel® IXA SDK 3.0 Release Notes*
- *Intel® IXA Optimized Data Plane Libraries Reference Manual*
- *IXP2400/IXP2800 Development Tools User's Guide*
- *Help Topics: Developer Workbench*
- *Intel® IXP2400/IXP2800 Network Processor Microengine C Compiler Language Support Reference*
- *Intel® IXP2400/IXP2800 Network Processor Microengine C Compiler LIBC Reference*
- *Intel® IXP2400/IXP2800 Network Processor Programmer's Reference Manual*
- *Intel® IXP2800 Network Processor Datasheet*
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2400 Network Processor Datasheet*
- *Intel® IXP2400 Network Processor Hardware Reference Manual*

## 1.4 Documentation Conventions

This tutorial contains the following text and typography conventions:

**Table 1-1. Documentation Conventions**

| Convention | Description |
|---|---|
| *Italics* | New terms and titles of documents or help systems appear in italics. |
| **bold** | Special text for labels of items appears in bold type. |
| `monospace` | Code and names of drives, directories, and files appear in boldface monospace type. |
| table of contents<br>table of figures<br>table of tables | Each item in the tables of contents, figures and tables is a hyperlink. |
| > | > links successive menu items. For example, the menu items for Developers Workbench are **Start > Programs > IXA SDK 3.0 > DevWorkbench**. |
| blue | When this document is viewed with Adobe* Acrobat*, blue text is for headings and hyperlinks. Hyperlinks can either provide a jump to another part of the document or launch Developer's Workbench or WindRiver* Tornado* for examining a project with those tools, providing a convenient way to navigate the SDK. |

# Framework Overview & Directory Tour 2

This chapter begins with a brief overview of the IXA Portability Framework and provides a tour of certain key directories installed with the IXA SDK. This overview provides a foundation for understanding the following concepts:

- Applications written with the framework
- Microblocks
- Combining microblocks to form an application

## 2.1 IXA Portability Framework Overview

The IXA Portability Framework comprises a software infrastructure for writing modular and portable code for network applications which can be used in VxWorks and Linux configurations. The 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 Intel® IXP2400, Intel® IXP2800, and Intel® IXP2850 network processors.

For more details on the Portability Framework, refer to the *Intel® Internet Exchange Architecture (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 tutorial 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 Applications in General

With the applications provided by the Applications 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, as shown in Figure 2-1.

**Figure 2-1. src Directory Structure Contents**


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 dataplane 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 Roadmap, \documentation\CP-PDK\Document Roadmap.pdf. |
| 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. |

**Table 2-1. src Subdirectory Descriptions (Continued)**

| Directory | Description |
|---|---|
| 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 Applications CD of the IXA SDK 3.0, 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 and the basic building blocks can be common to several projects. The applications and building blocks are explained in the following subsections.

### 2.3.1 Applications Directory

The applications are in the directory **`IXA_SDK_3.0\src\applications`** with subdirectories identifying the application. In our example of the IPV4 forwarder on POS media at OC-48 on an ingress processor, the path is as follows:
c:\ixa_sdk_3.0\src\applications\ipv4_forwarder\oc48_pos\ingress

There are two project-specific subdirectories: **`wbench_c_project`**, written in Microengine C, and **`wbench_project`**, written in microcode. In this Tutorial, 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 |

### 2.3.2 Building Blocks Directory

The building blocks (microblocks) are installed in separate directories under the **`IXA_SDK_3.0\src\building_blocks`** 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 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 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_3.0\src\library`** 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 XScale™ core, for example, fragmentation, route table manager, and L2 table manager. |

# Using the oc48_pos_ipv4_ingress Application 3

This chapter discusses a sample application that implements IPv4 forwarding in an ingress processor running on POS media at the OC-48 rate. The application is written using the IXA Portability Framework. For more details on this application, refer to Chapter 2, "POS Application Overview," in the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*. This chapter contains the following topics:

- A high-level description of the oc48_pos_ipv4_ingress application, including the directories and the files used.
- A set of exercises to help you become familiar with the contents and structure of the oc48_pos_ipv4_ingress application.

