C-Ware Application Design and Building Guide

C-WARE SOFTWARE TOOLSET, VERSION 2.4

CSTADBG-UG/D

REV 01

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by that customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyers purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyers shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

TABLE OF CONTENTS

About This Guide

CHAPTER 1 CST Application Overview

CHAPTER 2 Application Design and Coding Options

CHAPTER 4

Building and Packaging An Application

FIGURES

TABLES

ABOUT THIS GUIDE

Guide Overview

The *C-Ware Application Design and Building Guide* describes several tools that are included with the C-Ware Software Toolset (CST) and that allow you to build executable programs targeted for a C-Port Network Processor (NP) and for the CST's C-Ware Simulator.

This guide is primarily intended for software application developers who must produce communication applications that run on a C-Port NP, assuming the reader's basic understanding of the C-5 NP architecture and the CST's development tools, as documented in the CST documentation set listed in Table 3 on page 16.

This guide is organized as follows :

- Chapter 1 provides an overview of the structure of a CST application as it relates to the Build System structure of the C-Ware Software Toolset and the C-Port Network Processor architecture.
- Chapter 2 describes "design" approaches for C-coding and compiling an application within the CST build system structure and also for partitioning and packaging it for maximal implementation of the C-Port Network Processor.
- Chapter 3 describes specific directory structure requirements and build system conventions for using the Makefile build system utility within the CST.
- Chapter 4 describes how the C-Ware compiler and the **dcpPackage** tool work together to support partitioning an application into separate, dynamically loadable initialization executables and main processing executables. It also explains why and how to use the **dcpPackage** tool for producing package files. A *package file* contains all software and data for the application that is loaded into a C-Port NP.
- Appendix A describes the C-Ware Software Toolset's special table-building tool, the Offline Table Building Library, for building routing and switching tables used by your NP-based network application. It also provides additional details about key modules in the Main Phase program.

DF 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:

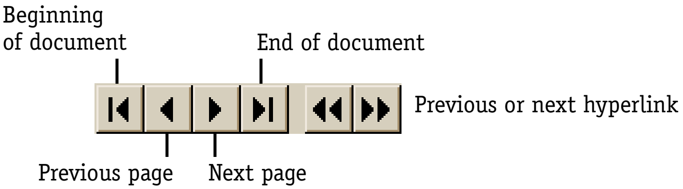


Table 1 summarizes how to navigate within an electronic document.

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.

ces to CST nes

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 2000/XP)
- **/usr/**_yourlogin_**/C-Port/Cst**_x_**.**_y_**/** (on Sun SPARC Solaris and Linux)

  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 2000/XP system might be **C:\C-Port\Cst2.1\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.

ision History

Table 2 provides details about changes made to create this new guide.

Table 2 *C-Ware Application Design and Building Guide* Revision History

REVISION	DATE	CHANGES
01	6/2004	This document was revised to replace internal references to ’Motorola’ with ’Freescale Semiconductor’. Copyright Freescale Semiconductor, Inc. 2004. Deleted "ver" and "mfg" as unsupported variants for target environments.
00	3/2004	Created a new, unified document after collating 4 documents published separately for previous CST releases (*Application Guidelines*, *Build System Conventions*, *Application Design Guide*, and, the forerunner of this manual, *Application Building Guide*).

For Revision History details on the now-retired Application Building Guide, refer to its previously published version, released on June 12, 2002.

Product ntation

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
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 and Linux)
	C-Ware Debugger User Guide	Describes the GNU-based tool for debugging software running on either the C-Port network processors simulators.	CSTDBGUG-UG
	C-Ware Integrated Performance Analyzer User Guide	Describes use of the Integrated 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

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

JMENT SUBJECT	DOCUMENT NAME	PURPOSE	DOCUMENT ID
Application Development	*C-Ware Application Design and Building Guide*	Describes tools to build executable programs for the C-Port network processors or simulators and design guidelines and trade-offs for implementing new C-Port NP-based communications applications.	CSTADBG-UG
	C-Ware API Reference Guide	Describes the subsystems and services that make up the C-Ware Applications Programming Interface (API) for C-Port NP-based communications applications.	CSTAPIREF-UG
	C-Ware API Programming Guide	Provides practical guidance in programming the C-Ware API services.	CSTAPIPROG-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
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

Chapter 1

CST APPLICATION OVERVIEW

Overview

The overall programming process involves at least two, and more likely three main phases: “writing”, “building” (also called “compiling”), and “testing & revising”. This manual is a guide to application “design” and “building” to suggest an overall process that is *repeated* — that is, that most programmers during application development will cycle *several times* through all the phases of writing, compiling, testing, and revision.

Design refers, not only to the overall development of an idea for an application using a C-Port Network Processor (NP), but also to the proper layout of modules within an application so that they can access and activate different parts of the NP during program initiation and ongoing processing. Design, therefore, also involves adherence to C-Ware Software Toolset (CST) directory structure requirements, naming conventions, and partitioning conventions for using various parts of the NP. Design also involves adherence to Makefile rules as well as to the rules for using other tools within the CST, such as cwsim, cwipa, and so on.

Building refers to all aspects of compiling, linking, and testing a CST application. Implicit is the use of all tools available within the CST environment, however this manual only discusses aspects of the Makefile tool and the GNU C compiler. Refer to other tool manuals in the CST documentation set for additional details, especially for simulation and debugging.

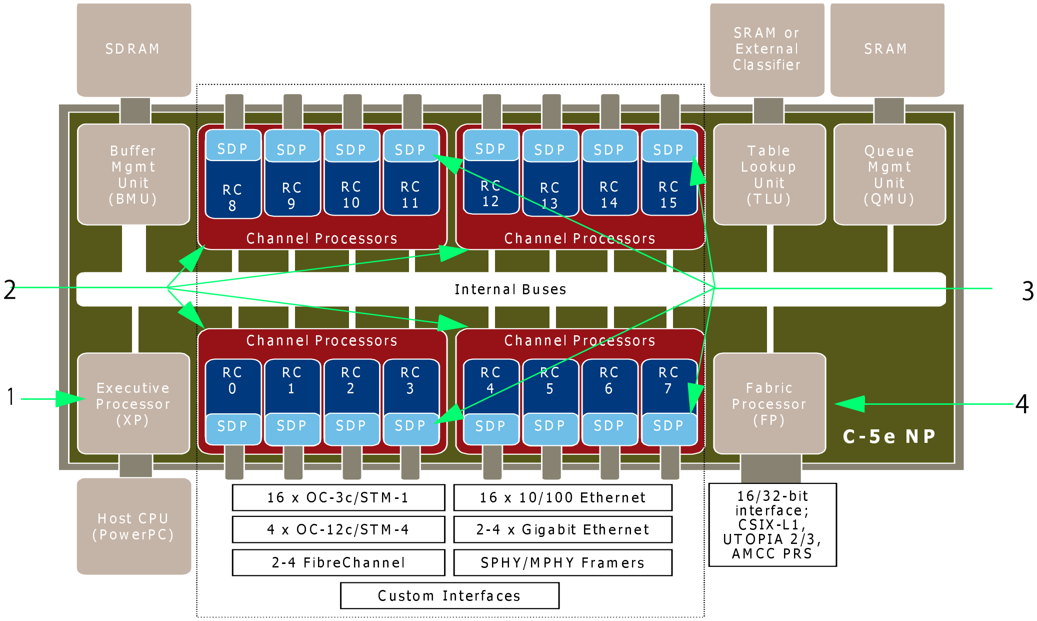
The programmer partitions the NP application into some or all (and more) of the following sections, each of which refer to processors within the NP:

XPRC
CPRC
SDP
FP
.
.
.

The sections refer to Executive Processor Risc Core (XPRC), Channel Processor Risc Core (CPRC), Serial Data Processor (SDP), and Fabric Processor (FP). The purpose for each is summarized below. For additional details on these sections, refer to "Functional Distribution" on page 3.

- Modules that a programmer specifies in the section for XPRC will govern the behavior of the Executive Processor Risc Core, that is, the processor assigned the task of initializing and maintaining the NP. See "1" in the figure below.
- Modules that a programmer specifies in the section CPRC will govern the behavior of each of the16 Channel Processors in the NP. See "2" in the figure below.
- Modules that a programmer specifies in the section SDP will govern the behavior of the 5 serial processors that are assigned to each Channel Processor in the NP (5 X 16 = 80 serial processors). See "3" in the figure below.
- Modules that a programmer specifies in the section FP will govern the behavior of the Fabric Processor (if implemented) in the NP. See "4"in the figure below.

The overall layout of the typical Network Processor is as follows:



ctional Distribution

Each processing activity already identified has functional characteristics that lends itself to being implemented in certain NP components or on a host processor. This section briefly presents the functional capabilities of each of the NP's processing components and how those capabilities support the required processing activities' functional characteristics.

SDP

The SDPs are proprietary microsequencers that contain a series of 8bit machines that process one byte at a time of network data traffic. These machines are optimized for bit-level computation and manipulation of the data stream. The SDPs have specialized support for CAM matching, CRC validation and generation, ALU computation, and so on.

Of the 5 microsequencers (also called serial processors) that make up the SDP for each Channel Processor:

- Two are used for Serial Data Processing Transmissions (TxSDP): TxBit and TxByte.
- Three are used for Serial Data Processing Receptions (RxSDP): RxBit, RxSync, and RxByte.

In addition, the SDPs' microsequencers can read and write information to a small set of registers that are shared with the SDP's corresponding Channel Processor RISC Core (CPRC). These registers can be used to exchange information such as status code reporting, transmit instructions, header extraction, and so on.

The SDPs' microarchitecture make them particularly apt for bit-level manipulation before writing information into these shared memory spaces. The benefit is that the CPRC's code can be made more efficient because that code need not spend cycles on bit-shifting or ANDing, operations that are typically inefficient in a RISC processor.

One of the SDP microsequencers (the RxByte serial processor) also has the special ability to launch transactions to the Table Lookup Unit (TLU) over the NP's Ring Bus. This allows for pipelining the process of looking up information, while the contents of a packet or cell are being received across the channel.

Finally, certain SDP microsequencers (RxByte and TxByte) have access to an area of the Channel Processor's DMEM that is used as temporary storage for packets or cells that must also be stored in buffer memory that is external to the NP. These areas of DMEM are written to (RxByte) or read from (TxByte) by these microsequencers in order to move the frames payload either to or from the external buffer memory.

Some operations that are typically handled by a MAC or framing layer device or activity are appropriately performed by the NP Channel Processors' SDPs. Some of these operations are very inefficient when implemented on a general-purpose RISC processor.

Given these capabilities, the SDPs typically implement the following types of operations:

- CRC validation and calculation
- Packet and cell header validation
- Extraction of cell or packet headers or fields in those headers
- Launching of lookups to the NP's TLU via the Ring Bus
- Writing or reading of packet or cell payload data to/from DMEM for DMA to/from external memory

In keeping with their intended function, the SDPs' programming language appears as familiar C code, but it is a microcoded, proprietary language. When implementing SDP microprograms, one thing to consider in your software design is the degree of difficulty in writing, debugging, and maintaining microcode as such. The case can be made for trading off some application performance for ease of application maintenance and understandability, by having functionality exist in the CPRCs instead in the SDPs. However, most system designers put significant functionality into the SDPs primarily to support different external interfaces (Ethernet, GbE, OC3, 12, etc). Some designers even employ the SDP to maximize the application's performance across all NP processing resources, but there is not much room to move functionality from CP to SDP. For example, RxByte within SDP only has "512 words" storage for microcode programming. For additional details on how to microcode the SDPs , refer to the *C-Ware Microcode Programming Guide* .

CPRC

The NP 's Channel Processor RISC Core (CPRC) is contained in each NP Channel Processor (CP). These RISC cores implement an instruction set that is relatively simple (that is, MIPS I compliant) and that doesn't contain instructions for multiplication, division, or floating-point operations.

Each CPRC has a dedicated local instruction memory, or IMEM (24KB for C-5 and 32KB for C-5e/C-3e), and local data memory, or DMEM (48KB). The somewhat limited size of these data stores can lead to CPRC programs that consume the lion's share of instruction and data memory in the CP.

The CPRC can access many of the NP's resources, such as event timers, cycle timers, interrupts, DMA to/from external memory, the TLU, QMU, and so on. Thus, the user's CPRC programs can be very flexible and surprisingly complex, despite its ease-of-use via the C code language. This level of detailed accessibility can greatly improve the efficiencies of the resources available to the overall C-Port NP-based system.

The CPRCs also have the role of having to complete certain tasks in a finite amount of time to meet the performance criteria for wire-speed operation set by many networking data path protocols. Thus, the amount of work and amount of code that a CPRC can execute over a single packet or cell is small. This is precisely the reason that the SDP, not the CPRC, implements some of the application's more time-critical tasks (launching lookups, data validation, and so on).

The CPRCs are instrumented to perform some of the application's more complex logic because of their powerful instruction set and access to certain other NP system resources. For example, the CPRC program typically collects lookup results that were launched from that CP's SDP, evaluate them along with other information parsed by the SDP, and finally make a classification or policy decision (that is, whether to enqueue something for an egress port, filter it, count a statistic, and so on).

The CPRC also has access to other system features like fast context-switching, shared DMEM, and aggregation (see the section "Aggregation" on page 24) that can aid in processing particular types of traffic -- high speed, loss of processing, and so on.

Among the C-Ware Reference Library applications, here are examples of functionality that is implemented on the CPRCs:

- Collection of lookup results
- Inspection of results of packet processing and validation
- Keeping of statistics
- Notification of the XP upon the occurrence of certain events, such as bridge address learning or reception of SONET overhead
- Forwarding and drop decisions
- Error handling in the data path

XPRC

The NP's Executive Processor (XP) contains its own Executive Process or RISC Core (XPRC). As a RISC processor core, the XPRC is functionally identical to the CPRCs. However, the XPRC is intended to perform different application activities than implemented on the CPRCs.

For example, the XPRC is not used in the data path forwarding function between physical interfaces. This is because the CPRCs typically notify the XPRC either by a shared memory write over the Global Bus or by DMA of data to XPRC's DMEM. The XPRC also has similar instruction and data memory size limitations as the CPRC (32KB of IMEM for C-5 and 48K of IMEM for C-5e/C-3e, and 32KB of DMEM for all NPs), so the XPRC-resident program is also limited in size.

