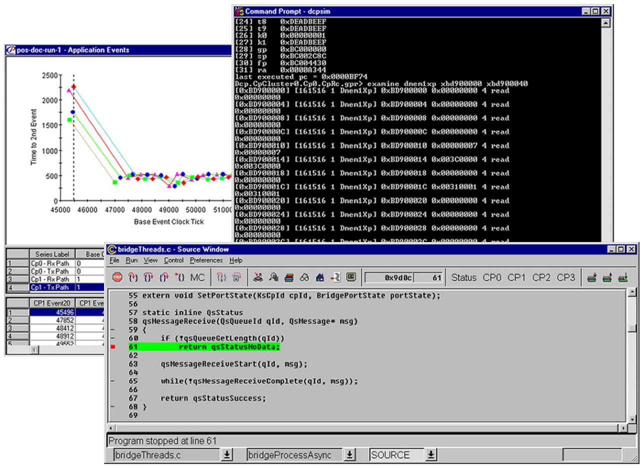
User Guide

C-WARE SOFTWARE TOOLSET
TUTORIAL WORKBOOK

C-WARE SOFTWARE TOOLSET
VERSION 2.2

CSTTW-UG/D
Rev 00

MOTOROLA
intelligence everywhere™



```
[24] t8    0xDEADBEEF
[25] t9    0xDEADBEEF
[26] k0    0x00000001
[27] k1    0xDEADBEEF
[28] gp    0xBC000000
[29] sp    0xBC002C8C
[30] fp    0xBC004430
[31] ra    0x0000A344
last executed pc = 0x0000BF74
Dcp.CpCluster0.Cp0.CpRc.gpr> examine dmem1xp xbd900000 xbd900040
[0xBD900000] [161516 1 Dmem1Xp] 0xBD900000 0x00000000 4 read
0x00000000
[0xBD900004] [161516 1 Dmem1Xp] 0xBD900004 0x00000000 4 read
0x00000000
[0xBD900008] [161516 1 Dmem1Xp] 0xBD900008 0x00000000 4 read
0x00000000
[0xBD90000C] [161516 1 Dmem1Xp] 0xBD90000C 0x00000000 4 read
0x00000000
[0xBD900010] [161516 1 Dmem1Xp] 0xBD900010 0x00000007 4 read
0x00000007
[0xBD900014] [161516 1 Dmem1Xp] 0xBD900014 0x003C0000 4 read
0x003C0000
[0xBD900018] [161516 1 Dmem1Xp] 0xBD900018 0x00000000 4 read
0x00000000
[0xBD90001C] [161516 1 Dmem1Xp] 0xBD90001C 0x00310001 4 read
0x00310001
[0xBD900020] [161516 1 Dmem1Xp] 0xBD900020 0x00000000 4 read
0x00000000
[0xBD900024] [161516 1 Dmem1Xp] 0xBD900024 0x00000000 4 read
0x00000000
[0xBD900028] [161516 1 Dmem1Xp] 0xBD900028 0x00000000 4 read
0x00000000
```

```
0x9d0c  61

55 extern void SetPortState(KsCpId cpId, BridgePortState portState);
56
57 static inline QsStatus
58 qsMessageReceive(QsQueueId qId, QsMessage* msg)
59 {
60     if (!qsQueueGetLength(qId))
61         return qsStatusNoData;
62
63     qsMessageReceiveStart(qId, msg);
64
65     while(!qsMessageReceiveComplete(qId, msg));
66
67     return qsStatusSuccess;
68 }
69

Program stopped at line 61
bridgeThreads.c    bridgeProcessAsync    SOURCE
```

C-Ware Software Toolset Tutorial Workbook

C-WARE SOFTWARE TOOLSET, VERSION 2.2

CSTTW-UG/D

Rev 00

CONTENTS

ABOUT THIS GUIDE

Guide Overview

This *C-Ware Software Toolset Tutorial Workbook* offers intensive instruction on how to use the Toolset to develop, debug, and tune C-Port NP-target application software.

This workbook is intended for new users of the CST who wish to quickly gain hands-on experience using CST software development tools.

This Workbook is divided into chapter-like sections called *lessons*. Each lesson assists the reader in learning a new set of skills for working with the CST.

This workbook contains nine chapters, which cover the following topics:

- Lesson 1 describes the structure of the lessons and provides details about the contents of each lesson.
- Lesson 2 describes the structure of the CST Toolset and identifies the locations of the CST tools and components.
- Lesson 3 provides instruction on the building of C-Ware applications.
- Lesson 4 describes the features of the C-Ware Simulation Environment (CSE) and provides instruction on its use.
- Lesson 5 describes the features of the C-Ware Debugger and provides instruction on its use.
- Lesson 6 describes the purpose and use of pattern files and pattern generation scripts.
- Lesson 7 describes the purpose of trace files and provides instruction on the use of the **printTrace.pl** script to filter trace files.
- Lesson 8 provides instruction on programming the Serial Data Processors (SDPs).
- Lesson 9 provides instruction on how to modify and debug the Gigabit Ethernet Switch (**gbeSwitch**) application.

Using PDF Documents

Electronic documents are provided as PDF files. Open and view them using the Adobe® Acrobat® Reader application, version 3.0 or later. If necessary, download the Acrobat Reader from the Adobe Systems, Inc. web site:

http://www.adobe.com/prodindex/acrobat/readstep.html

PDF files offer several ways for moving among the document's pages, as follows:

- To move quickly from section to section within the document, use the *Acrobat bookmarks* that appear on the left side of the Acrobat Reader window. The bookmarks provide an expandable 'outline' view of the document's contents. To display the document's Acrobat bookmarks, press the 'Display both bookmarks and page' button on the Acrobat Reader tool bar.
- To move to the referenced page of an entry in the document's Contents or Index, click on the entry itself, each of which is "hot linked."
- To follow a cross-reference to a heading, figure, or table, click the blue text.
- To move to the beginning or end of the document, to move page by page within the document, or to navigate among the pages you displayed by clicking on hyperlinks, use the Acrobat Reader navigation buttons shown in this figure:

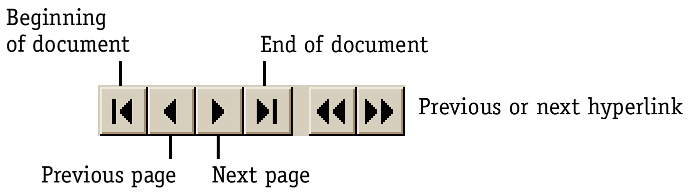


Table 1 summarizes how to navigate within an electronic document.

Table 1 Navigating Within a PDF Document

TO NAVIGATE THIS WAY	CLICK THIS
Move from section to section within the document.	A bookmark on the left side of the Acrobat Reader window
Move to an entry in the document's Contents or Index.	The entry itself
Follow a cross-reference (highlighted in blue text).	The cross-reference text
Move page by page.	The appropriate Acrobat Reader navigation buttons
Move to the beginning or end of the document.	The appropriate Acrobat Reader navigation buttons
Move backward or forward among a series of hyperlinks you have selected.	The appropriate Acrobat Reader navigation buttons

Guide Conventions

The following visual elements are used throughout this guide, where applicable:

This icon and text designates information of special note.

Warning: *This icon and text indicate a potentially dangerous procedure. Instructions contained in the warnings must be followed.*

Warning: *This icon and text indicate a procedure where the reader must take precautions regarding laser light.*

This icon and text indicate the possibility of electrostatic discharge (ESD) in a procedure that requires the reader to take the proper ESD precautions.

References to CST Pathnames

You typically install the C-Ware Software Toolset (CST) on your development workstation in a directory path suggested by the installation procedure, such as:

- **C:\C-Port\Cst***x***.***y***\** (on Windows NT)
- **/usr/***yourlogin***/C-Port/Cst***x***.***y***/** (on Sun SPARC Solaris)

  or:

  /usr/cport/C-Port/Cst*x***.***y***/**

  or:

  /opt/C-Port/Cst*x***.***y***/**

where ‘x’ is a major version number and ‘y’ is a minor (or intermediate) version number.

You typically install each CST version under some directory path **...\C-Port\Cst***x***.***y***\**. However, the user can install the CST in any directory on the development workstation. The user can also install more than one CST version on the same workstation.

Therefore, to refer to installed CST directories, we use pathnames that are relative to the **...\C-Port\Cst***x***.***y***\** path, which is the “root” of a given CST installation.

For example, the **apps\gbeSwitch\** directory path refers to the location of the Gigabit Ethernet Switch application that is installed as part of the CST. The full path of this directory on a Windows NT system might be **C:\C-Port\Cst2.2\apps\gbeSwitch\**, so this convention is convenience for shortening the pathname.

Other top-level directories that are installed as part of the CST include **bin\**, **diags\**, **Documentation\**, **services\**, and so on. These directories are described in the *C-Ware Software Toolset Getting Started Guide* document, which is part of the CST documentation set.

Revision History

Table 2 provides details about changes made for each revision of this guide.

Table 2 *C-Ware Software Toolset Tutorial Workbook* Revision History

REVISION DATE	CST REVISION	CDS REVISION	CHANGES
February 4, 2003	2.2	2.0	Original version.

Related Product Documentation

Table 3 lists the user and reference documentation for the C-Port silicon, C-Ware Development System, and the C-Ware Software Toolset.

Table 3 C-Port Silicon and CST Documentation Set

DOCUMENT SUBJECT	DOCUMENT NAME	PURPOSE	DOCUMENT ID
Processor Information	*C-5 Network Processor Architecture Guide*	Describes the full architecture of the C-5 network processor.	C5NPARCH-RM
	C-5 Network Processor Data Sheet	Describes hardware design specifications for the C-5 network processor.	C5NPDATA-DS
	C-5e/C-3e Network Processor Architecture Guide	Describes the full architecture of the C-5e and C-3e network processors.	C53C3EARCH-RM
	C-5e Network Processor Data Sheet	Describes hardware design specifications for the C-5e network processor.	C5ENPDATA-DS
	M-5 Channel Adapter Architecture Guide	Describes the full architecture of the M-5 channel adapter.	M5CAARCH-RM
	Q-5/Q-3 Traffic Management Coprocessor Architecture Guide	Describes the full architecture of the Q-5and Q-3 traffic management coprocessors.	Q5Q3ARCH-RM
Hardware Development Tools	*C-Ware Development System Getting Started Guide*	Describes installation of the CDS.	CDS20GSG-UG
	C-Ware Development System User Guide	Describes operation of the CDS.	CDS20UG-UG
Software Development Tools	*C-Ware Software Toolset Getting Started Guide*	Describes how to quickly become acquainted with the CST's software development tools for a given CST platform.	CSTGSGW-UG (Windows) CSTGSGS-UG (Sun SPARC Solaris)
	C-Ware Application Building Guide	Describes tools to build executable programs for the C-Port network processors or simulators.	CSTABG-UG
	C-Ware Debugger User Guide	Describes the GNU-based tool for debugging software running on either the C-Port network processorsor simulators.	CSTDBGUG-UG
	C-Ware Integrated Performance Analyzer User Guide	Describes use of theIntegrated Performance Analyzer tool for gathering performance metrics of a C-Port NP-based application running under the simulator.	CSTIPAUG-UG
	C-Ware Simulation Environment User Guide	Describes how to configure and run a simulation of a C-Port NP-based application using simulator tools.	CSTSIMUG-UG

Table 3 C-Port Silicon and CST Documentation Set (continued)

DOCUMENT SUBJECT	DOCUMENT NAME	PURPOSE	DOCUMENT ID
Application Development	*C-Ware Application Design Guide*	Describes design guidelines and trade-offs for implementing new C-Port NP-based communications applications.	CSTAPDG-UG
	C-Ware API User Guide	Describes the subsystems and services that make up the C-Ware Applications Programming Interface (API) for C-Port NP-based communications applications.	CSTAPIUG-UG
	C-Ware Host Application Programming Guide	Describes the CST software infrastructure and APIs that support host based communications applications.	CSTHAPG-UG
	C-Ware Microcode Programming Guide	Describes programming the C-Port network processor's Serial Data Processors and Fabric Processor.	CSTMCPG-UG
	C-Ware Q-5 TMC API User Guide	Provides a reference for the Q-5 Traffic Management Coprocessor (TMC) API. This guide includes a discussion and example of creating a scheduler hierarchy.	CSTQ5API-UG
Other Documents	*Answers to FAQs About C-Ware Software Toolset Version 2.0*	Describes how the directory architecture provided in C-Ware Software Toolset Version 2.0 differs from previous CST releases.	CSTOAFAQ-UG
	Build System Conventions	Describes the key features of the C-Ware Software Toolset's provided environment for building software.	CSTOBSC-UG
	C-Ware Software Toolset Application Guidelines	Describes the criteria for how software components comply with the C-Ware Software Toolset's provided environment for building software.	CSTOCAG-UG

Lesson 1

WORKBOOK OVERVIEW

Overview

The C-Ware Software Toolset (CST) contains a powerful set of software development tools, simulation and performance analysis tools, and sample C-Ware Application Library reference applications that can greatly increase your capability to develop applications for the C-Port family of network processors.

This Tutorial Workbook offers intensive instruction on how to use the Toolset to develop, debug, and tune C-Port NP-target application software.

This Workbook assumes that you are already familiar with the architecture and features of Motorola's C-Port family of network processors.

Lessons

This Workbook is divided into chapter-like sections called *lessons*. Each lesson assists the reader in learning a new set of skills for working with the CST.

Each lesson presents the following:

- **Goal** — The purpose of the lesson; that is, which skills you will learn.
- **Prerequisites** — The knowledge and skills you should bring before performing this lesson.
- **Modules** — A combination of knowledge to learn and steps to follow in order to learn a new skill. Each lesson can present one or more modules.
- **Summary** — A restatement of the purpose of the lesson, how each module in the lesson support the purpose of the lesson, and an explanation of how the new skills you learned in this lesson relate to previously learned skills and to other skills you can learn later in the Workbook.
- **Exercises** — One or more suggested questions to answer, or suggested procedures to follow, that refresh and reinforce the reader's knowledge of the skills presented in this lesson.

Lessons in This Workbook

This Workbook presents the following lessons.

CST High-Level Directories

This lesson presents how the CST's high-level directories and files are organized. Included informational topics are:

- Purpose and contents of all CST high-level directories
- Locations of CST directories that contain C-Ware applications (**apps\**) and components (**apps\components\**), tools (**bin\**), and CST product documentation (**Documentation\**)
- Organization of the CST documentation set
- Locations of directories from which to run CST-supplied applications under the C-Ware Simulation Environment
- How the CST's **apps\***application***\** subdirectories support the CST's build system conventions
- Locations of CST diagnostics programs for the C-5 NP

Building C-5 Applications From Programs

This lesson shows how to build a C-5 NP-targeted application package from a C-Ware application's individual programs, including these modules:

- Building the application's package
- Building the application's XPRC and CPRC programs and microprograms
- Performing dependency checking when building the application
- Building programs that perform offline support for C-Ware applications, such as table building

Simulating the Execution of C-5 Programs

This lesson presents how to use the C-Ware Simulation Environment as a “target” for running your C-Ware applications, including these modules:

- Learning Simulator command primitives
- Identifying the simulated C-5 NP components
- Running simulations using the C-Ware Simulation Environment and C-Ware Simulator tool
- Purpose of the configuration file (**config**) and command input file (**sim.in**)
- Preprocessing the Simulator’s configuration and command input files
- How the Simulator supports C-5 NP component tracing
- Generating diagnostic and trace messages during a simulation

Debugging C-5 RC Programs

This lesson shows how to use the C-Ware Debugger , which is a custom implementation of the GNU **gdb** tool, to debug your C-5 applications, including these modules:

- Understanding high-level Debugger features
- Starting the Debugger, from within the C-5 Simulator or from outside the Simulator
- Understanding how the Debugger interoperates with the Simulator
- Operating the Debugger, including targeting an RC program, setting breakpoints, stepping through code, and distinguishing the contexts of execution for a program in a CP cluster
- Changing the target of a debugging session from one RC to another

Generating Simulated Ingress Traffic

This lesson shows how to use the CST-provided set of Perl language "pattern generation" (or "pg") scripts to produce simulated ingress traffic data. You configure the Simulator to input this data to your C-Ware application as it runs under the C-Ware Simulation Environment. Included modules are:

- Description of all "pg" scripts: pattern-producing, API scripts, and Ethernet framer scripts
- Generating pattern files using "pg" scripts, with 100Mb Ethernet/IP and PPP over SONET examples
- Review of "pattern file" format
- Complementarity of using "pg" scripts and the **printTrace.pl** tool (for post-processing egress pattern files from running applications under simulation)

(An "ingress pattern file" is a representation of a stream of simulated ingress data traffic from a C-Ware application that ran under the Simulator.)

Post-Processing Simulated Egress Traffic

This lesson shows how to use the **printTrace.pl** tool to filter an egress pattern file into a format that presents ATM cell data and payload, or Ethernet packet data and payload, in human-readable format. Included modules are:

- Purpose of "trace files"
- Analyzing trace files
- Examining sample output from the **printTrace.pl** tool

(An "egress pattern file" is a representation of a stream of simulated egress data traffic from a C-Ware application that ran under the Simulator.)

Programming the C-5 Serial Data Processors

This lesson presents how to use the powerful features of the C-5 NP Channel Processor's Serial Data Processor (SDP) microsequencers to support detailed application requirements. This lesson also presents techniques for debugging SDP microcode programs within applications that run under the C-Ware Simulator. Included modules are:

- Overview of SDP architecture
- Overview of the common microsequencer architecture
- Overview of nine SDP microsequencers
- Overview of SDP microsequencer inputs and outputs
- Aids to SDP microcode programming
- Visibility of SDP state in the C-Ware Simulator, including microcode messaging and SDP tracing
- Analyzing SDP performance
- Examining SDP microcode source code

Modifying the Gigabit Ethernet Switch Application

This lesson walks you through several specific steps for making a basic modification to the CST's Gigabit Ethernet Switch (**gbeSwitch**) application.

In this extended exercise, you modify both microcode and CPRC program code to implement a rearrangement of header bytes in a series of Ethernet frames that pass through the C-Ware Simulator as configured to model the C-5 NP Version D0 device.

CST Helpful Hints

This section presents helpful hints for using the features of the CST.

Referring to CST Directory Pathnames

You typically install the CST in a default directory path provided by the installation procedure, such as:

- **C:\C-Port\Cst2.2\** (on Windows NT)
- **/usr/C-Port/Cst2.2/** (on Sun SPARC Solaris)

However, the user can install the CST in any directory on the system. The user can also install more than one CST version on the same system. You typically install those versions under the **...\C-Port\Cst***x.x***** path.

To refer to CST directories in this guide, we use a relative pathname to refer to a "top-level" CST directory—that is, a directory path that is relative to the **...\C-Port\Cst***x.x***** path. The CST's top-level directories typically contain subdirectories as well.

For example, we use the **apps\gbeSwitch\** directory path to refer to the location of the Gigabit Ethernet Switch application that is included in the CST. The full path of this directory on a Windows NT system might be **C:\C-Port\Cst2.2\apps\gbeSwitch\**, so shortening the path by this convention is convenient.

Other top-level directories that are installed as part of the CST include **bin**, **Documentation**, **diags**, **services**, and so on. These directories are described in "Top-Level Directory Contents" on page 25.

Windows NT™ File Completion

In the Windows NT registry, you can define a keyboard key to perform “filename completion” for commands typed at the DOS shell prompt. This is a helpful feature because it allows you to avoid retyping long filenames and directory names, as found in the CST software development environment.

To enable the “filename completion” feature at the DOS command prompt:

1. From the Window desktop, click the ‘Start’ menu and select ‘run...’, then type ‘regedit’ as the command to perform. This opens the Windows registry edit (program name is **regedit**) program.
2. Press **Ctrl+F** to open the Registry Edit tool’s Find dialog box.
3. In this dialog, search for the registry key ‘CompletionChar’.
4. Edit the value of this registry key to be ‘9’. This enables use of the **Tab** key on the keyboard for filename completion inside the DOS shell.

Setting the CST Environment Variables

You might have the need to install more than one version of the CST on your workstation. For instance, this might be required while you port your application from one CST version to another.

To support users who develop C-Ware based applications using more than one CST version, the CST defines and uses operating system environment variables that allow CST tools to differentiate among the directories of different CST installations.

The CST provides the **sv.bat** script (**sv.**[**c**]**sh** script on Sun Solaris™) in its **bin\** directory to set the environment variables used by CST program-building tools. In this Workbook there are instructions on how to use **sv.bat** from a DOS shell to set these variables.

To use a shell other than the DOS shell (such as the *bash* shell for Windows), you can put the set of environment variables that the **sv.bat** script sets in your shell initialization script (such as the **.bashrc** file for the *bash* shell).

If you prefer to use the DOS shell (as demonstrated in this Workbook), you must run the **sv.bat** script as the first command in each new DOS shell window that you open using CST program-building tools.

The syntax for calling the **sv.bat** program is as follows:

```
D:\C-Port\Cst2.2\bin> sv
```

Use this command if there is only one installation of the CST on your workstation, or if the most recent CST installation is the one that you are presently using.

If you have multiple CST installations on your workstation and you want to use a CST version other than the most recently installed version, then invoke **sv** with the path to the other installation:

```
D:\cd \C-Port\Cst2.0\bin> echo %CPORT%

D:\C-Port\Cst2.0\bin> sv d:\c-port\cst2.0
```

Details about using the **sv.bat** *script are found in the* C-Ware Software Toolset Getting Started Guide, *which is part of the CST documentation set.*

Text Editors

As you work with the CST's software development tools, you will open many text files for viewing and modification.

Though Windows NT includes text editors such as Microsoft Word™ and Microsoft WordPad™, these are not necessarily the most effective editors for ASCII text.

We recommend that you install your favorite programmer's text editor on your workstation. Many users prefer *GNU Emacs*, *Epsilon*, or *vi* for text editing.

Lesson 2

EXAMINING THE TOOLSET'S TOP-LEVEL DIRECTORIES

Goals

In this lesson you will learn:

- The purposes of each top-level directory provided by the C-Ware Software Toolset (CST)
- How to identify the locations of CST tools, C-Ware Application Library software, CST services software, C-Ware diagnostics software, and the CST documentation set.

Prerequisites

To perform this lesson's exercises, you should have access to a workstation on which the CST Version 2.2 product has been installed.

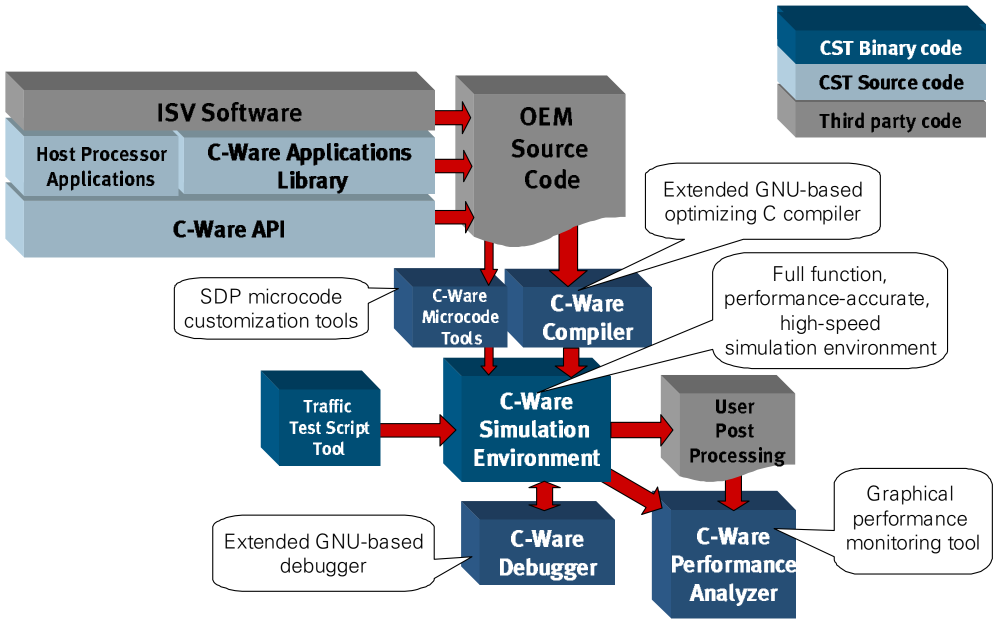
Top-Level Directory Contents

The CST contains a number of tools, software applications, and a set of documentation that helps you develop systems based on the C-Port family of network processors.

This section describes the organization of the software in the CST, as well as where to find the tools, documentation, and applications that will be required for software development.

Figure 1 on page 26 depicts the workflow for using the CST's software development tools to build and work with a C-Ware application.

Figure 1 User Workflow for CST Tools and Applications

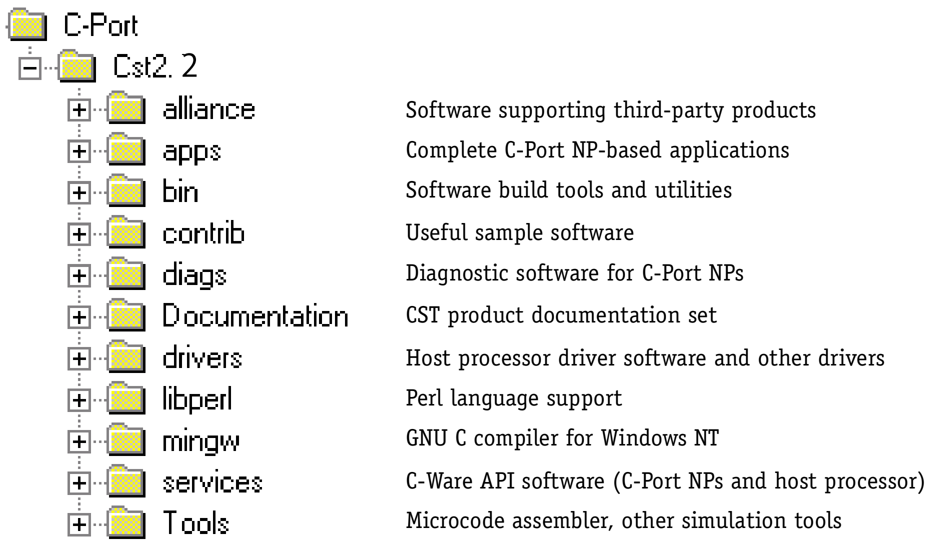

CST Binary code
CST Source code
Third party code
ISV Software
Host Processor Applications
C-Ware Applications Library
C-Ware API
OEM Source Code
Extended GNU-based optimizing C compiler
SDP microcode customization tools
C-Ware Microcode Tools
C-Ware Compiler
Full function, performance-accurate, high-speed simulation environment
Traffic Test Script Tool
C-Ware Simulation Environment
User Post Processing
Graphical performance monitoring tool
Extended GNU-based debugger
C-Ware Debugger
C-Ware Performance Analyzer

Under your CST installation's "root" directory, which is typically a path such as **C:\C-Port\Cst***x.x*****, there are a set of top-level directories shown in Figure 2 on page 27.

Your CST installation's top-level directories might include the 'patch1' directory, which indicates that a *patch release* was also installed on this workstation after a giver CST product version was installed. A patch release updates only certain files in an existing CST installation on your workstation. Its directory contains its own **README** file and installation script.

alliance\ This directory contains software that allows a C-Ware based application to work with products provided by third-party vendors. This is software is provided, documented, tested, and supported by members of Motorola's Smart Networks Alliance Program. See the *CST Release Notes* document for a description of this software.

Figure 2 CST Version 2.2 Installation's Top-Level Directories (Windows NT Platform)



C-Port
Cst2. 2

alliance	Software supporting third-party products
apps	Complete C-Port NP-based applications
bin	Software build tools and utilities
contrib	Useful sample software
diags	Diagnostic software for C-Port NPs
Documentation	CST product documentation set
drivers	Host processor driver software and other drivers
libperl	Perl language support
mingw	GNU C compiler for Windows NT
services	C-Ware API software (C-Port NPs and host processor)
Tools	Microcode assembler, other simulation tools

apps\ This directory contains all software for a set of C-Ware Application Library applications supported by Motorola in this CST release. Most of these applications demonstrate software that implements a Layer 3 or higher protocol on the C-Port family of network processors and support chips. Other applications implement tests that demonstrate basic functional capabilities of the C-5 Network Processor, such as the software found in the CST's **apps\hostDmaTest\** directory.

Each application supports the C-5 NP and/or the C-5e/C-3e NP, either on actual hardware or under the C-Ware Simulator or both.

Each application's software is contained in its own subdirectory under **apps\**. Each application's directory contains these subdirectories:

- **chip\np\** — Contains subdirectories that contain all source code, organized by build target, as explained in the *Build System Conventions* document (**apps\doc\Build_System.pdf**).

  Under each **.\chip\np\** directory subtree, find the application's main XPRC and CPRC programs under the relevant "terminal" (or end node) **.\src\** subdirectory — that is, the **.\chip\np\cprc\src\** and **.\chip\np\xprc\src\** directories. The terminal **.\src\**

directory might be found directly under the **.\chip\np\xprc\** directory or under a build target-specific subdirectory such as **.\chip\np\xprc\c5\** or **.\chip\np\cprc\cxe\**.

The source file names for XPRC and CPRC main programs can depend on how the application uses the network processor's Channel Processors. Look for files with names such as **cpMain.c** (versus **xpMain.c**), **cpMainRx.c** (versus **cpMainTx.c**), **atmCpMain.c** (versus **atmRxMain.c** or **atmTxMain.c**), and so on. C-Ware applications that use a multi-phase (that is, overlaid) RC program might use both the source file names **xpMain.c** and **xpMainInit.c** to implement the two programs phases.

- **doc** — Contains documentation about the application, typically including:
  - A formatted user guide about the application's architecture and functionality
  - A **README** file that summarizes the pertinent information about building the application
- **inc**— Contains C language header (***.h**) files used to build the application that are common to all *build targets* for this application.

For a C-Ware-compliant application, a build target *specifies a combination of chip architecture, chip product, chip generation (or version), chip processor, chip revision, runtime environment, and runtime configuration. These terms are defined and explained in the* Build System Conventions *document (***apps\doc\Build_System.pdf***).*

- **offline** — (Provided only for certain C-Ware Application Library applications) Contains software that is used to specify, create, and populate routing tables in a manner that uses a minimum of network processor cycles. This software reduces the number of cycles performed early in the application when run under the C-Ware Simulator. For a given application, see the file **apps\***application***\doc\Readme** for the details about using offline table building (if available).
- **run** — Contains files that allow you to build the entire application and to run the built application under simulation. That is, this directory contains:
  - The top-level **Makefile** for building the application
  - The package description file (that is, ***.dsc**) for specifying the content of the application's package file

- The pair of ***.expected** and ***.out** files used for this application's installation acceptance test
- The C-Ware Simulator configuration and (optional) default command input file provided by the application developer.

Also contains the subdirectories (under **.\bin**) that organize for each build target the network processor RC and SDP executable files (that is, ***.dcp** and ***.sdp**) and application package file (that is, ***.pkg**) that result from building the application.

Also contains subdirectories (**.\inPatterns** and **.\outPatterns**) that organize the files that contain representation of simulated ingress and egress traffic data. The application developer has provided these files to use (or to capture) when running this application under the C-Ware Simulator.

Also contains these subdirectories:

- **deps** — Collects makefile dependency information produced by the **make** tool during application builds. Used by programmer-friendly text editors.
- **lib** — Collects library (***.a**) files produced by the **make** tool during application builds.
- **obj** — Collects object (***.o**) files produced by the **make** tool during application builds.

The **apps** directory also contains a **.\components** subdirectory that contains the software for each of a set of reusable components that are (or can be) used to build the C-Ware Application Library applications. Each **.\components** subdirectory contains its own subdirectories for building that component; these subdirectories (that is, the **.\doc**, **.\inc**, **.\offline**, **.\run**, and **.\chip\np\** directories) are organized identically to C-Ware applications.

bin This directory contains most of the tools necessary to develop and build applications for the C-Port family of network processors, including running the C-Ware Simulator, and debugging and analyzing applications. You shouldn't need to modify the contents of this directory.

The **bin** directory must also contain a **license.db** file that is required for running the C-Ware Simulator. This license file can be obtained from your Motorola representative.

contrib\ This directory contains a range of software components, from complete applications to bits of source code. These components might have been created by Motorola or by third-party sources. *These applications and components are not supported by Motorola* but are deemed useful to developers of C-Ware applications. Any application provided here has minimal documentation.

The **contrib\archive\** directory contains previously released C-Ware Application Library applications that are *not supported by Motorola* in this CST release. Any application provided here has minimal documentation.

diags\ This directory contains the environment for building a large set of diagnostics programs for testing particular C-Port network processor components. Building each diagnostic results in a C-Ware package file that can be packloaded into an actual C-Port network processor device or loaded into the C-Ware Simulator.