## 3.1 Overview of the oc48_pos_ipv4_ingress Application

The oc48_pos_ipv4_ingress application receives PPP frames carrying IPv4 datagrams over a POS interface. Data comes into through the MSF into RBUFs. The data in the RBUFs are reassembled into PPP frames. For each PPP frame, the layer-2 PPP header is removed to yield an IP datagram.

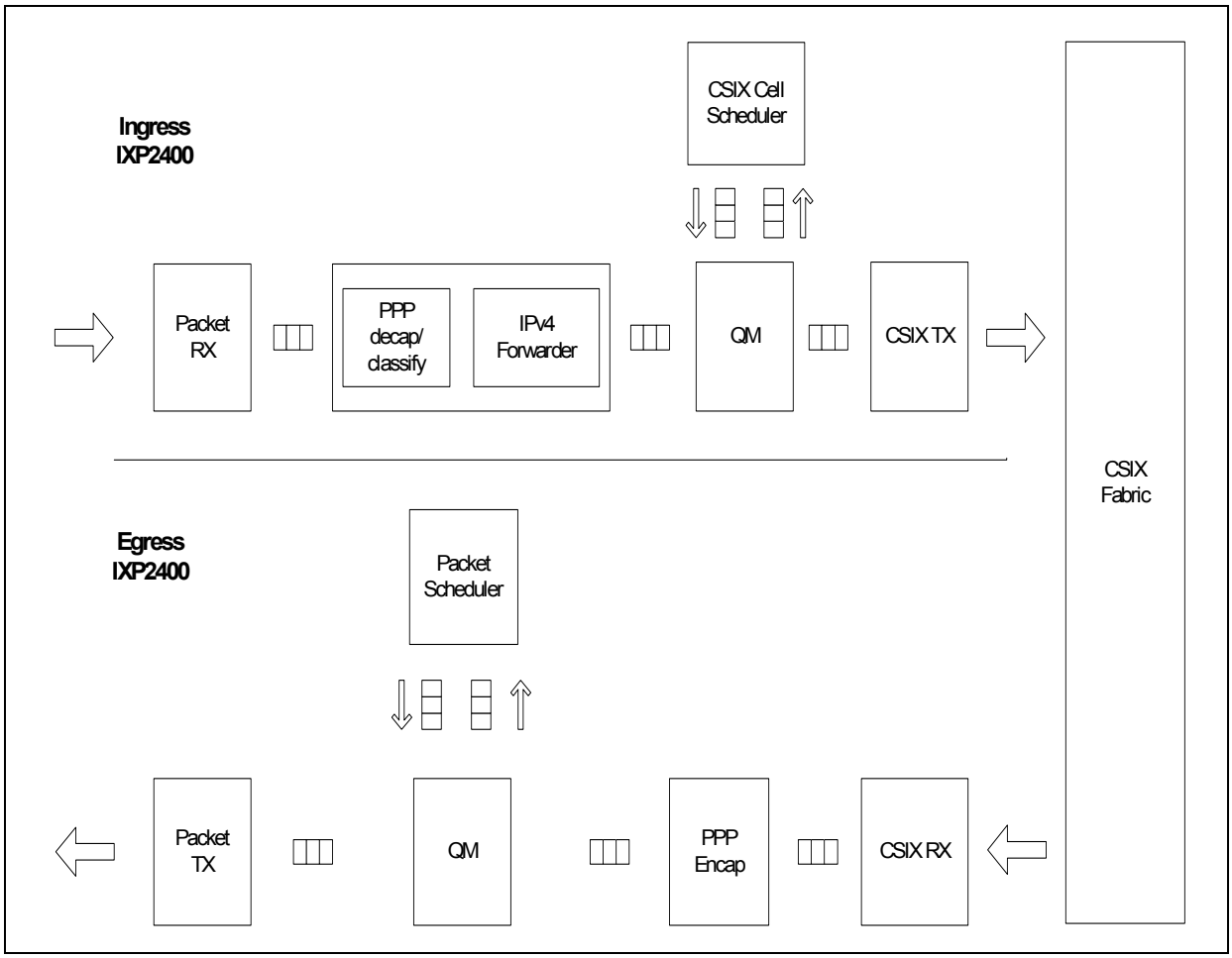
The IPv4 microblock performs a RFC 1812 header check. Next a Longest Prefix Match (LPM) lookup is performed and the packets are segmented into CSIX C-Frames and transmitted to the CSIX fabric. The result of the LPM lookup determines which Intel® IXP2400 network processor connected to the fabric receives the packet and which port of the selected processor is used to transmit the packet.