Though the name might imply it, the XPRC does not run any type of real-time executive program. The primary function of the XPRC is system boot, initialization, and code download, but actually the most important function for XP is to communicate with the external Host processor. When the NP is taken out of reset, the XPRC is the first processor to load and execute its program. Because of the XPRC's small amount of available IMEM and DMEM, XPRC programs must be relatively small.

The XPRC is also responsible for reading the NP *package* (a structure that is the concatenation of all executable code for all NP processors and certain additional memory structures) out of either PROM or PCI memory, then loading that program into the instruction and data memories of all the NP's CPRCs and SDPs. Additionally, the XP distributes any data from the package to all the CAMs and instruction storage for all the NP's microsequencers (in the CPs and FP).

After initialization, the XPRC begins running the user program on the XPRC. The XPRC does not usually place itself in the application's data path for all packets. Instead, the XPRC often implements low-latency control protocols that must run periodically in the system but that also must complete in a reasonable amount of time and for cases where the host processor shouldn't be involved.

Examples of user XPRC programs that require low latency:

- Bridge Address Learning and Again in for Ethernet MAC bridging
- SONET overhead monitoring for applications that support SONET
- An agent that performs communication between the NP and the a debugger running on the host

Additionally, the XPRC runs a portion of the C-5 Device Driver that enables the host processor to exchange information with NP. The XP manages this communication using the PCI bus and memory-mapped accesses and using DMA between itself and the host processor.

Using the PCI interface, the XP can move very small to very large units of data between the host and NP. Depending on the size of the data, you might prefer to use different methods for moving the data. The PCI interface on the XP supports DMA from external buffer memory to local XP DMEM to host memory over the PCI. The NP can also perform read and write memory-mapped transactions from the host over the PCI bus, 32 bits at a time.

Depending on the type of data transaction, you might use one or the other technique. The C-Ware Reference Library applications typically use PCI DMA to move packet data between the NP's processing resources and use memory-mapped reads/writes for small control information (status reporting, and so on).

Each of these techniques is by choice of the application software running on the XPRC and on the host processor.

Host Processor

The host processor in a C-Port NP-based system runs a Real-Time Operating System (RTOS), which, in turn, must utilize the C-5/5e/3e Device Driver via the C-Ware Host API's Host Services routines. These routines and the Device Driver must be built into the Board Support Package (BSP) that is part of the *host application program* that runs on the host processor. The host application program also contains the software that will functionally "connect" the network stack running on the host processor with that software running on the NP.

The NP Device Driver allows the host processor to programmatically access and control the NP. That is, the Driver allows the host processor to interrupt the NP, take a NP out of reset, to interrupt the NP's XPRC and CPRC programs, to download a package to the NP, to initiate a DMA data transfer between host processor memory and NP memory, and so on. The host application program typically accesses Device Driver functionality via the C-Ware APIs Host Services. The C-Ware Host API Host Services, the C-5 Device Driver, and their relationship are described in the *C-Ware Host Application Programming Guide* document.

There are additional features for particular system requirements that are typically implemented in the host application program, such as routing table updates, console driver, a SNMP agent, and so on.

Chapter 2

APPLICATION DESIGN AND CODING OPTIONS

Overview

Developing a CST application requires many decisions regarding "design" of the program to make most effective use of all Risc processors and SDPs within the C-Port Network Processor. In this sense, design is fundamentally directed toward maximizing NP architecture.

While this manual addresses "design and building" an application in a slightly different sense (see “Overview” on page 1), an element of design is clearly involved in all facets of application development. Within this chapter, design is treated in that broader approach.

The following sections discuss design to maximize the NP architecture **during C code module development**, using C coding requirement and recommendations:

- “C Coding Recommendations” on page 10
- “Tokens Versus Semaphores” on page 17
- “Recirculation” on page 20
- “Aggregation” on page 24
- “Shared DMEM” on page 26
- “IMEM Optimization” on page 27
- “Table Design and Table Building” on page 31

The following sections discuss design to maximize the NP architecture **during compilation/linking/packaging** (partitioning and overall building):

For a full description of the application "building" procedure itself , which involves all the above elements, refer to Chapter 3

- "Partitioning An XPRC Program For Initialization" on page 35
- "Partitioning the Program" on page 36
- "Coding the Package Description File" on page 37
- "Passing User Data From Init Phase Program to the Main Phase Program" on page 38
- "Address Resolution Among XPRC, CPRC, and XP Primary Bootstrap" on page 39
- "Valid API Routines to Call From an Initialization Phase Program" on page 42
- "Design Tradeoffs" on page 44

C Coding Recommendations

These coding and compilation recommendations can help reduce the size of your C programs and result in faster code execution.

Suppress Automatic Inlining of Functions

When optimization is in use, the **gcc** compiler aggressively *inlines* functions — that is, it integrates that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead, but it can also drastically increase the size of the generated code.

The CST uses this **gcc** switch to suppress automatic inlining of functions:

```
-fno-inline-functions
```

Functions explicitly declared as *inline* are inlined as usual. Here are some guidelines for using explicit function inlining in your programs:

- If the function is very short (about two to five opcodes, excluding the return instruction), it is a good candidate for inlining.
- If the function is relatively short and you are absolutely sure that it is now, and will be forever, called in only one place, it is a possible candidate for inlining.
- Functions that use *varargs* or *stdarg* calling sequences (such as *printf* and its variants) or called indirectly through a pointer cannot be inlined.

- Recursive functions are usually not good candidates for inlining.
- A function body that appears in a header file should always have the keywords *static inline* in its definition. This prevents the compiler from instantiating a separate function body that never gets called if all uses of the function are successfully inlined.
- Do not declare as *inline* a function prototype if the function's body is not available in the compilation unit. The compiler cannot inline code that it cannot see.
- For **gcc**, inlining of a function works only if that function appears lexically before any uses of it. Put the "leaf" *inline* routines at the top of the source file, followed by the "stem" *inline* routines that might use these leaf routines, and so on.

Avoid Branches

A branch can cause between zero and three IMEM fetch stall cycles, depending on the target address. In general, avoid branches in your code where possible.

Example 1

A cascaded if-then-else statement is a common example of code where dynamic branches can be eliminated:

```
if (bar == 0)
   // Almost never gets here
else if (bar < 0)
   // Sometimes gets here
else
   // Almost always gets here
```

The above code can be rewritten for better results as the following:

```
if (bar > 0)
   // Almost always gets here
else if (bar < 0)
   // Sometimes get here
else
   // Almost never gets here
```

Example 2

Here is another example:

```
if (cond){
   // Common path here
   x  = 1;
}
else
   x++;

if (x == 1)
   // Do something
```

The above code can be rewritten for better results as the following:

```
if (cond){
   // Common path here
   x = 1;
   // Do something
}
else
   x++;
```

Use the Test and Set Operation for Locking Support

The “Test and Set” operation enables any of the RISC cores (that is, the Executive Processor RISC Core, or XPRC, and the Channel Processor RISC Core, or CPRC) on the C-5 NP to provide lock support that an application can use to implement semaphore-like behavior.

The Test and Set operation is accomplished in software by executing the lwc0 opcode. The format of the *lwc0* instruction is:

```
   lwc0 rt, offset (base)
```

where *rt* is the target register, *base* is the register that contains the base address, and *offset* is the byte offset from the base address in DMEM.

This opcode operates as a standard load instruction, reading the DMEM at the given address by *offset* (*base*) and storing the result in the target register.

Regardless of the value read from memory at the address in DMEM specified, the location is written with 0xFFxxxxxx, where ‘x’ represents the value prior to the read. The value transferred to the target register is ‘xxxxxx’.

The value available in the target register reflects the success/failure of the Test and Set operation. If the most significant byte of the target register is zero, the location has not previously been used for a Test and Set. If the most significant byte is 0xFF, the location has been used as a Test and Set destination.

This scheme allows branching based on the sign of the target register.

To reset the location, software can reset the location (via the *sw* instruction or another instruction).

This functionality is available in the C-Ware API library via the functions *ksMutexInit()*, *ksMutexLock()*, *ksMutexLockTry()*, and *ksMutexUnlock()*.

After the *lwc0* instruction is executed, like any other memory load instruction, there must be a one-cycle delay before accessing the destination register.

Avoid Globals, File Statics and Pointers

A local variable is much easier for the compiler to disambiguate, so it usually gets promoted to a register for the lifetime of the routine (unless register pressure gets too high, which generally isn't the case). In some cases it might be beneficial to copy such a variable to a local. For example, look at the following code fragment:

```
// Compiler can’t tell if someOtherGlobal and *p might overlap,
// so it generates a "load *p" instruction each time through the loop.

foo (int* p) {
   for (expensive loop){
      if (someOtherGlobal == *p){
         // Do something
      }
    someOtherGlobal = something
   }
}

// If someOtherGlobal and *p never overlap, copy *p before entering
// the loop. Here the compiler knows a local can never interfere
// with someOtherGlobal and should put local into a register.
// This avoids a load each time through the loop.

foo (int* p) {
   int local = *p;
   for (expensive loop){
      if (someOtherGlobal == local){
         // Do something
      }
   someOtherGlobal = something
   }
}
```

Avoid ‘Volatile’ Storage Qualifier

Use the *volatile* storage qualifier only on control, input/output, and other addresses that the NP hardware changes asynchronously or that have a side-effect when read, such as the rxMsg FIFO.

From the compiler’s view, a *volatile* object cannot be promoted to a register, cannot be eliminated, cannot be part of a common subexpression, and so on. In a nutshell, every reference to a *volatile* object generates a load or a store instruction.

If you load a *volatile* object and know that it can't change because of some semaphore (that is, the *Avail* bit indicates that you own it), copy the field into a local and reference the local instead. In this case, you might also consider removing the *volatile* storage qualifier from the Creg that you only read when its “safe”.

void Shared and Global DMEM Accesses

“Shared” means non-local but within the same CP cluster. “Global” means across a CP cluster. Shared accesses always use one extra cycle. A global bus access can take 100 cycles in the worst case.

“Global” also refers to a variable that is accessible outside its compilation link unit. Such an object has an “optimizer penalty”. Use of global variables typically prevents optimized access of that data.

It might be better to use a local and just pass it around to its using routines, particularly if the using functions can be inlined.

If a choice is available, select the storage class for your program’s variables in this order of preference:

1. Local
2. Parameter (particularly if can be inlined or passed in register; that is, no more than four arguments)
3. C file *static*
4. C compilation unit global

Read the Generated Code

The **gcc** compiler is not always as efficient as it could be. Extra moves and branches could be eliminated by the compiler but sometimes aren’t. In these cases you might explicitly inline a function or otherwise explore what is preventing optimization.

For example:

```
// Simply changing the return type of foo() to int reduces the
// number of instructions from 19 to 14!  It appears that different
// sized operands have prevented optimization.

int ext;


void halt();
void ksPrintf(char *format, ...);


static  __inline__
signed char foo() {
   if (ext > 0)
      return 1;
   else
      return 0;
}


void start() {
   if (foo()){
      ksPrintf("pass");
   }
   else{
      ksPrintf("fail");
   }
   halt();
}


void halt() {
   asm("add $0, $0, $0");
}


void ksPrintf(char *format, ...) {
   asm("add $0, $0, $2");
}
```

Avoid Literal Masking of Control Registers

Instead of masking control registers with literal constants, use the macros in **services\chip\inc\dcpRegisterDefs.h**. These macros have been optimized to produce efficient MIPS code when extracting bits.

For example, this expression produces terrible code:

```
(txmsg_ctl & 0x80000000)
```

Use this macro instead:

```
BitIsSet(txmsg_ctl, RB_AVAILABLE)
```

Avoid Function Calls

A function call is slower for obvious reasons — it requires instructions for argument setup, stack setup, and so forth — but also may cause shared IMEM stalls.

Functions Take a Maximum of Four Arguments

The first four arguments for a called routine are passed in registers and can be referenced directly. Each argument after the fourth is stored in non-register memory and is accessed in the called routine indirectly via pointer.

Avoid Loads/Stores to Local DMEM After Initiating DMA

A load or store instruction on a CP's local DMEM can stall during payload transfer. If possible, pre-fetch local data before initiating transfer or defer it until DMA already complete.

Avoid Serializing Tasks

Look for independent tasks and try to initiate them without waiting for them to complete.

Tokens Versus Semaphores

From time to time, application developers are faced with an application requirement of providing serialized access to shared data structures. Examples of this would be structures that represent queues, table entries, and so on.

In the C-Ware Software Toolset (CST) the programming environment supplies two types of support for these types of sharing issues: tokens and semaphores.

Tokens

A *token* is a mechanism that can be used to protect a shared data structure. There are Hardware Receive tokens and Software Receive tokens. For details, refer to Appendix B in your NP *Architecture Guide*. A token is useful in an application when access to the shared data involve *one writer and multiple readers*. In a C-Ware application an example of this is where there is an aggregated set of CPRCs accessing a shared data structure (such as queues), but only one of them at a time is updating the data structure.

The following code fragment illustrates the use of a HW token in he SDP:

```
/* Wait for my RC to have the token before updating the shared data
 * structure */
```

```
while (!ksTokenPresent(SHARED_TOKEN));

/* Now update the shared data structure */
*sharedStruct++;

/* Pass the token, now that I am done with the resource */
ksTokenPass(SHARED_TOKEN);
```

When using tokens by calling the CPI routine *ksTokenPass()*, the passing sequence within a cluster is:

0 -> 1 -> 2 -> 3 -> 0

When passing tokens by calling the CPI routine *ksTokenPassBack()*, the passing sequence within a cluster is:

0 -> 3 -> 2 -> 1 -> 0

Semaphores

A *semaphore* is also a mechanism that can be used to protect a shared data stucture. A semaphore is typically used in an application situation where there are *multiple writers*.

The type of semaphores supported by the C-Ware Software Toolset are *binary semaphores* (as opposed to *counting semaphores*). Counting semaphores are supported by the NP hardware, but no software support for them exists in this CST release.

A semaphore executes a special instruction that locks an item of data when accessing it before returning control, if the resource is available. This primitive operation is typically known as the Test-And-Set instruction.