Building and running these diagnostics is described in the *C-Ware Development System User Guide* document.

Documentation\ This directory contains the entire CST documentation set, provided in Adobe Acrobat PDF file format for either online reading or offline printing.

We recommend that CST users first read the *CST Release Notes* document found in this directory. Included in the *Release Notes* is information on features that have been added, removed, or modified in this CST version.

The **Documentation\** directory also contains the **C-Ware_Start.html** file, which is an HTML page with hyperlinks to each item of CST product documentation, which are categorized as listed in Table 1 on page 31.

drivers\ This directory contains the software for "device driver" software to be incorporated into a host processor application program or host operating system (such as the Wind River Systems VxWorks real-time operating system). One important driver software object, the C-5 Device Driver, is described in the *C-Ware Host Application Programming Guide* document.

libperl\ This directory contains software for Perl language support for Windows NT.

mingw\ This directory contains a version of the GNU C compiler (**gcc**) that produces applications that run under Windows NT. Used by the CST's Microcode Assembler tool.

Table 1 Organization of the CST Documentation Set

TYPE OF DOCUMENT	NAME OF DOCUMENT
Release Notes	*C-Ware Software Toolset Release Notes*
Application Development	*C-Ware Application Design Guide* *C-Ware API User Guide* *C-Ware Host Application Programming User Guide* *C-Ware Microcode Programming Guide*
Applications and Components	*AAL-5 SAR Application Guide* *AAL-5 SAR to Gigabit Ethernet Switch Application Guide* *ATM Cell Switch Application Guide* ...
Development Tools	*C-Ware Application Building Guide* *C-Ware Debugger User Guide* *C-Ware Development System Getting Started Guide* *C-Ware Development System User Guide* *C-Ware Performance Analyzer User Guide* *C-Ware Simulation Environment User Guide* *C-Ware Software Toolset Getting Started Guide*
Other Documents	*Answers to Frequently Asked Questions About C-Ware Software Toolset Version 2.0* *Build System Conventions* *C-Ware Software Toolset Application Guidelines* ...

services\ This directory contains all of the C-Ware application programming interface (API) code, including host-resident API code. The C-Ware API routines are described in the *C-Ware APIs User Guide* document and the *C-Ware Host Application Programming Guide.*

This directory's **.\inc\** subdirectory contains all the public header files for the C-Ware APIs.

In the subdirectory under **services\** for a given set of API services (such as **.\buffer\** or **.\queue\** or **.\table\**), look for Host API code under its **.\host\** subdirectory.

The organization of the directory subtrees under **services** parallels that for the C-Ware Application Library applications provided under the CST's **apps** directory. For more information, see the *Build System Conventions* document (**apps\doc\Build_System.pdf**).

Tools\

This directory contains several software objects, some of which support the use of the C-Ware Simulator in a multiple C-Port network processor scenario. See the *C-Ware Simulation Environment User Guide* for more information about using this software.

This directory's **.\common** subdirectory contains other subdirectories that contain software used to perform multiple C-Port network processor system simulations.

This directory's **.\pcisim** subdirectory contains software for simulating a PCI bridge.

Under this directory's **.\uasm** subdirectory are the libraries used by the CST's Microcode Assembler tool.

Summary

In this lesson you learned the purposes of each top-level directory provided by the C-Ware Software Toolset (CST).

In subsequent lessons you will use this knowledge to quickly identify the locations of CST tools, C-Ware Application Library software, CST services software, C-Ware diagnostics software, and the CST documentation set.

Exercises

Here are some exercises to help ensure that you are familiar with the organization of the CST product.

- ❑ **Find and examine the XPRC's main program(s) for the C-5 NP target for the Gigabit Ethernet Switch application.**

  `apps\gbeSwitch\chip\np\xprc\src\xpMain.c and xpMainInit.c`

- ❑ **Find and examine the CPRC's main program for the C-5 NP target for the Gigabit Ethernet Switch application.**

  `apps\gbeSwitch\chip\np\cprc\src\cpMainRx.c (for receive channels) and cpMainTx.c (for transmit channels)`

- ❑ **Find and examine the public header file that contains the interfaces for the Buffer Services portion of the C-Ware API.**

  `services\inc\dcpBufferSvcs.h`

- ❑ **Find and examine the set of pattern files for simulated ingress traffic that are provided with the Gigabit Ethernet Switch application.**

  `apps\gbeSwitch\run\inPatterns\*.pat`

- ❑ **Find and examine the package descriptor file for the Gigabit Ethernet Switch application.**

  `apps\gbeSwitch\run\gbeSwitch.dsc`

- ❑ **Find the package file that is generated after building the debuggable C-5 NP Version D0 for simulation (the c5-d0-sim-debug variant) build target for the Gigabit Ethernet Switch application.**

  `apps\gbeSwitch\run\bin\c5-d0-sim-debug\gbeSwitch.pkg`

- ❑ **Find and examine the C-Ware Simulator configuration file that is used for running a simulation of the Gigabit Ethernet Switch application.**

  `apps\gbeSwitch\run\config`

Lesson 3

BUILDING C-WARE APPLICATIONS

Goals

In this lesson you will learn:

- How the C-Ware Software Toolset's (CST) build system tools support building executable programs from these source files:
  - C language programs for the network processor's Executive Processor (XP) and Channel Processor (CP)
  - C language-like microcode programs for the network processor's Serial Data Processors (SDPs) and Fabric Processor FDPs
- How the CST provides automated building of a C-Ware application's software
- What a C-Ware package file is and how it is built from compiled Executive Processor RISC Core (XPRC) and Channel Processor RISC Core (CPRC) programs and from SDP/FDp microprograms
- What a build target variant is, and how the CST's **apps\** subdirectories segregate an application's compilation products by build target variants
- How the organization of the CST's **apps\** subdirectories support the CST build system
- How the CST's build system provides automatic dependency checking
- How some CST-supplied C-Ware applications provide "offline" programs for running the application more conveniently under the C-Ware Simulator

Prerequisites

You should already be familiar with the organization and purposes of the CST's top-level directories.

To perform this lesson's exercises, you should have access to a workstation on which the CST Version 2.2 product has been installed.

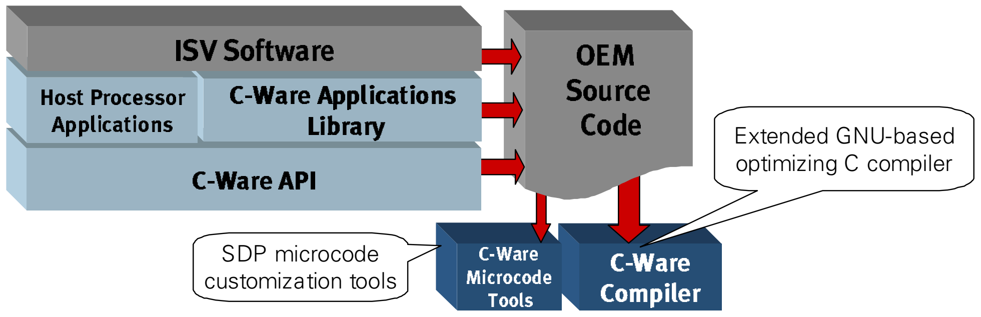
Source Files Correspond to Network Processor Embedded Processors

The C-5 Network Processor (NP) has a number of embedded RISC cores (RCs)—one Executive Processor RISC Core (XPRC) and 16 Channel Processor RISC Cores (CPRCs). In addition, each Channel Processor (CP) includes a microcode programmable Serial Data Processor (SDP, with both a set of ingress microsequences and blocks and a set of egress microsequencers and blocks). The C-5 NP also includes an embedded Fabric Processor (FP) whose Fabric Data Processors (FDPs), one for ingress and another for egress, are microcode programmable.

Each of these programmable entities can run a different software program. For example, there will be different software that runs in the XPRC, CPRCs, Channel Processor SDPs, and Fabric Processor FDPs.

The C-Ware Software Toolset (CST) allows you to use C language (and C language-like) source files for all these software programs and provides build tools to produce executable image files for each programmable entity.

Figure 3 Source Code Packaging


ISV Software
Host Processor Applications
C-Ware Applications Library
C-Ware API
OEM Source Code
Extended GNU-based optimizing C compiler
SDP microcode customization tools
C-Ware Microcode Tools
C-Ware Compiler

Automated Building of Packages

This CST product provides complete automation of the program building activity. Each C-Ware Application Library application can be built using one high-level makefile, found in the file **apps\***application***\run\Makefile**. This makefile produces a package file for the application.

A *package file* (that is, ***.pkg**) contains a concatenation of the application's executable images. Each image is downloaded into the C-5 NP processors when the chip is booted. When the C-5 NP device is installed in a C-Ware Development System (CDS), the use can "packload" the package file into the NP, or the package file can be declared in a configuration file (**config**) for loading into the C-Ware Simulator for application simulation.

To produce this package file, this makefile uses other makefiles to build an XPRC program executable (that is, a MIPS-like ELF file), some number of CPRC program executables (that is, MIPS-like ELF files), some number of Channel Processor SDP microprogram executables (that is, CAM contents, WCS contents, and so on), and Fabric Processor FDP microprogram executables. The software components used to build these executables are placed under the appropriate subdirectories found under the application's **apps\***application***\chip\np\** directory.

When the network processor is taken out of reset, it is loaded with the contents of a package file. Or, when you start the C-Ware Simulator, you typically configure it to load a package file that is specified in a Simulator configuration file (**config**). Under software control (typically in an "init phase" program that is loaded first into the XPRC), the package's contents are loaded and distributed to the appropriate C-5 NP embedded processors, memories, and so on.

The application's makefile leaves the package file in a "build target variant" subdirectory of the **apps\***application***\run\** directory. See the section "More About Package Files and Build Target Variants" that follows.

After you have set this command shell process's build target variant *environment variables*, described in the next section, you typically enter only one command to build the application:

```
C:\C-Port\...\application\run> make all
```

For a list of all makefile arguments (and their purposes) supported by an application's top-level **Makefile**, see the **Readme** file found in the application's **.\doc\** subdirectory.

More About Package Files and Build Target Variants

The application's package file is the ultimate product of using the **make** tool against the application's top-level makefile. The package file is typically placed in the specified build target variant's directory— that is, the **apps\***application***\run\bin\variant\** directory.

The application's *build target variant* is an identifier that segregates different builds of the application based on different combinations of several parameters that relate to the application's execution target—that is, whether the package is loaded into:

- A host processor or a C-Port network processor
- If a C-Port network processor, which C-Port network processor device
- If a C-Port network processor, which version of that device
- If a C-Port network processor, whether an actual NP device or the Simulator
- Whether the package was built for debugging or production use

The exact design of the CST's build target variant scheme is described in the CST's Build System Conventions *document.*

The CST build system's makefiles depend on a set of environment variables for specifying the build target variant. Use the command scripts found in the **bin\configure\** to set these variables for a given build target variant. Alternatively, you can specify the build target variant for the next build on the **make** command line, as follows:

```
C:\C-Port\...\gbeSwitch\run\> make REV=c5-d0-sim-debug
```

Of course, set (or specify) the appropriate build target variant for the **apps\***application***\run\** subdirectory where you issue the make command.

The CST provides the **dcpPackage** tool for creating a package from a set of MIPS-like ELF images for the XPRC and CPRC, SDP images, FP images, and so on. Find the **dcpPackage** tool in the CST's **bin\** directory.

All makefiles that build the executable components of a C-5 NP application are set up to build their respective images and to use **dcpPackage** to produce a C-5 NP package file.

Using the dcpPackage *tool and specifying the contents of a package description file are discussed in the CST's* C-Ware Application Building Guide *document.*

Compiling the Application's Programs

The application's top-level makefile automatically rebuilds each constituent program or microprogram with a dependency whose instance has changed.

You typically build C language programs to run on the XPRC or CPRCs. C language-like source files are used to express microcode programs for the Channel Processors' SDPs and Fabric Processor's FDPs.

The C language programs and the microcode programs use different compilation steps, as described in the following sections.

Notice these important facts about each application's programs and their compilation products:

- Source code that is common to the application's XPRC and CPRC programs is typically located under a subdirectory named **apps\***application***\\chip\\np\\src\** and/or **apps\application\\chip\\np\\inc\**.
- The compilation products of the application's programs, as binaries and like the application's package file, are left in subdirectories that are segregated under build target variants. That is, each application has its own **apps\***application***\\run\\obj\variant\** and **apps\application\\run\\lib\variant\** subdirectory trees.

XPRC Programs

Source code that is unique to the application's XPRC program is typically located under a subdirectory named **apps\***application***\\chip\\np\\xprc\\src\**.

The object files (***.o**) and library files (***.a**) that result from a build are typically placed in the **apps\***application***\\run\\obj\variant\\xprc\** and **apps\application\\run\\lib\variant\\xprc\** directories, respectively.

The application's XPRC executable file (***.dcp**) and its corresponding map file (***.map**) are typically placed in the **apps\***application***\\run\\bin\variant\** directory.

Note that the application typically uses a naming convention for the XPRC executable that includes the application identifier and the suffix "Xp", such as **gbeSwitchXp.dcp**.

CPRC Programs

Source code that is unique to the application's CPRC program(s) is typically located under a subdirectory named **apps\***application***\chip\np\cprc\src\**.

Notice that some applications will require more than one CP program, when different Channel Processors (CPs) on the C-5 NP are used for different purposes.

For example, in one application some CPs are dedicated to processing ingress traffic and others to egress traffic; in a different application certain CPs in the same CP cluster might process ingress traffic in a tightly synchronized manner, while other CPs in another CP cluster that also process ingress traffic do not explicitly cooperate with each other or with other clusters. In these cases you should expect to find (or to provide) different CP programs and their source files should follow a naming convention that distinguishes their respective roles.

The object files (***.o**) and library files (***.a**) that result from a build are typically placed in the **apps\***application***\run\obj\variant\cprc\** and **apps\application\run\lib\variant\cprc\** directories, respectively.

The application's CPRC executable file(s) (***.dcp**) and its corresponding map file (***.map**) are typically placed in the **apps\***application***\run\bin\variant\** directory.

Note that the application typically uses a naming convention for the CPRC executable(s) that includes the application identifier and the suffix "Cp", such as **oc3SarQCp.dcp** in the AAL-5 SAR application.

For an application that uses different CPs to handle different protocols, look for (or provide) a naming convention that distinguishes the executable files, such as **enetCp.dcp**, **gbeRxCp.dcp**, **gbeTxCp.dcp**, and **posCp.dcp** in the Packet Over SONET to Gigabit Ethernet Switch application.

Microcode Programs

Microcode programs (or *microprograms*) that are targeted to run on the Channel Processor SDPs or on the Fabric Processor's FDPs are typically located under a subdirectory named **apps\***application***\chip\np\sdp\src\**.

There is a distinct microprogram for each SDP microprocessor or FDP microprocessor that participates in the application's processing of ingress and egress traffic. Notice the naming conventions that each C-Ware Application Library application uses to differentiate its SDP and FDP microprograms.

SDP and FDP microprograms are built in a different fashion from XPRC and CPRC programs. Specifically, the output of compilation of microcode is a binary file that contains writable control store (WCS) and CAM contents. In contrast, the output of a program for one of the RCs is a binary file in MIPS-like ELF format.

The compilation for SDP microprograms happens in two phases:

1. Compilation of the program by the C-Ware Microcode Assembler (which is actually a C compiler) to verify syntax and generation of an executable that is native to the development workstation (that is, Windows NT or Sun SPARC Solaris).
2. Execution of that workstation-native executable to produce a binary image of the SDP's or FDP's WCS and CAM contents that become included in the application's package file.

By default, the CST's Microcode Assembler suppresses output of all build-time messages except error messages. The only information echoed to standard output is the execution of the executable that produces the WCS and CAM contents.

A by-product of every microprogram build is a log file (***.log**) that lists all output produced by the Microcode Assembler. This file is placed in the directory where the microcode executable file is written. The log file lists:

- The names and offsets of each label defined in the microprogram
- The compiled form of all CAM contents
- The number of CAM entries that were compiled
- The number of microcode instructions used in the program and the percentage of the WCS that the program uses
- The ID number of each general purpose register for the target SDP/FDP microprocessor that was not used.

Dependency Checking

An important task of a build system for software development is to ensure that, when producing an executable image, the object files are up to date with respect to the other object files in the application — especially those that share structures in header files. The CST provides this feature automatically by specifying *dependency checking* in each Makefile that builds an executable file.

The CST-supplied **cport-apps-rules.mk** and **cport-rules.mk** files contain the support for dependency checking. No changes are required to application makefiles in order to utilize this CST build system feature, as long as these two make include files are included.

The build system automatically generates dependency information for all files that are included in the build on a per-variant basis. It does this by putting the dependency information in the subdirectories under the application's **.\run\deps\** subdirectory.

The following directory tree shows where the dependency information is stored:

```
application\
   chip\
      np\
         cprc\
            c5\
            ...
   ...
   doc\
   host\
   offline\
   run\
      deps\
         c5-d0-sim-debug\
            cprc\
            xprc\
            sdp\
            fdp\
         ...
```

The first time you build the application, certain directories (those shown above in boldface) are populated with the dependency checking information, so that subsequent builds require only a rebuild of the source files when a dependent file has changed (such as a header file).

The dependency check information is contained in files named ***.d** in the appropriate directory.

Building Programs That Perform Offline Support for Applications

Some C-Ware Application Library applications offer software that performs “offline” functionality, such as software that builds a utility (or other code that will be built into the application) for specifying the static contents for the application’s routing tables. You can use this software to reduce the number of network processor cycles used early in the application’s execution when run under simulation. *This feature is not provided for all C-Ware Application Library applications.*

Notice whether each CST-provided application provides an **.\offline** subdirectory. If so, move to a subdirectory of **.\offline** that contains its own **Makefile**. Use the **make** tool to execute that makefile to build the application’s offline software.

The procedures for building and using the application’s offline software are described in the application’s **Readme** file—that is, the **apps\***application***\doc\Readme** file.

Summary

In this lesson you were introduced to how the source files for a C-Ware application are organized and to the CST’s build system commands. You were also introduced to the notion of a build target variant and learned that the CST’s implementation of build target variants depends upon a set of environment variables. You also examined the CST directory scheme that supports building C-Ware applications for C-Port network processors.

You will use this knowledge in subsequent lessons to navigate an application’s directory tree to setup a simulation and debugging environment for working with a C-Ware application.

You will also use this knowledge in a later lesson as you rebuild a new version of the CST-supplied Gigabit Ethernet Switch application that you have modified.

Exercises

The following exercises help you become familiar with the CST's program building tools for C-Ware applications.

- ❏ **Locate the "c5e-a1-sim-debug" build target variant for the Gigabit Ethernet Switch application and rebuild its application package.**

```
C:\C-Port\...\gbeSwitch\run\> make REV=c5e-a1-sim-debug
```

- ❏ **Follow the directions given in the file** apps\gbeSwitch\doc\Readme **to build the Gigabit Ethernet Switch application's offline table building code.**

```
// Set the build target variant first, for subsequent make commands
C:\C-Port\...\gbeSwitch\offline\cxe\>
   C:\C-Port\...\bin\configure\c5e-a1-sim-debug.bat

// Build the tables.exe executable file
C:\C-Port\...\gbeSwitch\offline\cxe\> make REV=c5e

//Run tables.exe to produce the files tle_writes.h and Tlu.state
C:\C-Port\...\gbeSwitch\offline\cxe\> tables.exe

// Use this tle_writes.h file to replace the same file found in
//   ...\apps\gbeSwitch\chip\np\xprc\cxe\inc\
C:\C-Port\...\gbeSwitch\offline\cxe\>
   copy tle_write.h ..\..\chip\np\xprc\cxe\inc\

// Build the Gigabit Ethernet Switch application again
C:\C-Port\...\gbeSwitch\run> make

// (The next run of this application under the C-Ware Simulator uses
// the table data specified in the file
// ...\apps\gbeSwitch\offline\cxe\tables.c.)
```

Lesson 4

SIMULATING THE EXECUTION OF A C-WARE APPLICATION

Goals

In this lesson you will learn:

- That the key components of the C-Ware Simulation Environment (CSE) are the C-Ware Simulator (**bin\cwsim.exe**), the pattern generation Perl scripts (**bin\pg*.pl)**, the **bin\printTrace.pl** tool for filtering pattern files into a human-readable format, and other CST-provided software objects that support certain kinds of "system" simulations
- What are the featured capabilities of the C-Ware Simulation Environment (CSE)
- How to start and stop the C-Ware Simulator
- How to direct the C-Ware Simulator at start-up to emulate a particular C-Port network processor (NP)
- The purpose of the contents of the Simulator's configuration file and command input file
- How commands in the Simulator's command input file are performed
- How preprocessor commands alter the flow of execution of a Simulator command input file
- How to halt the Simulator interactively
- What are the typically used Simulator commands
- How to navigate the Simulator's model of a C-Port NP's components
- How to direct the Simulator to produce "trace" information about the state of a C-Port NP's embedded components
- How to view performance-related counters that are maintained by the Simulator
- What are the alternative Simulator techniques for viewing application-generated messages during a simulation session

Prerequisites

Before attempting this lesson, you should already be familiar with the following:

- The purposes of the C-Ware Software Toolset's (CST) top-level directories
- The organization of a C-Ware application's subdirectory tree
- How to build a particular variant of a C-Ware Application

To perform this lesson's exercises, you should have access to a workstation on which the CST Version 2.2 product has been installed.

Components and Capabilities of the C-Ware Simulation Environment

To allow you to develop an application (or any software) for a C-Port network processor (NP) before an actual device is available, the CST provides the C-Ware Simulation Environment (CSE). The CSE is a set of loosely integrated tools for target system simulation. The tool at the "core" of the CSE is the C-Ware Simulator.

Starting the Simulator

As mentioned in Lesson 3, when preparing to run a C-Ware application under simulation, you typically start the Simulator in the application's **.\run** subdirectory.

See the CST's *C-Ware Simulation Environment User Guide* for an explanation of the command-line arguments to **cwsim.exe**, **c5sim.exe**, and **cxesim.exe**.

Using the cwsim.exe Wrapper

To start the C-Ware Simulator, run **bin\cwsim.exe**:

```
C:\C-Port\...\application\run> cwsim
```

cwsim.exe automatically invokes a more specific Simulator tool—either **c5sim.exe** (for simulating the C-Port C-5 NP) or **cxesim.exe** (for simulating the C-Port C-5e NP or C-3e NP)—based on the current settings of your command shell's CST environment variables. This **cwsim.exe** feature makes it convenient to switch your development efforts among applications that are targeted to different C-Port network processors and therefore use different C-Ware Simulators.

Starting a Simulator Explicitly

Alternatively, you can start the C-5 Simulator (**bin\c5sim.exe**) or the C-5e/C-3e Simulator (**bin\cxesim.exe**) explicitly by entering one of these commands:

```
C:\C-Port\...\application\run> c5sim
```

or:

```
C:\C-Port\...\application\run> cxesim
```

Invoking **c5sim.exe** or **cxesim.exe** directly can be convenient when you are developing applications for only one C-Port NP at a time. However, invoking a Simulator directly is not recommended if you must frequently switch your attention between developing C-Ware applications for the C-5 NP and the C-5e/C-3e NP.

At the command line prompt, you can invoke **c5sim.exe** and **cxesim.exe** using either a fully qualified pathname or using only its command name (via your workstation's normal PATH variable mechanism).

Automatic Use of the Configuration File and/or Command Input File

Regardless of which C-Ware Simulator you start, it automatically looks for and uses a configuration file (typically named **config**) and a command input file (typically named **sim-c5.in** or **sim-c5e.in**).

You can specify a configuration file or command input file with a different name or path in respective Simulator command-line switches.

Each line in the configuration file contains a keyword followed by one or more arguments. These keywords determine the settings for several run-time Simulator parameters, the locations of pattern files to contain simulated ingress and egress traffic, whether to preload certain data into the simulation, and several other parameters.

The command input file contains a series of command lines for the Simulator to execute in sequence, with these capabilities:

- No branching among the command lines is supported.
- If the last command in the file is 'quit', then the Simulator stops reading the command input file, stops the simulation, and exits.
- If there is no 'quit' command in the command input file, after the last command in the file is performed, the Simulator prompts the user interactively for the next command to perform.

Featured Capabilities of the C-Ware Simulator

The C-Ware Simulator is an interactive tool that offers access, via a command-line interface, to a *performance-accurate* simulation of the following C-Port network processors:

- C-5 NP Version D0 (Versions A0, B0, and C0 also supported)
- C-5e NP Version A1
- C-3e NP Version A1

Awareness of CST Build System Environment Variables

After you use the **sv** command script to set your CST environment variables (as you do to set the build target variant before building an application), on startup the Simulator automatically starts the specific Simulator tool for the C-Port NP model and version specified in those variables.

Alternatively, you can specify the '-rev *revspec*' command line switch to the **cwsim.exe** command, where revspec can be one of:

- **c5-d0** (or **c5-a0** or **c5-b0** or **c5-c0**)
- **c5e-a1**
- **c3e-a1**

However, if you specify a revspec when starting the Simulator, you must ensure that the build target variant of the application to be run under simulation *matches* the following:

- The current settings of the CST's build environment variables
- The C-Port NP model and version that the new Simulator session is modeling

Interactive Simulator Control

Press the **Ctrl-c** key to halt (without resetting) a running simulation. This leaves the simulation in the same state as if you had directed the simulation to proceed for a given number of NP clock cycles, as supported by the Simulator's 'go' command.

Support for Simulated Ingress and Egress Traffic

The Simulator models the C-Port NP's Channel Processor ports and Fabric Processor ports so that they support input and output of pattern files. A *pattern file* is a CST-defined specification for simulating a raw data stream.

Each pattern file is an ASCII text file in the following format:

```
20.000000 0 0 0 0
20.000000 0 0 0 0
20.000000 0 0 0 0
...
```

The first column is the time in nanoseconds that elapsed since the previous entry in the file. A file is always assumed to start at time zero. (This example is for 100Mbit Ethernet, so the RMII bit-pairs are specified as 20 nanoseconds apart.) The second column is the traffic data. (In the above example they are all zeros, as would be the case during an inter-frame gap for Ethernet.) The remaining three columns of data are reserved.

Supported by Pattern Generation Tools

The CSE includes a set of Perl scripts, called *pattern generation* (or **pg*.pl**) scripts, that programmatically produce pattern files suitable as simulated ingress traffic to the Simulator. Other CSE tools also support this pattern file specification.

The CSE also includes the **bin\printTrace.pl** tool for filtering a pattern file into a human-readable format. This is useful for examining the output from the CST's pattern generation scripts as well as the egress traffic streams produced by your network processor application running under simulation.

For example, the **printTrace.pl** tool filters pattern files that contain ATM cells and Ethernet frames so that they appear as follows:

ATM:

```
time=11419.751 vpi=21 vci=21 pti=0 clp=0 HEC=197
Payload String =[Vc3 #1 in(Cp0 21/21) out(Cp1 28/28) ]
Payload Numeric=[566333202331206 96e28437030203231 2f323129206f7574
28437031203238 2f32382920202020202020202020202020]
```

Ethernet:

```
MAC da:000000000007 MAC sa:000000000018 type:0800 crc:51275b5c
IP sa:868d0701 IP da:868d0701
Payload
Numeric=[4500002e000000000e0191b3868d0701868d0701202a2a2a2a5061636b65
742f43656c6c2023302a2a2a2a200000]
Time Elapsed: 5760 ns
```

Modeling the C-Port Q-5 Traffic Management Coprocessor

The C-Ware Simulator includes the capability to model the state of the C-Port Q-5 Traffic Management Coprocessor (TMC) device. You determine how the Q-5 TMC is initialized via settings in the Simulator configuration file.

Modeling the C-Port M-5 Channel Adapter

The C-Ware Simulator includes the capability to model the state of the C-Port M-5 Channel Adapter device. You determine how the M-5 Channel Adapter is initialized via settings in the Simulator configuration file.

The simulated M-5 Channel Adapter can produce its own trace output, can consume simulated ingress traffic in the form of CSE pattern files, and can produce simulated egress traffic in the form of CSE pattern files.

Preloading Executables in XP, CP, SDPs, and FP

When starting an application, the Simulator performs a network processor boot sequence that closely mimics the actual chip's boot process. During this process the XP executes code that loads portions of the specified package file, then downloads the appropriate executable images from the loaded package to the various CPRCs, to the Channel Processor SDPs, and to the Fabric Processor's FDPs.

When running application simulations, you might prefer to avoid performing this one-time activity of handling the package in the XP and CPs. To do so, the Simulator can "preload" the contents of the specified package into the XP, CPs, and FP.

This behavior is enabled in the application's Simulator configuration file.

Preloading a Saved State of the NP's Table Lookup Unit

The Simulator can preload a set of state for the network processor's Table Lookup Unit (TLU), via settings in the application's Simulator configuration file.

Preloading a Saved State of the NP's Buffer Management Unit

The Simulator can preload a set of state for the network processor's Buffer Management Unit (BMU), via settings in the application's Simulator configuration file.

Starting the C-Ware Debugger Within the Simulator

To support convenient starts and stops of debugging activity for a simulated application, the Simulator offers commands for starting a Debugger session after the application is loaded. See Lesson 5 for instruction in using the C-Ware Debugger.

Running Simulations Under Control of CWIPA Performance Analyzer

The Simulator can be started, controlled, and stopped under the control of the CST's C-Ware Integrated Performance Analyzer (CWIPA) tool.

Supporting External Loopback for SDPs and the Fabric Processor

The Simulator supports an external loopback capability for the network processor's Channel Processor's Serial Data Processors and Fabric Processor. This capability is intended to be roughly analogous to using a loopback cable on real hardware.

When enabled for an SDP, traffic flows directly from a given SDP's TxPinLogic block to the same SDP's corresponding RxPinLogic block. Similarly, when enabled for the FP, traffic flows directly from the FP's TxPinLogic block to its corresponding RxPinLogic block.

You enable/disable this capability via the network processor application's Simulator configuration file; no other software modifications are required.

Other CSE Components and Capabilities

The CSE incorporates other software tools that, when used together in certain combinations and configurations, enable you to perform "system" simulations.

Simulating PCI Bus Transactions To/From the Network Processor

Although the C-Se doesn't support simulation of a host processor, the C-Se includes a software tool for modeling PCI bus transcations. See the descriptions of the CST's **pcisrv** and **tmhost** tools in the CST's *C-Ware Simulation Environment User Guide* document.

Use the CST's **pcisrv** and **tmhost** tools to simulate PCI transactions to/from a simulated C-Port network processor. (**tmhost** simulates the API of the Q-5 Traffic Management Coprocessor (TMC)).

Providing Blocking TCP Clients and Servers to External Processes

In the typical configuration the C-Ware Simulator automatically creates internal threads that support blocking TCP clients and TCP servers on each configured network processor Channel Processor port.

Alternatively, you can configure the Simulator to open socket connections and accept connections on one or more RxSDP ports and to connect on one or more TxSDP ports (more specifically, after the point in time during the simulation at which each such port is enabled) to another process running on your development workstation. Each such process would run a separate program, provided and configured by you, that establishes a socket connection to the Simulator on a given C-Port NP port.

Each simulated network processor receive port blocks until the socket connection is made with the external process. (Note that accepting connections takes place only after the SDP receive port is enabled.) The transmit port blocks on an *accept()* socket call and waits for an external process to connect.

Supporting Nonblocking I/O for Socket Interfaces

The C-Ware Simulator also offers nonblocking I/O capability for users who write their own Tx client and Rx server programs.

This capability is supported by the *RxPortServer* and *TxPortClient* classes defined in the CST's **Tools\common\simIO\portConnection.h** file. These classes provide an interface to the C-Port NP's SDPs that can be programmed into your own tools for producing and consuming simulated NP ingress and egress traffic.

System Simulations That Are Supported

The CSE supports simulations of these combinations of C-Port network processors and other C-Port products:

- C-5 NP Version D0 (no coprocessor or adapter support)
- C-5e NP Version A1 with Q-5 TMC Version A1 and/or with M-5 Channel Adapter Version A0
- C-3e NP Version A1 with Q-5 TMC Version A1

The C-5e supports simulations of "systems" comprised of the following modeled system components and with the following interactions among components:

- Configuration A (hostless, no C-Port coprocessor or C-Port adapter) — A single network processor running a C-Ware package, with some number of network processor Channel Processor ports in use for traffic ingress and/or egress
- Configuration B (hostless, with C-Port coprocessor and/or C-Port adapter) — Same as Configuration A, except that the Simulator is configured to model the C-Port Q-5 Traffic Management Coprocessor (TMC) device and/or the C-Port M-5 Channel Adapter device (*Note: Only the C-5e NP and C-3e NP can interoperate with the Q-5 TMC and/or M-5 Channel Adapter.*)
- Configuration C (PCI-aware, no C-Port coprocessor or C-Port adapter) — Same Configuration A, except that the NP can produce and consume data across the Executive Processor's PCI interface.
- Configuration D (PCI-aware, with C-Port coprocessor and/or C-Port adapter) — Same Configuration B, except that the NP can produce and consume data across the Executive Processor's PCI interface.