### 3.1.1 Building Blocks Relationship

Figure 3-1 shows the relationship of the blocks in a dual chip configuration and the destination of the packets. The microblocks that comprise the oc48_pos_ipv4_ingress application run on the Ingress Intel® IXP2400 Network Processor as shown in the upper half of the figure. For more

details on this application, including details of the interfaces between building blocks, refer to Chapter 2, "POS Application Overview," in the *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*.

**Figure 3-1. Block Diagram: oc48_pos_ipv4_ingress Application**




The oc48_pos_ipv4_ingress project uses the following microblocks, coded in Microengine C:

| Microblock | Path and Description |
|---|---|
| `csix_tx` | C:\IXA_SDK_3.0\src\building_blocks\tx\microengine\csix_tx\microc<br>CSIX transmit operations |
| `ipv4_fwder` | C:\IXA_SDK_3.0\src\building_blocks\ipv4\microc<br>IPV4 forwarder |
| `packet_rx` | C:\IXA_SDK_3.0\src\building_blocks\rx\microengine\packet_rx\microc<br>Packet receive operations |
| `qm` | C:\IXA_SDK_3.0\src\building_blocks\queue_manager\qm_packet\microc<br>Queue manager |
| `scheduler` | C:\IXA_SDK_3.0\src\building_blocks\scheduler\scheduler_csix\microc<br>CSIX scheduler |

## 3.1.2 Application-Specific Files

The directory C:\IXA_SDK_3.0\src\applications\ipv4_forwarder\oc48_pos\ingress\wbench_c_project contains the files specific to this application. The following subsections examine the contents of this directory.

### 3.1.2.1 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:<br>• Flags specific to running this application in simulation or hardware<br>• Sizes of queue arrays<br>• Base addresses for the lookup and statistics tables<br>• Block IDs for the microblocks included in this application<br>dl_system.h also sets the source and the destination microblocks. For example, PACKET_RX_NEXT1 is set to BID_POS with the implication that Packet_Rx will forward the packets to PPP_classify. |
| dl_source.c | dl_source.c is the actual implementation of the packet data flow. This file sets the values for the outputs of a microblock. dl_source.c contains functions used to perform:<br>• transfer of packets from the Packet_Rx microblock to PPP_Classify (sub-routine DlSink)<br>• receive of packets from Packet_Rx (sub-routine dl_source) and from IPv4Fwd to Queue_Manager (sub-routine dl_qm_sink)<br>These functions 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_ipv4.c | Contains main( ) which binds the PPP_Classify and IPv4Fwd into an application such that the PPP_classify receives the packet from Packet_Rx through dl_source(). After removing the POS header, the packet undergoes IP lookup and header update, and the updated packet is sent to Queue Manager through dl_qm_sink. |