Due to execution of the Test-And-Set instruction, using a semaphore is slightly more costly in performance terms than using a token. Also, there is limitation in the NP on the memories (that is, whether XPRC DMEM, CPRC DMEM, or other memory addresses) that can be used with a semaphore.

The following is the functionality that the CST provides for utilizing semaphores in C-Ware applications:

- Semaphore initialziation, via the CPI routine *ksMutexInit()*.
- Semaphore asynchronous access, via the CPI routine *ksMutexLockTry()*.
- Sempahore synchronous access, via the CPI routine *ksMutexLock()*.
- Semaphore release, via the CPI routine *ksMutexFree()*.

Application code to synchronously lock a sempahore (that is, spin and wait for it) might includes this sample excerpt:

```
/* Initialize mutex */
ksMutexInit(mySemaphore, "noname");

/* Spin and wait for semaphore */
ksMutexLock(mySemaphore);

/* Now that we have semaphore, update data */
*mySharedData++;

/* Release semaphore */
ksMutexFree(mySemaphore);
```

An optional code segment that could be used to lock asynchronously a semaphore (test and return to caller if semaphore is not available, to prevent spin loops) is as follows:

```
/* Initialize mutex */
ksMutexInit(mySemaphore, "noname");

/* Test for availability */
if (ksMutexLockTry(mySemaphore)) {
   ksPrintf("got semaphore, updating data structure\n");
   *sharedData++;
   ksMutexFree(mySemaphore);
   return success;
} else { /* couldn't lock */
      ksPrintf("couldn't get semaphore\n");
   /* no need to free since we didn't get it */
}
```

Each of these two techniques has pluses and minuses:

- The tokens perform more quickly but are more restrictive in how they can be used.
- The semaphores are general purpose, but perform slightly worse.
- With poor application design, either technique can cause run-time deadlocks.

Recirculation

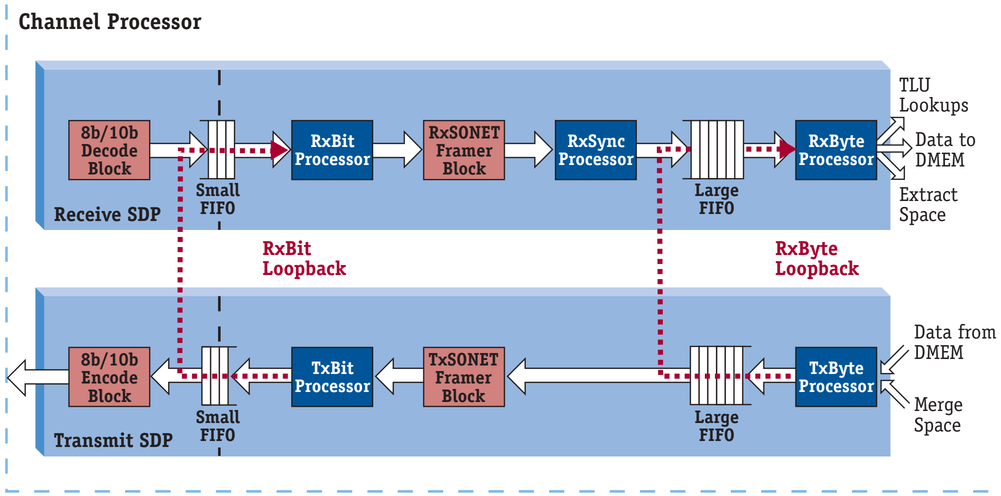
Certain applications for C-Port NPs have processing requirements that cannot always be met by a single Channel Processor (CP) because of computational complexity and complex reformatting of the data.

The NP provides a mechanism for using each of the CPs in a loopback mode called *recirculation*.

Recirculation allows users to send data through the transmit path of a CP (TxSDP) and, rather than sending it to the channel's physical interface, loop the data back to the receive path of the same CP (RxSDP). Enabling recirculation for an SDP means to configure its RxSDP and TxSDP so that the output from the TxSDP is routed to the input of its corresponding RxSDP.

Figure 1 on page 21 depicts the SDP configurations that support bit-oriented and byte-oriented recirculation data flows.

Figure 1 SDP Configuration and Data Flows for Bit-Oriented and Byte-Oriented Recirculations



The SDP can be configured to permit recirculation of the data path in either of two ways:

- Byte Processor loopback
- Bit Processor loopback

Byte Processor loopback changes the data path in the TxSDP to go from the base of the TxLargeFIFO to the bottom of the RxLargeFIFO, instead of going to the TxSONETFramer and TxBit. This is configured by calling the CPI function *ksSdpByteLoopbackConfigure()* routine. This function configures this data path by setting the CP's *SDP_MODE3* register.

Bit Processor loopback changes the data path in the TxSDP to go from the base of the TxSmallFIFO to the bottom of the RxSmallFIFO, instead of going to the PHY. This is configured by calling the CPI function *ksSdpBitLoopbackConfigure()* routine. This function configures this data path by setting the CP's *SDP_MODE3* register.

Benefits

Recirculation allows the application designer to tradeoff processing power for ports.

For applications where there are complex processing requirements, it is possible to commit some CPs to the task of post-processing packets received by other CPs that are directly connected to a physical network line. The cost is in reduced port density.

Scenarios

Enabling recirculation for a given CP's SDP can be useful in two distinct kinds of application scenarios.

In a first scenario, the CP that performs incoming packet demuxing on a given channel offloads the cell or packet hader processing to another, unused CP. This is useful when the former CP's role is to prepare the series of cells or frames from a multiplexed data stream for another CP whose SDP can perform header field extraction. In the latter CP's TxSDP, the TxByte serial processor would process the header fields out of DMEM in the conventional manner, but would forward its data traffic by recirculating via the corresponding RxSDP's RxByte processor whose output is routed back to DMEM for forwarding by the former CP.

Figure 2 on page 23 illustrates this recirculation scenario.

In a second scenario, recirculation between an SDP's TxBit and RxBit processors (or between its TxByte and RxByte) can support debugging output from the CP's transmit side without routing the data traffic through the external PHY, with its attendant requirements for traffic analysis.

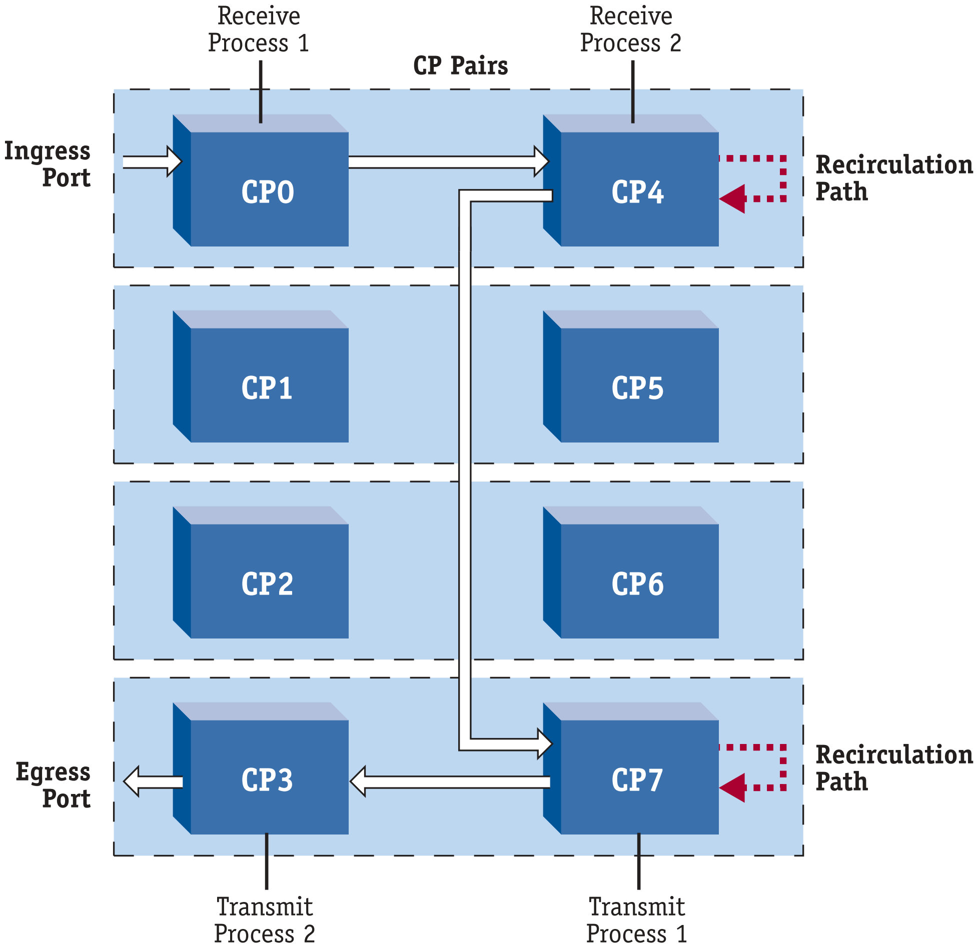
The NP's recirculation features and functions can be used for any application that requires greater degrees of packet processing, such as the C-Ware Reference Library applications that support multi-channel HDLC and segmentation.

Elasticity

An important property of the recirculation path through the SDP is that it provides a full chain of backpressure signalling through the CP. The RxByte serial processor stalls until the CPRC provides Extract Space resource. While RxByte stalls, the RxLarge FIFO block and then the TxLarge FIFO block will back up with data bytes. When they are full, TxByte will block on the next write to payload out. While it is blocked the TxDMA engine will also block. Also, the transmit side of the CPRC code will block because the Merge Space will not become available. As the transmit side of the CPRC stalls, the inbound queues will fill up with descriptors.

The key distinction between this scenario and a non-recirculation scenario is that in a non-recirculation scenario the network is supplying bits/bytes at a fixed rate, and there is no facility for backpressure. In a recirculated application the SDP's FIFOs provide some elasticity, and ultimately backpressure causes the transmit side to stall.

Figure 2 Recirculated Traffic Routed to an Unused Channel Processor



Eventually the inbound queues reach their limit and not more can be enqueued for processing. Before that hard limit is reached, the system has a large amount of elasticity. This is a benefit in that it provides the flexibility to apply precious cycle budget across multiple PDUs. However, the cost is in latency variation.

ation

Aggregation is a technique where multiple Channel Processors (CPs) in a NP can be configured to work together to process a single traffic stream. Aggregation can be broken down into the following individual techniques:

- Queue sharing
- Shared DMEM-resident data structures
- Shared IMEM resources

Queue Sharing

Multiple CPs in a cluster can share the same set of input and output queues using the queue sharing code found in the QueueManager library. Clearly this implies sharing of queues within the QMU. You may also want to consider using Serial Bandwidth schaling or sharing; refer to the “Purpose of the C-5e NP Channel Aggregate Mode” in your *Architecture Guide*.

Uses and Limitations

The queue sharing mechanisms ensure that the CPs access the queues in a coordinated fashion, either for the serialization or the scheduling of access to queues.

Serialization means that a sequence of descriptors must be processed in strict order. There may be multiple CP working to process pieces of work that are described by the descriptors in the queue. Under queue sharing the CPs take turns either enqueueing to or dequeueing from the queue. The CP typically use software tokens to ensure the ordering of their dequeue operations.

In this context, “scheduling” of queues means that the sending process draws packet descriptors from multiple egress queues and must use a scheduling algorithm to determine which queue to draw from on each packet transmit opportunity.

Using the queue sharing library, queues can be shared between two or four CPs in the same cluster. There is nothing preventing three CPs from sharing a queue, but the queue sharing library does not support this configuration as presently designed.

Costs and Benefits

The costs associated with the use of this technique:

- Increased application design complexity
- Increased processing overhead resulting from the use of queue sharing library code

Potential benefits of queue sharing include:

- Improved performance through the use of multiple parallel CP programs, all performing the same task
- Serialization of processing by multiple parallel tasks

Code Examples

The **apps\components\queueUtils\** directory contains the source code for a C-Ware application component that provides Serial Bandwidth scaling functionality.

Shared DMEM-Resident Data Structures

There are many application problems that can be addressed through the use of DMEM-resident data structures that are referenced and updated by all the CPs in a cluster, or even in different clusters.

The use of shared data structures by multiple parallel processors typically requires the use of some access arbitration mechanisms. The C-Ware CPI library provides token-passing and test-and-set mechanisms that can be used to effect such access arbitration. These mechanisms are described in more detail in the section “Shared DMEM” on page 26.

There are CPI routines that allow a CP program to construct the address of a location in another CP’s DMEM, given the location of the same data structure in the calling CP’s local DMEM.

Costs and Benefits

All four CPs within a cluster can access DMEM through a local bus with a latency that is generally 2 to 5 cycles, but in extreme cases can be 9 or 13 cycles. A CP can access the DMEM in another cluster through the NP’s Global Bus with a latency of from 10 to 110 cycles, depending on the Global Bus’s load.

Code Examples

The queue sharing library source code provides an example of a DMEM-resident data structure that is shared within a cluster.

For another example, the ATM receive port in the Gigabit Ethernet to ATM OC-12c SAR Switch application maintains a shared cache of Virtual Circuit information that is referenced and updated by all the CPs in the cluster.

DMEM

Because of the shared-memory architecture of the NP, there is visibility into all the local data memories (DMEM) on the chip (XP and CP). This access may be done either on an efficient local bus (like within a cluster of CPs sharing DMEM) or over the NP's Global Bus (when sharing DMA through the between BMU [Buffer Management Unit]).

Shared DMEM is typically used for implementing software tokens using software semaphores.

Software Tokens

Software tokens are used by application programs running in the CPRCs that have one writer at one time and multiple readers. Because of the fact that there is only writer at one time, the data does not need to be protected. An example of where software tokens might be used is in aggregation of CPs within a cluster.

With aggregation, one CP within its cluster is typically waiting for its turn in line to access a particular resource. Because it won't use that resource unless it has the software token, it only writes when it owns the token. A code example of how this would work follows:

Program 1 -- Waiting For Token

```
// I am waiting for the token.
while (!kstokenPresent(myToken)) {
   doSomethingElse();
}
// I now have the token -- proceed.
doMyTokenOperation();
```

Program 2 -- Owns Token

```
// I have token.
doMyTokenOperationWhileHavingTheToken();

// Done with the token - I'll give it up.
ksTokenPass(myToken);

// Ensure that what the program does next doesn't require the token.
...
```

Software Semaphores

Software semaphores are another way for implementing a mutual exclusion disicpline for a program that doesn't follow the same serial access rules as a program using software tokens.