- Configuration E (hostless and back-to-back) — Two network processors of the same C-Port model, where:
  - The two NPs are set up in a manner corresponding to any combination of Configuration A and Configuration B.
  - The two NPs exchange traffic with each other via their respective Fabric Processor ports, each of which is configured to operate in “back-to-back” mode.
- Configuration F (PCI-aware and back-to-back) — Two network processors of the same C-Por t model, where:
  - The two NPs are set up in a manner corresponding to any combination of Configuration C and Configuration D.
  - The two NPs exchange traffic with each other via their respective Fabric Processor ports, each of which is configured to operate in “back-to-back” mode.

Table 2 summarizes the properties of the alternative systems configurations that are supported by the CST’s C-Ware Simulation Environment. In Table 2 the letters in the first column correspond to the configurations detailed in the list preceding this paragraph. A checkmark indicates the presence of the component named at the head of the column in the configuration named in the leftmost column.

Table 2 Alternative Systems That Can Be Modeled Using the C-Ware Simulation Environment

	COMPONENT CAPABILITIES AND INTERACTIONS			
DISTINCT CONFIGURATIONS	**HOSTLESS**	**COPROCESSOR, ADAPTER**	**PCI-AWARE**	**BACK-TO-BACK**
A	✔			
B	✔	✔		
C			✔	
D		✔	✔	
E	✔	✔		✔
F		✔	✔	✔

Understanding Simulator Commands

Table 3 summarizes the commands available in the C-Ware Simulator.

Table 3 Simulator Command Groupings

COMMAND GROUPING	COMMAND	ACTION
Stop the Simulator	**quit**	Stop and end the Simulator process.
Simulator information	**help**	Display help about Simulator command syntax.
	version	Display version information about the Simulator.
Debugging support	**gdb**	Start the C-Ware Debugger in a separate process and attach it to the executable on the specified XPRC or CPRC in this Simulator session.
	symbols	Load microcode symbol information into the Simulator for the specified SDP or FDP from the specified file.
Start/stop NP cycle activity	**go**	Advance the entire simulation for a specified number of NP cycles, or perform NP cycles until a specified register has the specified value.
	hold	Suspend execution of the specified component and reset it.
	release	Allow execution of the specified component that was reset using 'hold'.
	reset	Reset the NP to its post-boot initial state
Navigate to NP components	**moveto**	Change the NP component that is the target of commands.
Change an NP component's contents	**deposit**	Deposit data into a component.
Load NP component's executable	**load**	Load an executable into the specified Channel Processor, Executive Processor, Channel Processor SDP, or Fabric Processor FDP.
Save/restore an NP component's entire state	**restore**	Load previously saved state into a component from a file.
	save	Save a component's state into a file.

Table 3 Simulator Command Groupings (continued)

COMMAND GROUPING	COMMAND	ACTION
Information about NP components	**clock**	Display the current count of network processor core clock cycles that have elapsed during this simulation session.
	counter	Display values of all performance-related counters for the NP's XPRC and CPRCs.
	examine	Display the value at the specified address (or range of addresses) in the specified component.
	list	Display the subcomponents of the specified component.
	trace	Set the degree (0 to 4) of listing verbosity for a component.
	umsg	Enable/disable microcode message for the specified SDP or FDP microsequencer.
	utilization	Display utilization information for a component.

Navigating Among Simulated NP Components

The C-Ware Simulator models the network processor's components as a hierarchy of objects. For example, after starting the Simulator but before navigating to a particular component, you can list the following objects:

```
Dcp> list
Dcp
  RingBus
  Tlu
  Qmu
  Bmu
  XpCluster
  CpCluster0
  CpCluster1
  CpCluster2
  CpCluster3
  FpCluster
  GlobalBus
  PayloadBus
```

These are the "top-level" network processor components that the Simulator models.

To obtain more detailed information from one of the components, use the Simulator's 'moveto' command to navigate to that component . Next, you can display all the components at that level. For example:

```
Dcp> moveto cpcluster0
Dcp.CpCluster0> list
CpCluster0
  ImemStorage0
  Cp0
  Cp1
  Cp2
  Cp3
Dcp.CpCluster0> moveto cp0
Dcp.CpCluster0.Cp0> list
Cp0
  GbNode0
  CpRc0
  Dmem0
  Creg0
  Sdp0
  PbNode0
  Imem0
Dcp.CpCluster0.Cp0> moveto cprc0
Dcp.CpCluster0.Cp0.CpRc> list
CpRc0
  gp, sp, fp, ra,
   0..31, v0..v1,
  a0..a3, t0..t7,
  s0..s7, t8..t9,
  k0..k1
```

The Simulator's 'help' command provides additional information about the relevance of the current NP component (or "context") for entering Simulator commands.

Running Simulations

You must run a simulation from a directory that also contains several other Simulator-related files. These files include:

- The Simulator configuration file, typically named **config**
- (Optionally) A Simulator command input file, typically named **sim-**[NPvariant]**.in**, used for running a series of Simulator commands in "batch" mode

These files can refer to other files used during the simulation, such as the application's ingress and egress pattern files (***.pat**). These files might reside in other directories. See the application's Simulator configuration file for the details.

Simulator Configuration File

The application's Simulator configuration file (typically named **config**) is the most important file that supports running the simulation. It specifies a list of *Simulator configuration variables* that specify information such as:

- System parameters, such as those related to the CSE's External Clock Server or an optional external Trace Client, that relate to extensible Simulator features including multiple NP simulations and "system" simulations as described in "System Simulations That Are Supported" on page 53
- Load-time parameters, such as the boot flags and saved state files, which parts of the NP preload with static data
- "Dialable" simulation parameters, such as core clock rate, port speeds, and QMU/Q-5 configuration
- File paths, such as those for the application's package file, the Simulator command input file (sometimes named **sim-***NP_variant_component***.in** ,where *NP_variant_component* can be 'c5' or 'c5e'), pattern files for ingress and egress traffic, and Simulator console output

Figure 4 on page 60 lists the contents of the Simulator configuration file for the Gigabit Ethernet Switch (**gbeSwitch**) application.

Figure 4 Simulator Configuration File for Gigabit Ethernet Switch Application

```
XpBootFileName $(CPORT)/apps/gbeSwitch/run/bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbeSwitch.pkg
DcpInstance 0
DcpCoreClockRate 200
ApplicationEventDetail 0
InfoMsg 0

#ifndef FULL_BOOT
BootFlags PreLoadAll

// Preload SDRAM and skip code to copy PROM to SDRAM
McSdramFileName $(CPORT)/apps/gbeSwitch/run/bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbeSwitch.pkg
BootFlags SdramLoaded
#endif

sdp0_TxDrainRate 8
sdp0_RxPatternFile $(CPORT)/apps/gbeSwitch/run/inPatterns/gbeAccept0.pat
sdp0_TxTraceFile $(CPORT)/apps/gbeSwitch/run/outPatterns/$(CPORT_MODEL)/gbeTrc0.pat

Sdp1_TxDrainRate 8
Sdp1_TxTraceFile $(CPORT)/apps/gbeSwitch/run/outPatterns/$(CPORT_MODEL)/gbeTrc1.pat

Sdp2_TxDrainRate 8
Sdp2_TxTraceFile $(CPORT)/apps/gbeSwitch/run/outPatterns/$(CPORT_MODEL)/gbeTrc2.pat

Sdp3_TxDrainRate 8
Sdp3_TxTraceFile $(CPORT)/apps/gbeSwitch/run/outPatterns/$(CPORT_MODEL)/gbeTrc3.pat

Sdp4_TxDrainRate 8
Sdp4_TxTraceFile $(CPORT)/apps/gbeSwitch/run/outPatterns/$(CPORT_MODEL)/gbeTxBase4.pat

Sdp5_TxDrainRate 8
Sdp5_TxTraceFile $(CPORT)/apps/gbeSwitch/run/outPatterns/$(CPORT_MODEL)/gbeTrc5.pat

Sdp6_TxDrainRate 8
Sdp6_TxTraceFile $(CPORT)/apps/gbeSwitch/run/outPatterns/$(CPORT_MODEL)/gbeTrc6.pat

Sdp7_TxDrainRate 8
Sdp7_TxTraceFile $(CPORT)/apps/gbeSwitch/run/outPatterns/$(CPORT_MODEL)/gbeTrc7.pat

sdp8_RxPatternFile $(CPORT)/apps/gbeSwitch/run/inPatterns/gbeAccept8.pat
Sdp8_TxDrainRate 8
Sdp8_TxTraceFile $(CPORT)/apps/gbeSwitch/run/outPatterns/$(CPORT_MODEL)/gbeTrc8.pat

...
[Lines removed]
...

BmuSdramSize 0x00650000
```

Command Input File

When it is launched, the Simulator reads the configuration file for a line that specifies the location of a command input file. As a result:

- If such a line is found and if that file exists, the Simulator reads its command input from that file until it reaches the end of file.
- If no such file is found, the Simulator prompts for command input from the user at the console.
- If such a line is not found in the configuration file, the Simulator looks for a file named **sim-***NP_variant_component***.in** in the current directory (where *NP_variant_component* can be 'c5' or 'c5e'). If found, the Simulator reads its command input from that file until it reaches the end of file.

When the Simulator reaches the end of a command input file, it prompts the user for the next command. When the Simulator encounters any 'quit' command at any point in a command input file, it ends the simulation session and exits.

Figure 5 on page 62 lists the Simulator command input file (**sim-c5e.in**) for the Gigabit Ethernet Switch (**gbeSwitch**) application.

Figure 5 Simulator Command Input File (sim-c5e.in) for the Gigabit Ethernet Switch Application

```
//*************************************************************
// This is the simulator init file. Place simulator commands
// for tracing and running an experiment in here.

//
// Add Tracing commands here.

//
// The following ifdef allows this init file to be used
// for both Quality Assurance Tests (from /QA/Simulator)
// and interactive use. To enable QA issue the simulator
// command:
//
//  dcpsim -c config -D _BATCH
//
cd Tlu
restore ./Tlu.State

#ifdef _BATCH
 go 350000
 quit
#endif
symbols sdp0 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp1 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp2 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp3 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp4 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp5 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp6 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp7 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp8 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp9 bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp10
bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp11
bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp12
bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp13
bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp14
bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
symbols sdp15
bin/$(CPORT_MODEL)-$(CPORT_REV)-$(CPORT_TGT)-$(CPORT_CFG)/gbe.sdp
```

Preprocessing Configuration and Command Input Files

At run-time the C-Ware Simulator “preprocesses” the configuration file and command input file—that is, the Simulator includes or excludes lines in these files based on the CST environment variables at the time the simulation starts or based on symbols passed to the Simulator in the command line. In the file listings shown in Figure 5 on page 62, notice the lines coded as:

```
...
#ifdef _BATCH
 go 350000
 quit
#endif
...
```

These lines are processed by the Simulator in the same manner as a C language compiler’s C preprocessor on a C language source program. For example, running this command:

```
   C:\C-Port\...\application\run\>
      C:\C-Port\...\bin\cwsim -c config -D _BATCH
```

causes a particular sequence of commands to be executed in the command input file shown in Figure 5. The Simulator’s ‘-D’ command line switch and argument specify the symbol to define. To define more than one symbol, you can specify more than one ‘-D’ switch in the command line.

Obtaining Information About NP Components

The C-Ware Simulator provides several techniques for obtaining information about the execution state and data content of the NP components as your application runs.

These features are generally termed *tracing* techniques. The Simulator provides built-in tracing for all modeled NP components. Other forms of tracing require that you *instrument*, or include special trace-producing calls in, your application's XPRC, CPRC, and/or microcode source code.

Component Tracing

A commonly used Simulator feature when debugging an application is *component tracing*, which means producing output about the state of an NP component.

As the Simulator runs the application, it can produce trace output about the state of any NP component (including RISC cores, microcoded sequencers, buses, BMU, TLU, and so on) at a given level of detail, or *trace level*. You can set any simulated NP component with a trace level from '0' (no tracing) to '1' (minimal detail) to '3' (very detailed). Each NP component also has a default trace level.

Use the Simulator's 'trace' command to turn tracing on or off for a given NP component, as follows:

- To turn a *default* level of tracing output on for a given NP component, specify the command:

  ```
  Dcp> trace NP_component
  ```

- To turn a specific level of tracing output on for a given NP component, specify the command:

  ```
  Dcp> trace NP_component trace_level
  ```

  Where *trace_level* is an integer from 0 to 3.

- To disable tracing for a given NP component, specify the command:

  ```
  Dcp> trace NP_component 0
  ```

By default, all NP component tracing is *disabled*. Enable tracing for components before the Simulator begins running your application or when the simulation is paused. The Simulator produces trace output only when simulated NP cycles are being executed.

In general, the Simulator does not support trace output from any NP component that "contains" other NP components, such as the CpCluster0 object. You should use component tracing for any NP component that performs distinct processing or discrete transactions or that has its own distinct state.

For example, to trace the activity of the Table Lookup Unit (TLU), you would execute the following:

```
cp.CpCluster0.Cp0.Creg0.0xBC004604> trace tlu 1
Dcp.Tlu> g
[148335 1 Tlu] TLU input message is ready
[148335 1 Tlu] TLU Request msg:RbMsg627 (error=0 mult=0 type=2 len=1
seq=0 dest=31 src=24 datau=0x10000100 datal=0x19000000)
[148346 1 Tlu] Message Ready For TLU,  (message length=1)
[148346 1 Tlu] TLU Command Received: (cmd=WRITEREG address=x0100
data=x19000000) Rb length=1 (slots)....
```

This allows the user to view, or to capture to a listing file, all transactions to the TLU. In particular, you can see that there is a TLU command (WRITEREG) that is received in this trace.

For example, to trace the execution of the XPRC, you can trace the following:

```
Dcp.Tlu> trace xprc 1
Dcp.XpCluster.Xp.XpRc> g 10
[287832 1 XpRc] 0x00008C58 0x8E040008 lw   a0, 0x0008 (s0)
[287833 1 XpRc] 0x00008C5C 0x00000000 nop
[287834 1 XpRc] 0x00008C60 0x14960027 bnea0, s6, 0x8D00   # 0x00000000 a0 c1 writeback
[287835 1 XpRc] 0x00008C64 0x00000000 nop
[287838 1 XpRc] 0x00008D00 0x26730001 addiu   s3, s3, 0x0001
[287839 1 XpRc] 0x00008D04 0x2E62000C sltiu   v0, s3, 0x000C    # 0x00000005 s3 c1 bypass
[287840 1 XpRc] 0x00008D08 0x1440FFB0 bnez    v0, 0x8BCC   # 0x00000001 v0 c1 bypass
```

To increase the detail in the XPRC trace output, you can specify trace level '3' as follows:

```
Dcp.XpCluster.Xp.XpRc> trace 3
Dcp.XpCluster.Xp.XpRc> g 5
[287841 3 XpRc] 0x00008D0C 0x26100010 addiu   s0, s0, 0x0010
[287841 3 XpRc] 0x00008BD0 0x00000000 I read
[287842 3 XpRc] 0x00008D10 0x00000000 nop                        # (squashed)
[287842 3 XpRc] 0x00008BD4 0x14400020 I read
[287843 3 XpRc] 0x00008BCC 0x8E020008 lw     v0, 0x0008 (s0)  # 0xBD9008BC s0 c1 writeback
[287843 3 XpRc] 0x00008BD8 0x00000000 I read
[287844 3 XpRc] 0x00008BD0 0x00000000 nop
[287844 3 XpRc] 0x00008BDC 0x8E02000C I read
[287844 3 XpRc] 0xBD9008C4 0x00000000 D read
[287845 3 XpRc] 0x00008BD4 0x14400020 bnez    v0, 0x8C58 # 0x00000000 v0 c1 writeback
[287845 3 XpRc] 0x00008BE0 0x00000000 I read
Dcp.XpCluster.Xp.XpRc>
```

At this level of tracing, you can observe very detailed information, such as the fetch-execute cycle of the XPRC.

You can enable tracing for more than one NP component during a simulation session. However, to do so you must enter a separate 'trace' command for each component to be traced.

ksPrintf() Diagnostic Messages

Sometimes component tracing provides more detail than is necessary to gain a high-level view of what is happening in the application.

The application's programs can issue their own console output messages. An XPRC program or CPRC program can call the C-Ware API function *ksPrintf()* to output a diagnostic message to the Simulator's standard output.

If you add this call to an RC program in the Gigabit Ethernet Switch application, the program produces a diagnostic message similar to the following:

```
Dcp.CpCluster0.Cp0.Creg0.0xBC004604> g
[277 0 XpRc] {Application output: Hello there, world}
```

This type of diagnostic message is helpful for high-level visibility into an RC program's execution without attaching the C-Ware Debugger or turning on the Simulator's component level tracing.

Application-Defined Tracing

The Simulator supports another technique called *application-defined tracing* for producing text message output to the Simulator console. This is similar to producing *ksPrintf()* text messages; however, unlike *ksPrintf()* messages, application-defined tracing messages are produced only when the application is run under the Simulator, and they must be explicitly enabled at the time the application is built.

Use the *dcpTrace1()* and *dcpTrace2()* macros to include application-defined tracing in your NP application. Having two macros allows your application to produce two levels of messaging. The two levels can represent different message verbosity levels or different message priorities. The meaning of the two message levels is for the application to define.

To illustrate, each C-Ware Application Library application includes code such as the following:

```
dcpTrace1(("Trace level 1 message output here!\n"));
```

and:

```
dcpTrace2(("Trace level 2 message output here!\n"));
```

To enable the output from these calls to be produced when the application runs under the C-Ware Simulator, the application must be recompiled and must define the build environment variable EXTRA_CFLAGS.

To build the application to produce these message for the application-defined trace level ‘1’ only, use the following make command:

```
make "EXTRA_CFLAGS+=-DDCP_TRACE_LEVEL_1"
```

To build the application to produce these message for the application-defined trace level ‘2’ only, use the following make command:

```
make "EXTRA_CFLAGS+=-DDCP_TRACE_LEVEL_2"
```

To build the application to produce these message for both application-defined trace level ‘1’and trace level ‘2’, use the following make command:

```
make "EXTRA_CFLAGS+=-DDCP_TRACE_LEVEL_1 -DDCP_TRACE_LEVEL_2"
```

All source files that contain these application-defined tracing messages must be recompiled. Before doing so, you must delete all existing compilation products (via a ‘make clean’) for the source files where the application-defined tracing messages appear.

Viewing the Simulator's Performance-Related Counters

The Simulator maintains its own set of "counter" objects that describe several performance-related XPRC and CPRC attributes of a simulation session.

In the Simulator, enter the command 'counter' to view these attributes. Figure 6 on page 69 presents an example of the output from the 'counter' command.

In this output, each line lists the 17 (for the 16 CPRCs and 1 XPRC) current values for a given kind of counter. For example:

```
Ireads  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 320
```

This line presents the 17 values of the 'Ireads' counter (that is, "instruction read operations") for the 16 CPRCs and 1 XPRC. The last value on each line is for the XPRC.

Remember that these counters are calculated only by the Simulator; they do not represent memory locations, registers, or components of the network processor.

The meaning of each counter and how to interpret them are presented in the CST's *C-Ware Simulation Environment User Guide* document.

Figure 6 Sample Output From the Simulator's 'counter' Command

```
...
[Lines removed]
...
Ireads  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 320
IreadsStalls  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 27
IreadsStallCycles  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 27
Dreads  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 43
DreadsStalls  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
DreadsStallCycles  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Dwrites  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 31
DwritesStalls  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6
DwritesStallCycles  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 153
StackReads  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11
StackReadsStalls  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
StackReadsStallCycles  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
StackWrites  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 16
StackWritesStalls  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
StackWritesStallCycles  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
LocalReads  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11
LocalReadsStalls  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
LocalReadsStallCycles  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
LocalWrites  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 16
LocalWritesStalls  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
LocalWritesStallCycles  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
BranchesTaken  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 29
BranchesMissed  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5
DependentBranchesTaken  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8
DependentBranchesMissed  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2
InstructionsExecuted  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 347
InstructionsSquashed  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 40
InterruptPending  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
InterruptsDelivered  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
LoadDelayFilled  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 22
BranchDelayFilled  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 39
FlowControlChange  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 45
ContextSwitches  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Context0  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 500
Context1  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Context2  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Context3  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Cycles  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 500
LastExecPc  -17958194 -17958194 -17958194 -17958194 -17958194 -17958194
-17958194 -17958194 -17958194 -17958194 -17958194 -17958194 -17958194 -17958194
-17958194 -17958194 65136
```

Summary

In this lesson you learned:

- That the key components of the C-Ware Simulation Environment (CSE) are the C-Ware Simulator (**bin\cwsim.exe**), the pattern generation Perl scripts (**bin\pg*.pl)**, the **bin\printTrace.pl** tool for filtering pattern files into a human-readable format, and other CST-provided software objects that support certain kinds of “system” simulations
- What are the featured capabilities of the C-Ware Simulation Environment (CSE)
- How to start and stop the C-Ware Simulator
- How to direct the C-Ware Simulator at start-up to emulate a particular C-Port network processor (NP)
- The purpose of the contents of the Simulator’s configuration file and command input file
- How commands in the Simulator’s command input file are performed
- How preprocessor commands alter the flow of execution of a Simulator command input file
- How to halt the Simulator interactively
- What are the typically used Simulator commands
- How to navigate the Simulator’s model of a C-Port NP’s components
- How to use the Simulator to produce “trace” information about the state of a C-Port NP’s embedded components
- How to view performance-related counters for the NP’s XPRC and CPRCs that are maintained by the Simulator
- What are the alternative Simulator techniques for viewing application-generated messages during a simulation session

Before you continue to a subsequent lesson, you can use this knowledge to explore the Simulator's features as you run the **gbeSwitch** application.

You will also use this knowledge in subsequent lessons, as follows:

- When you set up the C-Ware Debugger to work with a simulation session

Exercises

The following are some exercises to make sure that you are familiar enough with running simulations using the C-Ware Simulator.

❑ **Locate the 'creg0' object in the C-Ware Simulator's component hierarchy. Note what the absolute path is to where that object is in the hierarchy. What is that path?**

❑ **What information does the Simulator produce at trace level '2' for the C-5 NP's Buffer Management Unit (BMU)?**

❑ **In the "ipRx()" function (found in the "ipRxCp.c" source file in the Gigabit Ethernet Switch application) add a call to "ksPrintf()" to produce a diagnostic message. After the first invocation of the "ipRx()" function (regardless of the network processor RC where it is running), on what core clock cycle does the "ksPrint()" function emit the message?**

Lesson 5

DEBUGGING XPRC AND CPRC PROGRAMS

Goals

In this lesson you will learn:

- What are the featured capabilities of the C-Ware Debugger
- How to start the Debugger inside a Simulator session, to debug a Simulator-targeted C-Port NP XPRC/CPRC program
- How to establish interaction between the Debugger and a Simulator session
- How to start the Debugger outside a Simulator session, to debug a hardware-targeted C-Port NP XPRC/CPRC program
- How to use several typically useful Debugger commands and operations
- How to retarget the Debugger among different C-Port NP XPRC/CPRC programs in the same Simulator session
- How to distinguish program instances in the target C-Port NP Channel Processor cluster's shared IMEM

Featured Capabilities of the C-Ware Debugger

Because you can write software for a C-Port network processor (NP) in a high-level language such as C, the C-Ware Software Toolset (CST) includes the C-Ware Debugger to support source-level debugging of programs running on any of the NP's RISC cores (RCs)—that is, the NP's XPRC and CPRCs.

The C-Ware Debugger (**bin\cport-gdb.exe**) is a customized implementation of the Free Software Foundation's GNU 'gdb' debugger.

You can use the C-Ware Debugger to debug C-Port NP RC programs running under the C-Ware Simulation Environment or running on C-Port NP hardware. *However, this chapter focuses on techniques for debugging RC programs running under the Simulator.*

You should be aware of the benefits of the following high-level features of the C-Ware Debugger:

- The Debugger allows you to interact with *high-level language programs* (such as C language programs) running in the C-Port NP's RCs.

You can't use the Debugger tool to interact with the C-5 NP's microprograms running on the various CPRCs' SDPs or Fabric Processor's FDPs; however, other debugging techniques (such as Simulator traces and hard-coded output messages) are available for those programs.

- The Debugger closely *interoperates* with the C-Ware Simulator, which is part of the C-Ware Simulation Environment—that is, the Debugger can start, resume, and otherwise control the execution of C-Port NP RC programs that are loaded in the Simulator.
- You can use the Debugger to *debug remotely* (that is, across the local-area network to which your development workstation is connected) a target Simulator session or target C-Port NP-based system. The Debugger "attaches" to the target via TCP/IP sockets.

- When debugging a C-Port NP RC program running under the Simulator, you have these options for launching the Debugger:
  - Launching the Debugger *within the Simulator* (using the Simulator's 'gdb' command) is the most convenient technique, if you are operating both the Simulator and Debugger at your workstation. Launching the Debugger within the Simulator automatically establishes interoperation between the Simulator and the Debugger.
  - Launching the Debugger *outside the Simulator* (using the CST's 'cport-gdb' shell command) is possible when debugging RC programs running under a Simulator session on your workstation. However, launching the Debugger outside the Simulator is *required* when debugging across your network—that is, when debugging a program that is running under a Simulator session hosted on a physically remote workstation or server.
- When debugging C-Port NP RC programs running under the Simulator, you use that Debugger session to target *one XPRC program or CPRC program at a time*. That is, you can launch, run, and quit the Debugger while targeting one RC program, then launch the Debugger again to target a different RC program during the same Simulator session.
- When using the C-Ware Debugger to debug C-Port NP RC programs running on C-Port NP hardware (such as under the C-Ware Development System), you can launch multiple Debugger sessions with each targeted to a different RC program.

Starting the Debugger

You can start the Debugger from within the Simulator or from a command shell outside the Simulator, as described in the following sections.

Starting Within a C-Ware Simulator Session

To launch the C-Ware Debugger within a Simulator session:

1. Open a new command window or command shell.
2. In the new command window, run the CST's **bin\sv.bat** Windows batch file (or the CST's **bin\sv.[c]sh** Unix command script) to set up this command shell's CST environment variables.
3. Change directory to the location of your C-5 NP application "run" subdirectory.
4. Launch the Simulator, subject to these caveats:
   - Do not specify the Simulator's '-batch' command line option.
   - Do not specify the Simulator's '-maxCycles' command line option.
   - Do not specify the Simulator's '-timeOut' command line option.
   - Do not use a Simulator command input file, typically named **sim.in**, that causes the Simulator session to operate without a halt.
5. After the Simulator loads the specified C-5 NP application, and after it displays its "Dcp>" command prompt:
   - **a** Issue the Simulator's 'move' command, to give focus to the XPRC ('move xp') or CPRC ('move cp0' up to 'move cp15') whose RC program you want to debug.
   - **b** Issue the Simulator's 'gdb' command, to launch the Debugger.

Launching the Debugger in this manner causes it to automatically begin interoperating with the Simulator and to target the program image in the XPRC or CPRC that has focus.

Starting Outside a Simulator Session

If launched outside the Simulator, the Debugger "attaches" to a program running under a Simulator session via the Simulator process's TCP/IP network node id and port number. Also, after the Debugger has been launched in this scenario, you must specify:

- The path of the package file where the target Simulator session loaded the application
- The path of the "symbols" file for the RC program that you want to debug; this file was produced previously as a by-product of building that RC program

Important: After you have started a Simulator session, you must halt its execution of the loaded application before attempting to attach the Debugger to the simulation session.

To launch the C-Ware Debugger and attach it to an existing Simulator session:

1 In the Simulator, issue a command to prepare it to be the target of a Debugger session:

```
Dcp> appdebug imem-spec host_name:TCP_port_number
```

Where:

– *imem_spec* represents the Simulator component identifier of the location of the RC program that you want to debug in this Debugger session. Examples of an imem_spec are: ' imemxp' for the XPRC's IMEM, 'imem0' for CPRC0's IMEM, and so on.

Important: If you want to debug an RC program on a CPRC that is a member of a CP cluster, specify the IMEM for the first member of that cluster (that is, one of IMEM0, IMEM4, IMEM8, or IMEM12).

– *host_name*:*TCP_port_number* designates the network identifier for the workstation where the Simulator is running and the TCP port on that workstation to use to accomplish a connection to the C-Ware Debugger.

2 Open a new command window or command shell.

3 In the new command window, run the CST's **bin\sv.bat** Windows batch file (or the CST's **bin\sv.[c]sh** Unix command script) to set up this command shell's CST environment variables.

4 In the new command window, change directory to the location of your C-5 NP application's "run" subdirectory.

5 In the new command window, enter the command "cport-gdb" to start the C-Ware Debugger.

6 After the Debugger launches, in the Debugger's console window enter the command to attach the Debugger to the Simulator session:

```
(gdb) target dcpsim host_name:TCP_port_number
```

Where:

– *host_name*:*TCP_port_number* designates the same workstation network identifier and the TCP port on that workstation that you just declared using Simulator commands.

7 After the Debugger has established its connection with the Simulator session, use the C-Ware Debugger's 'file' command to identify which C-5 RC executable program is running on the target:

```
(gdb) file gbeRxCp.dcp
```

8 Continue debugging the specified RC executable program.

Establishing Debugger Interoperation With the Simulator

For the Debugger to interoperate with the Simulator, you must:

1 Start the Simulator to set up execution of, and then load, a specified application. The application is loaded from a specified package file.

2 In the Simulator, after you specify which of the RC programs loaded in the simulated C-5 that you want to debug (that is, after you "move focus" in the Simulator to the XPRC, any CPRC, or a shared CPRC cluster), you launch the Debugger using the Simulator's 'gdb' command.

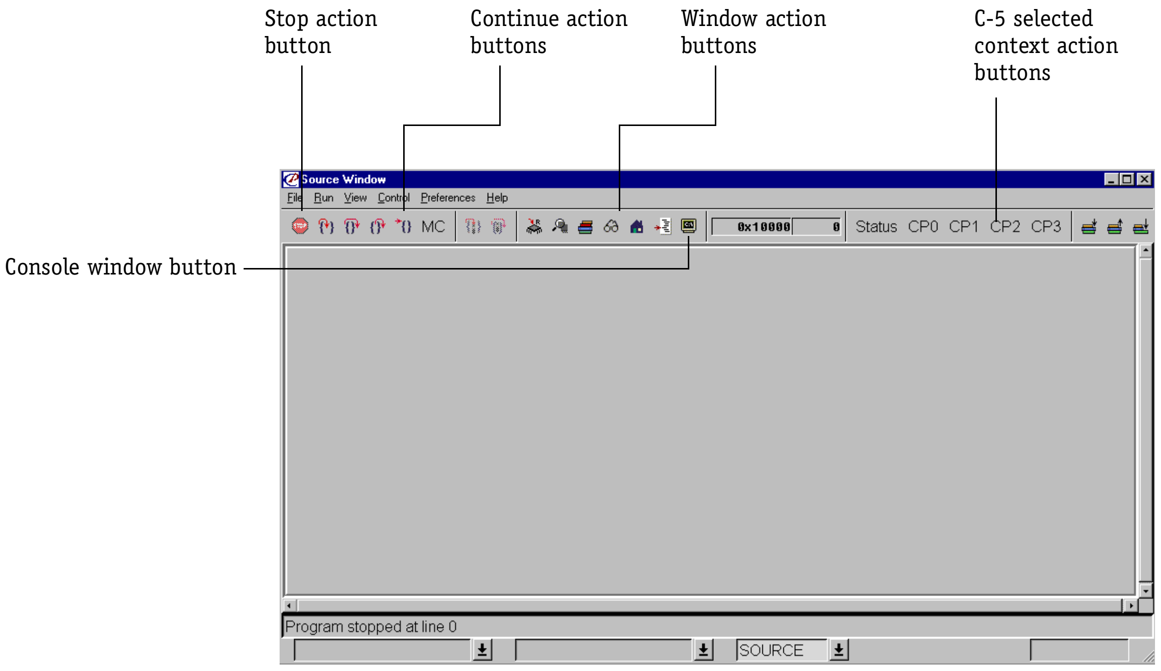
Launching the Debugger from the Simulator automatically establishes interoperation between the Simulator and the Debugger.

After you have started the Simulator, you can launch the Debugger only when the Simulator has been *halted*, for instance, *before* the Simulator begins running the programs it has loaded or *after* you have stopped the Simulator's execution of the loaded programs (such as after a Debugger breakpoint has been encountered).