### 3.1.2.2 list Subdirectory

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

### 3.1.2.3 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 for correctness of the output.

#### 3.1.2.4 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.

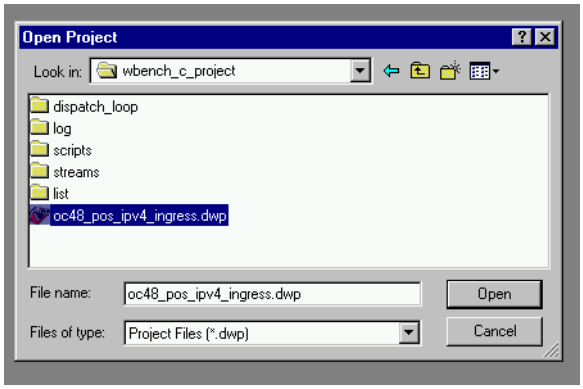
#### 3.1.2.5 streams Subdirectory

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

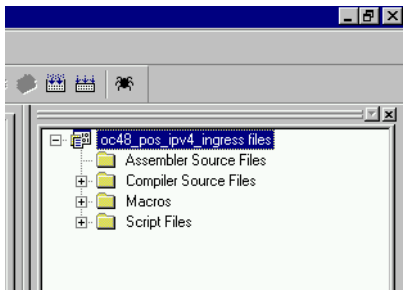
## 3.2 Working with the oc48_pos_ipv4_ingress Application

Let's reinforce some of the concepts we have learned in the preceding sections. 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 3.0 > DevWorkbench
2. Click on File > Open Project.
3. The oc48_pos_ipv4_ingress application is in the following directory: C:\IXA_SDK_3.0\src\applications\ipv4_forwarder\oc48_pos\ingress\wbench_c_project. Select the oc48_pos_ipv4_ingress.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 *.c files in the application. Double-click on any file to open it in the Workbench.




Exercises:

a. Open the file dl_ingress_system_default.h and search for POS_RX_NEXT1. What is this set to?

b. Open the file packet_rx.c. Where is this file located? Search for DlSink.

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

d. Open the file pos_ipv4.c. Where is the dispatch loop for PPP_Classify and IPv4Fwd? (Hint: search for dl_source.)

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

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

g. In the ipv4_fwder.c file, search for IPV4_NEXT1. What is IPV4_NEXT1 set to? (Hint: Search in dl_ingress_system_default.h.)

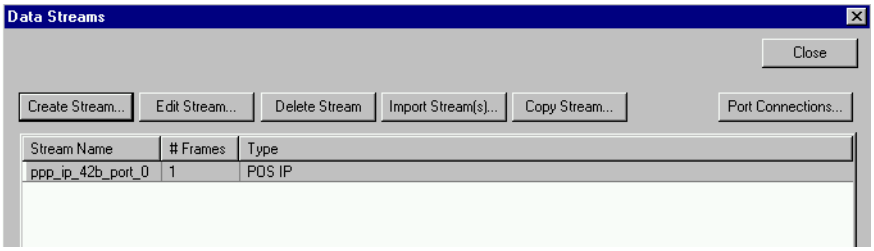
h. Open the file qm_packet.c. What does this file contain?

i. Open the file csix_tx.c. Where is this file located?

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

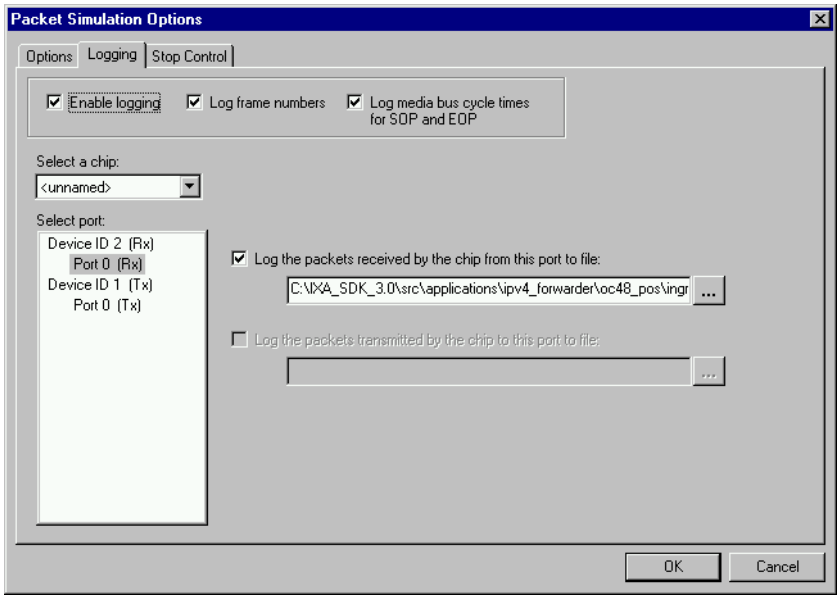
5. To build the application, click the Build button on the toolbar.  The application should compile with 0 errors and warnings.
The list files can be examined in the subdirectory:
C:\IXA_SDK_3.0\src\applications\ipv4_forwarder\oc48_pos\ingress\wbench_c_project\list