Programs try to ‘obtain’ a software semaphore when it wants to access a data structure. This is implemented by the RISC core by executing a test-and-set instruction that atomically reads the data and writes a pattern into it if no other program is using it. Programs that must access a shared data structure would first obtain the semaphore, then modify the shared data structure, then ‘release’ the semaphore.

An example of this follows:

```
// I need to access dataStructure.

while (!ksMutexLock(&semaphore)) {
   doSomethingElse;
}

// I have obtained the semaphore.

updateSharedMemory();

// Finished updating the shared data structure; release the semaphore.

ksMutexUnlock(&semaphore);
```

IMEM Optimization

To fit more functionality into the limited IMEM space available on the C-5 family of chips you can take some of the following actions to reduce the amount of IMEM a program uses.

Not all methods are appropriate in all circumstances. Some would not be acceptable in a production/commercial configuration, but may be useful for debugging and bring-up testing.

As a prerequisite, you should be familiar with the C-5 Architecture and the CST documentation set.

1 Generate and look at the **memUsage.txt** file for the application, to get an overview of the IMEM usage. In the reference applications, this file is generally found in the *application_name***\run\bin\variant\memUsage.txt** file. Specifically, an example is in

enetOc3Switch\run\bin\c5-d0-sim-debug\memUsage.txt

To see how this file is generated, look in the **bin\cport-apps-rules.mk** file for "memUsage". Notice how a perl script is invoked on a .map file to generate the **memUsage.txt** file.

2 Review the use of INLINE functions. Although INLINE functions save cycles by avoiding stack frame push/pop instructions, they use up more IMEM than the normal method of calling a subroutine. Note that for very small functions, an INLINE is both smaller and faster (that is, when the function length is small compared to the push/pop/return overhead).

3 Whenever possible, move non-forwarding path processing off the CPs to the XP, and further off the XP to the host. This reduces the IMEM required on-chip, in favor of host IMEM, which is abundant.

4 Use the init/main mechanism to conserve CP IMEM space by moving initialization and configuring operations to an "Init" program phase. See Chapter 4, “Building and Packaging An Application”. See also the “Kernel Services” chapter of the *C-Ware API User’s Guide*.

5 Use the following linker switches to get a map output, even if the link fails due to IMEM size. This will help in determining which functions are being loaded into memory and the size of each function.

```
#
#  Linker map, debug, and tracing options
#
make EXTRA_CFLAGS='--verbose -Wl,--verbose -Wl,--trace
-Wl,--print-map'
```

In some cases, depending on how the application Makefile is written, you use the following addition to the Makefile:

```
LDFLAGS_enetOc3Switch = --verbose -Wl,--verbose -Wl,--trace
-Wl,--print-map
```

6 Consider using a macro which will cause all *ksPrintf()* and *ksPanic()* calls to go away:

```
#define ksPrintf(a,...)
#define ksPanic(msg)
```

7 Use the following bash shell script (for example, on UNIX or NT-Cygwin) to find functions linked in but not called. To run it, enter **ispace** *filename***.map**.

```
#! /bin/sh
# ispace -- find unused functions in imem
#
rm -f xxcalls xxdefs
# func calls
grep jal $1.map | awk '{FS="<|>"; print $2}' | sort | uniq > xxcalls

# func definitions
grep "O .text" $1.map | awk '{print $6}' | sort > xxdefs

# show funcs defined but not called
comm -23 xxdefs xxcalls
```

8 Compile Switches. Using the -g switch for gdb does *not* increase IMEM usage. You can save some IMEM by compiling without defining the DCP_APPLICATION_EVENTS macro.

Diagnosing Memory Problems

Occasionally a given .dcp executable cannot be built and the linker complains of a memory shortage (usually IMEM). For example:

```
/vobs/sw/ssbin/Gnu/mips-cport-elf/bin/ld: region IMEM is full
(bin/c5-d0-sim-debug/xpCodeSep1InitXp.dcp section .text)
collect2: ld returned 1 exit status
```

This can be caused by the linker bringing in unwanted libraries due to finding the wrong instance of a symbol in the code. The following procedure will help identify symbols that may be referencing the wrong library.

1 Rebuild Your Executable Passing 2 Extra ld Flags

```
make EXTRA_LDFLAGS="-Wl,-T
/vobs/sw/ssbin/Gnu/mips-cport-elf/lib/ldscripts/rc-large -Wl,-Map foo"
```

The first flag tells the linker to use a special linker script (**rc-large**) which has all memories set to their maximum levels (128 kbytes for IMEM, 64 kbytes for DMEMs).

The second flag tells the linker to create a linker map file called **foo**. This map file identifies what object file was pulled from what library as an effect of a reference from what object file. Thus, it is possible to answer the question, “Why is **foo.o** from **bar.a** being linked in?”

2 Generate a Report

After you have an executable built you can get a quick look at all functions present with the following command:

```
cport-objdump --syms foo.dcp | grep ’\.text’ | sort
```

The output produced will look something like this :

```
00000000 g     O .text  00000000 _ftext
00000000 l    d  .text  00000000
00000008 g     O .text  000138f4 DCPmain
000138fc g     O .text  000000c0 bsInitialize
000139bc l       .text  00000190 bsPoolAllocateInternal
00013b4c g     O .text  00000180 ksProcLoadXp
00013ccc g     O .text  00000020 ksProcLoadMain
00013cec g     O .text  000000e4 _waitForDmemTransfer
...
```

Note that the compiler has been told to emit size information for all functions (column 5 above - for example, DCPmain is 0x138f4 bytes long).

3 Trace Dependencies

Using this output in combination with the linker map file you can follow all sorts of dependencies. For example:

a *bsInitialize()* is referred to in **xpCodeSep1InitXp.o** and that causes linking in of **bsInit.o**, as evidenced by this line of linker map file :

```
../../../../../../services/lib/c5-d0-sim-debug/xprc/
                  services.a(bsInit.o)
obj/c5-d0-sim-debug/xprc/xpCodeSep1InitXp.o (bsInitialize)
```

b *bsInitializeInternal()* is defined (and used) in **bsInit.o** by the virtue of including **bsMachDep.h**. *bsInitializeInternal()* calls *bsPoolAllocateInternal()* (defined in the same header), which references _dcpRevRevision - that causes linking in of init.o :

```
../../../../../../services/lib/c5-d0-sim-debug/xprc/services.a(init.o
)
../../../../../../services/lib/c5-d0-sim-debug/xprc/
                  services.a(bsInit.o) (_dcpRevRevision)
```

e Design and Table ding

Applications that use C-Port NPs have specific requirements for the routing/switching table lookups that must be performed. These requirements are typically one of the following:

- Indexed storage
- Exact matches
- Variable prefix matches

For additional details on table-building refer to Appendix A, “Offline Table Building Libraries”.

Indexed Storage

This type of table is general data storage that can be thought of as a large array of data. The records can be accessed by the ‘index’ of the array. This index is usually a small number, expressable in 16 bits or so, because the number of records in this table are a direct function of the size of this index. For example, a 16bit index makes possible indexed access to a table of 64K entries.

A practical example of this type of table would be a IEEE 802.1Q VLAN ID table, which is a 12bit quantity that is referenced directly by the VLAN ID as the index.

The NP implements this type of table as a TLU ‘data’ table. The data table’s entries are referenced directly by the index number, and it consumes one TLU table. Because the maximum addressability of TLU tables is 1M entries, the maximum key size is 20bits.

Exact Matches

This type of table is used for exactly matching data as well, but the difference from a data table is that the size of the key is larger. Exact match tables in the NP can be up to 112bits in length.

Overview

In most system architectures, this would be a much larger table that is sparsely populated. How this type of lookup works is to take the key, run it through some type of hash function and generate a smaller index that could be used to reference entries in another table — much like an indexed table.

A practical example of what type of lookup this would be is an Ethernet MAC address or a IPv4 flow. These are larger keys (48bits and 112bits, respectively) that can be reduced to a small domain that can point to entries in a smaller table.

In the C-5 NP's TLU, this type of function is implemented by chaining three tables together: a hash table, a trie table, and a key table, as follows:

- Hash table —The hash table is used to take the key of up to 112bits and have it point to another entry in another table of a smaller domain size. For example, a 48bit key is run through a hash function that produces an index that is an entry in the hash table. That entry will point to one of: an entry in a trie table or an entry in a key table.
- Trie table — The trie table is used to resolve any collisions by the hash table. Collisions can occur if two different keys hash to the same location in the hash table. The trie table is used to differentiate the two keys one bit at a time from each other since they hashed to the same entry in the hash table. The leaf trie entries point to an entry, where the key and associated data are kept, called a key table.
- Key table — The key table is used to store both the key for the entry and its associated data. The key is kept in the entry to do a full comparison with the one that it being looked up by the TLU. If there is a match the TLU will return the associated data part of the entry in the same record. If there is no match (which can happen if non-matching entries point to the same entry in the hash table), the TLU will indicate that in its response.

As for C-5e NP's table lookups, the functionality has been greatly expanded yet simplified. For details, refer to "Supported Table Types" in the *C-5e/C-3e Network Processor Architecture Guide*.

Design Tradeoffs and Performance

The chaining together of these types of tables by the TLU allows the system designer to choose among speed versus space tradeoffs during system architecture and design.

The performance of these types of tables is equated with the number of entries in all tables that have to be visited to get to the associated data. The biggest impact to that metric here is the number of collisions — which is the number of trie table entries that must be referenced. The number of trie table entries is really a function of how good (or how bad) the hash function is when it is hashing the key and how many entries the hash table contains.

The hash table is thought to be a large, sparsely populated table. The larger the table, the fewer collisions and fewer references to entries in the trie table. Typically, the hash table is some number of times bigger than the key table, where the associated data is kept. A hash table that is 2X the number of data entries will perform worse than a hash table with

4X the number of entries. Obviously, the table size will increase with a hash table 2X the size of the key table.

The number of entries in a trie table is based on the number of collisions that are anticipated to happen. This will be slightly more in the 2X case than in the 4X case, but probably linear.

As an example, assume two hash-trie-key tables that support up to 64K entries. Table 1 shows the memory consumption and approximate performance of each.

Table 1 Resource Usage of Hash-Trie-Key Tables With Different Hash Table Sizes

HASH-TRIE-KEY TABLE SIZE OR PERFORMANCE METRIC	HASH TABLE SIZE	
	2X KEY TABLE SIZE	**4X KEY TABLE SIZE**
Storage of hash table entries	1M (that is, 128K * 8Bytes)	2M (that is, 256K * 8Bytes)
Storage of trie table entries	64K (that is, 8K * 8B)	32K (that is, 4K * 8Bytes)
Storage of key table entries	1M (that is, 64K * 16Bytes)	1M (that is, 64K * 16Bytes)
Total table storage	> 2M	> 3M
Performance estimate of lookups*	N SRAM references	N-2 SRAM references

* See the *C-53/3e Network Processor Architecture Guide* for a complete analysis.

Variable Prefix Matches

The other primar y type of table that is required by certain applications are called variable (or longest) prefix match tables. This type of table is specifically for protocols that find the best match for lookups. This is the default behavior for IPv4 address lookups in systems that support IP.

Some systems require that this type of table (IP routing table) have a wide range of entries. Some systems are as little as 1K and some are can be up to 1M entries. The C-5 provides a flexible manner in which system parameters may be dialed to make the speed versus space tradeoff for these types of tables.

The following information is specific to the C-5 NP. (For the C-5e, TLU lookup is so improved that use of 8-bit and 16-bit index tables is obsolete.) Variable prefix tables are constructed in the TLU by chaining together one of the following series of tables:

- VP Trie - Data

- 8bit Index - VP Trie - Data
- 16bit Index - VP Trie - Data

VP Trie - Data Table Chain

This type of table chain has the VP Trie table implement the variable prefix and keeps track of the best match. The data table contains just the associated data. This is different from hash-trie-key because the VP Trie table resolves the bits of the key, and it isn't needed to be stored in the data table entry.

The advantage of this type of table is that it is compact and only consumes two tables. The disadvantage is that it can contain more table entry references the more keys that are installed in it.

8-Bit Index - VP Trie - Data

This type of table extends above the concept with an optimization. The first eight bits of the lookup key are evaluated as a direct index into a 256-entry index table. That index table's entries point into the VP Trie table, which points to the associated data in the data table.

This type of table is a compromise between speed and space, because it improves the number of memory references, but adds a new 256-entry table that the TLU must allocate.

16-Bit Index - VP Trie - Data

This final table configuration for doing best matches is the most efficient of the three, but consumes the most table space. Like the 8-bit index table, the first 16 bits of the lookup key are evaluated as a direct index into a 64K entry index table. That index table's entries point into the VP Trie table that points to the associated data in the data table.

Design Tradeoffs and Performance

The performance and memory consumption of these three different table types vary greatly. Table 2 shows the differences between the three.

Table 2 Resource Usage of Table Chain Types With Different Lookup Index Sizes

	TABLE CHAIN TYPE*		
SIZE OR PERFORMANCE METRIC	**NO INDEX**	**8-BIT INDEX**	**16-BIT INDEX**
Index table entry storage	0	2K (that is, 256 * 8Bytes)	524K (that is, 64K * 8Bytes)
VP Trie table entries storage	1.5M (that is, 64K * 24Bytes)	1.5M (that is, 64K * 24Bytes)	1.5M (that is, 64K * 24Bytes)
Data table entries storage	0.5M (that is, 64K * 8Bytes)	0.5M (that is, 64K * 8Bytes)	0.5M (that is, 64K * 8Bytes)
Total storage	2M	> 2M	> 2M
Estimated performance (min/typical/max) in SRAM references	2/11/33	3/16/26	3/12/18

* The examples below show the three table types with 64K of associated data.

Partitioning An XPRC Program For Initialization

For an Executive Processor RISC core (XPRC) program to use available IMEM resource to its fullest, the C-Ware Software Toolset (CST) supports placing the program's initialization code in a separate executable (called the *init phase* executable) and supports a mechanism for allowing that executable (called the *main phase* executable) to load another executable that performs the XPRC's primary processing.