Operating the Debugger

After the Debugger starts, it displays the default Debugger client window, shown in Figure 7. This window, titled “Source Window,” displays the source code for the program being debugged. This window displays as empty until the Debugger completes its connection to the Simulator session.

Figure 7 C-Ware Debugger’s Default Window



Program stopped at line 0

As it opens, the Debugger reads in the debugging symbols from the specified symbol file (specified in the Simulator’s ‘symbols’ command) and attaches to the C-5 Simulator.

After the Debugger launches, the Simulator session’s execution of its loaded application remains halted. This means that you begin using the Debugger to examine the target RC programs execution context (memory contents, register contents, program counter, and so on).

Tip: An important way to gain control of the target RC program at this point is to set one or more *breakpoints*. See the section “Setting Breakpoints” on page 82 for more information.

By convention, the "main" routine in each C-5 RC program is *DCPmain()*

```
int DCPmain(KsPackage package);
```

Debugger Control of a Simulator Session

After the Debugger establishes interoperaton with the Simulator, the Debugger controls, or "drives," that Simulator session for as long as the interoperation remains established.

More importantly, as long as the Debugger is interoperating with a Simulator session, you cannot issue commands from the Simulator's command line interface to control the Simulator session.

Closing the Debugger ends its interoperation with a Simulator session. The Simulator's execution of the simulated C-5 must be *halted* (for instance, at a Debugger breakpoint) before you can close the Debugger. After you close the Debugger, you can also close the Simulator, if you wish.

Remember that halting the Simulator's execution of your application "freezes" all run-time state (including the run-time states of all RCs, SDPs, and the FP) of the simulated C-5 NP.

Targeting an RC or Shared CPRC Cluster

The Debugger can target the program running on any individual RC (that is, the XPRC or any CPRC) or the program running in the shared IMEM for any shared CPRC cluster.

When you launch the Debugger from the Simulator, the Debugger always assumes that the C-5 RC that "has focus" in the Simulator session is running the program that you want to debug.

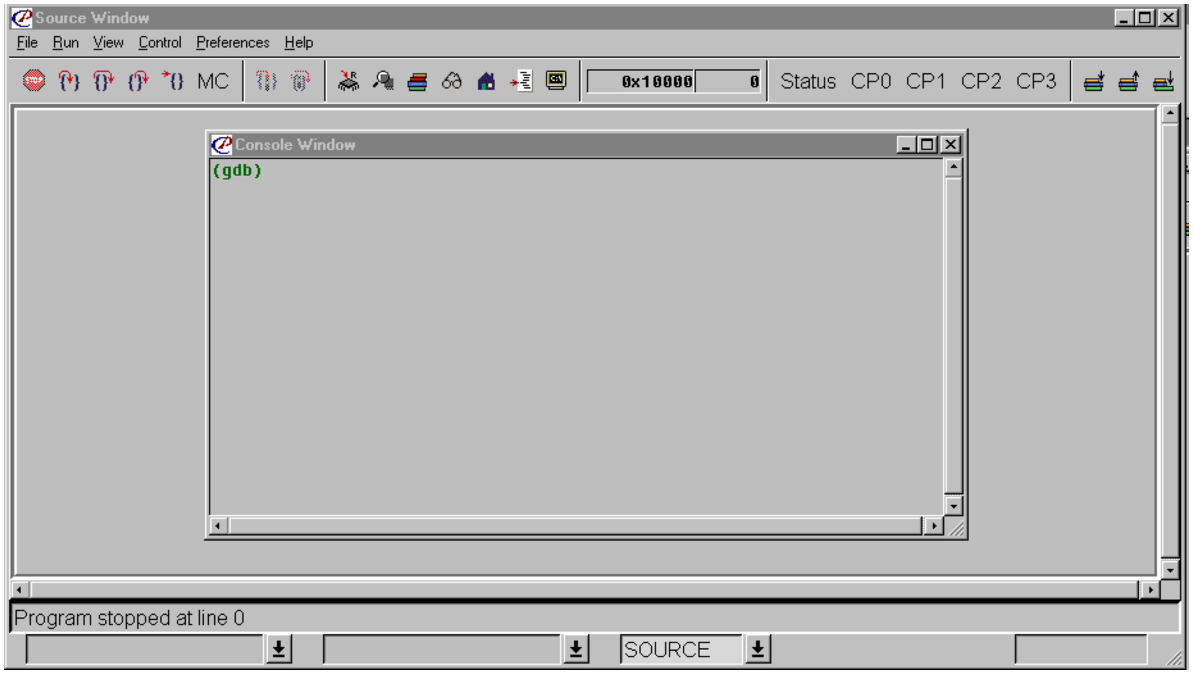
Important: If you want to debug an RC program on a CPRC in a shared CP cluster, change the Simulator's focus to the lower CP in that cluster—that is, to CPRC0, CPRC4, CPRC8, or CPRC12.

In contrast, when you launch the Debugger outside of the Simulator, you must first specify in the Simulator session which RC program the upcoming Debugger session will attach to.

Opening the Debugger's Console Window

To open the Debugger's command, or *console*, window, select the **View > Console** pull-down menu command, or enter the **Ctrl+N** keystroke.

Figure 8 C-Ware Debugger's Default Source Window and Console Window



```
(gdb)
```

Program stopped at line 0

Setting Breakpoints

A breakpoint is a Debugger object that causes the execution of the target program to be paused at a particular program location. Breakpoints are an important way for you to control the program being debugged.

You can set a breakpoint using the Debugger's button bar or using commands in the Debugger's console window.

You can set a breakpoint at the top of a routine, at a particular line of source code, or at a specific address.

Using the Button Bar

To set a breakpoint in a program using the button bar:

1 Click on a source code line in the Source Window.

2 Click on the "Stop" icon.

Using a Console Window Command

To set a breakpoint in a program using a console window command, enter the 'break' (or 'b') command:

```
(gdb) b DCPmain
Hardware assisted breakpoint 1 at 0x8010: file xpMainInit.c, line 81.
```

Viewing Information About Breakpoints

Use the 'info break' command:

```
   (gdb) info break
```

Setting a Thread-Specific Breakpoint

For a program running in a CPRC that is part of a shared CP cluster, you can set a breakpoint that is thread-specific. That is, the breakpoint pertains to only one of the shared program's contexts of execution.

Enter this command to set a thread-specific breakpoint at line 45 in the target program's source code:

```
(gdb) b 45 thread 2
```

You can view information about the thread-specific breakpoints you have set:

```
(gdb) info break
Num Type           Disp Enb Address    What
1   hw breakpoint  keep y   0x0000800c in DCPmain at cpMain.c:80
   breakpoint already hit 1 time
2   hw breakpoint  keep y   0x00009d4c in transmit at macTx.c:89
   stop only in thread 2
```

Setting a Temporary Breakpoint

You can also set a *temporary breakpoint*. That is, the Debugger cancels this breakpoint after the first time it is encountered. Enter this command:

```
(gdb) tb DCPmain
```

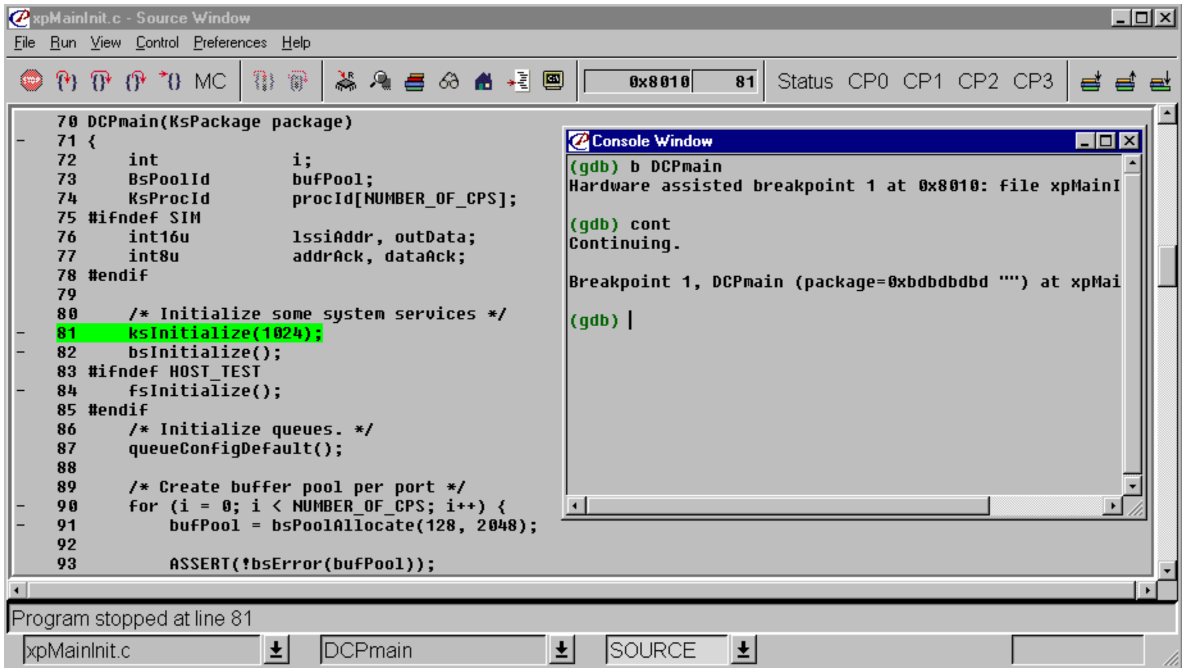
Encountering a Breakpoint

After the target RC program encounters a breakpoint the Debugger display should appear similar to Figure 9.

When the program's execution reaches the breakpoint, you will have access to the GUI debugger again and it will have something similar to the following on the display:

You are now free to step through the code running on the XPRC, inspect variables, and so on as GDB allows.

Figure 9 Debugger Display After Encountering a Breakpoint


xpMainInit.c - Source Window
File Run View Control Preferences Help
MC
0x8010 81
Status CP0 CP1 CP2 CP3
Console Window

```
70 DCPmain(KsPackage package)
71 {
72     int             i;
73     BsPoolId        bufPool;
74     KsProcId        procId[NUMBER_OF_CPS];
75 #ifndef SIM
76     int16u          lssiAddr, outData;
77     int8u           addrAck, dataAck;
78 #endif
79
80     /* Initialize some system services */
81     ksInitialize(1024);
82     bsInitialize();
83 #ifndef HOST_TEST
84     fsInitialize();
85 #endif
86     /* Initialize queues. */
87     queueConfigDefault();
88
89     /* Create buffer pool per port */
90     for (i = 0; i < NUMBER_OF_CPS; i++) {
91         bufPool = bsPoolAllocate(128, 2048);
92
93         ASSERT(!bsError(bufPool));
```

```
(gdb) b DCPmain
Hardware assisted breakpoint 1 at 0x8010: file xpMainI

(gdb) cont
Continuing.

Breakpoint 1, DCPmain (package=0xbdbdbdbd "") at xpMai

(gdb) |
```

Program stopped at line 81

xpMainInit.c | DCPmain | SOURCE

Continue Program Execution

To continue the target program's execution, enter the Debugger's 'continue' command:

```
(gdb) cont
Continuing.
```

Additional Topics

Targeting a New RC Program in the Same Simulator Session

After you have used the Debugger to control a given RC program, you might wish to target a different C-5 RC program. To do so, you must take the following steps:

1. Use the Debugger to halt the Simulator session (such as by encountering a Debugger breakpoint, or by pressing the "Stop" button on the Debugger's button bar).
2. Close the Debugger.
3. In the Simulator change focus to another RC, such as the XPRC, another CPRC, or the based CPRC in a shared CPRC cluster.
4. Start a new Debugger session, either from this Simulator session (using the Simulator's 'gdb' command) or outside this Simulator session (as described in the section "Starting Outside a Simulator Session" on page 77).

Distinguishing RC Program Instances in Shared IMEM

When you want to debug the RC program running on a CPRC that is a member of a shared CP cluster, the program loaded in that cluster's shared IMEM is the actual target.

For each CPRC that is enabled in that CP cluster, the Debugger distinguishes a "thread" of execution of the program loaded in the cluster's shared IMEM. That is, the Debugger assigns a "thread number" to each CPRC's *context of execution* that takes place as the application runs. (Do not confuse this "context of execution" phrase with a CPRC's own hardware-implemented "contexts".)

Viewing CPRC Thread Numbers

To view the thread numbers for the CPRCs in a targeted CP clusters, enter this Debugger command:

```
(gdb) info threads
```

Changing Focus to One CPRC Within the Cluster

You can change the Debugger's “focus” to a given CPRC in the cluster by entering this Debugger command:

```
(gdb) thread thread_#
```

Where:

– *thread_#* refers to a thread (that is, a CPRC context of execution) shown in the output of a ‘info threads’ command.

Each Variable has Distinct Addresses in Cluster's DMEM Resources

The RC program that runs in a CP cluster use a shared IMEM resource but use the clustered CPRCs' respective DMEM resources. Thus, the same RC program variable has a distinct address in each of the CPRC's DMEM resource.

For example:

```
(gdb) thread 1
[Switching to process 1]
...
(gdb) p/x &nblock
$1 = 0xbc000004
(gdb) thread 2
[Switching to process 2]
...
(gdb) p/x &nblock
$2 = 0xbc100004
(gdb) thread 3
[Switching to process 3]
...
(gdb) p/x &nblock
$3 = 0xbc200004
```

Notice that the address of the same variable changes as you change the Debugger's focus among the “threads” within the target shared CP cluster.

Summary

In this lesson you have learned:

- What are the featured capabilities of the C-Ware Debugger
- How to start the Debugger inside a Simulator session, to debug a Simulator-targeted C-Port NP XPRC/CPRC program
- How to establish interaction between the Debugger and a Simulator session
- How to start the Debugger outside a Simulator session, to debug a hardware-targeted C-Port NP XPRC/CPRC program
- How to use several typically useful Debugger commands and operations
- How to retarget the Debugger among different C-Port NP XPRC/CPRC programs in the same Simulator session
- How to distinguish program instances in the target C-Port NP Channel Processor cluster's shared IMEM

Exercises

The following exercise make the user familiar with debugging programs using the C-Ware Debugger.

- ❑ **Start a Simulator session that runs the "Packet Over SONET to Gigabit Ethernet Switch" (posGbeSwitch) application. Change the Simulator's focus to CP8, then launch the Debugger from within the Simulator. In the Debugger set a breakpoint at the "posOc12Receive()" function. Step through part of this function in the "posOc12Cp.c" source file.**

- ❑ **The 'Gigabit Ethernet Switch' application (gbeSwitch) uses aggregated channels, which means that clusters of CPs are configured during application initialization. In the documentation for this application (open the file "...\apps\gbeSwitch\doc\gbeSwitch.pdf"), in the "Application Control and Data Flow" section notice in the control/data flow diagram the roles for each of the C-5 NP's four configured CP clusters. (Each cluster either receives or transmits one GbE port.) Pick one of the four CP clusters to debug.**

  Start a Simulator session that runs the "gbeSwitch" application. Change the Simulator's focus to the lowest CP (that is, CP0, CP4, CP8, or CP12) in the CP cluster of your choice, then launch the Debugger within the Simulator. After the Debugger launches, set a breakpoint at the "DCPmain()" function in the target CP program, then continue execution. After the session reaches the "DCPmain()" function (that is, after the first CPRC context of execution in the target cluster reaches the "DCPmain()" function), set a breakpoint in the "receive()" function on CPRC2 only. Step through some of the receive path.

Lesson 6

GENERATING SIMULATED INGRESS TRAFFIC

Goals

In this lesson you will learn:

- What is the purpose of pattern files
- What is the purpose of patter generation scripts

Overview

To test different features of the applications running on the C-5 Network Processor (NP), you must provide different stimuli for the simulation. Thus there is the need to simulate the ingress and egress traffic data that an application produces and consumes while running under simulation.

The C-Ware Software Toolset (CST) provides a three-part solution for this need:

- **Pattern files** — An efficient ASCII file format for representing raw traffic data
- **Pattern generation scripts** — A programmatic means to use specifications of traffic data patterns to produce an equivalent representation as raw data

Pattern files are ASCII files whose lines are representations of raw traffic data. These files can be presented to the C-5 Simulator for the purpose of altering the processing that the application performs. This allows you to change the data packets and cells that are received by the C-5 NP's Channel Processors (CPs) and Fabric Processor (FP) while running under the C-5 Simulator. The format of pattern files is described in "Pattern File Format" on page 93.

The *pattern generation scripts* (or "pg" scripts) are Perl routines. They are organized into logical groups by the protocols (IP, Ethernet, Physical, SONET) they produce. You invoke routines for one of the groups in a particular order to produce a pattern file.

The scripts are designed so that they can be used in a "pipeline" fashion to filter, first, a minimally specified data set and, later, to filter an intermediate data set and produce a pattern file. That is, some scripts are designed to produce only intermediate data, others both consume and produce intermediate data, and others produce pattern file output.

You are free to modify these scripts, provided in source form, in order to generate custom data formats and protocols that accommodate your application and system needs.

pg Scripts

The CST provides three types of “pg” (pattern generating) scripts:

- Pattern-producing scripts
- API scripts
- Ethernet-type framer scripts

These three types of scripts are used together to generate the types of pattern files that are required by applications that run on the C-5 Simulator.

Pattern-Producing Scripts

Pattern-producing scripts are invoked from the command line and perform a specific user-call visible duty. An example of this is the script to apply an IP header to a datagram or the script to apply SONET overhead to a PDU. The set of pattern-producing scripts is described in the following sections.

General

- **pgGenData.pl** — Produces raw data PDUs based on length and count.
- **pgFmtDcpsim.pl** — Formats data for consumption by software simulator.
- **pgFmtBusWidth.pl** — Formats data width according to physical layer protocol.

Protocols

- **pgEnet.pl** — Formats data for Ethernet.
- **pgAtm.pl** — Formats data for ATM cells.
- **pgIp.pl** — Applies IPv4 header to datagrams.
- **pgPpp.pl** — Applies PPP header and byte-stuffing to datagrams.
- **pgSonetPayScram.pl** — Applies SONET payload scrambling to datagrams.
- **pgSonet.pl** — Applies SONET framing to datagrams.

Fabric Port

- **pgFpCport.pl** — Converts data stream into C-5 DCP FP format.
- **pgFpPrizma.pl** — Applies IBM PRIMZMA header to fabric cells.
- **pgFpUtopia.pl** — Applies proper control signals for Utopia-3 for simulator.

API Scripts

API scripts are called from the pattern-producing scripts. They perform a more detailed function than the pattern-producing scripts. An example of this is for the SONET protocol. The **pgSonet.pl** pattern-producing scripts uses other API scripts (**pgSonetPath.pl**, **pgSonetLine.pl** and **pgSonetSection.pl**).

The set of API scripts are listed here:

- **pgWeightedRange.pl** — Provides routines for creating and retrieving values from a weighted range expression.
- **pgPtnParse.pl** — Provides access routines to the data that is passed between scripts.
- **pgOptions.pl** — Provides a routine to parse command line options.
- **pgCrc.pl** — Provides routines to initialize and calculate CRCs.
- **pgSonetContext.pl** — Defines common routines for SONET pattern generation.
- **pgSonetPath.pl** — Defines routines related to SONET path overhead.
- **pgSonetSection.pl** — Defines routines related to SONET section overhead.
- **pgSonetLine.pl** — Defines routines related to SONET line overhead.

Ethernet Framer Scripts

These routines manipulate Ethernet frames based on different values of the Ethernet type field (802.1Q, IP, and so on).

Here are the provided Ethernet framer scripts:

- **pgEnetTypeIp.pl** — Support for Ethernet/IP frames.
- **pgEnetTypeMacCtl.pl** — Support for 802.3 MAC control frames.
- **pgEnetTypeVlan.pl** — Support for 802.1Q MAC frames.

Generating Pattern Files

To generate patterns using the previously mentioned scripts, you invoke them on the command line in the following fashion:

100Mb Ethernet/IP Example

```
C:\C-Port\...\bin> perl pgGenData.pl -numReps 10 -size 64 -rand | perl
pgIp.pl -srcAddr 0x868d0101 -destAddr 0x868d0101 | perl pgEnet.pl
-macDA 0x000000000011 -macSA 0x000000000033 -type 0x0800 | perl
pgFmtBusWidth.pl -100MbEther | perl pgFmtDcpSim.pl > outFile.pat
```

This command line invokes a series of scripts that generate 10, 64-byte Ethernet frames with random data, the IP addresses specified, the MAC addresses specified formatted for 100Mb Ethernet.

PPP Over SONET Example

```
C:\C-Port\...\bin> perl pgGenData.pl -numReps 10 -size 44 -byteCount |
perl pgIp.pl -srcAddr 0x868d0101 -destAddr 0x868d0101 | perl pgPPP.pl
| perl pgSonet.pl -pointer 89 | perl pgFmtBusWidth.pl -bit | perl
pgFmtDcpSim.pl > out.pat
```

This command line invokes a series of scripts that generate 10, 44-byte IP framed PPP over SONET frames for OC-3c.

Pattern Files

Pattern files are output from the “pg” scripts previously described.

Pattern File Format

Because the CST’s C-5 Simulator completely models the physical interface that it is configured for, the data going into and coming out of that interface has the same format of the physical layer protocol.

That is, instead of packets or cells being fed and transmit out of these interfaces, the Simulator requires use of a physical layer protocol representation, which is admittedly much harder to read and understand upon visual inspection.

The format of these pattern files follow:

```
20 257 0 0 0
20 257 0 0 0
20 257 0 0 0
20 259 0 0 0
```

The first column in each of these lines is the number of nanoseconds (ns) between bits on the wire. In this case, this number is a 20 since in RMII there are 20ns per di-bit.

The second column in each of these lines is the value of the data for that particular clock cycle. In the case of Fast Ethernet, there are only two bits of interest, since it is a two-bit interface (bits 0 and 8).

Benefits

A benefit of using the pattern file approach is that the CST’s **printTrace.pl** utility— which takes a pattern file as input and outputs a formatted, timestamped representation of that data that distinguishes ATM cell/Ethernet frame header data from payload data— can operate on the output from an application run under the Simulator *and* on the output from an invocation of a set of “pg” scripts.

For example, after running the Gigabit Ethernet Switch application under simulation, you could use **printTrace.pl** to format the simulated egress traffic data:

```
C:\C-Port\...\apps\gbeSwitch\run\outPatterns\c5i\>
  perl ..\..\..\..\bin\printTrace.pl gbeTrc7.pat > gbeTrc7.formatted
```

You can do likewise after generating a pattern file from the command line using the “pg” scripts:

```
C:\C-Port\Cst2.2\bin> perl pgGenData.pl -numReps 10 -size 64 -rand |
perl pgIp.pl -srcAddr 0x868d0101 -destAddr 0x868d0101 | perl pgEnet.pl
-macDA 0x000000000011 -macSA 0x000000000033 -type 0x0800 | perl
pgFmtBusWidth.pl -100MbEther | perl pgFmtDcpSim.pl >
C:\C-Port\Cst2.2\apps\gbeSwitch\run\outPatterns\c5i\outFile.pat

C:\C-Port\Cst2.2\apps\gbeSwitch\run\outPatterns\c5i> perl
..\..\..\..\bin\printTrace.pl outFile.pat > outFile.formatted
```

Exercises

❑ **Use the pattern generation scripts to generate a set of five 512-byte IPX frames. Generate these frames with a MAC DA of: 0x000000000011, MAC SA of: 0x000000000030 and Ethernet type of 0x8137.**

To do this, you will need to run the scripts with the following arguments:

```
C:\C-Port\Cst2.2\bin> perl pgGenData.pl -numReps 5 -size 512
-rand | perl pgEnet.pl -macDA 0x000000000011 -macSA
0x000000000030 -type 0x8137 | perl pgFmtBusWidth.pl -100MbEther |
perl pgFmtDcpSim.pl > test.pat
```

❑ **Run the “printTrace.pl” post-processing tool on the output pattern that you just generated to see that the packets are correct. You should see output as follows:**

```
C:\C-Port\Cst2.2\bin> perl printTrace.pl test.pat
```

```
MAC da:000000000011 MAC sa:000000000030 type:8137 crc:6f0b3c9a
Payload
Numeric=[9031cf957a59e5d2bf2cdbb5834d03175d252afd721e01026088929a9b2aa
9735a0e9bc8cd854de0baf4ec8a24763cdc35c7d7ffff9c64444cd7066017ad0e02eb4
64596b0d6b97c34be7775f2be1b9962bc9b925c26396ccd84fdc0582ba87d10b38125f
324e7b14d6d12f7ae27e0d295302dd1792781bb674791aec1b8791f5ed50884a96d1af
3eb8c58785fd8517445fb4cbd9132c2d66580e307fe920c8831d7a0a832d71f1cbe50f
0495623bbd5b599bf4024000fceda351d8d031d74c0af8b126f33b24a6f3b9388a0298
1f6b2ec30562dfe75ff18a01870ee0ce54a3ac46933a09a737799a2dad49fb890602fb
c8ee73e309ab295597e14bd9c9eb0cd2693dee99dba0baaf950914e2c1bded9be27531
413a4d18b72684c77802752bc50d3f5dfb94cf12010c8869cf412e0a7521a813a4aeb8
da91d7e617fcb8261b0889b6501b5199fdc7dbf7f61c98d5bf4a12d5f21bef39c07540
ea321d8dd98b8da0320b59d37102b9f57515ea9cdce869ccce625a1674022da106d924
d8c394f1cce2248c9e5cabe9d5cdb3add3a3f8afc0d14866d1842e43b2520ee140c0f5
6ea666ef2d688d7b165420186f4663d9541aff16fe301f099c993243862016a15a8da1
0cfa9b1cd87af24b0bac707dea4b4158df20f4625fb9e4aec5eb137273d85e61be7711
4c82bf9c6de357400c01d674ffe0940303f279ee2f032c7a5a8e8]
Time Elapsed: 43040  ns


MAC da:000000000011 MAC sa:000000000030 type:8137 crc:dc154652
Payload
Numeric=[ebaa26a2916cf18a948941655909cb32116397123227a4749ab270af65d59
733f3e064fc2fe59371a0b50c494268e5b5bae56465e74f63925abbbebcbd233345aee
95e841ce83384a770b0171413075a4eb2246511acb932b19da78b198b0cf976f8ba87a
eb15f630ccf2b0f4fc836527c0270829bc26659d7f19fff0d90dd634e48069c640407
74dbef66ba9ccee0959814cb76979691d0ac570244573bc55fde1db9a1e42e89e6c7f
732e6ccf4d9e3a02f9d4340bbc92bb67b14ceda1f4e9279c331bafe03a5b12c1246ad
d68e5fc73ba33b34f84805c4260c0259a4576a6a178bb356a8e089910d77426ddd7c1
690b5c7ddd94bdda5bfd4c3dfe8e985701b5ad26d2b00a8c6101662afce27240c5804
faa50deaa36e053e64a7c805d323a28cfb76364cc8577b9b1afcc12674b6ffc6a49ff
2b04a914a8cb63f8b7ef3abc39f72ccbefd34a1e897eb817d73e3f02e4033bbf0a00a
a4088e8b720ffe20990c2ca95550a7696d38ad2a7f81cd6f01566020d3bbe4a4a683b
bbcf5775c1ce7d2741bdec20a6aef112a361d971e9e94b0bbeaab4ccf60ffc5342b63
2e9432f530614d1e320bc5c2e100726dcd767ebf3d8f17ece4db0f7da1bfc141e8ca3
38d0d85cab406bb8deae0cc4e657b0da600c4c138270eab0858b2cc6ffba24b7adf14
192424eb4f5fd00bd03de54144353ad8dfe3d89b0a2c2f5b732fe49f8be985bb]
Time Elapsed: 43040  ns


MAC da:000000000011 MAC sa:000000000030 type:8137 crc:6e188a85
Payload
Numeric=[f8ded7255f482bf9bdb707a5d0bcd11fb860a6eae308c672013a6b28eb80
3ec1e83608c53a8eceda5489fddea53ad05e5d088f651e53e633f020d9e47238a28f9
f6c04dd23444aae8ff59bc96cd1bd22840501bf55e676c40ca106aef880b444d2a598
43d875bf5a0379a06762074998e26eee628802f9dd6fb3559a7f8f09603befdcf69b9
513684e7f14df4dc8ee9637478fdf404dbadd3b3072e39307fe0f9e412b1d3b7ef50c
145647837d69ec994a7966f38cd5cbcd97de51f044f1b820c6792d060a74dc220336d
1ccc7a84cc8ebd73e7f0513075bb7682e4c2ebe220e865bce28e621072ed13f92588a
aadeeef11a278ecce512dfd2638cac7348f3fbba98452132997c5a3a7ab1f2abfb3c8
33ce57b53899256592b4a38ecbdcea6af45c410b8db570b7b05d69e8a9a91c7a3b2c2
de286cc905700a8144113efb819d125ef4dc563d4a0dfe2c11384dc9a615432237902
da11b74c2550e48b1df91de14cbffa29e989731fa3b0a3d4168cb45e661ba340d9a5a
5cabf939e558f4aa33485a884a38191bb5d8abd1017d9aef4a9bbfdfe9d49b4ec8268
ab065fe1a6907af422b3a30dcc30c7b0ef61590f692e0686769a8d2f148009d0ca76c
f49a336a579fce011a644651d3da4f329078fe6e5e6b6294e05664f3826b9bc615263
5165178a155149d2447d370fe381b84430193c51bf85ee785394d7a4f6ac3d7636d]
Time Elapsed: 43040  ns
```

```
MAC da:000000000011 MAC sa:000000000030 type:8137 crc:3bc2e99a
Payload
Numeric=[bc1714c0d8a35c79186a3057e93a4f27572e45342c551080f6cf72e92d7b7
8b33fef474cd2d1a1cb01424af3dc84b541eb7d17d3134862a4836e24f91301ea69c13
01a6cad87d2d6457fde26f889381134323133f497bd4baec9ebedd8421a0fe725e45cf
46154afe3616eac447b87cdcbbf7cd38f47580de9c35e4836530ce13f6725e09bc8327
3c36ea2cd2c52f6b532dc0cbe0921158d918838134d10692240d28244fcea1f48b4784
b5886cf8411d48d1a82f2b97973eb8c5de2a81f811c478bca5edbbe765c719561a18e7
bd62f021b0a384169b571228bdbce37ec50249b02f48963033ce297270489eaa73967f
71866815554a0404a88e67a54ac759569ec1eaa263f1c1ab0d37d74a76ba319d45ee08
5f665b9a15fd0da10be5c3cd1c846902c1fc169472b8b3d5139a8b2aba40e54f3524e2
95e46f63c429fc19545b656b605b3a120b86186dd977e07ab8d5992c305008cbf48cbd
5b94b89671a6318ab29b0abbde3c191431c44bbfa7c37cd9c597bff5ff238a19325ee7
5a40ca7be19a6333bbbfec1e48ce4de9db8bb26c127679db7d4e210659a7c73c394559
55cbf1418ff0b651b6a6f45733942a4e4e173693bf6bd57b99a9fe4ae3c1b88b791038
275f8e3b33e1a44779b0c621dffbd0a01aef4b4c024ad9ca9f912e8dca0e39326e6924
32ca0d5abdb72fba8b50ed0bff9d5501e344af2d73f37dd6e34ed]
Time Elapsed: 43040  ns


MAC da:000000000011 MAC sa:000000000030 type:8137 crc:ad3d7cee
Payload
Numeric=[635f8ea1f0bcbc638cfaa0536ea0b27dab441a1023742ea5db44d19fea1a4
17fb3b41c5f7bf61974178553cad2bc900d2ffd6da4b8e6d7ea7b7abac162be465c07d
2c26f391544a0c453417cc08f59b2f76645e3979b58cdd4d06c14a1fbb07f2f913ddc4
9fdd1675266a8eee747c386f3eaa40c4349ca42495aead13e523af63eeed11a933a603
1c9e54f0dc0b88e8c5e1c11c506514e258653bd9994eda9b594124b03c5ff944e79228
32f8c6082adb9b59d1ff4bb0e3a7f22f974c89cc69cd649ffdb56281bf6fa951d11546
56ebccd98e090e8343a2d705237d5c8db6571f20cc94c8438ab4a2cdf78d7e9952872a
ed1fbf783160c93dc979a9448c963e11ef70e6b585b005136c87d0fe3659171b491959
b3260c6e790cfccf2c532a183d7d3bab09e65e0fb884099aea4f38f458233e02b26b8b
18e9fffff9983860bd5e48d4101eed11921da61c3105304135e013a93bf5a701dd3655
1ef9890df953c523ba0a7a8c27f822c6b78fbe30512176351eed5927e65eb851cdc6ac
0fe3a07771e5a57f105f1c7c6c6c09d0b0d8db22249bb545982e9f1ecdc91f0b14efd6
cf09a7d2ab90c5280380328894dd98b59001d498c00c413eace356272c6dfea89a43f9
ca4b6f4912558dbc7bdf1ce1b1e06f17033a33255c2a6f427a22d6c1364c1c4d4b1687
fa415482e3b9f08fa55fb870e926a2465274d370d4398db5595ec]
Time Elapsed: 43040  ns
```

Lesson 7

POST-PROCESSING SIMULATED EGRESS TRAFFIC

Goals

In this lesson you will learn:

- What is the purpose of trace files
- How to use the **printTrace.pl** script to filter trace files into a human-readable format

Overview

After using the C-Ware Software Toolset's (CST) C-Ware Simulator to run an application under simulation, one of the most important tasks that a developer would perform is to verify the correctness of the simulated egress traffic produced by the application. Doing so requires some degree of "post-processing" because the output from each simulated C-5 NP port is a pattern file.