6. Before running the application, let's check the data stream the application will receive. Click Simulation > Data Stream 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 > 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 spider 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. 

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

# Debugging the oc48_pos_ipv4_ingress Application 4

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

## 4.1 Application Packet Flow Overview

The 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_system_ingress_default.h file (found in C:\IXA_SDK_3.0\src\library\microblocks_library\include) contains the hash defines for microblock IDs, packet metadata, and packet buffer.

### 4.1.1 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. 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 meta data 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 meta data 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.

### 4.1.2 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 meta data 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.

### 4.1.3 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 a head room of 128 to 512 bytes. This allows room to prepend headers without having to move the packet within the DRAM.

## 4.2 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*.

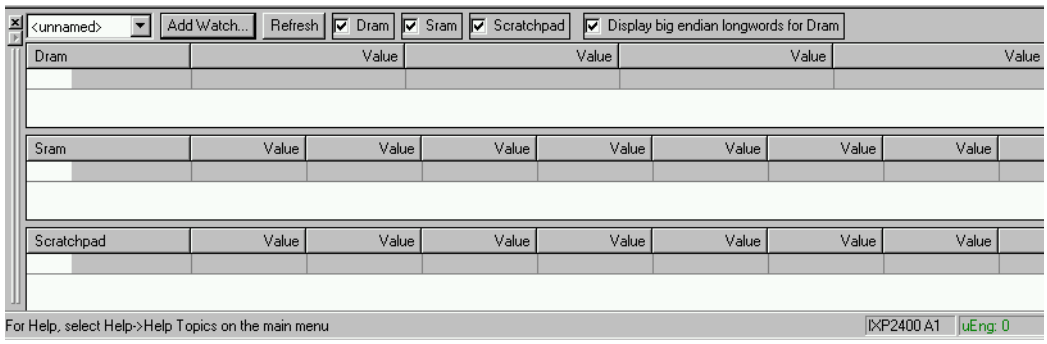
The exercises in this section will demonstrate:

- 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

Peform the following steps to debug the oc48_pos_ipv4_ingress application:

1. Build the oc48_pos_ipv4_ingress application project and click the debug button. 
2. Click the Memory Watch button on the toolbar. 

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

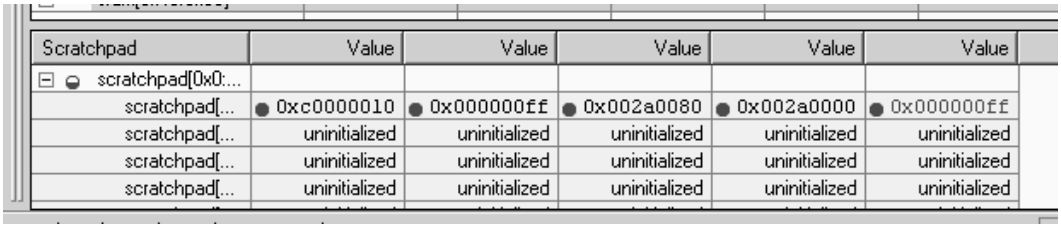



Set a breakpoint on change on scratchpad. This breakpoint will be hit when a microblock writes the dispatch loop variables to the next microblock. In this specific example, the breakpoint will be hit when the PacketRx microblock writes the dispatch loop variables to the scratch ring between PacketRx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

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

4. 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.

5. 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.