When program and processor configuration and initialization are complete, the primary processing executable is loaded on top of the initialization executable. Thus, the primary executable can contain only the code used for post-initialization processing.

Thus, you can design your XPRC's primary program to exclude the code for some of the larger C-Ware API services, such as:

- The code to load the Channel Processors and Fabric Processor from the package
- The code to allocate buffer pools, to assign them to Channel Processors, and to initialize their BTags
- The code to initialize the queues and allocate them to the various processors

Remember that the XPRC initialization feature is intended to make more IMEM available for the XPRC's predominant runtime processing. It is not intended to be a general checkpoint/restore mechanism or an overlay scheme that would allow a program to progress to an arbitrary point, then stop and save the NP's entire state before loading the next executable and continuing processing.

Partitioning the Program

Most larger XPRC programs have a natural partitioning between code that is run once as the NP boots and the overall NP application starts, and code that runs to support the forwarding application for as long as the forwarding processor is running.

Among the initialization and one-time activities you would want to implement in an init phase executable:

- Calls to the various C-Ware API initialization routines (that is, *ksInitialize()*, *bsInitialize()*, *qsInitialize()*, and so on)
- Calls to configure the system resources (that is, *bsPoolAllocate()*, *bsPoolInitialize()*, *qsQueueCreate()*, and so on)
- Calls to load the NP's embedded processors (that is, *ksProcLoad()*).

It is important to note that the initialization code should *not* start the processors (that is, should not call *ksProcStart()*).

Coding the Init Phase Program

The init phase program is simply an XPRC program that ends with a call to the API routine *ksProcLoadXp()*, which overlays the XP's IMEM and DMEM with the code and corresponding data images for either another XPRC init phase executable or the XPRC main phase executable.

To preserve the in-memory data structures created by the initialization routines, these services arrange for their data areas to be written to SDRAM by *ksProcLoadXp()* before the main program is loaded. Be aware that *ksProcLoadXp()* does not return to its caller.

Coding the Main Phase Program

The main phase program comprises the remainder of the application. Before execution starts at the application's *DCPmain()*, the system reloads all preserved data that is used in both the initialization code and in the main program. The main program must call *ksInitialize()*; otherwise, it need not call any C-Ware API service initialization routine that was called in the initialization program.

zing Multiple Contexts n Main Phase Program

If the XPRC's main phase program (that is, the remainder of the application) uses multiple CPU contexts (by calling *ksContextCreate()*), the contexts must be created *in the main phase program* rather than in any init phase program. Be aware that the call to *ksProcLoadXp()* resets all contexts and stacks.

Allocating Queues

The XPRC init phase program should allocate queues if it needs them, but should not use those queues until the main phase program is running. This is because loading the main phase program resets the status of any pending messages on the queues.

Accessing the TLU

Though the init phase program code may use the TLU, that code should ensure that the TLU is quiescent before calling *ksProcLoadMain()*.

Coding the Package Description File

After you have created an application with both init phase and main phase programs, you build them into a package by including lines like the following in the package description file:

```
XPINIT "filename" "optional description";
XP "filename" "optional description";
```

For a NP application that does not have a separate init phase program, the package description file should contain only the 'XP' statement.

See Chapter 4 for a description of how to code the entire package description file.

User Data From se Program to the Main Phase Program

The XPRC init phase program can create user-defined data structures and pass them to the main phase program. To do so, user data must be placed in one or more structures that are aligned on a 64Byte boundary by using the 'ALIGNED64' (or 'ALIGNED128') macro. Any data that, in fact, must be aligned should of course appear at the beginning of the structure. For an example, see the structure 'BufSvcsData' in this file:

services\kernel\chip\np\xprc\inc\kernelSvcsData.h.

Typically, the declaration of the user data to be passed to the main phase program appears in a header file with the 'extern' keyword. Any module including that file can then refer to the data in the structure. For example:

```
typedef struct {
/* Global variable that keeps track of initialized services. */
   int32    serviceInit;
   pkgInfo PKGinfo;

/* Number of buffers allocated in pool 29 */
   int32u  pool29Buffers;
} KernelSvcsData;

extern KernelSvcsData ALIGNED64 kernelSvcsData;
```

In both the init phase and main phase programs, there must be exactly one module that defines the data structure. If there is a common source module that is linked into both the initialization and main programs, it can contain the definition. Otherwise, the definition must appear once in an init phase-only source module, and once again in a main phase-only source module. The linker issues an error message if this rule is violated.

The definition in the main phase-only source module must repeat the declaration, but without the 'extern' keyword. It must also include a use of the KS_INIT_DATA macro so that the data will be preserved from one program to the next. (This macro is defined by including **dcpKernelSvcs.h** in your program.) Thus, to continue our example, the definition in the main phase-only source module contains:

```
UserData ALIGNED64 userData;
KS_INIT_DATA(data_identifier, &userData, sizeof(userData),
   windDownRtn);
```

The *data_identifier* is a small number between 32 and 63, and must be different for each user data structure defined in this way. (The values 0 to 31 are reserved for use by the

CST's system services.) Thus, there can thus be up to 32 separate data areas passed, though for efficiency, fewer is better.

The *windDownRtn* parameter is either zero (0) or the name of a function taking no arguments and returning void. If present, this function will be called before the data is written. It can ensure that any asynchronous operations are complete, and that their results are reflected in the userData area.

Address Resolution Among XPRC, CPRC, and XP Primary Bootstrap

The definitions in the file:

services\chip\inc\dcpImportExport.h

allow the programmer a limited form of address resolution at package build time among the XPRC program(s), the CPRC programs, and the XP primary bootstrap.

You can define a value (usually the address of a variable) in one module and specify that it be "exported." You can "import" that value into another module. These "export" and "import" statements (defined as C language macros) add information to the ELF file that is used by the package build tool to resolve the imports.

To use this feature, add this statement in your program:

```
#include <dcpImportExport.h>
```

For additional information see the comments in **dcpImportExport.h**.

For example, assume that you have a host application program that must write directly to a specific location in a particular Channel Processor DMEM region. To export an address within that DMEM for use by the host program, include code such as the following in the CPRC program:

```
/* Internal IP Address */
int32u ipInternalAddr;
#ifdef HOST_TEST
DCPEXPORT(ipInternalAddr);
#endif
```

The 'DCPEXPORT' macro causes this symbol and its CP DMEM address to be included in a reserved section of any package that uses this CPRC program.

To import the address within that DMEM region into the host program, include code such as the following in the host program:

```
   char * symName = "ipInternalAddr";
...
   status=hsSymbolImport(ipStack->dcpHandle,symName,(void**)&valPtr);
```

This makes the address in the DMEM region, represented by the exported symbol 'ipInternalAddr', available to the host program.

Exporting a Value

To export a value, use the DCPEXPORT macro or DCPEXPORT_VALUE macro in the code. These statements must appear at the top level of the code, outside any function scope.

DCPEXPORT Macro

```
   DCPEXPORT(var_name)
```

The *var_name* argument is the name of an external variable or procedure. Its address can be imported into other programs as *var_name*.

It is usual that *var_name* be defined in the current compilation unit, but it is possible to export a name declared here but defined elsewhere in the program being linked.

It is valid for multiple programs in a package to export the same *var_name*, but **only** if the value is the same in all of the exports. Similarly, the same *var_name* can be exported multiple places in a single program, but only if the same value is exported everywhere.

DCPEXPORT_VALUE Macro

```
   DCPEXPORT_VALUE(value, name)
```

The *value* argument is an expression to be exported that must be constant at compile time. The value of the expression *value* is available to be imported elsewhere using the name *name*.

Importing a Value

To import a value, use the DCPIMPORT macro or DCPIMPORT_WEAK macro in the code. These statements must appear at the top level of the code, outside any function scope.

DCPIMPORT Macro

```
DCPIMPORT(import_name, name)
```

This declares a variable named *import_name* of type 'void*'. At package build time, the value of *name* exported from some other program is stored in *import_name*.

If the other program exported *name* using the 'DCPEXPORT(*name*)' macro, the void* *import_name* contains a pointer to the variable in the other program.

If a CPRC program is exporting a variable's address, the imported pointer value contains the address of the variable in CP0. The *importing program* must explicitly relocate it to point to some other CP's memory if necessary.

If the other program exported *name* using the 'DCPEXPORT_VALUE(*value*, *name*)' macro, the void* *import_name* contains the value from the other program. In this case it may be necessary to cast the void* to some other type.

DCPIMPORT_WEAK Macro

```
DCPIMPORT_WEAK(import_name, name)
```

Same as the DCPIMPORT macro, but *import_name* contains 0 if *name* is not exported anywhere.

PI Routines to Call Initialization Phase Program

The C-Ware API routines are documented in the *C-Ware API User Guide* document.

For more information about restrictions on the functionality for partitioning the XPRC program, see the *C-Ware Software Toolset Release Notes* document for this C-Ware Software Toolset version.

Only the following C-Ware API routines can be called within an init phase executable.

Kernel Services

ksInitialize() — Must be called in each init or main executable, but not more than once in each executable

ksLssiRead()
ksLssiReadComplete()
ksLssiWrite()
ksLssiWriteComplete()
ksMdioRead()
ksMdioReadComplete()
ksMdioWrite()
ksMdioWriteComplete()
ksPanic()
ksPrintf()
ksProcIdCreate()
ksProcLoad()
ksProcLoadXp()
ksSerialBusConfigLssi()
ksSerialBusConfigMdio()

Buffer Services

bsInitialize() —Must be called no more than once across all init/main executables

bsError()
bsPoolAllocate()

Queue Services

qsInitialize() — Must be called no more than once across all init/main executables

qsEnable()
qsQueueConfig()
qsQueueCreate()
qsQueueLevelSet()
qsQueuePool()

Protocol Services

psInitialize() — Must be called no more than once across all init/main executables

All other routines can be called from the init executable.

PDU Services

pduInitialize() — Must be called no more than once across all init/main executables

All other routines can be called from the init executable.

Fabric Services

fsInitialize() — Must be called no more than once across all init/main executables

All other routines can be called from the init executable.

Table Services

InitializeTableServices() — Must be called no more than once across all init/main executables

and/or

CreateTable()

Tradeoffs

There are certain features that can be implemented in any of various components of a C-Port NP-based system.

Some of these features are in the control path, and it is desirable that they execute quickly. But such features also cost memory and cycle time when implemented on the NP. These types of features might present a classic speed/space tradeoff by being implemented on the NP's XPRC versus the host processor.

Examples of protocol features that fit into this category are:

- Bridge Address Table management
- Bridge Learning and Aging
- Ethernet autonegotiation

These types of features can be implemented in a XPRC program that can run quickly (almost at the line interface speed); however, they cost the system in instruction and data memory.

Some designers of NP-based systems choose to implement, for example, the Bridge Address Table management in the XP to allow for quick lookup access times. This is because the application can learn a large number of bridge addresses quickly. The down side of implementing this function on the XP is that it consumes instruction memory and cycles that could otherwise be used by other features of the system running on the XP.

Chapter 3

APPLICATION DIRECTORY STRUCTURE AND BUILD SYSTEM CONVENTIONS

Overview

The C-Ware Software Toolset (CST) is an easily-accessible environment for implementing C code, as soon as the user learns how to employ its software directory structure and build system.

This chapter provides the necessary information for mastering use of both CST directory structure and of the build system (**make**) utility:

Because the C-5e and C-3e Network Processors have very similar architectures, there is considerable sharing of code between builds for these devices, even where the same code would not be shared with a build for the C-5 Network Processor. Directories named "cxe" are used to contain this code.

Table 3 provides definitions of terms used in this document and their descriptions.

Table 3 Build System Terminology

TERM	DEFINITION
Architecture	The architecture of the chip specified. Today, this is just "np" (network processor)
Model	The chip being specified. Today this is "C-5", "C-5e", or "C-3e"
Processor	The processor on a given chip. For example, "cprc", "xprc", "sdp", and so on.
Revision	The revision of a given chip. For example, "a1", "b0"
Environment	The environment that the given software runs on. For example, software simulation ("sim"), hardware ("hw")
Configuration	The configuration of the build ("debug" or "release") that controls the usability and optimization level of the code
Variant	Combination of the components (that is, the architecture, model, processor, revision, environment, and configuration) that make up a build target. A variant is specified by the settings of a related set of environment variables provided by the C-Ware Software Toolset.

Application Directory Structure

Applications that run under the environment provided by the CST are required to support multiple chip versions (C-5, C-5e, others), different revision levels (A1, B0, and so on) for each chip version, and different run-time environments (software simulation, hardware).

The directory structure accommodates software that is:

- Common between the CPRC and XPRC and any chip or chip revision
- Processor dependent (that is, specific to CPRC, XPRC, SDP, FDP, or host)
- Processor and chip dependent (for example, specific to CPRC on C-5e)
- Processor, chip and rev dependent (for example, specific to CP on C-5 Version D0)

There are a set of rules that allow a flexible implementation and interpretation of the structure of the directories in the build tree.

The CST uses this directory structure for both its C-Ware Reference Library *applications* (found in subdirectories under the CST's **apps** directory) and *application components* (found in subdirectories under the CST's **apps\components\** directory).

Structure Requirements for File and Directory Names

The build system has a certain set of requirements of the directory structure in order for it to work properly.

The build system consists of a set of Makefiles that do the following:

- Automatically generate **obj**, **lib**, and **bin** subdirectories for a certain *variant*. The variant is defined by the settings of a set of related environment variables provided by the CST.
- Automatically generate **deps** directories for dependency information at build time.

To accomplish this, the build system requires that the following rules are followed:

1. If the variant to be built contains a directory named **chip** every subdirectory below **chip** is of a well-known name, so that the build system knows how and when to build it.
2. The only valid leaf directories that contain sources and headers involved in the build are to be named **src** and **inc**.
3. The **src** directory should contain only source files (***.c**, ***.cpp**, ***.S**).
4. The **inc** directory should contain only header files (***.h**).
5. There should be no empty directories. For instance, if the **src** or **inc** directory exists, it should contain at least one source or include file (respectively). This is not an absolute requirement but will slow down builds, as empty directories may be checked for building.
6. All paths in Makefiles should be relative.