One utility provided with the CST, the **printTrace.pl** Perl script, post-processes the pattern file output, called *trace files*, from a simulation and outputs human-readable and time-stamped representation of the data.

The format of pattern files is described in "Pattern Files" on page 93.

Trace Files

In the Simulator configuration file (**config**) that is used to run a simulation, for each Channel Processor SDP that transmits data, there must be an entry that specifies that interface's output trace filepath. For example:

```
Port0_TxTraceFile $(CPORT)\apps\gbeSwitch\gbeTrc0.pat
```

As the simulation executes and transmits data out the physical interface of a given CP, the specified file is written with the information that was transmitted out that CP.

Generating the Trace Files

To generate the trace files, simply run an application under the CST's C-Ware Simulator. You can run this simulation either interactively or in batch mode.

After you run this simulation, trace files (specified in the **config** file) will be written in the current directory, as follows:

```
C:\C-Port\Cst2.2\apps\gbeSwitch\run\outPatterns\c5i> dir *.pat
...

10/15/01  04:48p               521,882 gbeTrc13.pat
10/15/01  04:48p               519,279 gbeTrc14.pat
10/15/01  04:48p               512,232 gbeTrc15.pat
10/15/01  04:48p               554,479 gbeTrc5.pat
10/15/01  04:48p               553,030 gbeTrc6.pat
10/15/01  04:48p               523,632 gbeTrc7.pat
10/15/01  04:48p             1,613,245 gbeTxBase12.pat
10/15/01  04:48p                23,432 gbeTxBase4.pat
...
```

Analyzing the Trace Files

The contents of each of these pattern files contains the simulated egress data from the physical layer protocol for some CP. So, the first few bytes of the Fast Ethernet packets that were transmit out CP0 look like:

```
20.000000 257 0 0 0
20.000000 257 0 0 0
20.000000 257 0 0 0
20.000000 257 0 0 0
20.000000 259 0 0 0
20.000000 256 0 0 0
20.000000 256 0 0 0
20.000000 256 0 0 0
20.000000 256 0 0 0
20.000000 256 0 0 0
```

As stated previously, these trace files are difficult to read and analyze.

You can use the CST's **printTrace.pl** tool to turn this data into a formatted listing of human-readable packet or cell data. You can call **printTrace.pl** with a variety of command line arguments that transform the trace file's simulated physical layer bit stream into human-readable text.

Invoke **printTrace.pl** from a command line to produce output such as the following from the Gigabit Ethernet trace file that can be more easily visually inspected:

```
C:\C-Port\Cst2.2\apps\gbeSwitch\run\outPatterns\c5i>
  perl ..\..\..\..\..\bin\printTrace.pl gbeTrc13.pat

MAC da:000000000001 MAC sa:000000000012 type:0800 crc:804877f0
IP sa:868d0101 IP da:868d0101
Payload
Numeric=[4500002e000000000e019db3868d0101868d0101202a2a2a2a5061636b657
42f43656c6c2023302a2a2a2a200000]
Time Elapsed: 5760  ns


MAC da:000000000001 MAC sa:000000000012 type:0800 crc:1e48dd3c
IP sa:868d0101 IP da:868d0101
Payload
Numeric=[4500002e000000000e019db3868d0101868d0101202a2a2a2a5061636b657
42f43656c6c2023312a2a2a2a200000]
Time Elapsed: 5760  ns
```

```
MAC da:000000000001 MAC sa:000000000012 type:0800 crc:fd4f52b2
IP sa:868d0101 IP da:868d0101
Payload
Numeric=[4500002e000000000e019db3868d0101868d0101202a2a2a2a5061636b65
742f43656c6c2023322a2a2a2a200000]
Time Elapsed: 5760  ns


MAC da:000000000021 MAC sa:000000000151 type:0000 crc:7891f7e1
Payload
Numeric=[202a2a2a2a5061636b65742f43656c6c2023332a2a2a2a20000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000]
Time Elapsed: 10880  ns


MAC da:000000000021 MAC sa:000000000030 type:0000 crc:78cd8de1
Payload
Numeric=[202a2a2a2a5061636b65742f43656c6c2023342a2a2a2a20000000000000
00000000000000000000000000000000]
Time Elapsed: 5760  ns


MAC da:000000000021 MAC sa:000000000151 type:0000 crc:0bc3330e
Payload
Numeric=[202a2a2a2a5061636b65742f43656c6c2023352a2a2a2a20000000000000
00000000000000000000000000000000]
Time Elapsed: 5760  ns


MAC da:ffffffffffff MAC sa:000000000151 type:0000 crc:ad1d6cf3
Payload
Numeric=[202a2a2a2a5061636b65742f43656c6c2023362a2a2a2a20000000000000
00000000000000000000000000000000]
Time Elapsed: 5760  ns
```

You can see that this output presents the headers of the packets, interesting addresses, the CRC, and so on.

The **printTrace.pl** script produces other interesting information for other protocols, such as SONET overhead information, inter-packet gaps, and so on. This script is fully documented in the *C-Ware Simulation Environment User Guide* document.

Exercises

The following are some exercises to make sure that you are familiar enough with post-processing the simulated egress traffic produced by a CST Reference Library application.

- ❏ **Run the Gigabit Ethernet Switch application under the C-Ware Simulator, then use the "printTrace.pl" tool to produce a human-readable file that formats the simulated egress data cells produced by the application out of CP0.**

Lesson 8

PROGRAMMING THE NP SERIAL DATA PROCESSORS

Overview

Among the most powerful components of the C-Port Network Processor (NP) are the Channel Processor's (CP) Serial Data Processors (SDPs). The SDPs are a set of configurable components and microprogrammable sequencers that are optimized for data processing tasks. Duties for which general-purpose processor cores (especially RISC) are inefficient are optimized in the SDP components.

Due to the high degree of flexibility and "pipelined" nature of these processors, coupled with the fact that they are programmed in a proprietary microcode, the task of programming the SDPs is complex.

To aid programmers in developing software for the SDPs, the CST provides a set of tools and documentation to help with this task.

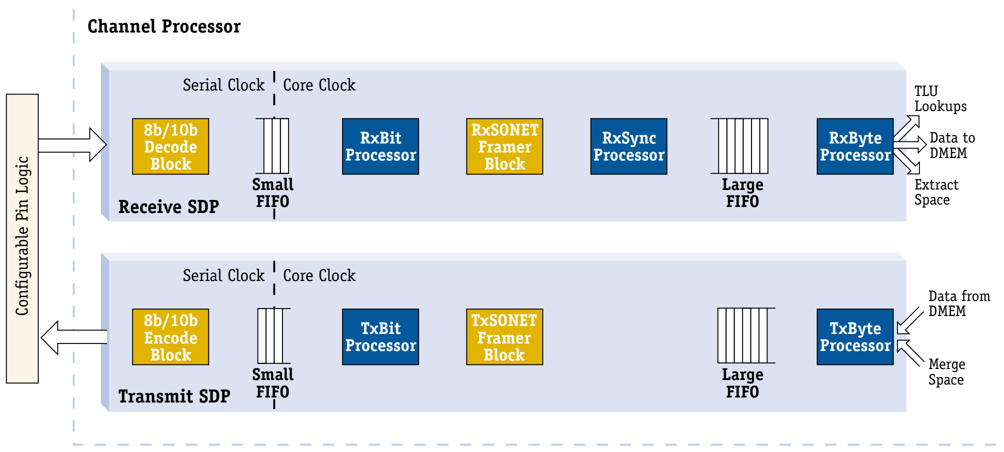
SDP Architecture Overview

As mentioned previously, the SDP architecture consists of a set of microsequencers that are either configured or programmed using a proprietary microcode that is coded in a manner similar to C language function calls. There are two SDPs per Channel Processor (CP): a receive SDP (RxSDP) and transmit SDP (TxSDP). Each has its own blocks and sequencers.

As depicted in Figure 10, the RxSDP comprises five sequencers, and the TxSDP comprises four.

Each of the dark colored sequencers are microprogrammable. This document describes that in much detail. Each of the light-colored blocks is a configurable (not programmable) block.

Figure 10 SDP Architecture



Common SDP Architecture

Each programmable microsequencer in the RxSDP and TxSDP has a common architecture.

The programmable microsequencers in the RxSDP are:

- Receive Bit (RxBit) Processor
- Receive Sync (RxSync) Processor
- Receive Byte (RxByte) Processor

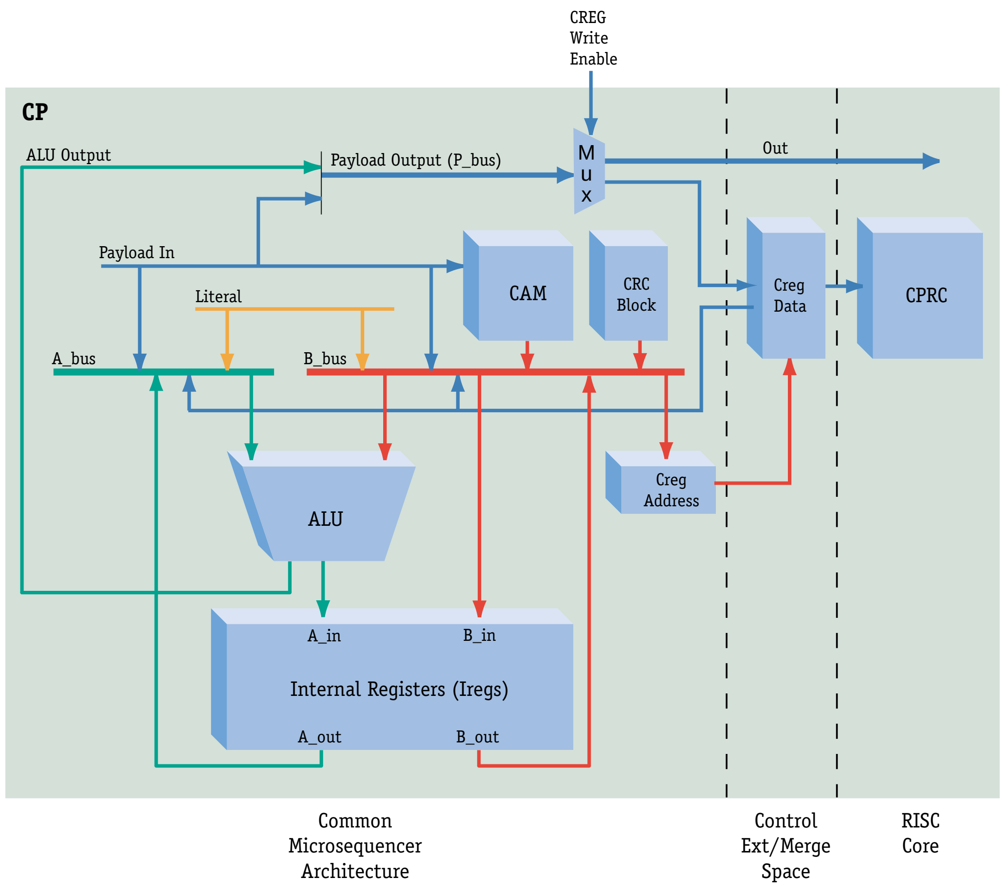
The programmable microsequencers in the TxSDP are:

- Transmit Byte (TxByte) Processor
- Transmit Bit (TxBit) Processor

Common Microsequencer Architecture

Each of the SDP microsequencers has the same architecture. They vary as to the amount of writable control store (WCS), Content Addressable Memory (CAM) storage, Cyclical Redundancy Check (CRC) engines and Internal Registers (Iregs).

Figure 11 Common Microsequencer Architecture



Each microsequencer in the SDP embodies this concept:

> Information is received from the downstream element from the 'Payload In' bus. Information is transmit upstream via the 'Payload Output' (*P_bus*). The microsequencer has access to various resources for decision-making, and performs pattern-matching for this task.

Microsequencer Components

The following are the components of the common microsequencers:

Payload In bus/Payload bus

The 'Payload In', or just 'Payload bus', is the bus on which the data is received from the downstream processing element. Depending on the position of the micro-sequencer in question, this downstream element may be another micro-sequencer, a FIFO, DMEM, and so on.

The payload bus delivers data in at up to 9-bits wide. As data comes in on this bus, it can be sent to match an entry in the CAM, put on the A-bus, put on the B-bus and send to the output bus.

Payload Out/P_bus

The 'Payload Out', or 'P_bus', is the bus on which the data is transmit to the upstream processing element. Depending on the position of the micro-sequencer in question, this upstream element may be another micro-sequencer, a FIFO, DMEM, the physical interface, and so on.

The Payload out bus transmits data up to 9-bits wide.

A-Bus

The A-bus is one of the two main internal busses inside the micro-sequencer. Data may be placed on the A-bus either from the Payload input, a synthesized literal or an Internal Register (Ireg).

The A-bus can only output to the ALU.

B-Bus

The B-bus is the other main internal bus inside the micro-sequencer. Data may be placed on the B-bus either from the Payload input, a synthesized literal, the CAM or the CRC engine.

The output of the B-bus can be the ALU, an Internal Register directly (without the ALU) or the Creg Address.

ALU

Each micro-sequencer has an ALU. Some of the functions that the ALU has in it are: add, subtract, pass-through, exclusive or, and so on. These commonly used functions can be used to match data and do testing for the data path implemented in the SDP microprogram.

Internal Registers (Iregs)

Each microsequencer has a set of internal registers (Iregs). These Iregs can be used for general purpose storage, and can be loaded onto either the A-bus or B-bus.

Certain of these Iregs may be used as counting registers with a single cycle increment function applied to them.

Certain counting registers may be concatenated together to form one 16-bit counting register.

Content Addressable Memory (CAM)

Certain microsequencers have available to them a CAM for pattern matching of Payload data. The CAMs (if present) are of varying sizes and can match up to 9 bits of data at one time in one core clock cycle.

Cyclic Redundancy (CRC) Block

Certain microsequencers have available to them a CRC block used for CRC calculation by the hardware. The number and types (CRC-5, CRC-10, CRC-16, CRC-32, etc.) vary according to the microsequencer.

At the end of a packet or cell time, under program control, the microprogram may test the result of the CRC with a single pass/fail bit inside the sequencer. Additionally, it may load the resulting bytes of the CRC from the B-bus.

Creg Address

Each micro-sequencer has the capability to write data to and read data from an address in configuration registers in the CP. This may be either control space, extract/merge space, ring bus registers. It may only access one 8-bit register at one time.

In order for the microsequencer to access the configuration registers, it must set up the Creg address to point to the correct register address.

Microsequencer Inputs and Outputs

As stated previously, each microsequencer's inputs and outputs vary depending on the position and configuration of the sequencer, or more specifically (depending on the sequencer being in the RxSDP or TxSDP) which components inside the SDP are enabled.

The following two tables present the inputs and outputs of each of the microsequencers based on configuration.

Microsequencers that are programmable are shown in italics.

Table 4 RxSDP Microsequencer Inputs/Outputs

MICROSEQUENCER	PAYLOAD IN	PAYLOAD OUT
Configurable Pin Logic	Rx Physical Interface	8b/10b decoder (if enabled) OR RxSmall FIFO
8b/10b decoder	Configurable Pin Logic	RxSmall FIFO
RxSmall FIFO	8b/10b encoder (if enabled) OR Configurable Pin Logic	*RxBit Processor*
RxBit Processor	RxSmall FIFO	RxSONET Framer (if enabled) OR *RxSync Processor* (if enabled) OR RxLargeFIFO
RxSONET Framer	*RxBit Processor*	*RxSync Processor* (if enabled) OR RxLargeFIFO
RxSync Processor	RxSONET framer (if enabled) OR *RxBit Processor*	RxLargeFIFO
RxLargeFIFO	*RxSync Processor* (if enabled) OR RxSONET framer (if enabled) OR *RxBit Processor*	*RxByte Processor*
RxByte Processor	RxLargeFIFO	CP DMEM

Table 5 TxSDP Micro-sequencer Inputs/Outputs

MICRO-SEQUENCER	PAYLOAD IN	PAYLOAD OUT
TxByte Processor	CP DMEM	TxLarge FIFO
TxLargeFIFO	*TxByte Processor*	TxSONET framer (if enabled) OR Base CP's *TxBit Processor* (if aggregation) OR *TxBit Processor*
TxSONET Framer	TxLargeFIFO	*TxBit Processor*
TxBit Processor	TxSONET framer (if enabled) OR TxLargeFIFO	TxSmallFIFO
TxSmallFIFO	*TxBit Processor*	8b/10b encoder (if enabled) OR Configurable Pin Logic
8b/10b encoder	TxSmallFIFO	Configurable Pin Logic
Configurable Pin Logic	8b/10b encoder (if enabled) OR TxSmallFIFO	Tx Physical Interface

Aids to SDP Programming

The brief topics below enumerate some of the most commonly encountered issues that arise when programming and debugging the SDPs under the C-Ware Simulator.

Microcode Language

As stated previously, since the SDPs are not general-purpose, they are programmed using their own microcode language.

This microlanguage is fully documented in the *C-Ware Microcode Programming Guide* document. One caveat for using a higher-level language for programming the primitives of the SDP is that there are often side-effects from using some of the functions in the language.

For example, any time a branch is performed to a microinstruction that is not the next instruction, the *Branch()* function uses the Literal bus. If the program attempts to use the Literal bus on the same instruction, the microassembler produces a warning.

For a full list of the issues that arise from programming the SDPs, see the *C-Ware Microcode Programming Guide* document.

Other Considerations

Below is a small list of the most commonly encountered problems with programming the SDP.

Creg Address Timing

When the microprogram sets the Creg address, a full two cycles are required for that address to fully propagate to the register. This means that any time the program reads from a Creg, it must ensure that there is a wait for two cycles.

When writing to a Creg, the SDP allows users to write the address and data to that address on the same clock cycle. The SDP guarantees that the address will be set before the data is written to it.

This rule also holds when modifying the Creg Address Pointer via an Creg increment function. This means that any time a piece of data is read, the pointer was written at least [n - 2] cycles ago.

ALU Testing

Running data through the ALU and producing a result can happen on the same cycle. However, if testing the ALU for that data, that data is not available until the following cycle.

This means that if the program performs an ADD command in the ALU, the status of that add (zero, non-zero, and so on) is not available until the next core clock cycle.

B-bus

The B-bus is the only bus in the SDP that can do certain necessary tasks, such as obtaining bytes from the CRC engine, programming the Creg Address, and so on. For this reason the programmer must be careful to ensure that the program does not overload the B-bus, else inconsistent results will occur in that microprogram.

P-bus

The Payload Output bus is used not only for payload output, but also for writing Creg data. In SDP programs, these two tasks operate independently under program control.

Microcode Symbols

Another issue that arises with simulating the SDPs is microcode symbols.

In every microprogram, there is a function named *LabelDef()* that defines a microcode label. These labels are actually program addresses that the microprogram and microassembler use for branching.

When debugging the SDP programs, these labels are also made available by the C-Ware Simulator, to make the program more readable.

However, the default behavior of the microassembler is to strip out the symbols before the SDP programs are put into the package file. Thus, you must edit the **sim.in** file used by the simulation to include these symbols for every SDP for which these symbols must be displayed.

To accomplish this, you must add one of the following lines to the **sim.in** files:

```
load sdp<n> <ucodeFileName.sdp>
```

or:

```
symbols sdp<n> <ucodeFileName.sdp>
```

This will ensure that the microprogram's symbols are properly displayed by the C-Ware Simulator when using microcode messaging and simulator tracing.

Visibility of SDPs in the C-Ware Simulator

Because the SDPs are not general-purpose, programmable components in conventional languages (like the C-5 NP's RISC cores), the SDPs aren't supported by the same types of debugging tools as the cores (such as a source code-based debugger).

For this reason the CST provides a variety of debugging aids for the SDPs, so that programmers can debug SDP programs under simulation before relying on the hardware for such tasks.

Microcode Messaging

One of most basic techniques for diagnosing SDP microprograms is *microcode messaging*. Microcode messaging is the equivalent for the SDPs of calling the C-Ware API *ksPrintf()* routine in a RISC core program.

The format of a microcode message is the following:

```
DCPSIM_MESSAGE(YOUR MESSAGE GOES HERE - NO QUOTES NEEDED);
```

Unlike a call to *ksPrintf()*, no evaluation can take place in this statement. Also, you must have loaded microcode symbols into the Simulator via the application's **sim.in** file before running the simulation and seeing the messages.

Additionally, you can enable or disable this feature *per sequencer* with the following syntax:

```
umsg <sequencerName> <0/1>
```

So, to enable this feature on RxByte0 (that is, the RxByte processor in CP0), you would issue the following Simulator command or include this command line in the **sim.in** file:

```
umsg RxByte0 1
```

To disable messaging in TxBit processor on CP 3:

```
umsg TxBit3 0
```

The output of microcode messaging is echoed to the Simulator's standard output. The format of such messages follows:

```
C:\C-Port\Cst2.2\apps\gbeSwitch\run> cwsim
c5sim, CST 2.2, C-5 [Rev D0] release build on Sept 12 2002 22:35:08
...
Dcp.CpCluster0.Cp0.Sdp> umsg rxbyte0 1
Dcp.CpCluster0.Cp0.Sdp.RxByte> g
```

```
[199853 0 RxByte0]  Microcode Message: START PROCESSING NEW FRAME
[199875 0 RxByte0]  Microcode Message: RECEIVED INTERNAL MAC DA
[199884 0 RxByte0]  Microcode Message: DETERMINING FID TO USE
[199887 0 RxByte0]  Microcode Message: NOT 8021Q TAGGED
[199909 0 RxByte0]  Microcode Message: MAC SA LOOKUP
[199912 0 RxByte0]  Microcode Message: PARSE MAC TYPE
[199920 0 RxByte0]  Microcode Message: MAC TYPE IS 80XX
[199924 0 RxByte0]  Microcode Message: PARSE IP
[200000 0 RxByte0]  Microcode Message: IPDA LOOKUP
[200009 0 RxByte0]  Microcode Message: STREAM PDU
[200060 0 RxByte0]  Microcode Message: SWITCH SCOPE
Halt: Dcp: Control C detected
```

This feature is useful when you want an overall view of what is happening in the microprogram without detailed Simulator tracing or stepping through code.

SDP Tracing

Because of the complexity of the SDP architecture and lack of other debugging tools, there is a great deal of tracing support about the state of the SDPs provided by the C-Ware Simulator.

Unlike other components supported in the Simulator, there are *four* levels of tracing available for the SDP microsequencers (including the FIFOs and configurable blocks).

Table 6 Software Simulator Tracing Levels for SDP Components

Tracing Level	Output
0	None.
1	Only data that is transferred on the Payload Out (P-bus) is printed.
2	Everything at level 1 plus Program counter, Creg Address, Number of bytes transmit on the P-bus and an indication if the sequencer did a read or write to the creg address.
3	All of levels 1 and 2 plus the entire micro-sequencer state.

Trace Level 1

Trace level 1 echoes information to the Simulator's standard output only when data is transmitted to the P-bus for the given sequencer. For example:

```
C:\C-Port\Cst2.2\apps\gbeSwitch\run> cwsim
c5sim CST 2.2, C-5e [Rev A1] release build on Oct 29 2002 22:35:08
...
Dcp.CpCluster0.Cp0.Sdp> trace rxbyte0 1
Dcp.CpCluster0.Cp0.Sdp.RxByte> g
[199926 1 RxByte0] (dataOut = 0x45)
[199935 1 RxByte0] (dataOut = 0x0)
[199937 1 RxByte0] (dataOut = 0x0)
[199938 1 RxByte0] (dataOut = 0x2E)
[199939 1 RxByte0] (dataOut = 0x0)
[199940 1 RxByte0] (dataOut = 0x0)
[199942 1 RxByte0] (dataOut = 0x0)
[199949 1 RxByte0] (dataOut = 0x0)
[199955 1 RxByte0] (dataOut = 0xF)
[199957 1 RxByte0] (dataOut = 0x1)
[199958 1 RxByte0] (dataOut = 0x9A)
[199959 1 RxByte0] (dataOut = 0xB3)
[199962 1 RxByte0] (dataOut = 0x86)
...
```

The 'dataOut' keyword in the output listing specifies the data that is sent "upstream" through the SDP. In this case, since the Simulator's focus is the CP's RxByte processor, this is DMEM.

Trace Level 2

Trace level 2 displays the information at trace level 1, plus certain additional information. For example:

```
C:\C-Port\Cst2.2\apps\gbeSwitch\run> c5sim
c5sim CST 2.2, C-5e [Rev A1] release build on Oct 29 2002 22:35:08
...
Dcp.CpCluster0.Cp0.Sdp> trace rxbyte0 2
Dcp.CpCluster0.Cp0.Sdp.RxByte> g
[199699 2 RxByte0] CPid: 0 PC=0x00 == COLD_START mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199700 2 RxByte0] CPid: 0 PC=0x01 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199701 2 RxByte0] CPid: 0 PC=0x01 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
```

```
[199702 2 RxByte0] CPid: 0 PC=0x01 ==  mBytesSent = 0 CregAddr=0x0
Token: TRUE
[199703 2 RxByte0] CPid: 0 PC=0x01 ==  mBytesSent = 0 CregAddr=0x0
Token: TRUE
[199704 2 RxByte0] CPid: 0 PC=0x02 == GO mBytesSent = 0 CregAddr=0x0
Token: TRUE
[199705 2 RxByte0] CPid: 0 PC=0x03 == INITIALIZE mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199706 2 RxByte0] CPid: 0 PC=0x04 == WAIT_FOR_FRAME mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199707 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199708 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199709 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199710 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199711 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199712 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199713 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199714 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199715 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199716 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199717 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199718 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199719 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199720 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199721 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199722 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199723 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
```

```
[199724 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199725 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199726 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199727 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199728 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199729 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199730 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199731 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199732 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199733 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199734 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199735 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199736 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
[199737 2 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0
CregAddr=0x0 Token: TRUE
...
```

For this level of tracing, the keywords in the output listing represent the following information shown in Table 7.

Table 7 Trace Level 2 Keywords

KEYWORD	MEANING
CPid	CP that emitted the trace message
PC	Value of the program counter (uPC). If there is a label on this microinstruction, it is also displayed.
mBytesSent	Number of bytes sent out the Payload Out bus
CregAddr	Value of the Creg address on this cycle

Trace Level 3

Trace level 3 displays all of the information at level 2, plus the entire state of the microsequencer. For example:

```
C:\C-Port\Cst2.2\apps\gbeSwitch\run> c5sim
c5sim CST 2.2, C-5e [Rev A1] release build on Oct 29 2002 22:35:08
Dcp> g 170000
Dcp> trace rxbyte0 3
Dcp.CpCluster0.Cp0.Sdp.RxByte> g 10
Dcp.CpCluster0.Cp0.Sdp.RxByte> g
[199699 3 RxByte0] CPid: 0 PC=0x00 ==  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x00000000 (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=0
                Iregs:    [0]=0x0   [1]=0x0   [2]=0x0   [3]=0x0
                          [4]=0x0   [5]=0x0   [6]=0x0   [7]=0x0
                          [8]=0x0   [9]=0x0   [10]=0x0   [11]=0x0
                          [12]=0x0   [13]=0x0   [14]=0x0   [15]=0x0
                CRC32:  CRCaccum = 0 CRCinput = 0 CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
*******************************************************************
[199700 3 RxByte0] CPid: 0 PC=0x01 ==  (stalled)  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x00000000 (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0   [1]=0x0   [2]=0x0   [3]=0x0
                          [4]=0x0   [5]=0x0   [6]=0x0   [7]=0x0
                          [8]=0x0   [9]=0x0   [10]=0x0   [11]=0x0
                          [12]=0x0   [13]=0x0   [14]=0x0   [15]=0x1
                CRC32:  CRCaccum = 0 CRCinput = 0 CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

[199701 3 RxByte0] CPid: 0 PC=0x01 ==  (stalled)  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x00000000 (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 1
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0   [1]=0x0   [2]=0x0   [3]=0x0
                          [4]=0x0   [5]=0x0   [6]=0x0   [7]=0x0
                          [8]=0x0   [9]=0x0   [10]=0x0   [11]=0x0
                          [12]=0x0   [13]=0x0   [14]=0x0   [15]=0x1
                CRC32:  CRCaccum = 0 CRCinput = 0 CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
*******************************************************************
[199702 3 RxByte0] CPid: 0 PC=0x01 ==  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x00000100 (valid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 1
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0   [1]=0x0   [2]=0x0   [3]=0x0
                          [4]=0x0   [5]=0x0   [6]=0x0   [7]=0x0
                          [8]=0x0    [9]=0x0    [10]=0x0    [11]=0x0
                          [12]=0x0    [13]=0x0    [14]=0x0    [15]=0x1
                CRC32:  CRCaccum = 0 CRCinput = 0 CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
*******************************************************************
[199703 3 RxByte0] CPid: 0 PC=0x01 ==  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x000001FF (valid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0    [1]=0x0    [2]=0x0    [3]=0x0
                          [4]=0x0    [5]=0x0    [6]=0x0    [7]=0x0
                          [8]=0x0    [9]=0x0    [10]=0x0    [11]=0x0
                          [12]=0x0    [13]=0x0    [14]=0x0    [15]=0x1
                CRC32:  CRCaccum = 0 CRCinput = 100 CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
*******************************************************************
[199704 3 RxByte0] CPid: 0 PC=0x02 ==  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x000001FF (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0    [1]=0x0    [2]=0x0    [3]=0x0
                          [4]=0x0    [5]=0x0    [6]=0x0    [7]=0x0
                          [8]=0x0    [9]=0x0    [10]=0x0    [11]=0x0
                          [12]=0x0    [13]=0x0    [14]=0x0    [15]=0x2
                CRC32:  CRCaccum = 0 CRCinput = 1FF CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x1 0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

[199705 3 RxByte0] CPid: 0 PC=0x03 ==  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x000001FF (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0    [1]=0x0    [2]=0x0    [3]=0x0
                          [4]=0x0    [5]=0x0    [6]=0x0    [7]=0x0
                          [8]=0x0    [9]=0x0    [10]=0x0    [11]=0x0
                          [12]=0x0    [13]=0x0    [14]=0x0    [15]=0x3
                CRC32:  CRCaccum = 0 CRCinput = 1FF CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x2 0x1 0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0
*******************************************************************
[199706 3 RxByte0] CPid: 0 PC=0x04 ==  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x000001FF (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0    [1]=0x0    [2]=0x0    [3]=0x0
                          [4]=0x0    [5]=0x0    [6]=0x0    [7]=0x0
                          [8]=0x0    [9]=0x0    [10]=0x0    [11]=0x0
                          [12]=0x0    [13]=0x0    [14]=0x0    [15]=0x4
                CRC32:  CRCaccum = 0 CRCinput = 1FF CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x3 0x2 0x1 0x1 0x0 0x0 0x0 0x0 0x0 0x0
*******************************************************************
[199707 3 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x000001FF (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
              Abus=0xFF Bbus=0x0 ALUout=0xFF Pbus=0xFF ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0   [1]=0x0   [2]=0x0   [3]=0x0
                          [4]=0x0   [5]=0x0   [6]=0x0   [7]=0x0
                          [8]=0x0   [9]=0x0   [10]=0x0   [11]=0x0
                          [12]=0x0   [13]=0x0   [14]=0x0   [15]=0x5
                CRC32:  CRCaccum = 0 CRCinput = 1FF CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x4 0x3 0x2 0x1 0x1 0x0 0x0 0x0 0x0 0x0
*******************************************************************
[199708 3 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x000001FF (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
              Abus=0xFF Bbus=0x0 ALUout=0xFF Pbus=0xFF ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0   [1]=0x0   [2]=0x0   [3]=0x0
                          [4]=0x0   [5]=0x0   [6]=0x0   [7]=0x0
                          [8]=0x0   [9]=0x0   [10]=0x0   [11]=0x0
                          [12]=0x0   [13]=0x0   [14]=0x0   [15]=0x5
                CRC32:  CRCaccum = 0 CRCinput = 1FF CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x4 0x3 0x2 0x1 0x1 0x0 0x0 0x0 0x0 0x0
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

[199709 3 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x000001FF (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
              Abus=0xFF Bbus=0x0 ALUout=0xFF Pbus=0xFF ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0   [1]=0x0   [2]=0x0   [3]=0x0
                          [4]=0x0   [5]=0x0   [6]=0x0   [7]=0x0
                          [8]=0x0   [9]=0x0   [10]=0x0   [11]=0x0
                          [12]=0x0   [13]=0x0   [14]=0x0   [15]=0x5
                CRC32:  CRCaccum = 0 CRCinput = 1FF CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x4 0x3 0x2 0x1 0x1 0x0 0x0 0x0 0x0 0x0
*******************************************************************
[199710 3 RxByte0] CPid: 0 PC=0x05 ==  (stalled)  mBytesSent = 0 CregAddr=0x0 Token: TRUE
                DataIn: 0x000001FF (invalid)  mDataOut: 0x0 (invalid)
                Input FIFO depth: 0
              Abus=0xFF Bbus=0x0 ALUout=0xFF Pbus=0xFF ALU flags (from prev cycle ALU op):
N=0 C=0 C(link)=0 Z=1
                Iregs:    [0]=0x0   [1]=0x0   [2]=0x0   [3]=0x0
                          [4]=0x0   [5]=0x0   [6]=0x0   [7]=0x0
                          [8]=0x0   [9]=0x0   [10]=0x0   [11]=0x0
                          [12]=0x0   [13]=0x0   [14]=0x0   [15]=0x5
                CRC32:  CRCaccum = 0 CRCinput = 1FF CRCnext = 0 CRC32 Load State: 0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF0931943
                uPC trace:  0x4 0x3 0x2 0x1 0x1 0x0 0x0 0x0 0x0 0x0
*******************************************************************
```

For this level of tracing, the keywords in the output listing represent the information shown in Table 8.