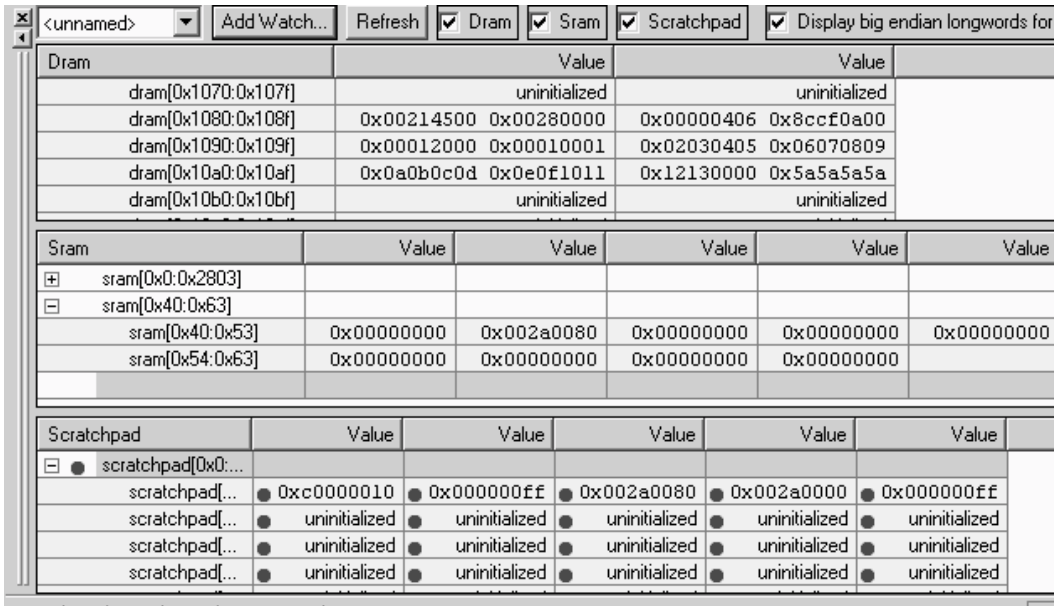

6. Using the dispatch loop variables, we can locate the packet metadata in SRAM and packet buffer in DRAM. The dlBufHandle variable tells that the packet buffer has both SOP and EOP, which implies that the packet length is less than 2K in size, therefore we can ignore the dlEopBufHandle. From the lower 24 bits we can derive the SRAM address for the packet metadata and DRAM address for the packet buffer as follows:
packet meta data 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 meta data and the library function Dl_BufGetData in dl_buf.c to obtain the packet address.

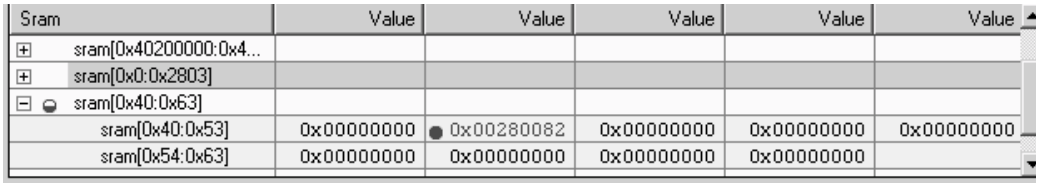
7. Given the dlBufHandle = 0xc0000010, the packet meta data resides at SRAM address 0x40. Add a SRAM watch point for address 0x40:+32. The second long word shows the packet size and buffer offset from where the packet data starts in the packet buffer.

8. 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.