Directory Keywords

There are a number of keywords used in the directory trees to help control the build and to provide context for the code included in them. As a general rule, there can be an **inc** or **src** directory at any level.

- Top Level:
  - **chip**, **host**, and **offline**

- Architectures:
  - **np** and **offline**
- Processors:
  - **xprc**, **cprc**, **sdp**, **fdp** (found only under **chip**)
  - **ppcVxworks** and **i86Linux** (found only under **host**)
- Chips:
  - **c5** (for C-5), **c5e** (for C-5e), **c3e** (for C-3e), **cxe** (for C-5e and C-3e)
- Revisions:
  - **a1**, **b0**, **d0**
- Environment
  - **sim**, **hw**

This list will be added to over the course of time as Motorola introduces new products in the C-Port family.

TAGS Support

The build system also supports the TAGS feature that many editors (such as GNU emacs) use.

The TAGS feature allows users to highlight variables, structures, and so on and have a file that contains the definition of that item automatically loaded. For software development activities, this is a very handy feature.

Once again, no special support is required in individual Makefiles to support this feature. Users need only make sure that they include the CST file **bin\cport-rules.mk** make include file in Makefiles.

To use this feature, first build the TAGS files as follows:

```
D:\C-Port\CST2.2\apps\gbeSwitch\run> make tags
```

This step will build a TAGS file in the **bin\***Variant***\\$(CPORT_PROC)\** directory with the name $**(APPNAME).tags** for both the cprc and xprc.

In editors that support this feature, this is the file that should be loaded when using the feature.

The TAGS files are not automatically built by the build system.

Makefiles

The application Makefile will include several standard targets. These targets are necessary so that a parent Makefile will be able to invoke a child Makefile in a consistent manner. Table 4 summarizes these targets.

Table 4 Standard Makefile Targets

TARGET NAME	DESCRIPTION
default	First target in the Makefile. Has a single dependency of “all” and no rules. Invoking make on the command line with no targets will build the target “all.”
all	Builds all necessary code images. (e.g. the *.pkg file)
patterns	Generates all input pattern files used for regression testing.
test	Runs the regression test.
accept	Builds the code image, generates the patterns, and runs the regression test. Equivalent to “make all patterns accept.”
clean	Removes all files created by the make process from all of the standard targets.

Also note that Makefiles must support GNU emacs TAGS.

Here is a portion of an example Makefile that illustrates the use of the standard targets:

```
default:       all

all:           $(TGT)/widget.pkg

patterns:   $(INPAT_DIR)/widgetIn.pat
              cd $(INPAT_DIR) && $(MAKE)

test:
              $(DCPSIM) -batch
               .
               .
               .

accept:     all patterns test
```

```
clean:
              rm -r $(TGT)
              rm -f $(INPAT_DIR)/*.pat
              .
              .
              .
```

Directory Example for Applications

The following is one example of a directory structure that the build system can accommodate.

Table 5 provides a hierarchical description of the directory tree for one of the sample applications found in the apps directory of the CST. While the table illustrates the directory tree for only one application (**gbeSwitch**), another application will employ the same subdirectory structure layout, although it might not use all those those listed in the table.

As you look at your CST installation and compare its contents to that in the following table, please notice that one directory below apps is <u>not</u> an application name. Its name, **components** *suggests what it holds — subdirectories each named for a service component provided to applications needing that service (examples are ip, sonet, atm, phy).*

Notice that the directory hierarchy next described for applications also applies to **components**. *For this reason, after* Table 5, *the section "*File Placement Guidelines*" describes all top-level filenames as "app_or_component".*

5 Directory Structure Naming Hierarchy for Applications

TORY NAME AND HIERARCHY POSITION	DESCRIPTION (EXAMPLE)
application	The top-level module directory positioned below the **apps** directory in the CST directory structure. The actual name for *application* (forthis example, it is **gbeSwitch**) usually suggests its functionality.
run*	Holds simulation files, which are automatically generated by the CST build system, creating target-dependent subdirectories (like **bin, obj**, **lib**, and **deps**) for such files. The gbeSwitch subdirectories are not listed here to maintain readability. See "Variants" on page 56 for details about the contents of these subdirectories.
host†	Holds files for a host application. Subdirectories for gbeSwitch are not listed here for simplicity, but note that this directory would hold an **np** directory with subdirectory for whatever OS was being used on the host (for example, **Linux** and/or **ppcVxworks)**
offline	Holds offline table-building code
doc	Holds documentation for the application (README, app design spec)
chip	Always so named for modules related to NP
np	Always so named, may refer to C-5, C-5e, or C-3
sdp	Always so named, holding subdirectories for functionality specific to SDP
src	Holds source files for SDPs
inc	Holds include files for SDPs
cprc	Always so named, holding subdirectories for functionality specific to CPRC
c5	Always so named, but only present for applications built to use the c5
src	Holds source files for the c5
inc	Holds include files for the c5
cxe	Always so named, but only present for applications built to use eitherthe c5e or the c3e
src	Holds source files for either c5e or c3e
inc	Holds include files for the c5e or c3e
xprc	Always so named, holding subdirectories for functionality specific to XPRC
c5	Always so named, but only present for applications built to use the c5
src	Holds source files for the c5
inc	Holds include files for the c5
cxe	Always so named, but only present for applications built to use eitherthe c5e or the c3e
src	Holds source files for c5e or c3e
inc	Holds include files for c5e or c3e

*Most files in subdirectories under **run** are automatically generated by the CST build system, dependent on values you feed to it in order to describe the particular target for this current "variant". See "Object File Generation" on page 53 for details on variants.

†This directory may also contain an **inc** and **src** subdirectory, depending upon the application. Files inserted at this upper level would be for code that has no direct dependencies on the NP "chip".

It is important that any body of software limit the amount of environment-dependent code (version to version) that is different for a given application, so this structure works well.

File Placement Guidelines

To benefit from this organization, it is important that the files are all placed properly.

Here are guidelines to assist you in determining where source files should be placed in a directory tree that the build system can support:

- `app_or_component/run` — This optional directory is used by some software (applications) that contain the high-level **Makefile**, simulation files, and so on. This is the directory where your both build an application and run simulations of it, if a "sim" variant is present.
- `app_or_component/inc` — This include directory should contain header files that are architecture, model, processor, version and environment independent. Header files must match ***.h**. An example would be a header file that contained definitions that ran on any architecture, any version, any processor and any environment. Otherwise, the header file must be placed at a lower level **inc** directory.
- `app_or_component/doc` — This optional directory should contain documentation for the given component. This is primarily for applications.
- `app_or_component/chip` — This directory must be present so that the build system knows where to start conditionally determining which files to build. This directory should contain only subdirectories.
- `app_or_component/chip/np` — This directory (reserved keyword) contains subdirectories that contain files that are only relevant to a network processor architecture.
- `app_or_component/chip/np/<cprc|xprc|sdp|fdp>/<src|inc>` — If present, these directories should contain at least one file (source or header) that is specific for the processor that is specified (that is, cprc, xprc, sdp, fdp).
- `app_or_component/chip/np/<cprc|xprc|sdp|fdp>/<c5|cxe>/<src|inc>` — If present, these directories should contain at least one file (source or header) that is specific for the chip model(s) and processor(s) that is specified.

- *app_or_component*/chip/np/<cprc|xprc|sdp|fdp>/<c5|cxe>/<a1|b0|d0>/<src|inc> — If present, these directories should contain at least one file (source or header) that is specific for the chip model(s), processor(s), and processor revision that is specified.
- *app_or_component*/chip/np/<cprc|xprc|sdp|fdp>/<c5|cxe>/<a1|b0|d0>/<sim|hw/<src|inc> — If present, these directories should contain at least one file (source or header) that is specific for the chip model(s), processor(s), processor revision, and environment that is specified.

Object File Generation

A major benefit of the build system is the automatic generation of object file directories. This allows users to specify a certain set of environment variables and build for a specific target *variant*. See the section "Variants" on page 56.

In the build system, builds of multiple variants can coexist without corrupting each other.

Object Directory Location

The build system automatically creates target-dependent directories for the storage of object files, libraries, and executables. These directories are:

- **obj** — Stores object files that end in ***.o**
- **lib** — Stores archived libraries that end in ***.a**
- **bin** — Stores binaries, executables and package files that end in ***.exe**, ***.dcp**, ***.pkg**, and so on

These directories are located underneath the **run** directory. All corresponding files would be located under this directory as shown in Table 6.

Table 6 Directories Containing Files Generated by Build System

DIRECTORY	CONTENTS
app/**run/obj/***variant***/cprc/**	Contains all the object files for the CPRC for the application for the given variant.
app/**run/lib/***variant***/cprc/**	Contains any archived libraries for the CPRC for the application for the given variant.
app/**run/obj/***variant***/xprc/**	Contains all the object files for the XPRC for the application for the given variant.
app/**run/lib/***variant***/xprc/**	Contains any archived libraries for the XPRC for the application for the given variant.

Table 6 Directories Containing Files Generated by Build System (continued)

DIRECTORY	CONTENTS
app/**run**/**obj**/*variant*/**sdp/**	Contains all the object files for the SDPs (***.ucode**, ***.log**) for the application for the given variant.
app/**run**/**obj**/*variant*/**fdp/**	Contains all the object files for the FDPs (***.ucode**, ***.log**) for the application for the given variant.
app/**run**/**bin**/*variant*/	Contains all binary and executable type files for the application for the variant. This includes: • ELF binary files (***.dcp**) • Memory maps for those files (***.map**) • SDP image files (***.sdp**) • FDP image files (***.fdp**) • C-Ware package files (***.pkg**) • Emacs-like 'tags' files

For example, the following shows where the **obj**, **bin**, and **lib** directories would be located:

```
application
  run
    obj
      c5-d0-sim-debug
        cprc
        xprc
        sdp
        fdp
    lib
      c5-d0-sim-debug
        cprc
        xprc
    bin
      c5-d0-sim-debug
        cprc
        xprc
```

Environment Variables for Object File Generation

A few environment variables are required for the build system to properly operate.

The environment variables that are supported by the build system fall under two categories:

- Those required to define the type of target being built
- Those required by the build system, regardless of the configuration (these will be discussed in detail later in this document)

Table 7 shows the set of environment variables required by the build system that may be defined to customize the target of the build. Note that variables set in a **Makefile** will take precedence over variables set in the environment.

Table 7 Environment Variables Required by the Build System

ENVIRONMENT VARIABLE NAME	PURPOSE
`CPORT_ARCH`	Used by the build system to identify the architecture that is being built for. This variable can be defined on the command line, in the environment or in the Makefiles themselves. Arguments on the command line will take precendence, then those in the Makefile, then those in the environment. Valid values for this variable are **np**. The default value for CPORT_ARCH is **np**.
`CPORT_MODEL`	Used by the build system to identify the chip model that is being built for. This variable can be defined on the command line, in the environment or in the Makefiles themselves. Valid values for this variable are: **c5, c5e, c3e**. The default value for CPORT_MODEL is: **c5**.
`CPORT_REV`	Used by the build system to identify the revision of a particular version of a chip that is being built for. This variable can be defined on the command line, in the environment or in the Makefiles themselves. Valid values for this variable are: **a1, b0, d0** The default value for CPORT_REV is dependent on the value of CPORT_MODEL. There is no default for rev. For c5, the most recent rev is **d0**. For c5e and c3e, the most recent rev is **a1**.
`CPORT_TGT`	Used by the build system to identify the target for the architecture/model/revision of a chip that is being built. This differentiates software that is built for simulation vs. hardware. This variable can be defined on the command line, in the environment or in the Makefiles themselves. Valid values for this variable are: **hw**, **sim**. The default value for CPORT_TGT is: **sim**.

Table 7 Environment Variables Required by the Build System (continued)

ENVIRONMENT VARIABLE NAME	PURPOSE
CPORT_CFG	Used by the build system to identify the configuration being built (release, debug). This variable can be defined on the command line, in the environment or in the Makefiles themselves. Valid values for this variable are: **debug**, **release.** The default value for CPORT_CFG is: **debug**.

Variants

To automatically generate the object file directories and to search for source files in the correct places, the environment variables are used to generate something called an identifier called a *variant*. A variant is a uniquely named identifier that is a combination of chip architecture, model, processor, revision and environment.

For example, if a user had the following definitions:

```
CPORT_ARCH=np
CPORT_MODEL=c5
CPORT_REV=d0
CPORT_TGT=hw
CPORT_CFG=debug
```

The variant would be: **c5-d0-hw-debug**. This variant uniquely identifies the build target by its combination of chip architecture, model, revision, target and configuration in the source tree.

Because in the build system only one architecture currently exists, the value of CPORT_ARCH is not currently used in forming the variant.

Dependency Checking

The build system supports automatic dependency checking for **.c**, **.cpp**, and **.s** source files. No changes are required to the Makefiles provided in the CST that take advantage of the build system and include the proper ***.mk** files.

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

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

```
<application>
  run
    deps
      c5-d0-sim-debug
        cprc
        xprc
        sdp
        fdp
```

The first time that a build of a given target variant is performed, those directories in bold will be populated with the dependency checking information. This allows subsequent builds of this target variant to require only a rebuild of those source files for which a dependent file has changed (such as a header file).

The dependency checking information will be contained in files named: **filename.d** in the appropriate **cprc** and **xprc** subdirectory, and **filename.ud** in the **sdp** and **fdp** subdirectories.

The CST files **bin\cport-apps-rules.mk** and **bin\cport-rules.mk** support dependency checking. As long as they are included, no other support is required in the **Makefiles**.

If you change any file locations, we recommend that you perform the build system command 'make clean_all'. This causes the dependencies information to be automatically updated with the location of the new file and prevents generation of new build errors.

ıe Build System

To properly use the build system under the guidelines described above, you must follow certain practical rules for setting environment variables, and so on. To keep track of this manually would be cumbersome.

This section shows how you can leverage the build system with the least knowledge of its internals.

Setting Environment Variables

The build system defines and uses three types of environment variables:

- *Those required by the CST for valid operation of the software simulator, compiler and debugger* — These are set by running the **sv.bat** (or **sv.[c]sh** under Solaris and Linux) command file when creating a new shell. (See the *C-Ware Software Toolset Getting Started* document for the details.)
- *Those required by the build system that are set inside of Makefiles* — These are statically defined inside of **Makefiles** and the ***.mk** make include files. These variables are described in the file **bin\cport-base.mk**.
- *Those required by the build system to customize for different build configurations* — These are variables that affect if the system is built for software simulation or hardware, what chip model and revision, defining the variant, and so on (that is, CPORT_MODEL, CPORT_REV, and so on). These variables are described in the file **bin\cport-base.mk**.