Table 8 Trace Level 3 Keywords

KEYWORD	MEANING
CPid	CP that emitted the trace message.
PC	Value of the program counter (uPC). If there is a label on this micro-instruction, it will also be displayed.
mBytesSent	Number of bytes sent out the Payload Out bus.
CregAddr	The value of the Creg address on this cycle.
mDataIn	The value of the data on the Payload In bus. Note that this data is not necessarily valid on every single cycle. On cycles that this data is valid (and is waiting for the micro-program to read it), it will be marked with the *(valid)* marker.
mDataOut	The value of the data on the Payload Out (P-bus) bus. Note that this data isn't necessarily valid on every single cycle. It will only be valid on cycles that the micro-program has made it valid under software control. When this field is valid, it will be marked with the *(valid)* marker.
Abus	This is the value of the A-bus on the given cycle.
Bbus	This is the value of the B-bus on the given cycle.
ALUout	This is the value of the ALU output on the given cycle.
Pbus	This is the value of the Pbus on the given cycle.
ALU flags	These are the values of the ALU flags. Note that they are from the result of the ALU operation from the previous cycle (if there was one).
Iregs<n>	These are the values of all 16 Iregs for the given sequencer on the given cycle.
CRC registers	The values of the CRC accumulator and associated CRC registers. It also lists the CRC mode (i.e., CRC-32).
SDP Mode register	This is the value of the SDP mode register for this SDP (either 3 or 5 for receive or transmit, respectively).
uPC trace	The micro-PCs of the last 10 instructions executed.

Analyzing SDP Performance

Unlike measuring performance of a program running on a RISC core, when measuring performance on the SDP you do not simply count the number of processor cycles per packet or cell time. This is because the SDP does not do things *per packet time*, it does them *per byte*. Also, there are several FIFOs built into the SDP that absorb the processing latency, so it is a more complex analysis.

The best metric to observe for the SDP is the depth of the large FIFOs; this is a gauge of the performance of the RxByte and TxByte processors. The depth of the FIFOs indicates how close to the 'edge' the microprogram runs, before the overflow. The FIFOs have three levels of tracing available.

Table 9 FIFO Tracing Levels

TRACING LEVEL	OUTPUT
0	None.
1	Only data that is read out of the FIFO on that clock cycle is printed.
2	Everything at level 1 plus the depth of the FIFO on that cycle, plus the deepest the FIFO got during the simulation.

Trace Level 1

Trace level 1 echoes information to the Simulator's standard output only when data is transmitted out of the FIFO.

```
C:\C-Port\Cst2.2\apps\gbeSwitch\run> cwsim
c5sim CST 2.2, C-5e [Rev A1] release build on Oct 29 2002 22:35:08
...
Dcp.CpCluster0.Cp0.Sdp> trace rxlgfifo0 1
Dcp.CpCluster0.Cp0.Sdp.RxLgFifo> g
[199702 1 RxLgFifo0] (dataOut = 0x100)
[199703 1 RxLgFifo0] (dataOut = 0x1FF)
[199848 1 RxLgFifo0] (dataOut = 0x10A)
[199850 1 RxLgFifo0] (dataOut = 0x0)
[199862 1 RxLgFifo0] (dataOut = 0x0)
[199863 1 RxLgFifo0] (dataOut = 0x0)
[199864 1 RxLgFifo0] (dataOut = 0x0)
[199865 1 RxLgFifo0] (dataOut = 0x0)
[199866 1 RxLgFifo0] (dataOut = 0x12)
[199868 1 RxLgFifo0] (dataOut = 0x0)
[199878 1 RxLgFifo0] (dataOut = 0x0)
[199879 1 RxLgFifo0] (dataOut = 0x0)
[199880 1 RxLgFifo0] (dataOut = 0x0)
[199881 1 RxLgFifo0] (dataOut = 0x0)
[199882 1 RxLgFifo0] (dataOut = 0x21)
[199883 1 RxLgFifo0] (dataOut = 0x8)
Halt: Dcp: Control C detected
```

Trace Level 2

Trace level 2 echoes all the information from trace level one, plus additional information.

```
C:\C-Port\Cst2.2\apps\gbeSwitch\run> cwsim
c5sim CST 2.2, C-5e [Rev A1] release build on Oct 29 2002 22:35:08
...
Dcp.CpCluster0.Cp0.Sdp> trace rxlgfifo0 2
Dcp.CpCluster0.Cp0.Sdp.RxLgFifo> g
[199699 2 RxLgFifo0] Depth(0), Empty(1),
[199701 2 RxLgFifo0] Depth(1), Empty(0), *
[199702 2 RxLgFifo0] (dataOut = 0x100)
[199702 2 RxLgFifo0] Depth(1), Empty(0), *
[199703 2 RxLgFifo0] (dataOut = 0x1FF)
[199703 2 RxLgFifo0] Depth(0), Empty(1),  |1
[199847 2 RxLgFifo0] Depth(1), Empty(0), *
[199848 2 RxLgFifo0] (dataOut = 0x10A)
[199848 2 RxLgFifo0] Depth(0), Empty(1),  |1
[199849 2 RxLgFifo0] Depth(1), Empty(0), *
[199850 2 RxLgFifo0] (dataOut = 0x0)
```

```
[199850 2 RxLgFifo0] Depth(0), Empty(1),  |1
[199851 2 RxLgFifo0] Depth(1), Empty(0), *
[199852 2 RxLgFifo0] Depth(2), Empty(0), **
[199853 2 RxLgFifo0] Depth(3), Empty(0), ***
[199854 2 RxLgFifo0] Depth(4), Empty(0), ****
[199855 2 RxLgFifo0] Depth(5), Empty(0), *****
[199857 2 RxLgFifo0] Depth(6), Empty(0), ******
[199858 2 RxLgFifo0] Depth(7), Empty(0), *******
[199859 2 RxLgFifo0] Depth(8), Empty(0), ********
[199860 2 RxLgFifo0] Depth(9), Empty(0), *********
[199861 2 RxLgFifo0] Depth(10), Empty(0), **********
[199862 2 RxLgFifo0] (dataOut = 0x0)
[199862 2 RxLgFifo0] Depth(9), Empty(0), ********* |10
[199863 2 RxLgFifo0] (dataOut = 0x0)
[199863 2 RxLgFifo0] Depth(9), Empty(0), ********* |10
[199864 2 RxLgFifo0] (dataOut = 0x0)
[199864 2 RxLgFifo0] Depth(9), Empty(0), ********* |10
[199865 2 RxLgFifo0] (dataOut = 0x0)
[199865 2 RxLgFifo0] Depth(9), Empty(0), ********* |10
[199866 2 RxLgFifo0] (dataOut = 0x12)
[199866 2 RxLgFifo0] Depth(9), Empty(0), ********* |10
[199867 2 RxLgFifo0] Depth(10), Empty(0), **********
[199868 2 RxLgFifo0] (dataOut = 0x0)
[199868 2 RxLgFifo0] Depth(9), Empty(0), ********* |10
[199869 2 RxLgFifo0] Depth(10), Empty(0), **********
[199870 2 RxLgFifo0] Depth(11), Empty(0), ***********
[199871 2 RxLgFifo0] Depth(12), Empty(0), ************
[199873 2 RxLgFifo0] Depth(13), Empty(0), *************
[199874 2 RxLgFifo0] Depth(14), Empty(0), **************
[199876 2 RxLgFifo0] Depth(15), Empty(0), ***************
[199877 2 RxLgFifo0] Depth(16), Empty(0), ****************
[199878 2 RxLgFifo0] (dataOut = 0x0)
[199878 2 RxLgFifo0] Depth(16), Empty(0), ****************
[199879 2 RxLgFifo0] (dataOut = 0x0)
[199879 2 RxLgFifo0] Depth(15), Empty(0), *************** |16
[199880 2 RxLgFifo0] (dataOut = 0x0)
[199880 2 RxLgFifo0] Depth(15), Empty(0), *************** |16
[199881 2 RxLgFifo0] (dataOut = 0x0)
[199881 2 RxLgFifo0] Depth(15), Empty(0), *************** |16
[199882 2 RxLgFifo0] (dataOut = 0x21)
[199882 2 RxLgFifo0] Depth(14), Empty(0), **************  |16
[199883 2 RxLgFifo0] (dataOut = 0x8)
[199883 2 RxLgFifo0] Depth(14), Empty(0), **************  |16
[199884 2 RxLgFifo0] Depth(15), Empty(0), *************** |16
[199886 2 RxLgFifo0] Depth(16), Empty(0), ****************
```

```
[199887 2 RxLgFifo0] Depth(17), Empty(0), *****************
[199888 2 RxLgFifo0] Depth(18), Empty(0), ******************
[199890 2 RxLgFifo0] Depth(19), Empty(0), *******************
[199891 2 RxLgFifo0] Depth(20), Empty(0), ********************
[199893 2 RxLgFifo0] Depth(21), Empty(0), *********************
[199894 2 RxLgFifo0] Depth(22), Empty(0), **********************
[199896 2 RxLgFifo0] Depth(23), Empty(0), ***********************
[199898 2 RxLgFifo0] Depth(24), Empty(0), ************************
[199899 2 RxLgFifo0] Depth(25), Empty(0), *************************
[199900 2 RxLgFifo0] Depth(26), Empty(0), **************************
[199901 2 RxLgFifo0] Depth(27), Empty(0), ***************************
[199903 2 RxLgFifo0] Depth(28), Empty(0), ****************************
...
```

SDP Microcode Examples

Below are two SDP code examples from the Gigabit Ethernet Switch application: the ‘gmiiRxBit’ and ‘RxByte’ microprograms.

‘gmiiRxBit’ Code Example

The following code is found in this file:

apps\components\phy\chip\np\sdp\src\gmiiRxBit.c

```
/******************************************************************************
 * Copyright (c) 1999, 2000, 2001 C-Port Corporation, a Motorola Company     *
 * All Rights Reserved                                                        *
 *                                                                            *
 * The information contained in this file is C-Port Corporation confidential *
 * and proprietary.                                                           *
 * Any reproduction, use or disclosure, in whole or in part, of this         *
 * program, including any attempt to obtain a human-readable version of this *
 * program, without the express, prior written consent of C-Port             *
 * Corporation or Motorola Incorporated is strictly prohibited.              *

******************************************************************************/

/***************************************************************************/
/*                  Gigabit GMII RX Bit Processor                          */
/***************************************************************************/
/*
 * This file implements the SDP RX Bit Processor Ucode for gigabit over GMII.
 *
 * Following a series of bytes to the RXBYTE processor there will be a single
 * byte with Bit8 on to flag EOF.  Bits 7-0 of this EOF contain the framing
 * status code (0 = OK).
 *
 * Small FIFO details:
 * -------------------
 * 0-7  = data
 * 8    = RX_DV (also goes to PhyStatus0 condition)
 * 9    = RX_ER (goes only to PhyStatus1 condition, not CAM)
 *
 * Datapaths and CAM work like 10 bit mode (SR1, SR2 disappear, CAM input
 * is PayloadIn directly). However, the 9th CAM input bit is equal
 * to (RX_DV * ~RX_ER) so that both conditions can be monitored
 * on the same cycle while examining the data pattern (necessary during
 * preamble stripping, for example.)
 */

#define RXBIT

/***************************************************************************/
/*    T H E   M I C R O C O D E                                            */
/***************************************************************************/
```

```
#include <stdio.h>
#include <sdpUcode.h>
#include "ethernetDefinitions.h"
#include "gmiiBitIf.h"

void
SDPmain()
{
    /**********************************************************************/
    /*    Optional User-provided comment for Ucode and CAM output files      */
    /**********************************************************************/
    Description="GMII Full-Duplex Gigabit Ucode for the SDP RX Bit Processor.";

    /* The input rate is 9 ns / byte = 1.49 cycles/byte @ 166 Mhz.
     *
     * End of frame is indicated by RX_DV going low.  The 9th CAM bit
     * is RX_DV * ~RX_ER.  When this bit de-asserts, ucode examines
     * the 9th bit of Ireg0 to differentiate which signal caused the
     * EOF condition, ~RX_DV or RX_ER.
     *
     * When EOF occurs we send a byte with it's 9th bit set up stream to
     * serve as a EOF marker to the RX Byte processor. */


#define RX_DV                          PhyStatus0
#define RX_ER                          PhyStatus1

#undef  LastByteRead
#define LastByteRead                   Ireg0
#define BYTECOUNT_MS                   Ireg2
#define BYTECOUNT_LS                   Ireg3
#define BYTECOUNT                      BYTECOUNT_LS
#define Flags                          Ireg6

#define CRC_LEN                        4
#define VLAN_LEN                       4
#define FRAME_LENGTH_CHECK_MS          MAX_FRAME_LENGTH_MS
#define FRAME_LENGTH_CHECK_LS          (MAX_FRAME_LENGTH_LS+VLAN_LEN+CRC_LEN)

#define MARK_START_OF_FRAME            0x0A

#define LOG_UPC ;

/*
#define LOG_UPC \
    Bbus(Literal(3)); \
    Abus(IregsA(Ireg15)); \
    ALU(PassA); \
    Pbus(ALUout); \
    Actions(CregsWrite|CregAddrWrite); \
    Branch(ALWAYS,Upc+1);
*/
    /*****************************************************************/
    /*   C O L D _ S T A R T                                         */
    /*****************************************************************/

    /* Send the two required pipeline scrubbing byte patterns. */
    Abus(Literal(0));
    ALU(PassA);
    Pbus(ALUout);
    Actions(Merge9+DataOutValid);
    Branch(ALWAYS, Upc+1);

    Abus(Literal(0xFF));
    ALU(PassA);
    Pbus(ALUout);
    Actions(Merge9+DataOutValid);
    Branch(ALWAYS, Upc+1);

    IregORlit(STATUSreg,CONCAT1);
    Branch(ALWAYS,Upc+1);

    /*****************************************************************/
    /*   W A I T _ F O R _ I D L E                                   */
    /*****************************************************************/
LabelDef(WAIT_FOR_IDLE);

    IregInit(Flags,gmiiBitFrameStatusSuccess,UseBbus);
    Branch(ALWAYS,Upc+1);

    /* Init the 16-bit byte counter to the appropriate neg value
     * to count up.  If it hitrs all ones then frame is too long.
     */
    IregInit(BYTECOUNT_MS,~FRAME_LENGTH_CHECK_MS,UseBbus);
    Actions(CregAddrIncr);
    Branch(ALWAYS,Upc+1);

    IregInit(BYTECOUNT_LS,~FRAME_LENGTH_CHECK_LS,UseBbus);
    Branch(ALWAYS,Upc+1);

LabelDef(IDLE_LOOP);

    /*
     * Use CAM match on Data9 to detect start of frame
     */
    Actions(UnloadFIFO|RepeatUntil);
    CAMMATCH("111010101",SomeGroup,Label(RX_SFD));
    Branch(CAMmatch,CAMMATCH("101010101",SomeGroup,Label(STRIP_PREAMBLE)));

LabelDef(WAIT_FOR_EOF);
    /* We come here to discard whatever remains of a carrier event. */

    Actions(UnloadFIFO|RepeatUntil);
    BranchNot(PhyStatus0,Label(WAIT_FOR_IDLE));

    /****************************************************************/
    /*   S T R I P _ O F F _ P R E A M B L E                        */
    /****************************************************************/
LabelDef(STRIP_PREAMBLE);
    DCPSIM_MESSAGE(RECEIVING PREAMBLE);

    /*
     * Strip off the preamble, and be vigilant for early-end.
     *
     */

    /* Monitor RX_DV to detect early end and loop until SFD */
    Actions(UnloadFIFO);
    CAMMATCH("0xxxxxxxx", SomeGroup, Label(IDLE_LOOP));
    CAMMATCH("111010101",SomeGroup,Label(RX_SFD));                          /*
0xD5 (SFD) */
   Branch(CAMmatch,CAMMATCH("101010101",SomeGroup,Label(STRIP_PREAMBLE)));  /*
0x55 */

    /* Read something besides AA or AB, go sink the event */
    Branch(ALWAYS,Label(WAIT_FOR_EOF));

    /*------------------------------------------------------*/
LabelDef(RX_SFD);

    /* If we don't own the token then don't receive frame */
    BranchNot(AggToken,Label(WAIT_FOR_EOF));

    DCPSIM_MESSAGE(SFD RECEIVED);

/* Send start of frame marker upstream. */
    Abus(Literal(MARK_START_OF_FRAME));
    ALU(PassA);
    PayloadOut(ALUout);
    Actions(Merge9);
    Branch(ALWAYS,Upc+1);

    /* Start streaming bytes of the packet.
     * Save each unloaded byte in Ireg0 so that when the loop terminates
     * we can differentiate between the EOF and error condition. Note
     * that we must use Ireg0 since our intent is to capture Data9.
     */

    /* Prime the 1-deep payload pipeline */
    Bbus(Payload);
    IregsB(LastByteRead);
    Actions(IregsBwrite|UnloadFIFO|IregIncr(BYTECOUNT));
    Branch(ALWAYS,Upc+1);

LabelDef(STREAM_FRAME_LOOP_TOP);

    /*
     * Stream the data up.
     */

    Bbus(Payload);
    IregsB(LastByteRead);
    Abus(IregsA(LastByteRead));
    ALU(PassA);
    PayloadOut(ALUout);
    Actions(UnloadFIFO|IregIncr(BYTECOUNT)|IregsBwrite);

Branch(CAMmatch,CAMMATCH("0XXXXXXXX",SomeGroup,Label(CARRIER_EOF_OR_ERROR)));

    Bbus(Payload);
    IregsB(LastByteRead);
    Abus(IregsA(LastByteRead));
    ALU(PassA);
    PayloadOut(ALUout);
    Actions(UnloadFIFO|IregIncr(BYTECOUNT)|IregsBwrite);

Branch(CAMmatch,CAMMATCH("0XXXXXXXX",SomeGroup,Label(CARRIER_EOF_OR_ERROR)));

    Bbus(Payload);
    IregsB(LastByteRead);
    Abus(IregsA(LastByteRead));
    ALU(PassA);
    PayloadOut(ALUout);
    Actions(UnloadFIFO|IregIncr(BYTECOUNT)|IregsBwrite);

Branch(CAMmatch,CAMMATCH("0XXXXXXXX",SomeGroup,Label(CARRIER_EOF_OR_ERROR)));

    Bbus(Payload);
    IregsB(LastByteRead);
    Abus(IregsA(LastByteRead));
    ALU(PassA);
    PayloadOut(ALUout);
    Actions(UnloadFIFO|IregIncr(BYTECOUNT)|IregsBwrite);

Branch(CAMmatch,CAMMATCH("0XXXXXXXX",SomeGroup,Label(CARRIER_EOF_OR_ERROR)));
```

```
    BranchNot(AllOnes(BYTECOUNT),Label(STREAM_FRAME_LOOP_TOP));

LabelDef(FRAME_TOO_LONG);

    DCPSIM_MESSAGE(GIGABIT GMII RX FRAME TOO LONG);
    IregInit(Flags,gmiiBitFrameStatusLengthError,UseBbus);
    Branch(ALWAYS,Upc+1);

    Branch(ALWAYS,Label(SEND_EOP));

LabelDef(CARRIER_EOF_OR_ERROR);

    /* If RX_ER is on then frame was errored, otherwise it was a normal EOF. */
    Actions(UnloadFIFO);
    Branch(RX_ER,Label(CARRIER_ERROR));

LabelDef(CHECK_FOR_RUNT);
    IregADDlit(BYTECOUNT_LS,FRAME_LENGTH_CHECK_LS-64);
    Actions(UnloadFIFO);
    Branch(ALWAYS,Upc+1);

    IregADDClit(BYTECOUNT_MS,FRAME_LENGTH_CHECK_MS);
    Actions(UnloadFIFO);
    Branch(ALWAYS,Upc+1);

    Actions(UnloadFIFO);
    BranchNot(ALUneg,Label(SEND_EOP));

    DCPSIM_MESSAGE(FRAME TOO SHORT);
    LOG_UPC;

    IregInit(Flags,gmiiBitFrameStatusLengthError,UseBbus);
    Actions(UnloadFIFO);
    Branch(ALWAYS,Upc+1);

    Actions(UnloadFIFO);
    Branch(ALWAYS,Label(SEND_EOP));

LabelDef(CARRIER_ERROR);

    DCPSIM_MESSAGE(RX_ER RECEIVED FROM PCS LAYER);
    /* RX_DV is still on, so RX_ER was asserted.  Set the FrameException flag */
    IregInit(Flags,gmiiBitFrameStatusPCSdataError,UseBbus);
    Actions(UnloadFIFO);
    Branch(ALWAYS,Upc+1);

LabelDef(SEND_EOP);
```

```
    /* Send up the EndOfPacket marker with frame status embedded in it. */
    Abus(IregsA(Flags));
    ALU(PassA);
    Actions(Merge9);
    PayloadOut(ALUout);
    Actions(UnloadFIFO);
    Branch(ALWAYS,Upc+1);

    /* Pass the aggregation token. */
    IregORlit(STATUSreg,CSR_AggTokenOut);
    IregsB(TMP1);
    Actions(IregsBwrite);
    Branch(ALWAYS,Upc+1);

    IregXOR(STATUSreg,TMP1);
    Branch(ALWAYS,Label(WAIT_FOR_EOF));
}
```

'RxByte' Code Example

The following code is found in this file:

apps\gbeSwitch\chip\np\sdp\src\rxByte.c

```
/*****************************************************************************
 * Copyright (c) 1999, 2000, 2001 C-Port Corporation, a Motorola Company
*
 * All Rights Reserved                                                      *
 *                                                                          *
 * The information contained in this file is C-Port Corporation confidential
and proprietary.   *
 * Any reproduction, use or disclosure, in whole or in part, of this        *
 * program, including any attempt to obtain a human-readable version of this *
 * program, without the express, prior written consent of C-Port            *
 * Corporation or Motorola Incorporated is strictly prohibited.
*

****************************************************************************/

/*************************************************************************/
/*                  10/100 Ethernet RX Byte Processor                    */
/*************************************************************************/
/*
 * This file implements the SDP RX Byte Processor Ucode for all speeds of
 * Ethernet.
 *
 * The 9th bit of the input FIFO is EOP (EndOfPacket), which will
 * be asserted by the RX Bit processor on the status byte for the frame,
 * which follows immediately after the final data byte of the frame. */

#define RXBYTE

/*************************************************************************/
/*    T H E   M I C R O C O D E                                          */
/*************************************************************************/
#include <stdio.h>
#include "ipIf.h"

/* Need to define __inline__ as a null or else VC++ chokes on it */
#define __inline__

#include "sdpUcode.h"
#include "rcSdpApiIf.h"
#include "enetDefs.h"

#define LOG_UPC_ADDR ((int)&((MacHeader*)NULL)->unused2)

/*
#define LOG_UPC \
    CregsAddrWrite(9); \
    Abus(IregsA(Ireg15)); \
    ALU(PassA); \
    Pbus(ALUout); \
    Actions(CregsWrite); \
    Branch(ALWAYS,Upc+1);

*/

#define LOG_UPC ;


void
SDPmain()
{
    /************************************************************************/
    /*    Optional User-provided comment for Ucode and CAM output files      */
    /************************************************************************/
    Description = "Ethernet Ucode (all speeds) for the SDP RX Byte Processor.";

    /* These vars are used as parameters to the high-level call ExtractField().
*/
    char* l4EOFtarget;
    int l4Len,l4LoopReg,l4ToDmem,l4ToCregs,
    l4CregAddr,l4IregA,l4OtherActions;

LabelDef(COLD_START);
    Branch(ALWAYS,Upc+1);

    /* Flush the fifo. Any garbage on the fifo is flushed here. RxBit
     * puts the 0x1FF on the fifo to indicate that it is up and ready.
     * anything prior to that is garbage!!!
     */
    Actions(RepeatUntil|UnloadFIFO);
    Branch(CAMmatch,CAMMATCH("111111111",SomeGroup,Label(GO)));

LabelDef(GO);

    Actions(Merge9);
    Branch(ALWAYS,Upc+1);

    /************************************************************************/
    /*         I N I T I A L I Z E                                           */
    /************************************************************************/
LabelDef(INITIALIZE);

    /* Per-frame initialization:
     * FrameLength MS:LS = all ones (because count includes EOF marker)
    * STATUSreg = 0 (no errors, no token out, all counters in byte mode, CAMin
= Payload)
```

```
     * The frame length counter will be changed to 16-bit mode later
     * CRC = 48 ones */

    /* First half of initialization */
    IregClear2(STATUSreg, Flags);
    Branch(ALWAYS,Upc+1);

    /**************************************************************************/
    /*         W A I T _ F O R _ F R A M E                                    */
    /**************************************************************************/
LabelDef(WAIT_FOR_FRAME);

    // LOG_UPC;

    IregClear2(FrameLengthLS, FrameLengthMS);
    Branch(ALWAYS, Upc+1);

    Actions(RepeatUntil);
    Branch(RxDataValid, Upc+1);

    BranchNot(AggToken,Label(SINK_FRAME));

    Abus(Payload);
    Actions(RepeatUntil|UnloadFIFO|Merge9);
                    CAMMATCH("1xxxx1111",SomeGroup,Label(START_OF_AN));
    Branch(CAMmatch,CAMMATCH("100001010",SomeGroup,Label(START_OF_FRAME)));

LabelDef(SINK_FRAME);
    Actions(UnloadFIFO+RepeatUntil);
    Branch(Data9,Label(WAIT_FOR_FRAME));

LabelDef(START_OF_FRAME);

    LOG_UPC;

    /* FIFO has start of a frame in it.  Check if we own extract space scope */
    CregsAddrWrite(RxStatus);
    Branch(ALWAYS,Upc+1);

    /* Second half of initialization code while waiting for Creg read */
    IregInit(FrameLengthLS,0xFF,UseBbus);
    Actions(CRCinit);
    Branch(ALWAYS,Upc+1);

    /* Read RXstatus */
    CregToIreg(TMP2);
    Branch(ALWAYS,Upc+1);

    /* Check OWN bit of RXstatus, normal case is SDP owns (bit 7 == 0) */
```

```
Branch(ALUneg,Label(WAIT_FOR_OWN));

/* We have incoming data and we own the extract space scope. Set
   the busy flag */
DCPSIM_MESSAGE(START PROCESSING NEW FRAME);
Abus(IregsA(TMP2));
Bbus(Literal(RxStatus_Busy));
ALU(OR);
Pbus(ALUout);
Actions(CregsWrite);
Branch(ALWAYS,Upc+1);

/* All set, Proceed to parse the frame. */

/*
 * Pass the token here. Let the next CP pick up the next packet.
 */
Abus(IregsA(STATUSreg));
Bbus(Literal(TOKEN));
ALU(OR);
Actions(IregsAwrite);
Branch(ALWAYS, Upc+1);

Abus(IregsA(STATUSreg));
Bbus(Literal(TOKEN));
ALU(XOR);
Actions(IregsAwrite);
Branch(ALWAYS, Upc+1);

/* Init header status to zero */
Bbus(Literal(EXTRACT_HDRSTATUS));
Abus(SameLiteral);
ALU(XOR);
Pbus(ALUout);
Actions(CregAddrWrite|CregsWrite);
Branch(ALWAYS, Upc+1);

/* Init frame status to zero */
Bbus(Literal(EXTRACT_FRAMESTATUS));
Abus(SameLiteral);
ALU(XOR);
Pbus(ALUout);
Actions(CregAddrWrite|CregsWrite);
Branch(ALWAYS, Upc+1);

/* Init frame status to zero */
Bbus(Literal(EXTRACT_BFC));
Abus(SameLiteral);
ALU(XOR);
```

```
    Pbus(ALUout);
    Actions(CregAddrWrite|CregsWrite);
    Branch(ALWAYS, Upc+1);

    /**************************************************************************/
    /*         Parse Frame                                                    */
    /**************************************************************************/

#include "enetParse.h"

LabelDef(PDU_LOOP_BREAK);

    /* Prep address in case there is an error to report */
    CregsAddrWrite(EXTRACT_FRAMESTATUS);
    Abus(SameLiteral);
    ALU(XOR);
    Pbus(ALUout);
    Actions(CregsWrite);
    Branch(ALWAYS,Upc+1);

LabelDef(FINAL_FRAME_STATUS_CHECK);
    /* EOP_next was asserted, the Frame is completed.  */
    /* Check RXBIT framing status */
    Abus(IregsA(LastByteRead));
    ALU(PassA);
    Pbus(ALUout);
    Branch(ALWAYS,Upc+1);

    /* Clear error flag */
    IregXOR(Flags,Flags);
    Branch(ALUzero,Label(FRAMING_CHECK_DONE));

    /* EOF marker non-zero, which means that the received frame
     * was determined by rxBit to be mis-aligned or too long or
     * too short. Write the status code embedded in the EOF marker
     * to Cregs */
LabelDef(SET_FRAMESTATUS_FRAMING_ERROR);
    DCPSIM_MESSAGE(PAUSE FRAME OR FRAMING ERROR);

    Bbus(IregsB(LastByteRead));
    ALU(PassB);
    IregsA(Flags);
    Pbus(ALUout);
    Actions(CregsWrite+IregsAwrite);
    Branch(ALWAYS,Upc+1);

    Branch(ALWAYS,Label(FINI));

LabelDef(FRAMING_CHECK_DONE);
```

```
    /* Framing OK, so check for FIFO overrun condition */
    Actions(InvertCondition);
    Branch(FIFOoverrun,Label(OVERRUN_CHECK_DONE));

    /* FIFO overrun. Set status and release from Overrun Mode */
LabelDef(SET_FRAMESTATUS_OVERRUN);
    DCPSIM_MESSAGE(LARGE RX FIFO OVERRUN ERROR);

    IregORlit(Flags,macFrameStatusFifoOverrun);
    Pbus(ALUout);
    Actions(CregsWrite+Merge9);
    Branch(ALWAYS,Upc+1);

    Branch(ALWAYS,Label(FINI));

LabelDef(OVERRUN_CHECK_DONE);
    Actions(InvertCondition);
    Branch(CRCerror,Label(CRC_CHECK_DONE));

    /* CRC Error - set flag in status reg */
LabelDef(CRC_ERROR);
    DCPSIM_MESSAGE(CRC ERROR);

    IregORlit(Flags, macFrameStatusCrcError);
    Pbus(ALUout);
    Actions(CregsWrite);
    Branch(ALWAYS,Upc+1);

LabelDef(CRC_CHECK_DONE);
    /* **************************************************************** */
LabelDef(FINI);

   LOG_UPC;

    /* At this point we have detected/reported all errors.
     * If FrameLength MS|LS is not in concatenated mode then we know that
     * we had an early end before the MS half was cleared.  Do it now before
     * writing it to extract space. */

    /* Check status reg */
    Abus(IregsA(STATUSreg));
    Bbus(Literal(CSR_Count01));
    ALU(AND);
    Branch(ALWAYS,Upc+1);

    Actions(InvertCondition);
    Branch(ALUzero,Label(COUNT_MS_READY));

    /* Frame length still in 8 bit mode - clear out MS byte */
    IregInit(FrameLengthMS,0,UseBbus);
    Branch(ALWAYS,Upc+1);

LabelDef(COUNT_MS_READY);
    CregsAddrWrite(EXTRACT_FRAMELENGTH);
    Abus(IregsA(FrameLengthMS));
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite);
    Branch(ALWAYS,Upc+1);

    /* Write LS half of length */
    Abus(IregsA(FrameLengthLS));
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite+CregAddrIncr);
    Branch(ALWAYS,Upc+1);

LabelDef(EXIT_SCOPE);

    LOG_UPC;

    /* Write final frame status to Creg space */
    CregsAddrWrite(EXTRACT_FRAMESTATUS);
    Abus(IregsA(Flags));
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite);
    Branch(ALWAYS,Upc+1);

    DCPSIM_MESSAGE(SWITCH SCOPE);

    /* Set Done (A.K.A. Own) flag, but leave L1 and Busy on too. */
    CregsAddrWrite(RxStatus);
    Branch(ALWAYS,Upc+1);

    Abus(Literal(RxStatus_L1done | RxStatus_Own));
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite);
    Branch(ALWAYS,Upc+1);

    Branch(ALWAYS, Label(INITIALIZE));

    /**************************************************************************/
    /*        I P H D R _ P A R S E _ E R R O R                               */
    /**************************************************************************/
LabelDef(IPHDR_PARSE_ERROR);
```

```
    /* Did data9 come on unexpectedly ? */
    Abus(IregsA(HeaderStatus));
    Bbus(Literal(IP_FRAMING_ERROR));
    ALU(SUB);
    Branch(ALWAYS,Upc+1);

    Actions(InvertCondition);
    Branch(ALUzero,Label(IP_HEADER_ERROR_NOT_FRAMING));

LabelDef(IP_HEADER_ERROR_FRAMING);
    Branch(ALWAYS,Label(RX_DATA_ERROR));

LabelDef(IP_HEADER_ERROR_NOT_FRAMING);

    /* Report the header error to extract space */
    DCPSIM_MESSAGE(IP HEADER PARSING ERROR REPORTED - CONTINUE WITH FRAME READ);
    CregsAddrWrite(EXTRACT_HDRSTATUS);
    Branch(ALWAYS,Upc+1);

    Abus(IregsA(HeaderStatus));
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite);
    Branch(ALWAYS,Upc+1);

    /* Continue to read the rest of the frame, but without enthusiasm. */
    Branch(ALWAYS,Label(STREAM_PDU));


    /**************************************************************************/
    /*        R X _ D A T A _ E R R O R                                      */
    /**************************************************************************/
LabelDef(RX_DATA_ERROR_POP);
    Abus(Payload);
    ALU(PassA);
    IregsA(LastByteRead);
    Actions(UnloadFIFO+IregsAwrite+IregIncr(FrameLength));
    Branch(ALWAYS, Upc+1);

LabelDef(RX_DATA_ERROR);
    /* We come here when Data9 came on during header parsing.
     *  Bits 7-0 of the EOF marker may be an error code. */
    DCPSIM_MESSAGE(ENTERING EARLY-END ERROR HANDLER);

    LOG_UPC;

    CregsAddrWrite(EXTRACT_HDRSTATUS);
    Branch(ALWAYS,Upc+1);
    Abus(Literal(macHdrStatusFrmParseError));
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite);
    Branch(ALWAYS,Upc+1);

    /* Write the determined rx path to extract */
    Abus(IregsA(PathState));
    Bbus(Literal(EXTRACT_RX_PATH));
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregAddrWrite|CregsWrite);
    Branch(ALWAYS, Upc+1);

    /* RXBIT may have reported a framing error of some sort - copy
     * RXBIT's error code into the frame status flag and merge
     * with normal completion path.  If no error from RXBIT then
     * synthesize a length error code into the Frame Status. */

    Bbus(IregsB(LastByteRead));
    ALU(PassB);
    Pbus(ALUout);
    Abus(IregsA(Flags));
    Actions(IregsAwrite);
    Branch(ALWAYS,Upc+1);

    Actions(InvertCondition);
    Branch(ALUzero,Label(FINI));

    /* RXBIT status was OK.  Make this error a RXBYTE unexpected EOF */

    IregInit(Flags,macFrameStatusLengthError,UseAbus);
    Pbus(ALUout);
    Branch(ALWAYS,Upc+1);

    Branch(ALWAYS,Label(FINI));

    /* **************************************************************** */
LabelDef(START_OF_AN);

    /* We just received a start of AN marker. Wait for a scope.
     *  Setting the Creg address reg here may not be necessary.
     */

    LOG_UPC;

    CregsAddrWrite(RxStatus);
    Branch(ALWAYS,Upc+1);
    IregInit(Flags,0,UseBbus);
    Branch(ALWAYS,Upc+1);

    Abus(Creg);
    ALU(PassA);
    Branch(ALWAYS,Upc+1);

    /* If no scope then just go back and wait for the next thing to happen */
    Branch(ALUneg,Label(WAIT_FOR_FRAME));

    /* Turn on the busy bit in RxStatus */
    Abus(Literal(RxStatus_Busy));
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite);
    Branch(ALWAYS,Upc+1);

    /* Point Creg address reg at 6 byte block where 3 contiguous pages go. */
    CregsAddrWrite(AN_PAGES_BASE);
    Branch(ALWAYS,Upc+1);
    Branch(ALWAYS,Label(READ_PAGE1_MS));

    // LOG_UPC;

LabelDef(READ_PAGE1_MS);
    Abus(Payload);
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite|UnloadFIFO);
    Branch(CAMmatch,CAMMATCH("0xxxxxxxx",SomeGroup,Label(READ_PAGE1_LS)));

    Branch(ALWAYS,Label(AN_WAIT_FOR_FRAME));

LabelDef(READ_PAGE1_LS);
    Abus(Payload);
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite|CregAddrIncr|UnloadFIFO);
    Branch(CAMmatch,CAMMATCH("0xxxxxxxx",SomeGroup,Label(READ_PAGE2)));

    Branch(ALWAYS,Label(AN_WAIT_FOR_FRAME));

LabelDef(READ_PAGE2);
    /* Discard K */
    Abus(Payload);
    Actions(UnloadFIFO);
//
Branch(CAMmatch,CAMMATCH("0xxxxxxxx",SomeGroup,Label(AN_WAIT_FOR_FRAME)));
    Branch(ALWAYS,Upc+1);
    /* Discard CB */
    Abus(Payload);
    Actions(UnloadFIFO);
//
Branch(CAMmatch,CAMMATCH("1xxxxxxxx",SomeGroup,Label(AN_WAIT_FOR_FRAME)));
    Branch(ALWAYS,Upc+1);

LabelDef(READ_PAGE2_MS);
    Abus(Payload);
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite|CregAddrIncr|UnloadFIFO);
    Branch(CAMmatch,CAMMATCH("0xxxxxxxx",SomeGroup,Label(READ_PAGE2_LS)));

    Branch(ALWAYS,Label(AN_WAIT_FOR_FRAME));

LabelDef(READ_PAGE2_LS);
    Abus(Payload);
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite|CregAddrIncr|UnloadFIFO);
    Branch(CAMmatch,CAMMATCH("0xxxxxxxx",SomeGroup,Label(READ_PAGE3)));

    Branch(ALWAYS,Label(AN_WAIT_FOR_FRAME));

LabelDef(READ_PAGE3);
    /* Discard K */
    Abus(Payload);
    Actions(UnloadFIFO);
//
Branch(CAMmatch,CAMMATCH("0xxxxxxxx",SomeGroup,Label(AN_WAIT_FOR_FRAME)));
    Branch(ALWAYS,Upc+1);

    /* Discard CB */
    Abus(Payload);
    Actions(UnloadFIFO);
//
Branch(CAMmatch,CAMMATCH("1xxxxxxxx",SomeGroup,Label(AN_WAIT_FOR_FRAME)));
    Branch(ALWAYS,Upc+1);

LabelDef(READ_PAGE3_MS);
    Abus(Payload);
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite|CregAddrIncr|UnloadFIFO);
    Branch(CAMmatch,CAMMATCH("0xxxxxxxx",SomeGroup,Label(READ_PAGE3_LS)));

    Branch(ALWAYS,Label(AN_WAIT_FOR_FRAME));
```

```
LabelDef(READ_PAGE3_LS);
    Abus(Payload);
    ALU(PassA);
    Pbus(ALUout);
    Actions(CregsWrite|CregAddrIncr|UnloadFIFO);
    Branch(CAMmatch,CAMMATCH("0xxxxxxxx",SomeGroup,Label(READ_3_PAGES_DONE)));

    Branch(ALWAYS,Label(AN_WAIT_FOR_FRAME));

LabelDef(READ_3_PAGES_DONE);

    /* Pop end of AN marker */
    Actions(UnloadFIFO);
    Branch(ALWAYS,Upc+1);

    /* We have written 3 contiguous pages into extract space.  Set frame status
     * to "AN Pages ready" and give up the scope to the RC.
     */
    DCPSIM_MESSAGE(THREE CONTIGUOUS AN PAGES RECEIVED);

    LOG_UPC;

    IregORlit(Flags, macFrameStatusAutoNegotiation);
    Branch(ALWAYS,Upc+1);

    CregsAddrWrite(EXTRACT_RX_PATH);
    Branch(ALWAYS,Upc+1);

    CregInit(0);
    Branch(ALWAYS,Upc+1);

    LOG_UPC;

    /* We may have entered overrun mode during AN, so unconditionally clear it
*/
    Actions(Merge9);
    Branch(ALWAYS,Label(EXIT_SCOPE));

LabelDef(AN_WAIT_FOR_FRAME);
/* we come here when we got at least the start of one AN page but failed
 * to receive three contiguous well-formed ones. Give it a bad status and
 * return scope.
 */
    DCPSIM_MESSAGE(FAILURE TO PARSE THREE CONTIGUOUS AN PAGES);

    IregORlit(Flags, macFrameStatusAutoNegotiationBad);
    Branch(ALWAYS,Upc+1);

    LOG_UPC;
    /* We may have entered overrun mode during AN, so unconditionally clear it
*/
    Actions(Merge9);
    Branch(ALWAYS,Label(EXIT_SCOPE));



    /************************************************************************/
    /*        W A I T _ F O R _ O W N                                       */
    /************************************************************************/
LabelDef(WAIT_FOR_OWN);

    /* Coming here is a very bad thing - the RC has apparently failed to
     * keep pace with the SDP, or it has hung or otherwise abdicated.  We
     * count how many frames we miss while waiting for ownership, and when
     * we finally get it we report an error along with the Bad Frame Count
     * (BFC).
     *
     * Note that it is important while this is going on to unload the
     * ingress FIFO adequately so that we don't spill, and therefore miss,
     * any end of frame markers.
     *
     * It may be that we should generate an 'event' here as well to attract
     * the attention of the RC. TBD.
     */

    DCPSIM_MESSAGE(WAIT FOR OWN);
    Branch(ALWAYS,Upc+1);

    LOG_UPC;

LabelDef(SPILL_ANOTHER_FRAME);
    CregsAddrWrite(EXTRACT_BFC+1);
    Branch(ALWAYS, Upc+1);

    /* Spill the current frame */
LabelDef(SPILLING_FRAME);
    PayloadIn(FIFOout);
    Pbus(ALUout);
    Actions(UnloadFIFO|CregsWrite); /* Loop until a valid end of frame marker,
but not a start of frame marker */
    CAMMATCH("1xxxx1111",SomeGroup,Label(SPILLING_FRAME));
    CAMMATCH("100001010",SomeGroup,Label(SPILLING_FRAME));
    Branch(CAMmatch,CAMMATCH("0XXXXXXXX",SomeGroup,Label(SPILLING_FRAME)));

    /* Check if we own extract space scope */
    CregsAddrWrite(RxStatus);
    Branch(ALWAYS,Upc+1);
```

```
        Branch(ALWAYS,Upc+1);

        /* Read RXstatus */
        CregToIreg(TMP2);
        Branch(ALWAYS,Upc+1);

        /* Check OWN bit of RXstatus, normal case is SDP owns (bit 7 == 0) */
        Branch(ALUneg,Label(SPILL_ANOTHER_FRAME));


    LabelDef(DROPPED_FRAME_REPORT);
        DCPSIM_MESSAGE(DONE SPILLING FRAMES);
        /* We have no way of testing for a new frame, except for trying to pop
         * the FIFO and stalling in the absence of one, in which case we won't
         * be able to test for scope ownership until a new frame is upon us.
         * Therefore we are consigned to spill every frame until we sink a
         * frame that ends after scope ownership has been handed to us.
         */

        /* We have the scope now, report error to RC */
        LOG_UPC;

        /* For now, set the BFC to non zero to signal a drop event */
        CregsAddrWrite(EXTRACT_BFC);
        Branch(ALWAYS, Upc+1);

        Abus(Literal(0xFF));
        ALU(PassA);
        Pbus(ALUout);
        Actions(CregsWrite);
        Branch(ALWAYS, Upc+1);

        /* Clear dropped frames, since it is aliased over FrameLength */
        IregsA(DroppedFramesMS);
        IregsB(DroppedFramesLS);
        Bbus(Literal(0));
        ALU(PassB);
        Actions(IregsAwrite|IregsBwrite);
        Branch(ALWAYS,Upc+1);

        IregORlit(Flags, macFrameStatusFramesDropped);
        Branch(ALWAYS,Upc+1);

        /* Write final frame status to Creg space */
        CregsAddrWrite(EXTRACT_FRAMESTATUS);
        Abus(IregsA(Flags));
        ALU(PassA);
        Pbus(ALUout);
        Actions(CregsWrite);
        Branch(ALWAYS,Upc+1);

        /* Write final frame status to Creg space */
        CregsAddrWrite(EXTRACT_HDRSTATUS);
        Abus(IregsA(Flags));
        ALU(PassA);
        Pbus(ALUout);
        Actions(CregsWrite);
        Branch(ALWAYS,Upc+1);

        /* Clear the frame length field in Extract space */
        CregsAddrWrite(EXTRACT_FRAMELENGTH);
        Abus(SameLiteral);
        ALU(XOR);
        Pbus(ALUout);
        Actions(CregsWrite);
        Branch(ALWAYS,Upc+1);

        Abus(Literal(0));
        ALU(PassA);
        Pbus(ALUout);
        Actions(CregsWrite+CregAddrIncr);
        Branch(ALWAYS,Upc+1);

        LOG_UPC;

        /* Write the determined rx path to extract */
        CregsAddrWrite(EXTRACT_RX_PATH);
        Branch(ALWAYS, Upc+1);

        Abus(Literal(macFrameStatusAutoNegotiationBad));
        ALU(PassA);
        Pbus(ALUout);
        Actions(CregsWrite);
        Branch(ALWAYS, Upc+1);

        Branch(ALWAYS, Label(WAIT_FOR_FRAME));
    }
```

Lesson 9

MODIFYING THE GIGABIT ETHERNET SWITCH APPLICATION

Overview

This lesson shows how to implement a small change in packet-handling functionality in the CPRC code and SDP microcode of the C-Ware Software Toolset's (CST) Gigabit Ethernet Switch (**gbeSwitch**) application.

You will learn how to examine the application as supplied in the CST and how to debug changes to the application's code.

Application Overview

In the CST Version 2.2 release, the **gbeSwitch** application runs on C-5 NP Version D0 hardware and under simulation and on C-5e Version A1 under simulation.

The **gbeSwitch** application provides the following high-level functionality:

- Runs on CDS with a C-5 Switch Module (C-5 NP Version D0 only) and 2 x TBI PIM
- TBI Gigabit Ethernet PHY support
- 1000BaseT autonegotiation
- IEEE 802.3 Ethernet MAC support
- Enable/Disable port state support
- RMON EtherStats support
- Layer 2 MAC Bridging support:
  - Bridge forwarding
  - Bridge flooding
  - Bridge address learning
  - Bridge address aging
- IEEE 802.1p and 802.1Q VLAN support

- IPv4 Unicast Routing
- IPv4 Multicast Routing
- ICMP Support (TTL exceeded, no route, redirect)

The application uses the C-5 NP's four Channel Processor clusters to provide two Gbe ports, as follows:

• CP cluster 0:	Gbe Rx - port 1
• CP cluster 1:	Gbe Tx - port 1
• CP cluster 2:	Gbe Rx - port 2
• CP cluster 3:	Gbe Tx - port 2

The **gbeSwitch** application package also provides a host-enabled configuration of the software as well as an offline table-building configuration. The application's README file describes how to build the application for either of these configurations. Find the README file in this path:

apps\gbeSwitch\doc\Readme

For further information consult the *Gigabit Ethernet Switch Application Guide*. This document in included in the documentation set for the C-Ware Software Toolset. Look for this file in your CST installation:

Documentation\Modules\Applications_and_Components\gbeSwitch.pdf

Specific Functional Changes

The new functionality to be implemented in the application is to take a two-byte field from the payload section of an Ethernet frame and to prepend this field to the egress frame while removing the original field from the payload section. You will implement this functional change so that it is performed only on L2 bridged frames that are not VLAN tagged.

To summarize, you will change the application's functionality as follows:

- Add a two-byte field in the CPRC's Extract Space.
- Include the two-byte field in the enqueued descriptor.
- Insert the two-byte field in the CPRC's Merge Space.
- Manipulate the egress stream.

The format of the ingress frames is as follows:

MAC DA	MAC SA	Type	Two-Byte Field	Payload

The format of the egress frames is as follows:

Two-Byte Field	MAC DA	MAC SA	Type	Payload

Building the Application

Build the **gbeSwitch** application for debuggable execution under the C-Ware Simulation Environment for the C-5 NP Version D0 chip, as follows:

1. Open a new command shell window.
2. Move to the CST's **apps\gbeSwitch\run\** directory.
3. Run the **sv** command script for your platform. For instance, on Windows NT enter this command:

   ```
   ..\..\..\bin\sv
   ```

4. Build the **gbeSwitch** application as debuggable and runnable under the C-Ware Simulator for the C-5 NP Version D0 chip, as follows:

   ```
   ..\..\..\bin\configure\c5-d0-sim-debug.bat
   make
   ```

5. Build the application's simulated input traffic pattern files, as follows:

   ```
   make patterns
   ```

Setting Up the Simulation Environment

You first want to observe in action under the C-Ware Simulator the unmodified **gbeSwitch** application as it processes input traffic.

In the **gbeSwitch** application the XP program builds a small MAC bridge table. This table can be found in this file:

apps\gbeSwitch\chip\np\xprc\src\bridgeTableXp.c

The array *bridgeTableInit[]* holds the routes.

A MAC DA of 0x22 received at the Gbe port on the C-5 NP's cluster 0 will be bridged to the Gbe port on the C-5 NP's cluster 3. The command line of CST-supplied pattern generation Perl scripts to generate this traffic is:

```
perl ...\bin\pgGenData.pl -numReps 2 -size 46 -byteCount |
perl ...\bin\pgEnet.pl -macDA 0x000000000022 -macSA 0x000000000021 -type 0x0123 |
perl ...\bin\pgFmtBusWidth.pl -lGbEther |
perl ...\bin\pgFmtDcpSim.pl > outFile.pat
```

This command line produces the following frames:

```
MAC da:000000000022 MAC sa:000000000021 type:0123 crc:809b1905
Payload Numeric=[000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122
232425262728292a2b2c2d]
Time Elapsed: 568  ns

MAC da:000000000022 MAC sa:000000000021 type:0123 crc:809b1905
Payload Numeric=[000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122
232425262728292a2b2c2d]
Time Elapsed: 568  ns
```

You must update the Simulator configuration file (**config**) to change the input pattern file for Cp0 to the desired file (**outFile.pat**) and also to change the input file for Cp8 to a null file. The following two lines must be changed inside the file **apps\gbeSwitch\run\config**:

```
...
sdp0_RxPatternFile $(CPORT)/apps/gbeSwitch/run/inPatterns/outFile.pat
...
sdp8_RxPatternFile $(CPORT)/apps/gbeSwitch/run/inPatterns/null.pat
...
```

If the file **null.pat** is not already in the **apps\gbeSwitch\ run\inPatterns\** directory, create it by creating a new text file and renaming it to **null.pat**.

Running the Unmodified Application Under Simulation

To determine the flow of these packets through the C-5 NP, use the C-Ware Simulator to step through the program and view the application's debug-related microcode-generated messages (known as *microcode messages*). The **gbeSwitch** application was instrumented to produced these messages by the application's author.

You will make the first modifications to the application's transmit-side microcode, so producing a trace of this code's microcode messages before any modifications are made is a useful step.

The first CP to transmit from the aggregated cluster is Cp12. So enable the output of microcode messages from the Simulator's TxByte12 object (that is, notice the Simulator command 'umsg txbyte12 1' in the following listing).

```
C:\C-Port\CST2.2\apps\gbeSwitch\run> c5sim
c5sim CST 2.2  C5 [Rev D0] release build on Apr 16 2002 20:01:45
Dcp> cd TLU
Dcp.Tlu> restore ./tlu-c5.state
Dcp.Tlu> symbols sdp0 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster0.Cp0.Sdp> symbols sdp1 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster0.Cp1.Sdp> symbols sdp2 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster0.Cp2.Sdp> symbols sdp3 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster0.Cp3.Sdp> symbols sdp4 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster1.Cp0.Sdp> symbols sdp5 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster1.Cp1.Sdp> symbols sdp6 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster1.Cp2.Sdp> symbols sdp7 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster1.Cp3.Sdp> symbols sdp8 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster2.Cp0.Sdp> symbols sdp9 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster2.Cp1.Sdp> symbols sdp10 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster2.Cp2.Sdp> symbols sdp11 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster2.Cp3.Sdp> symbols sdp12 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster3.Cp0.Sdp> symbols sdp13 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster3.Cp1.Sdp> symbols sdp14 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster3.Cp2.Sdp> symbols sdp15 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster3.Cp3.Sdp> umsg txbyte12 1
Dcp.CpCluster3.Cp0.Sdp.TxByte> g
[172398 0 TxByte12]  Microcode Message: START TRANSMITTING NEW FRAME
[172407 0 TxByte12]  Microcode Message: NEW FRAME IS NON-IP
[172435 0 TxByte12]  Microcode Message: CHECK FOR VLAN TAGGING
[172443 0 TxByte12]  Microcode Message: PASSING IP/NON-IP MERGE POINT
[172549 0 TxByte12]  Microcode Message: SETTING TX BYTE SIGNAL
[172551 0 TxByte12]  Microcode Message: I OWN THE TOKEN
[172554 0 TxByte12]  Microcode Message: STREAMING DONE
[172555 0 TxByte12]  Microcode Message: BAD FRAME SIDE-CAR BIT WAS NOT SET
[172560 0 TxByte12]  Microcode Message: PADDING TO 60 BYTES DONE
[172562 0 TxByte12]  Microcode Message: TRANSMITTING CRC
[172575 0 TxByte12]  Microcode Message: SENDING END-OF-FRAME MARKER TO TXBIT
[172695 0 TxByte12]  Microcode Message: TXBIT SIGNAL RECEIVED - TRANSMISSION COMPLETE
[172696 0 TxByte12]  Microcode Message: TXBIT AOK SIGNAL RX - DE-ASSERTING BYTEtoBIT
[172700 0 TxByte12]  Microcode Message: SWITCH SCOPE
Halt: Dcp: Control C detected
Dcp.CpCluster3.Cp0.Sdp.TxByte> q
```

Notice in this listing that the user pressed the Ctrl-c key after viewing the microcode message "SWITCH SCOPE", then exited the Simulator.

Use the CST's **printTrace.pl** tool to filter (that is, into human-readable format) the frame that was output from the C-5 NP's Cp12 by the unmodified application:

```
C:\C-Port\CST2.2\apps\gbeSwitch\run\outPatterns\c5>
   perl ...\bin\printtrace.pl gbetxbase12.pat

MAC da:000000000022 MAC sa:000000000021 type:0123 crc:809b1905
Payload Numeric=[000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122
232425262728292a2b2c2d]
Time Elapsed: 568  ns
```

Identifying a Discrepancy in Egress Data

This single frame is equivalent to the two input frames produced by the command line of Perl scripts shown in "Setting Up the Simulation Environment" on page 153. However, we must account for the fact that feeding the application two input frames resulted in only one output frame from the C-5 NP. Two obvious possibilities are: the input pattern file is incorrectly specified, or there is an error in the **gbeSwitch** application.

In fact the problem is related to how the C-Ware Simulation Environment emulates the mechanics of the Gigabit Ethernet Ten Bit Interface (TBI). When no active data is being transmitted on the TBI, the interface is actually still required to transmit alternating idle characters. After initialization the **gbeSwitch** application's RxBit microcode synchronizes itself to the TBI stream by looking for idle characters before looking for the start of frame character. If the application was running on actual C-5 NP hardware on a fully populated Switch Module board (or similar), the PHY would be transmitting these characters during initialization, which would allow the pertinent C-5 NP channel's RxBit processor to synchronize to the stream. In this simulation environment the input stream is supplied directly from the input pattern file **outFile.pat**, which begins with a start of frame character rather than idle characters.

Correcting the Discrepancy

The application's simulated input stream can be modified to start with idle characters by adding some null, non-Ethernet, encapsulated data prior to the Ethernet encapsulated data, before formatting for the TBI stream. You must run a pattern generation Perl script command line that changes this null data into idle characters. (This can be accomplished by temporarily storing the outputs from one Perl script command line in an intermediate file, then use that file as input to a second Perl script command line's final formatting script.) In fact the **gbeSwitch** application's pattern makefile in the **run\inPatterns\** directory uses this technique to build up different types of Ethernet frames as an input file.

For example, you can use the Windows command shell to perform the task.

1 Generate some null data and placed it in a file **tmp1**:

```
perl \c-port\cst2.2\bin\pgGenData.pl -numReps 1 -size 2 -hexData 0 > tmp1
```

2 Generate the Ethernet frames as already demonstrated and append them to **tmp1**:

```
perl \c-port\cst2.2\bin\pgGenData.pl -numReps 2 -size 46 -byteCount |
perl \c-port\cst2.2\bin\pgEnet.pl -macDA 0x000000000022
-macSA 0x000000000021 -type 0x0123 >> tmp1
```

3 Use **tmp1** as the input to the final formatting scripts:

```
cat tmp1 |  perl \c-port\cst2.2\bin\pgFmtBusWidth.pl -1GbEther |
perl \c-port\cst2.2\bin\pgFmtDcpSim.pl > outFile.pat
```

Examining the first generated **outFile.pat** file shows that its first line is a start-of-frame character:

```
//  S
8.000000 507 0 0 0
```

After using the new method to produce the input pattern file, the second generated **outFile.pat** file contains two idle characters prior to the start-of-frame character:

```
//  IDLE A
8.000000 444 0 0 0
//  IDLE B
8.000000 197 0 0 0
//  S
8.000000 507 0 0 0
```

Re-running the simulation with the second **outFile.pat** file as input causes the application to produce two output frames that are identical to the two input frames.

Thus, you have observed the sequence of transmit-side microcode messages that occur when passing one frame through the unmodified **gbeSwitch** application from the C-5 NP's Cp0 to Cp12.

Modifying the Transmit Microcode

Begin modifying the application by changing the transmit-side microcode, so that it removes two bytes of payload and inserts two fixed bytes at the head of the egress stream.

The application's microcode source for the TxByte serial processor resides in this file:

apps\gbeSwitch\chip\np\sdp\src\txByte.c

Inspecting the series of microcode messages produced by the unmodified application allows you to determine the locations where additions should be made to the transmit-side microcode, as follows:

- The first byte of transmission of the Ethernet frame occurs just after the label ALG_RAW. (This is the point where the microcode message "NEW FRAME IS NON-IP" is output.) Figure 12 on page 158 (Listing 1) presents the source code listing from **txByte.c**, starting from the ALG_RAW label.
- By following the flow of microcode from where the microcode message "CHECK FOR VLAN TAGGING" is output, notice that the payload bytes must be removed after the microcode label INSERT_VLAN_MERGE_RAW. The code must be changed here to prevent IP frames from being modified. Figure 13 on page 159 (Listing 2) presents the source code listing from **txByte.c**, starting from the INSERT_VLAN_MERGE_RAW label.

Figure 12 Listing 1 From txByte.c in Gigabit Ethernet Switch Application

```
[Lines removed]
...
/***************************************************************/
/* ALG_RAW                                                     */
/***************************************************************/

LabelDef(ALG_RAW);
    DCPSIM_MESSAGE(NEW FRAME IS NON-IP);
    Branch(ALWAYS,Upc+1);


   /* This wait for data is necessary in half-duplex, otherwise CRC
    * errors occur.
    */
   WAIT_FOR_DATA(Upc+1);

    /* Send the MAC DA and SA  */

/* Send first byte twice for FIFO bug workaround */
    Pbus(Payload);
    Actions(DataOutValid);
    Branch(ALWAYS,Upc+1);

    XMITBYTE;
    XMITBYTE;
    XMITBYTE;
    XMITBYTE;
    XMITBYTE;
    XMITBYTE;
    XMITBYTE;
    XMITBYTE;
    XMITBYTE;
    XMITBYTE;
    XMITBYTE;
    XMITBYTE;

    /* Check merge space for the vlan tag. If it is 0xffff, that indicates to
     * NOT tag this frame. Otherwise, tag it with the appropriate vlantag
     * and send out */
    DCPSIM_MESSAGE(CHECK FOR VLAN TAGGING);
    CregsAddrWrite(TX_VLAN_TAG);
    Branch(ALWAYS,Upc+1);

    Actions(CregAddrIncr);
    Branch(ALWAYS,Upc+1);
...
[Lines removed]
```

Figure 13 Listing 2 From txByte.c in Gigabit Ethernet Switch Application

```
[Lines removed]
...
LabelDef(INSERT_VLAN_MERGE_RAW);

    /* Now examine MS type byte to see if a delay should be simulated */

/*    WAIT_FOR_DATA(Upc+1);*/

/*    Branch(CAMmatch,CAMMATCH("011111110",SomeGroup,Label(SIM_UNDERRUN)));*/


    Branch(ALWAYS,Label(ALG_IP_NONE));

    /***************************************************************/
    /* A L G _ I P _ R O U T E D                                   */
    /***************************************************************/

LabelDef(ALG_IP_ROUTED);

    DCPSIM_MESSAGE(NEW FRAME IS IP);
    /* Strip off bytes specified by RC */
    CregsAddrWrite(TXCONTROL_DHS);
    Branch(ALWAYS,Upc+1);
...
[Lines removed]
```

To depict your modifications to this source code, we use line numbers to refer to the the code as found in the original source file.

To insert bytes into the stream, we modify the XMITBYTE macro so that it adds literals into the stream:

```
143   #   define XMITBYTE \
144        PayloadIn(FIFOout);\
145        Abus(Payload);\
146        ALU(PassA);\
147        PayloadOut(ALUout);\
148 Actions(CRCaccum+IregIncr(TXbyteCount)+UnloadFIFO+IregIncr(CNTR));\
149        Branch(ALWAYS,Upc+1);
```

This macro takes the input byte from memory and passes it to the TxLargeFifo block. It accumulates the byte in the CRC engine and increments two counter registers. To modify the code to pass literal values we can use the following code:

```
    Abus(Literal(0xde));
    ALU(PassA);
    PayloadOut(ALUout);
    Actions(CRCaccum+IregIncr(TXbyteCount)+IregIncr(CNTR));
    Branch(ALWAYS,Upc+1);
```

The microcode contains a workaround for the downstream transmit that requires the first byte to be output twice. This leads to the following changes (shown in boldface) to add the two literals at the head of the egress stream:

```
    LabelDef(ALG_RAW);
    DCPSIM_MESSAGE(NEW FRAME IS NON-IP);
    Branch(ALWAYS,Upc+1);


  /* This wait for data is necessary in half-duplex, otherwise CRC
   * errors occur.
   */
   WAIT_FOR_DATA(Upc+1);

   /* Send the MAC DA and SA  */

  /* Send first byte twice for FIFO bug workaround */
     //NEW - Added code to insert 2 bytes of literals
  //    Pbus(Payload);
  //    Actions(DataOutValid);
  //    Branch(ALWAYS,Upc+1);
    DCPSIM_MESSAGE(NEW - Adding 2 bytes) ;
    // Still send first byte twice
    Abus(Literal(0xde));
    ALU(PassA);
    PayloadOut(ALUout);
    Actions(DataOutValid);
    Branch(ALWAYS,Upc+1);
   Abus(Literal(0xde));
   ALU(PassA);
   PayloadOut(ALUout);
   Actions(CRCaccum+IregIncr(TXbyteCount)+IregIncr(CNTR));
   Branch(ALWAYS,Upc+1);
    Abus(Literal(0xad));
    ALU(PassA);
    PayloadOut(ALUout);
    Actions(CRCaccum+IregIncr(TXbyteCount)+IregIncr(CNTR));
    Branch(ALWAYS,Upc+1);
  XMITBYTE;
```

Removing the two bytes of payload requires that we unload the input FIFO but not pass on the data to the output.