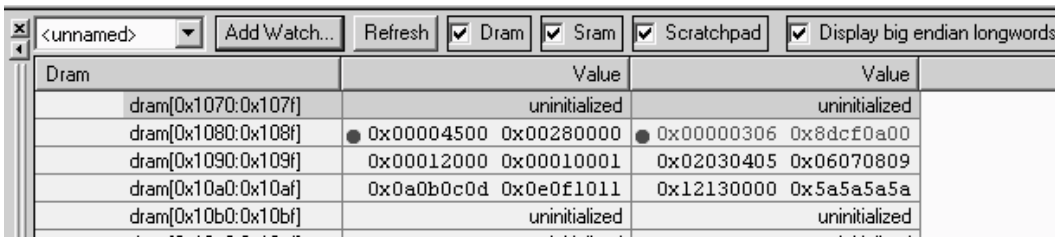
9. Set a break point on change on the SRAM address to see how the packet metadata gets updated.
10. Set a break point on change on the DRAM address to see how the PPP decapsulation and IPvFwder microblocks modify the IP header. The memory watch window will show DRAM, SRAM, and scratchpad as shown below.




11. 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.




12. 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 debugging exercise, refer to the following documents:

- *Intel® Internet Exchange Architecture (IXA) Portability Framework Developer's Manual*: Chapter 6 : Dispatch Loop
- *Intel® Internet Exchange Architecture (IXA) Portability Framework Reference Manual*: Chapter 8, Dispatch Loop

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

# Modifying the oc48_pos_ipv4_ingress Application 5

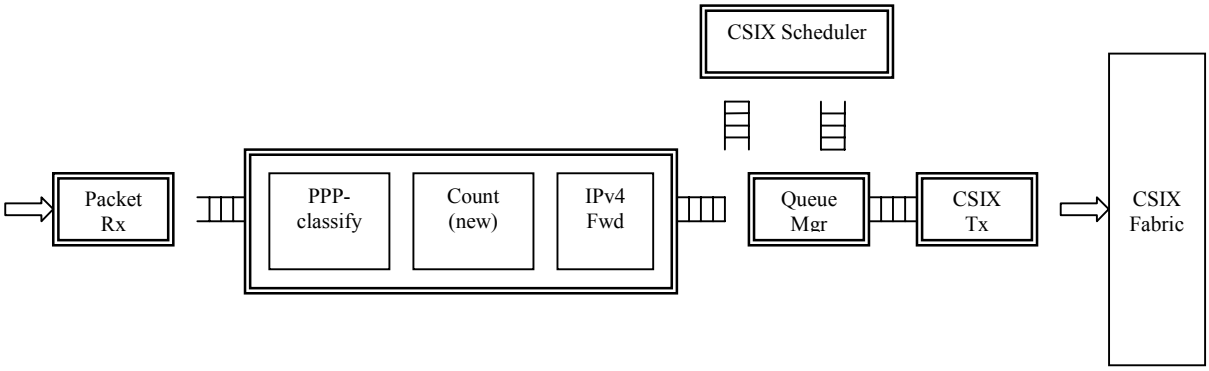
This chapter describes the steps involved to modify the existing oc48_pos_ipv4_ingress application by creating a new microblock, modifying the application's dispatch loop, rebuilding the project with new source files, and running the debugger.

## 5.1 Changing the Application

We will take 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 5-1.

*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 5-1. Block Diagram: Modified oc48_pos_ipv4_ingress Application**




## 5.2 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_tutorial in the directory C:\IXA_SDK_3.0\src\applications\ipv4_forwarder\oc48_pos\ingress
2. Copy all the folders from C:\IXA_SDK_3.0\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.

## 5.3 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 C:\IXA_SDK_3.0\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 C:\IXA_SDK_3.0\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.

```
/* TUTORIAL_begin */
#define BID_COUNT                   0x2A
/* TUTORIAL_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.

```
/* TUTORIAL_begin */
#define COUNT_SRAM_ADDR             0x40300200
/* TUTORIAL_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;
/* TUTORIAL_begin */
        dlNextBlock = BID_COUNT;
/* TUTORIAL_begin */

    }
...
return;

}
```

***Note:*** Remember to undo this modification after you complete the tutorial exercises.

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);
/* TUTORIAL_begin */
        count();
/* TUTORIAL_end */
        Ipv4Fwder(pkt_hdr,IP_HDR_OFFSET,pkt_hdr,IP_HDR_OFFSET);

        ------
```

```
    } // end while

    return 0;
}
```