Setting Variables Manually

Users may set the environment variables manually. This may be desirable if the build is initiated automatically by scripts, for example.

Users may set the environment variables manually in one of three ways:

- Define the variables in the user environment
- Statically assign the variables in **Makefiles**
- Call one of the configure scripts provided in the CST

To define the variables in the user environment, users can define the 'CPORT_*' variables wherever variables are defined on that particular system:

- Windows — Under System Properties, Advanced, and Environment Variables
- Solaris and Linux— In an initialization file (**.kshrc**, **.cshrc**)

To set these environment variables in a **Makefile**, call out the variables:

```
CPORT_ARCH=np
```

Under the CST's **bin** directory is the **configure** directory (for the CST on both Sun SPARC Solaris and Linux platforms it is the **ssbin/configure.[c]sh/** directories), which contains batch files (or Unix shell scripts) that set the environment variables for typical build target variants such as **c5-d0-hw-debug** or **c5e-a1-sim-debug**. The name of the script matches the variant directory that is utilized based on the values assigned to the variables in that script.

Note that for the CST on both the Sun SPARC Solaris platform and the Linux platform, you must "source" each configuration script, as described for the **sv.[c]sh** script in the *C-Ware Software Toolset Getting Started* document.

Setting Variables Automatically

Most users prefer to have the variables set automatically on the command line when invoking 'make'.

To do this, you can include the following variables-setting argument in the 'make' command line, if using the ***.mk** make include files provided in the CST's **bin** directory:

```
D:\C-Port\CST2.2\apps\<appName>\run> make REV=c5-d0-sim-debug
```

In the case of this example, specifying this 'REV' variable causes the build system to behave *as if* the following variables were set:

```
CPORT_ARCH = np
CPORT_MODEL = c5
CPORT_REV = d0
CPORT_TGT = sim
CPORT_CFG = debug
```

This allows flexible use of the build system without having to use different **Makefiles** or constantly changing the environment between builds.

You can also use these variables in the CST's simulation environment (for example, to find where a particular package files is located). This capability has also been extended to be used by the software simulator as follows:

```
D:\C-Port\CST2.2\apps\gbeSwitch\run> cwsim -rev c5-d0-sim-debug
```

In the case of this example, two things happen:

1. The software simulator behaves as if the 'CPORT_*' variables were set as specified in the argument to the '-rev' flag.
2. The correct simulator is invoked as specified by the argument to the '-rev' flag.

CST's Provided Application-Level Build Targets

The CST provides a set of make include files that define build targets for use in the **Makefiles** that reside at the application level. These targets, listed in Table 8, are intended to be run from the **.\run** subdirectory under any C-Ware application provided in the CST.

In Table 8 *notice that each build target's identifier is case-sensitive.*

For any of the build targets listed in Table 8, you can specify the variant using the command 'make REV=<variant>'.

Table 8 Build Targets Defined in CST-Provided Make Include Files

BUILD TARGET	WHERE DEFINED	PURPOSE
accept	**bin\cport-apps-rules.mk**	Perform the 'all' target's actions for this variant and run this variant's acceptance test (if any).
all	Application-level, component-level, or subsystem-level **Makefile**	Build all of this variant's generated components (that is, ***.o**, ***.a**, ***.out**, ***.map**, ***.dll**, ***.exe**, ***.sdp**, ***.dcp**, ***.dsh**, ***.d**). This is typically the *default target* in an application-level, component-level, or subsystem-level **Makefile**
checkEnv	**bin\cport-apps-rules.mk**	Run the CST's **bin\checkEnv.pl** script.
clean	**bin\cport-apps-rules.mk**	Perform the 'clean_local' target's actions for this variant, and delete ***.out** and ***.processed** files for this application.
clean_all	**bin\cport-apps-rules.mk**	Perform the 'clean' target's actions for all variants of this application.
clean_host	**bin\cport-apps-rules.mk**	Perform the 'clean_local' target's actions for the **host** subdirectory for this application.
clean_local	**bin\cport-apps-rules.mk**	Delete the subdirectory tree for this variant under the **obj**, **lib**,, **bin**,, and **deps** subdirectories for this application.
clean_local_all	**bin\cport-apps-rules.mk**	Perform the 'clean_local' target's actions for all variants of this application.
clean_patterns	**bin\cport-apps-rules.mk**	Delete all files in this variant's **inPatterns** subdirectory, and delete this variant's **outPatterns\s** directory.
compare	**bin\cport-apps-rules.mk**	Compare the **.out** files produced by this variant's acceptance test with this variant's **.expected** files.

8 Build Targets Defined in CST-Provided Make Include Files (continued)

D TARGET	WHERE DEFINED	PURPOSE
parseMap	**bin\cport-apps-rules.mk**	Run the CST's **bin\memMapParse.pl** script for this variant's XPRC and CPRC executables.
patternDir	**bin\cport-apps-rules.mk**	Create the **outPatterns** subdirectory for this variant.
patterns	**bin\cport-apps-rules.mk**	Build this variant's input pattern files (if any).
pkg_check	**bin\cport-apps-rules.mk**	Validate the contents of the package file (***.pkg**) for this variant.
test	**bin\cport-apps-rules.mk**	Perform the 'pkg_check' target's actions for this variant, perform the 'patterns' target's actions for this variant, run this variant's acceptance test (if any), then perform the 'compare' target's actions for this variant.

CST's Provided Make Include Files

Table 9 lists the CST's provided make include files (found under the CST's **bin**) and the purpose of each. These files support the operations defined in the application-level Makefiles that are provided in the CST for each C-Ware application.

Table 9 CST-Provided Make Include Files

ENVIRONMENT VARIABLE NAME	PURPOSE
cport-base.mk	This make include file processes the build environment variables and sets up the required build variables. It is used by **cport-rules.mk**. This file is typically not modified by users. This file should not be included directly in user **Makefiles**.
cport-base-port.mk	Portable portion of **cport-base.mk**. Modify this file when a new operating system and build targets are added. This file is typically not modified by users. This file should not be included directly in user **Makefiles**.
cport-make.mk	This make include file sets up the necessary build flags and variables. Configures the build by including the pertinent sub-Makefiles specific to the build target variant's elements. It is used by **cport-rules.mk**. This file is typically not modified by users. This file should not be included directly in user **Makefiles**.
cport-rules.mk	This make include file provides support for building object files, libraries, executables, and so on. Includes **cport-base.mk** and **cport-make.mk**. This file is typically not modified by users. This file should not be included directly in user **Makefiles**.

Table 9 CST-Provided Make Include Files (continued)

ENVIRONMENT VARIABLE NAME	PURPOSE
cport-apps-rules.mk	This make include file provides support for building object files, libraries, executables, and so on. Includes **cport-base.mk** and **cport-make.mk**. This file is typically not modified by users. This file should not be included directly in user **Makefiles**.
cport-<CPORT_MODEL>-<CPORT_REV>.mk	Contains definitions specific to target model and target revision. This file can be modified by users to accommodate changes in toolsets or command names.
cport-<CPORT_PROC>.mk or **cport-make-<CPORT_OS>-<CPORT_PROC>.mk**	Contains definitions specific to target processor. This file can be modified by users to accommodate changes in toolsets or command names.
cport-make-<CPORT_OS>.mk	Contains definitions specific to target operating system. This file can be modified by users to accommodate changes in toolsets or command names.
cport-make-<CPORT_BUILD_OS>.mk	Contains definitions specific to a particular build operating system. Defines the general commands and the native compiler. This file can be modified by users to accommodate changes in toolsets or command names.
cport-make-<CPORT_BUILD_OS>-<CPORT_OS>.mk	Contains definitions specific to a particular build operating system and target operating system. The cross-compiler is defined here. This file can be modified by users to accommodate changes in toolsets or command names.
cport-<CPORT_TGT>.mk	Contains definitions specific to build target. This file can be modified by users to accommodate changes in toolsets or command names.
cport-<CPORT_CFG>.mk	Contains definitions specific to build configuration. This file can be modified by users to accommodate changes in toolsets or command names.

Additional Makefile Functionality

This functionality is available only for CST releases after CST Version 2.0.

The CST's build system allows you to add extra compiler flags and extra include paths from both the command line and from individual Makefiles.

To add extra flags (which are assigned to the CFLAGS variable) from the command line, execute the following:

```
make EXTRA_CFLAGS=-DMY_EXTRA_CFLAG
```

To add extra include paths (which are assigned to the INCLUDES variable) from the command line, execute the following:

```
make EXTRA_INCLUDES=-I/home/me/MyTempIncludePath
```

Summary of Build System Requirements and Recommendations

The following tables list a set of requirements and recommendations for successfully integrating applications into CST build system, split out into the following categories:.

- "Build System Directory Requirements for Applications"
- "Naming Conventions"
- "Build System Checklist"

'd System Directory Requirements for Applications

Table 10 enumerates the directory requirements of applications in the CST.

Table 10 Requirements of CST Applications

DESCRIPTION	REQUIRED
The directory structure will meet the specifications in "Application Directory Structure" on page 46 and "Structure Requirements for File and Directory Names" on page 47.	Yes
The only leaf directories at the <family> level or below will be "inc" and "src".	Yes
The only leaf directories at the <module> level will be "inc", "src", "doc", and "sim".	Yes
The "sim" directory may contain a "patterns" directory.	No
Source code should be placed in the directory tree at the highest level as practical.	No
As a source code file becomes more target specific, it should be demoted down the directory tree as appropriate.	No
Source code which is expected to be portable across all (or many) architectures should be placed in the "<module>/src" directory and its public interface should be placed in the "<module>/inc" directory.	No
Source code which implements interfaces between processors in a single architecture should be placed in the ".../<family>/src" directory and its public interface should be placed in the ".../<family>/inc" directory.	No
Source code which is intended to run on a particular processor should be placed in the ".../<proc>/src" directory and its public interface should be placed in the ".../<proc>/inc" directory.	No
Source code which includes chip specific implementation should be placed in the ".../<chip>/src" directory and its public interface should be placed in the ".../<chip>/inc" directory.	No
Source code which included chip revision specific implementation should be place in the ".../<rev>/src" directory and its public interface should be placed in the ".../<rev>/inc" directory.	No

Naming Conventions

Table 11 enumerates the naming conventions/requirements of CST applications.

Table 11 Naming Conventions/Requirements of CST Applications

DESCRIPTION	REQUIRED
All files and directories will be named using a mixed case convention where the first letter is always lowercase and the beginning of each subsequent logical word is uppercase.	Yes
Filenames will meet the specifications in "Makefiles" on page 49.	Yes

Table 11 Naming Conventions/Requirements of CST Applications (continued)

DESCRIPTION	REQUIRED
Makefiles will be named "Makefile"	Yes
Simulation configuration files will be named "config[<desc>]".	Yes
Simulation input files will be named "sim[<desc>].in".	Yes
Filenames should not contain underscores.	No
Header files used to export public functions and data should have the same file name root as the source file in which the functions and data are contained.	No
Header files that describe interfaces between physical components should have a file name root that ends in "If".	No

Build System Checklist

Table 12 presents a checklist to aid developers in making sure that an application meets all the Build System requirements of the CST.

Table 12 Checklist of Build System Requirements of CST Applications

DESCRIPTION	COMPLETE?
proper directory structure	
properly named source code and header files	
properly named Makefile and simulation files	
standard targets in Makefile	
support for GNU emacs TAGS	
README file (recommended)	
regression test (recommended)	
design review (recommended)	
no errors or warnings during build (recommended)	

Chapter 4

BUILDING AND PACKAGING AN APPLICATION

Overview

This chapter provides an overview of the C-Ware versions of GNU tools that are provided in the C-Ware Software Toolset (CST) and how you use them to build application executables targeted to a C-Port Network Processor (C-5, C-5e, or C-3e NP). This document is not intended to be a complete guide to programming in the C programming language or to using the GNU software development tools.

This chapter — which, like the rest of this, assumes that you are an experienced programmer familiar with the C programming language, C compilers, and **make** tools — contains the following sections:

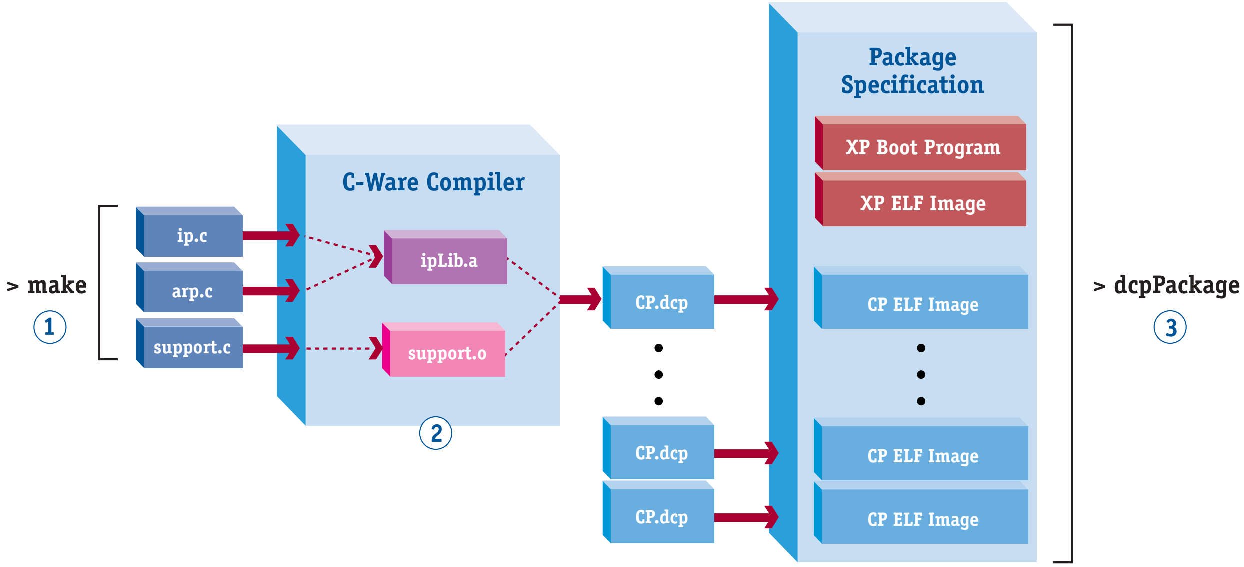
- "About Building a C-Ware Application" on page 68
- "Environment Variables" on page 70
- "Using the 'make' Tool" on page 70
- "Using the C-Ware Compiler" on page 74
  - "Compiler Support for Different Network Processors" on page 75
  - "Compiling a Program Targeted for the CPRC" on page 76
  - "Compiling a Program Targeted for the XPRC" on page 77
  - "Linker Recommendation" on page 78
- "Packaging and Loading An Application" on page 79
  - "About the dcpPackage Tool, Package Description Files, and Package Files" on page 79
  - "Understanding NP Application Configurations" on page 80
  - "Using the 'dcpPackage' Tool" on page 81
  - "RC Interface Code" on page 93
  - "How the XP Starts" on page 94
  - "Loading and Reloading the CPRCs" on page 96

uilding a C-Ware tion

After writing and modifying your code, you can build an application, or any component of an application, that is targeted for a C-Port Network Processor (NP). Figure 3 on page 68 shows the process for compiling and packaging C-Ware code to run on the C-Ware Simulator or on a NP.