Now the code also must transmit the Ethernet type field before removing the payload bytes. The original code at this label would branch unconditionally to ALG_IP_NONE, where the rest of the frame would be streamed to the output. Examining the microcode shows that if you remove the bytes at this point, the Ethernet type field would be removed, not the payload. Also VLAN-tagged flows would be modified if the code flow remained the same.

Thus, the code must be modified to redirect VLAN-tagged frames to ALG_IP_NONE and for non-VLAN-tagged frames modified to transmit two bytes before removing the two payload bytes. The code to do this follows:

```
/* Creg still pointing at low byte of VID */
    Abus(Creg);
    ALU(PassA);
    PayloadOut(ALUout);
    Actions(CRCaccum|IregIncr(TXbyteCount));
   Branch(ALWAYS,Upc+1);

     // NEW - Modified to remove two bytes here
   Branch(ALWAYS,Label(ALG_IP_NONE));
LabelDef(INSERT_VLAN_MERGE_RAW);

    /* Now examine MS type byte to see if a delay should be simulated */

/*    WAIT_FOR_DATA(Upc+1);*/

/* Branch(CAMmatch,CAMMATCH("011111110",SomeGroup,Label(SIM_UNDERRUN)));*/

   DCPSIM_MESSAGE(NEW - Removing two bytes from payload) ;
   // Still send the Type field
   XMITBYTE;
   XMITBYTE;
   // Remove here
   PayloadIn(FIFOout);
   Abus(Payload);
   ALU(PassA);
   Pbus(ALUout);
   Actions(UnloadFIFO);
   Branch(ALWAYS,Upc+1);
   PayloadIn(FIFOout);
   Abus(Payload);
```

```
        ALU(PassA);
        Pbus(ALUout);
        Actions(UnloadFIFO);
     Branch(ALWAYS,Label(ALG_IP_NONE));
```

Running this through the Simulator gives the following output:

```
C:\C-Port\CST2.2\apps\gbeSwitch\run\outPatterns\c5>
   perl ...\bin\printTrace.pl gbetxbase12.pat
```

```
MAC da:dead00000000 MAC sa:002200000000 type:0021 crc:4bd2571c
Payload Numeric=[0123020304050607080900a0b0c0d0e0f101112131415161718191a1b1c1d1e1f202122
232425262728292a2b2c2d]
Time Elapsed: 568  ns
```

Notice that the **printTrace.pl** tool still parses the frame as Ethernet, but also notice that:

- The 2Byte field has been inserted at the head of the stream.
- The MAC addresses have shifted by two bytes.
- The MAC SA continues into the type field.
- The type field shifts into the payload.
- The first two bytes of the ingress payload have been removed.

Further modifications to the transmit-side microcode require that an area in this Channel Processor's (CP) Merge Space is found for the 2Byte field to be inserted. The structure of Merge Space is defined in the overlying RISC core code. You must examine this structure to determine whether it can be used as is, or whether you must modify the structure. The structure definition for merge space is contained in this file:

apps\gbeSwitch\tgt\c5\inc\rcSdpApiIf.h

```
typedef struct {
    int16u pauseTime;
    int8u  txAlgorithm;
    int8u  datalinkHdrSize;
    int32u macDaHi;
    int16u vlanTag;
    int16u macDaLo;
    int32u macSaHi;
    int16u unused2;
    int16u macSaLo;
} MacMerge;
```

You could add an entirely new field in this structure to accommodate the 2Bytes to be added, but for ease of modification you can instead use the existing 16bit member named *unused2*.

To modify the microcode to access this area of Merge Space, you must first define its offset. As the microcode is actually translated by a C compiler, you can use a C cast to define the merge value. All other values for Merge Space are already defined by casts, so a simple text cut and paste in **txByte.c** is all that is required.

```
#define TX_MAC_SA_LO        TXMERGE + (int) &((MacMerge*)NULL)->macSaLo

   //NEW - Added define for unused2
   #define TX_UNUSED2        TXMERGE + (int) &((MacMerge*)NULL)->unused2

    /*******************************************************************/
```

The transmit-side microcode now must set up the *CregAddr* to point to this area and to source the data from here rather than use a literal value. When reading from Creg space, the address must be written two clocks before the data is read. The modified code now becomes:

```
    LabelDef(ALG_RAW);
    DCPSIM_MESSAGE(NEW FRAME IS NON-IP);
    Branch(ALWAYS,Upc+1);


   /* This wait for data is necessary in half-duplex, otherwise CRC
   * errors occur.
   */
   WAIT_FOR_DATA(Upc+1);

   /* Send the MAC DA and SA  */

  /* Send first byte twice for FIFO bug workaround */
     //ASM Added code to insert 2 bytes of merge fields
  //    Pbus(Payload);
  //    Actions(DataOutValid);
  //    Branch(ALWAYS,Upc+1);
     DCPSIM_MESSAGE(NEW - Adding 2 bytes) ;
    CregsAddrWrite(TX_UNUSED2);
     Branch(ALWAYS,Upc+1);
     // Mandatory wait
     Branch(ALWAYS,Upc+1);
     // Still send first byte twice
     Abus(Creg);
```

```
         ALU(PassA);
         PayloadOut(ALUout);
         Actions(DataOutValid);
         Branch(ALWAYS,Upc+1);
         // Increment Creg address 2 clocks before we need data
         Abus(Creg);
         ALU(PassA);
         PayloadOut(ALUout);

Actions(CRCaccum+IregIncr(TXbyteCount)+IregIncr(CNTR)+CregAddrIncr);
         Branch(ALWAYS,Upc+1);
         Branch(ALWAYS,Upc+1);
         // Send out second byte
         Abus(Creg);
         ALU(PassA);
         PayloadOut(ALUout);
         Actions(CRCaccum+IregIncr(TXbyteCount)+IregIncr(CNTR));
         Branch(ALWAYS,Upc+1);
584   XMITBYTE;
```

This concludes the changes to the application's transmit-side microcode. The microcode shown above can be tested using the CST's Debugger (gdb) on Cp12, as follows:

1 Enter a value into *merge->unused2* prior to the point at which the CP starts transmission.

2 Run the **printTrace.pl** tool on the output data to identify that the value is prepended to the stream.

Modifying the Receive Microcode

Changes to the receive-side microcode require a similar analysis of the flow of an Ethernet frame. In this case enable microcode messaging for Cp0's RxByte0 serial processor.

```
C:\C-Port\CST2.2\apps\gbeSwitch\run> c5sim
c5sim CST 2.2  C5 [Rev D0] release build on Apr 16 2002 20:01:45
Dcp> cd TLU
Dcp.Tlu> restore ./tlu-c5.state
Dcp.Tlu> symbols sdp0 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster0.Cp0.Sdp> symbols sdp1 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster0.Cp1.Sdp> symbols sdp2 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster0.Cp2.Sdp> symbols sdp3 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster0.Cp3.Sdp> symbols sdp4 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster1.Cp0.Sdp> symbols sdp5 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster1.Cp1.Sdp> symbols sdp6 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster1.Cp2.Sdp> symbols sdp7 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster1.Cp3.Sdp> symbols sdp8 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster2.Cp0.Sdp> symbols sdp9 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster2.Cp1.Sdp> symbols sdp10 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster2.Cp2.Sdp> symbols sdp11 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster2.Cp3.Sdp> symbols sdp12 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster3.Cp0.Sdp> symbols sdp13 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster3.Cp1.Sdp> symbols sdp14 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster3.Cp2.Sdp> symbols sdp15 bin/c5-d0-sim-debug/gbe.sdp
Dcp.CpCluster3.Cp3.Sdp> umsg rxbyte0 1
Dcp.CpCluster0.Cp0.Sdp.RxByte> g
[171333 0 RxByte0]  Microcode Message: START PROCESSING NEW FRAME
[171369 0 RxByte0]  Microcode Message: RECEIVED UNICAST MAC DA
[171377 0 RxByte0]  Microcode Message: DETERMINING FID TO USE
[171380 0 RxByte0]  Microcode Message: NOT 8021Q TAGGED
[171397 0 RxByte0]  Microcode Message: MAC DA LOOKUP
[171415 0 RxByte0]  Microcode Message: PARSE MAC TYPE
[171420 0 RxByte0]  Microcode Message: WRITE MAC TYPE TO DRAM
[171427 0 RxByte0]  Microcode Message: MAC TYPE IS NEITHER 08XX NOR 88XX
[171428 0 RxByte0]  Microcode Message: STREAM PDU
[171508 0 RxByte0]  Microcode Message: SWITCH SCOPE
Halt: Dcp: Control C detected
Dcp.CpCluster0.Cp0.Sdp.RxByte>
Dcp.CpCluster0.Cp0.Sdp.RxByte> q
```

As the payload for an Ethernet frame starts after the type field, you should examine the microcode after the line of code that produces the "MAC TYPE IS NEITHER 08XX NOR 88XX" message.

Again in this case, an area in the CP's Extract Space must be identified to hold the 2Byte code. The structure used is defined in the file:

apps\components\phy\inc\macHeader.h

This is the structure definition:

```
typedef struct {
    int8u  hdrStatus;
    int8u  frameStatus;
    int8u  rxPath;
    int8u  protocol;
    int16u macType;
    int16u frameLen;
    int32u unused2;
    int8u  priority;
    int8u  badFrameCount;
    int16u vlanId;
    int32u macDaHi;
    int16u macDaLo;
    int16u unused4;
    int32u macSaHi;
    int16u macSaLo;
    int16u unused5;
} MacHeader;
```

As in the case for the transmit-side microcode, you could add an entirely new field here to accommodate the two bytes to be added, but for ease of modification you can use the existing member *unused2*.

The RxByte microcode consists of several files. The code that we are looking for exists in this file:

apps\gbeSwitch\chip\np\sdp\inc\enetParse.h

This file is the target of a #include statement found in this file:

apps\gbeSwitch\chip\np\sdp\src\rxByte.c

rxByte.c uses definitions for Extract Space that are found in this file:

apps\components\phy\chip\np\inc\ethernetDefinitions.h

Modify **ethernetDefinitions.h** to add the value used when accessing *unused2*.

```
#define EXTRACT_VID         ((int)&((MacHeader*)NULL)->vlanId)

   //NEW - Added definition for unused2
   #define EXTRACT_UNUSED2      ((int)&((MacHeader*)NULL)->unused2)

#define EXTRACT_MAC_DA      ((int)&((macHeader*)NULL)->macDaHi)
```

This microcode source file provides examples of how to write fields to Extract Space:

1. The Creg address needs to be set up to point to *unused2*.
2. From previous microcode, copy the code that moves a byte to DMEM and modify it to add a write to Creg.

Notice that for writes the address is supplied along with the write data, so the address is incremented when the second byte is written.

```
#ifndef SIMOVERRUN

    DCPSIM_MESSAGE(MAC TYPE IS NEITHER 08XX NOR 88XX);
   /* MSB is some 'other' */
     //NEW - Added code to put next two bytes into unused2
     DCPSIM_MESSAGE(ASM - Putting 2 bytes into extract);
      CregsAddrWrite(EXTRACT_UNUSED2);
      Branch(ALWAYS, Upc+1);
      PayloadIn(FIFOout);
      Abus(Payload);
      ALU(PassA);
      PayloadOut(ALUout);
      IregsA(LastByteRead);
      Actions(IregIncr(FrameLength)|IregsAwrite|UnloadFIFO|CRCaccum|CregsWrite);
      Branch(Data9,Label(RX_DATA_ERROR));

      PayloadIn(FIFOout);
      Abus(Payload);
      ALU(PassA);
      PayloadOut(ALUout);
      IregsA(LastByteRead);
     Actions(IregIncr(FrameLength) | IregsAwrite | UnloadFIFO | CRCaccum |
        CregAddrIncr | CregsWrite);
     Branch(Data9,Label(RX_DATA_ERROR));
    Branch(ALWAYS,Label(STREAM_PDU));

#else
```

The microcode higher level function *CregsAddrWrite()* sets up the Creg address block, which breaks down to:

```
Bbus(Literal(cregAddr));
Actions(CregAddrWrite);
```

This could have been incorporated in the first payload write command, if there was no "branch relative" command. (Relative branches use the SDP's Literal Bus to provide the branch offset.)

In previous versions of the CST, modifying a microcode header file did not automatically cause the dependent files to be recompiled. To force **rxByte.c** to be recompiled, either open the file and save it or execute a "make clean_local" command prior to recompiling.

Verifying the Modifications

To verify that the microcode has written the correct value to the CP's Extract Space, examine the overlying C code with the C-Ware Debugger (gdb), or alternatively use the C-Ware Simulator's microcode messaging and trace facilities.

First, run the Simulator with microcode messaging enabled for RxByte0. This identifies the time that the microcode message appears:

```
[171427 0 RxByte0]  Microcode Message: NEW - Putting 2 bytes into extract
```

Next, rerun the simulation up to this point, then to verify the microcode execution, in the Simulator set trace level 3 on RxByte0 and advance the simulation one clock at a time, as follows:

```
Dcp.CpCluster3.Cp3.Sdp> umsg rxbyte0 1
Dcp.CpCluster0.Cp0.Sdp.RxByte> g 171426
[171333 0 RxByte0]  Microcode Message: START PROCESSING NEW FRAME
[171369 0 RxByte0]  Microcode Message: RECEIVED UNICAST MAC DA
[171377 0 RxByte0]  Microcode Message: DETERMINING FID TO USE
[171380 0 RxByte0]  Microcode Message: NOT 8021Q TAGGED
[171397 0 RxByte0]  Microcode Message: MAC DA LOOKUP
[171415 0 RxByte0]  Microcode Message: PARSE MAC TYPE
[171420 0 RxByte0]  Microcode Message: WRITE MAC TYPE TO DRAM
Dcp.CpCluster0.Cp0.Sdp.RxByte> trace rxbyte0 3
Dcp.CpCluster0.Cp0.Sdp.RxByte> g 1
[171427 3 RxByte0] CPid: 0 PC=0x0BC ==  mBytesSent = 14 CregAddr=0x5
Token: FALSE
                Microcode Message: ASM - Putting 2 bytes into extract
                DataIn: 0x00000000 (valid)  mDataOut: 0x23 (invalid)
                Input FIFO depth: 49
                Abus=0x0 Bbus=0x8 ALUout=0x8 Pbus=0x8 ALU flags (from
prev cycle ALU op): N=0 C=0 C(link)=1 Z=1
              Iregs:     [0]=0x0        [1]=0xD        [2]=0x2
[3]=0x0
                          [4]=0x3        [5]=0x1        [6]=0x0
[7]=0x23
```

```
                           [8]=0x21      [9]=0x0       [10]=0x0
[11]=0x0
                          [12]=0x0      [13]=0x80     [14]=0x0
[15]=0xBC
                CRC32:  CRCaccum = 9F020D00 CRCinput = 0 CRCnext =
9F020D00 CRC32 Load State: 0
                CRC44Partial: 0->0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF4931943
              uPC trace:  0xBB 0xBA 0xB9 0xB8 0xB7 0xB6 0xB5 0xB4 0xB3
0xB1
*******************************************************************
Dcp.CpCluster0.Cp0.Sdp.RxByte> g 1
[171428 3 RxByte0] (dataOut = 0x0)
[171428 3 RxByte0] CPid: 0 PC=0x0BD ==  mBytesSent = 15 CregAddr=0x8
(Creg Write) Token: FALSE
                DataIn: 0x00000000 (valid)  mDataOut: 0x0 (valid)
                Input FIFO depth: 49
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from
prev cycle ALU op): N=0 C=0 C(link)=1 Z=0
              Iregs:     [0]=0x0       [1]=0xD       [2]=0x2
[3]=0x0
                          [4]=0x3       [5]=0x1       [6]=0x0
[7]=0x23
                           [8]=0x21      [9]=0x0       [10]=0x0
[11]=0x0
                          [12]=0x0      [13]=0x0      [14]=0x0
[15]=0xBD
                CRC32:  CRCaccum = 9F020D00 CRCinput = 0 CRCnext =
9F020D00 CRC32 Load State: 0
                CRC44Partial: 0->0
                CRC5:   CRCaccum = 0 CRCinput = 8 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF4931943
              uPC trace:  0xBC 0xBB 0xBA 0xB9 0xB8 0xB7 0xB6 0xB5 0xB4
0xB3
*******************************************************************
Dcp.CpCluster0.Cp0.Sdp.RxByte> g 1
[171429 3 RxByte0] (dataOut = 0x1)
[171429 3 RxByte0] CPid: 0 PC=0x0BE ==  mBytesSent = 16 CregAddr=0x8
(Creg Write) Token: FALSE
                DataIn: 0x00000001 (valid)  mDataOut: 0x1 (valid)
                Input FIFO depth: 48
                Abus=0x1 Bbus=0x0 ALUout=0x1 Pbus=0x1 ALU flags (from
prev cycle ALU op): N=0 C=0 C(link)=1 Z=1
```

```
               Iregs:    [0]=0x0        [1]=0xE        [2]=0x2
[3]=0x0
                          [4]=0x3        [5]=0x1        [6]=0x0
[7]=0x0
                            [8]=0x21       [9]=0x0       [10]=0x0
[11]=0x0
                           [12]=0x0       [13]=0x80     [14]=0x0
[15]=0xBE
                CRC32:  CRCaccum = 9F020D00 CRCinput = 0 CRCnext =
1F5F8623 CRC32 Load State: 0
                CRC44Partial: 0->0
                CRC5:   CRCaccum = 0 CRCinput = 0 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF4931943
              uPC trace:  0xBD 0xBC 0xBB 0xBA 0xB9 0xB8 0xB7 0xB6 0xB5
0xB4
********************************************************************
Dcp.CpCluster0.Cp0.Sdp.RxByte> g 1
[171430 3 RxByte0] CPid: 0 PC=0x0BF ==  mBytesSent = 16 CregAddr=0x9
Token: FALSE
                DataIn: 0x00000002 (valid)  mDataOut: 0x1 (invalid)
                Input FIFO depth: 47
                Abus=0x0 Bbus=0x0 ALUout=0x0 Pbus=0x0 ALU flags (from
prev cycle ALU op): N=0 C=0 C(link)=1 Z=0
              Iregs:    [0]=0x0        [1]=0xF        [2]=0x2
[3]=0x0
                          [4]=0x3        [5]=0x1        [6]=0x0
[7]=0x1
                            [8]=0x21       [9]=0x0       [10]=0x0
[11]=0x0
                           [12]=0x0       [13]=0x0      [14]=0x0
[15]=0xBF
                CRC32:  CRCaccum = 1F5F8623 CRCinput = 1 CRCnext =
42D4A523 CRC32 Load State: 0
                CRC44Partial: 0->0
                CRC5:   CRCaccum = 0 CRCinput = 1 CRCnext = 0
                SDP Mode Register = CregSdpMode3=0xF4931943
              uPC trace:  0xBE 0xBD 0xBC 0xBB 0xBA 0xB9 0xB8 0xB7 0xB6
0xB5
********************************************************************
Dcp.CpCluster0.Cp0.Sdp.RxByte>
```

In this output notice the following:

- The value for CregAddr lags the microcode instruction by one clock; however, the write is directed to the correct Creg.
- Other values of interest are the *DataIn* and *mDataOut* fields, which show that the payload has been written to DMEM as well as to Extract Space.

Modying the Receive-Side and Transmit-Side CPRC Programs

You must next modify the C source code for the receive-side CPRC program and transmit-side CPRC program. Examine this code to determine whether there is space in the enqueued descriptor to pass the 2Byte field and where to remove this data for placement in the CP's Merge Space. This is best performed by using the CST's C-Ware Debugger (gdb) to step through the code.

Receive-Side CPRC Program Modifications

The code to be modified resides in this file:

...\apps\gbeSwitch\chip\np\cprc\src\macRxCp.c

To step through the receive-side CPRC code, start the Simulator, move to Cp0, then start the Debugger. From the Debugger console set a breakpoint on *DCPmain()* for thread 0, then continue.

The receive-side CPRC program performs some initialization before calling the function *receive()*, which in turn calls the function *macReceive()*. Thus, to work quickly, set a breakpoint at *macReceive()* for thread 0, then continue.

The following listing illustrates the steps to take in the Debugger:

```
gdb) b DCPmain thread 0
Hardware assisted breakpoint 1 at 0x8024: file
../chip/np/cprc/src/cpMainRx.c, line 69.
(gdb) c
Continuing.
[Switching to process 0]
Breakpoint 1, DCPmain () at ../chip/np/cprc/src/cpMainRx.c:69
(gdb) b macReceive thread 0
Hardware assisted breakpoint 2 at 0x8d7c: file
../chip/np/cprc/src/macRxCp.c, line 252.
(gdb) c
Continuing.
[Switching to process 0]
Breakpoint 2, macReceive () at ../chip/np/cprc/src/macRxCp.c:252
(gdb)
```

At this point the Debugger's source window displays the following code:

```
-  252    while(1){
   253        MSG_EVENT(START_GBE);
-  254        ipRxDescData->appData3 = 0xffff;
   255#if 0
   256        /* Default fid contains all ports untagged */
   257        fidMembers = macBcastVector;
   258        fidUntagged = macBcastVector;
   259        fidCount = macBcastCount;
   260        fidValue = macPVID;
   261#endif
-  262        returnFromAn = FALSE;
   263
   264        /* Check to see if there is data ready to process. */
   265   MSG_EVENT(WAIT_FOR_RX_SCOPE);
-  266        while ((pduHandle = pduRxAllocate()) == NULL) {
   267
-  268            rxBitCtl->cpRcReady = TRUE;
-  269            rxBitCtl->duplexMode = 0xBB;
   270           /* Check for control messages if there is free time.*/
-  271            if(!(ksCpId & 3)) {
-  272                while(qsQueueGetLength(ctrlQueue)) {
-  273                    processCtlCommand(ctrlQueue);
-  274                }
   275            }
   276        }
   277
   278        /* Catch any updates to these values */
-  279        fidMembers = macPvidMemberSet;
-  280        fidUntagged = macPvidUntaggedSet;
-  281        fidCount = macPvidMemberCount;
   282        fidValue = macPVID;
   283
   284 /* LOG("packet received"); */
   285
   286        /* Get the buffer handle for this PDU */
   287        bufHandle = pduRxBufHandleGet(pduHandle);
-  288                ipRxDescData->bufHandle = bufHandle;
```

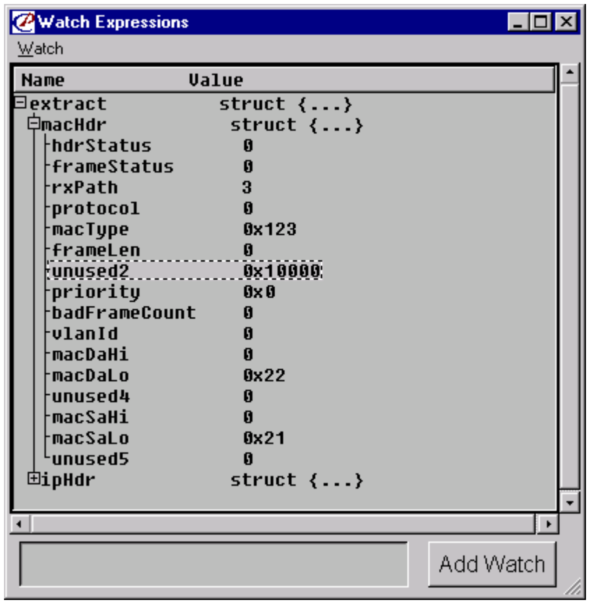
The CPRC program is polling the datascope ownership bit to determine when the RxSDP has completed header processing. Set a breakpoint on line 288 to break when the SDP has finished processing.

Step over the next few lines until the pointer for Extract Space is cast:

```
 290    /* Get extraction space */
 291            extract = (BridgeExtract *)pduRxHeaderGet(pduHandle);
 292
 293             /* Validate that the frame thus far and the header are valid
 294              * NOTE: This assumes that the header status and frame status are
 295              * adjacent in the extract space and that both use 0 to indicate
 296              * a successful frame and/or header status. */
 297
-298            if (*(int16u *)&extract->macHdr.hdrStatus == macHdrStatusSuccess) {
 299
 300                /* LOG("header status success"); */
```

After this executes, you can add to the Debugger’s Watch window to expand the values within the structure. In Figure 14 notice that the first two bytes of payload are in the first two bytes of *unused2*.

Figure 14 Debugger Watch Window for the ‘extract’ Structure



```
Name                Value
⊟extract             struct {...}
 ⊟macHdr              struct {...}
   hdrStatus           0
   frameStatus         0
   rxPath              3
   protocol            0
   macType             0x123
   frameLen            0
   unused2             0x10000
   priority            0x0
   badFrameCount       0
   vlanId              0
   macDaHi             0
   macDaLo             0x22
   unused4             0
   macSaHi             0
   macSaLo             0x21
   unused5             0
 ⊞ipHdr               struct {...}
```

Next, step over the next few instructions until you reach the line that calls the classification function:

```
   379            /* Call the appropriate function according to the rx path as set by the
SDP. */
-  380            rxStatus = macRxVectorTbl[extract->macHdr.rxPath](extract, &macFwdBlock,
   381                                                             ipRxDescData);
   382
```

This function call is determined by an array that holds a list of possible classifications that are required. The index to this array has been determined by the RxSDP. In this case it will call *MacRxUnicast()*, which performs some checks, retrieves the lookup results, and sets up the enqueued descriptor.

The enqueued descriptor *ipRxDescData* is of type *IpDescriptor*, which is defined in the file:

...\apps\gbeSwitch\tgt\c5\inc\ipIf.h

This is the structure definition:

```
typedef struct {
    BsBufHandle         bufHandle;
    int16u              length;
    int16u              VNID_client; /* 12bit VNID+4bits for clientId.*/
    int32u              appData1;
    int16u              appData2;
    int16u              appData3;
} IpDescriptor;
```

The descriptor is 16Bytes long and has three fields that can be used for application-specific data. Stepping through the code further shows that *appData1* is used to store the transmission algorithm, and *appData3* is used to determine whether VLAN tagging is required. *appData2* is unused.

Modify the code as shown at the bottom of the listing in Figure 15 on page 176 to set up *appData2* to the value in Extract Space. Notice that *unused2* is a 32bit field, while *appData2* is a 16bit field, so a shift is necessary to put the correct portion in the descriptor.

Figure 15 Listing of macRxCp.c to Be Modified

```
if(rxStatus == macRxFwdUnicast) {

                /* LOG("macRxFwdUnicast"); */

                /* Is the target local or over the fabric? */
                if(macFwdBlock.fabricId == dcpFabricId) {
                  /* If this is a VLAN tagged frame, check whether this port should
                     * send tagged or untagged.  If VLAN lookup succeeded, we are
                     * already set up to send out tagged */

                  /* NOTE this relies on the macBaseQueues values being spaced by 16
                     *      allowing us to get from queueId to port# by dividing */
#if 0
                    if( extract->macHdr.vlanId &&
                        ( fidUntagged & ( 1 << (macFwdBlock.queueId >> 4))) ) {
                        /* Port should go out untagged */
                        ipRxDescData->appData3 = 0xffff;
                    }
#else
                    if(extract->macHdr.rxPath == rxInternalMacDA)
                    {
                        /* Do nothing for routed frames */
                    }
                    /* Non-routed VLAN tagged */
                    else if( extract->macHdr.vlanId)
                    {
                        if ( fidUntagged & ( 1 << (macFwdBlock.queueId >> 4)))
                        {
                            /* Port should go out untagged */
                            ipRxDescData->appData3 = 0xffff;
                        }
                        /* tagged case is already set-up ! */
                    }

                    /* PVID case for non-routed frames */
                    else if(!(extract->macHdr.vlanId))
                    {
                     if ( macPvidUntaggedSet & ( 1 << (macFwdBlock.queueId >> 4)))
                       {
                           /* Port should go out untagged */
                        ipRxDescData->appData3 = 0xffff;
                       }
                       else
                       {
                           ipRxDescData->appData3 = macPVID;
                       }
                    }
#endif
                    macFwdBlock.queueId += extract->macHdr.priority;

   //NEW - Added setup of appdata2
                       ipRxDescData->appData2 = extract->macHdr.unused2>>16;
                }
```

Transmit-Side CPRC Program Modifications

Now you can perform a similar set of steps for the transmit-side CPRC program's code. In this case we are looking for the descriptor being dequeued and where to put the descriptor data into the transmit CP's Merge Space.

The transmit CP is sitting in a loop in function *transmit()* waiting for a QMU notification that a descriptor is available. Quit the Debugger session on the receive CP. In the Simulator move to Cp12 and start a new Debugger session. If the receive CPRC is not executing the code in **macTxCp.c**, step through the CPRC program until the Debugger reaches that code.

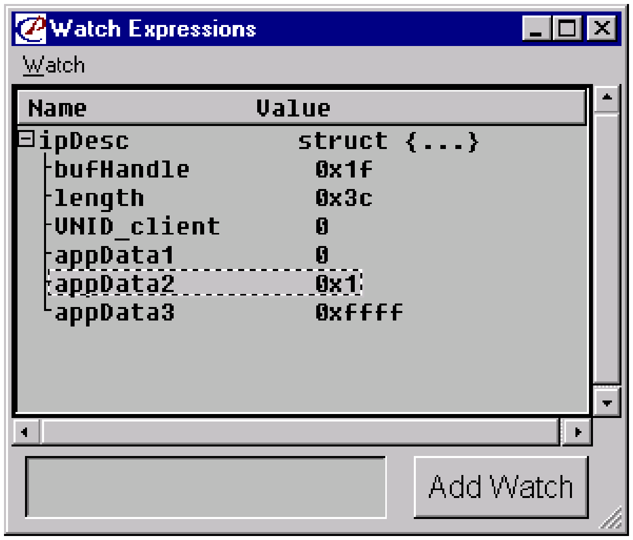
```
-   while (1) {

          MSG_EVENT(START_GBE_TX);

        /*
         * Get next queue to drain from. Note that this function is a blocking
         * function (no context switching used in this application). Also note
         * that it hides the version specific differences in dequeue support.
         */

-       LOG("Calling getNextQueue()");

-       getNextPacket(ctrlQueue, macBaseQueueId);

        ipDesc = (IpDescriptor*)qsMessageGetData((QsMessage*)&macTxDescStorage);

```

In **macTxCp.c**, step over the code until you see the pointer for the CP's Merge Space being set up.

```
        LOG("SDP transmitting");

        /* Get pointer to merge space */
        merge = (MacMerge*)pduTxHeaderGet(pduHandle);

       /* Prepare to update stats. Add back 4bytes of the FCS. */
-       frameLen = ipDesc->length + 4;
-       switch ( (ipDesc->VNID_client & 0x000F)) {

```

At this point the CP’s Merge Space is available to be examined, as is the dequeued descriptor *ipDesc*. Figure 16 shows the Debugger Watch window’s display of the *ipDesc* structure.

Figure 16 Debugger Watch Window for the ‘ipDesc’ Structure



```
Name                Value
⊟ipDesc             struct {...}
  bufHandle         0x1f
  length            0x3c
  UNID_client       0
  appData1          0
  appData2          0x1
  appData3          0xffff
```

You can verify that *appData2* contains the value extracted. Stepping through the code shows that in this case the code switches to case *macClientTxRaw*, and this is where the code to insert the data into Merge Space should be added.

```
case macClientTxIcmp:
case macClientTxRaw:  /* Transmit frame as is (ala Bridged) */
case macClientTxRawIgnorePortState:

  /* Set the transmit algorithm for bridging */
  merge->txAlgorithm = macTxAlgRaw;
     //ASM - Set up unused2
     merge->unused2 = (ipDesc->appData2<<16)+ipDesc->appData2;

  if( ipDesc->appData1 == macTxBcast ) {
         ++macStatsTx.txBcastPkts;
```

Here a 32bit assignment is created by duplicating the 16bit data on both the MS and LS halves. This ensures that the C compiler 16bit integer assignment is correct, regardless of the alignment of the two 16bit entities *merge->unused2* and *ipDesc->allData2*. (Tip: Stepping the Debugger through versions of the code with or without a shift will allow you to determine which is the correct assignment.)

A final run of the traffic through the Simulator produces the following results in the transmitted data out the transmit-side channel :

```
MAC da:000100000000 MAC sa: 002200000000 type: 0021 crc:5433448f
Payload Numeric=[0123020304050607080900a0b0c0d0e0f1011121314151617181191a1b1c1d1e1f202122
232425262728292a2b2c2d]
Time Elapsed: 568  ns
```

This indicates that the first two payload bytes 0x0001 are removed from the payload and prepended to the frame.