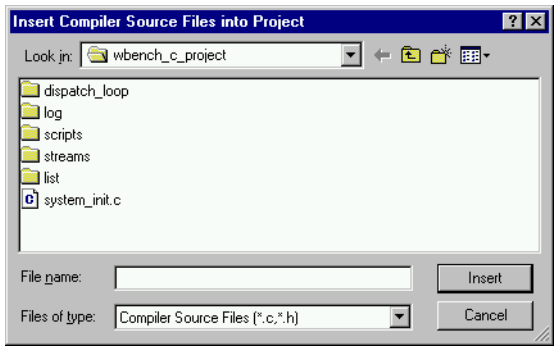
7. Include the count.c file in pos_ipv4.c.

```
#include "dl_source.c"
#include "ppp.c"
#include "ipv4_fwder.c"
/* TUTORIAL_begin */
#include "count.c"
/* TUTORIAL_end */
```

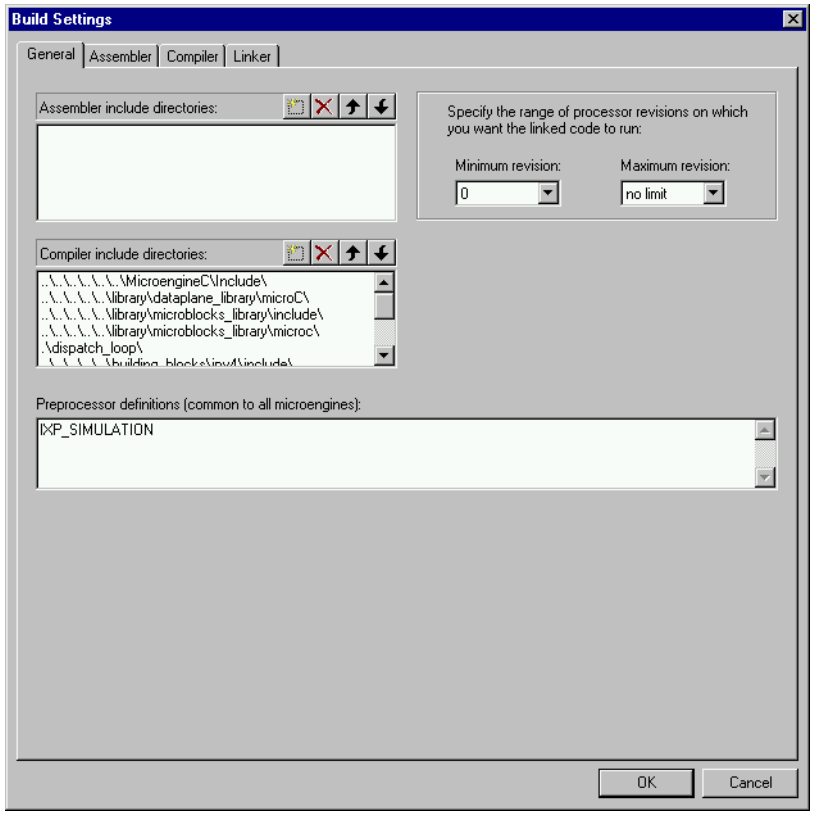
## 5.4 Building a New Project

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 bar.



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.

***Note:*** Remember to undo these modifications after you complete the tutorial exercises.

## 5.5 Additional Exercises

If you have reached this far, congratulations! You began this Tutorial with a brief overview of the IXP Portability Framework and ended with actually writing and debugging an application. In the process, we saw what an application looks like and saw how the application files are organized. We saw the role of the dispatch loop and how it provides the "glue" that combines building blocks to create a meaningful application. The rich set of building blocks can be reused and ported to different applications thus saving on development time. Finally we learned how to write a microblock and debug it using the Developer Workbench and Transactor. At this point you can

confidently try writing more complex microblocks, using the examples in the C:\IXA_SDK_3.0\src\EXAMPLES directory and the IXA SDK documentation set to support your activities.