Notice in Figure 3 that the ultimate product of building a network processor-based application is a C-Ware *package file*, which is directly loadable into a NP device or into the C-Ware Simulator.

Figure 3 C-Ware Application Building Process



To build a C-Ware package file:

1 Execute the **make** tool and specify a *build target*, which specifies a *target variant* as defined in the *Build System Conventions* document. The variant is a string formed as the concatenated combination of shorter keyword strings, each of which signify one of the following:

   - **Architecture** — Network processor
   - **Model** — C-5, C-5e, or C-3e
   - **Revision** — A1, B0, and so on
   - **Environment** — simulation or actual hardware
   - **Configuration** — debug or release

   make reads the specified **Makefile** (that is, typically specified *implicitly* and found in the current directory) and executes the compiler, the library manager, the **dcpPackage** packaging tool, and any other CST tools needed to build a C-Ware package file for the specified target. Using the **make** tool is described in more detail in "Using the 'make' Tool" on page 70.

2 **make** automatically builds dependency information as necessary about the software components that contribute to producing each generated software component, then automatically checks that information during each new build to determine which components to rebuild. Under **make** control, the various tools perform these operations:

   - Compile C and C++ source modules, creating object files (default extension **.o**). For more compiler details, refer to "Using the C-Ware Compiler" on page 74.
   - Create libraries of object files. You can combine related object files into a library to allow reuse and sharing of those objects. For example, if you had object files for various routine protocols (such as **ip.o**, **arp.o**, and **rarp.o**), you could combine them into a library called **libip.a**. This library could then be linked into multiple products.
   - Link the object files and libraries together to create executable files. For example, to create an ethernet switch application, you might link files such as **libip.a**, **boot.o**, **enet.o**, and so on. The resulting executable files conform to an industry-standard, portable, file format known as ELF (Executable and Linking Format).

3 **make** produces a C-Ware package file from a variety of pieces: the XP Boot Program (supplied by C-Port with the C-Ware Software Toolset), the application's XP executable file and CP executable files, and the SDP and FP microcode executable files.

   If the Makefile's dependencies dictate, **make** invokes the **dcpPackage** tool to create a new package file. The **dcpPackage** operates under the control of a *package description file*.

   The **dcpPackage** tool and package description files are described in "Packaging and Loading An Application" on page 79.

After a package is created, it can be specified in the configuration of a C-Ware Simulator session, so that its contents are automatically loaded into that simulation session. The Simulator also allows configuration of how the various pieces of the application found in the specified package file are loaded into the appropriate areas of the simulated network processor. For more information see the *C-Ware Simulation Environment User Guide* document.

Environment Variables

The C-Ware Software Toolset's installation process automatically updates the *user environment variable settings* on your machine. Consult the *C-Ware Software Toolset Getting Started* document for a list of the environment variables pertinent to your workstation's operating system.

These environment variables point to the directories and subdirectories for a given version of the C-Ware Software Toolset. Thus, their settings must change if you elect to start working in a different CST version (for example, CST Version 1.8 or CST Version 2.0).

Using the 'sv' Script

After opening a new command shell in which you intend to use any CST command line tool for any CST version, always run the 'sv' script as your first step, but *do not* run it more than once in the same shell process.

For more information about using the 'sv' script for your workstation's operating system environment, consult the *C-Ware Software Toolset Getting Started* document.

Using the 'make' Tool

The C-Ware Software Toolset includes a copy of the GNU **make** tool that you can use to automate the process of building software for the C-Ware Simulator or for a NP. **make** is located in the Toolset's top-level **bin** directory, and operates on GNU-compatible

Makefiles that specify the commands and file dependencies needed to build an application.

Makefiles

Makefiles are ASCII text files that are suitable as inputs to the GNU **make** tool. In the C-Ware Software Toolset there is a Makefile in each directory where work must be done to build the C-Ware libraries and Reference Library application executables. Makefiles can recursively invoke the **make** tool in other directories to ensure that a required component is up to date.

Makefiles contain information that specifies:

- Which files need to be built, based on which files have changed since the last time **make** was run. This *dependency checking* speeds up the builds because object files and libraries that have not changed will not be rebuilt. The Makefiles provided in the CST support automatic production and checking of dependency information.
- The commands to run to build a particular dependent file.

Makefiles can be performed from the command line:

```
C:\C-Port\CST2.2\apps\gbeSwitch\run> make
```

This **make** command builds the C-Ware Gigabit Ethernet Switch application.

A Makefile can also be performed by invocation from another Makefile or script:

```
myTarget:
   cd targets\sim && $(MAKE)
```

One Makefile can also "include" another Makefile, called a *make include file*. A make include file typically defines Makefile variables and certain "core" functionality that more specifically "targeted" Makefiles can use.

The CST includes a set of make include files that are intended to be included by the *application-level Makefile* for each C-Ware application provided in the CST. The included make include files, combined with the CST's well-defined build directory architecture, allows all application's Makefiles to be very similar to each other. See the *Build System Conventions* document for the details.

Targets

A *target* is a reference to the definition in a Makefile of a series of commands to perform when specified “rules” are satisfied. The CST’s build system supports two kinds of targets:

- **Build targets** — Commands to perform to produce (that is, compile, link, or otherwise filter) an application’s software component files based on other files. The CST build system defines a scheme for specifying a “variant” target (that i,s the combination of target processor, processor version, environment, and configuration). The variant, which can be specified in the make command line or via previously set environment variables, determines which tree of subdirectories (under the application’s root directory) are populated with software component files and which build-time switches are used by the C-Ware compiler s (and other build tools) to produce those files.

  Some targets of this kind are defined to produce *all* the software component files that comprise the application deliverable (that is, a package file), whereas other targets of this kind produce a narrowly defined *subset* of the necessary component files, such as only the simulated ingress traffic files for an application’s simulation sessions.

- **File maintenance targets** — Typically, commands to delete some, or for certain targets all, of the software components files used to produce the application deliverable.

Each target’s identifier is *case-sensitive*.

If you specify no target in a make command line, the ‘all’ target is the default. The operations performed for the ‘all’ target are defined in the application-level Makefile.

Table 13 summarizes the standard Makefile targets.

Table 13 Standard Makefile Targets

TARGET NAME	DESCRIPTION
default	First target in the Makefile. Has a single dependency of “all” and no rules. Invoking make on the command line with no targets will build the target “all.”
all	Builds all necessary code images. (e.g. the *.pkg file)
patterns	Generates all input pattern files used for regression testing.
test	Runs the regression test.
accept	Builds the code image, generates the patterns, and runs the regression test. Equivalent to “make all patterns accept.”
clean	Removes all files created by the make process from all of the standard targets.

Also note that Makefiles must support GNU emacs TAGS.

Here is a portion of an example Makefile illustrating the use of the standard targets:

```
default:          all

all:             $(TGT)/widget.pkg

patterns:    $(INPAT_DIR)/widgetIn.pat
               cd $(INPAT_DIR) && $(MAKE)

test:
               $(DCPSIM) -batch
               .
               .
               .

accept:      all patterns test

clean:
               rm -r $(TGT)
               rm -f $(INPAT_DIR)/*.pat
               .
               .
               .
```

Current Directory for Running Make

You typically build each of the CST-provided C-Ware applications from the application’s **run** subdirectory.

Building the Reference Library executables in this way causes recursive calls to **make** to build the executables for the NP’s various embedded processors, as well as the files that support a simulation session for the application. Also, if necessary, the C-Ware services libraries that the application uses will be rebuilt.

Finally, the **make** tool produces a C-Ware package file, which can be loaded either into the C-Ware Simulator or on C-Port NP hardware.

For example, to build a package file for the Gigabit Ethernet Switch application that incorporates debuggable executables and that is targeted to C-5 NP Version D0 hardware, enter this command from the application’s **run** subdirectory:

```
C:\C-Port\CST2.2\apps\gbeSwitch\run> make REV=c5-d0-hw-debug
```

ie C-Ware er

The C-Ware Software Toolset includes a version of the GNU C compiler (GCC) that has been customized for the C-Ware platform. GCC is a re-targetable and re-hostable compiler system with multiple front ends and a large number of hardware targets. GCC is complemented by a set of utility programs for object code translation and so on, and by the GNU Debugger (GDB).

The C-Ware Software Toolset's customized version of the GCC, called the C-Ware Compiler, supports the programming of the Channel Processor RISC Cores (CPRCs) and Executive Processor RISC Core (XPRC). You use the C-Ware Compiler to compile your C-Port Network Processor-targeted programs, creating NP-targeted image and map files, which identify the program and data sections of the image. You can examine this information for debugging purposes or for determining the actual size of the NP image.

C-Port's version of the GNU compiler is distributed under the GNU Public License (GPL).

The C-Port compiler executable is **bin\cport-gcc.exe**. This executable is automatically invoked by the **make** tool, which is described in "Using the 'make' Tool" on page 70.

By convention, the C-Ware Makefiles produce executable files with a .dcp filename extension.

The file **bin\cport-make.include** contains the switch settings that are used to compile and link all source, object, and library files provided in the C-Ware Software Toolset.

The commands you use for building programs for the C-Port NP's Channel Processor RISC Core (CPRC) and the Executive Processor (XP) are described in the following sections.

piler Support for erent Network Processors

The compiler expects a command line option to specify the C-Port NP (C-5, C-5e, or C-3e) you are targeting. See the sections "Compiling a Program Targeted for the CPRC" on page 76 and "Compiling a Program Targeted for the XPRC" on page 77 for the details.

Compilation for the C-5e supports additional XPRC/CPRC instructions compared with the C-5.

Note these difference in the results of compilation and linkage of programs for the C-5 and C-5e processors:

- Unlike for the C-5, the C-5e's 'k0' register (register 26) is not shared by the four register contexts in a cluster.
- Additonal optimizations are performed for C-5e code, such as use of 'Branch Likely' instructions where appropriate and removal of "stray" nops that follow memory accesses (LW and SW).
- For the C-5e the compiler does not generate workaround code for the SLT instruction problem (found only in the C-5).
- Starting address of loaded code:
  - 0x0000 for C-5e programs
  - 0x8000 for C-5 programs
- IMEM size is larger for the C-5e than for the C-5.

ng a Program d for the CPRC

To compile a program to run on a C-5 NP's Channel Processor RISC Core (CPRC), include the following switch on the compiler command line:

```
-mcpu=cport
```

To compile a program to run on a C-5e or C-3e NP's Channel Processor RISC Core (CPRC), include the following switch on the compiler command line:

```
-mcpu=cport2
```

In the file **bin\cport-make.include** you will notice that Makefiles for C-Ware applications additionally include these compiler switches:

```
-O3 -Wall -DDCP_APPLICATION_EVENTS -c
```

These switches specify optimization level, warning messages level, define the symbol 'DCP_APPLICATION_EVENTS' (to support generating event timing metrics), and compile without linking, respectively.

The CPRC does not support the multiply or divide opcodes and does not have a floating point unit. Avoid these operations. In an executable built to run on the CPRC, the main routine must be named DCPmain().

To compile a *debuggable* program to run on a CPRC, additionally include this compiler switch:

```
-g
```

In the file **bin\cport-make.include** you will notice that Makefiles for C-Ware applications additionally include this compiler switch to support debugging:

```
-DDCP_DEBUG
```

When building an executable file that will be debugged, C-Port recommends that you compile all source files to be debugged with a lower level of optimization, to allow more convenient interaction with your source code. That is, use the "no optimization" switch (-O0) to compile only the source files that you will be examining in source form.

It is usually not feasible to compile your entire CP application with the "no optimization" switch because this increases the probability that the resulting executable image exceeds the size of the CP's IMEM.

piling a Program geted for the XPRC

To compile a program to run on a C-5 NP Executive Processor (XP), include the following switch on the compiler command line:

```
-mcpu=cportxp
```

To compile a program to run on a C-5e or C-3e NP Executive Processor (XP), include the following switch on the compiler command line:

```
-mcpu=cportxp2
```

In the file **bin\cport-make.include** you will notice that Makefiles for C-Ware applications additionally include these compiler switches:

```
-O3 -Wall -c
```

These switches specify optimization level, warning messages level, and compile without linking, respectively.

The XP does not support the multiply or divide opcodes and does not have a floating point unit. Avoid these operations. In an executable built to run on the XP, the main routine must be named DCPmain().

To compile a *debuggable* program to run on the XP, additionally include this compiler switch:

```
-g
```

In the file **bin\cport-make.include** you will notice that Makefiles for C-Ware applications additionally include this compiler switch to support debugging:

```
-DDCP_DEBUG
```

When building an executable file that will be debugged, C-Port recommends that you compile the source files to be debugged with a lower level of optimization, to allow more convenient interaction with your source code. That is, use the “no optimization” switch (-O0) to compile only the source files that you will be examining in source form.

It is usually not feasible to compile your entire XP application with the “no optimization” switch because this increases the probability that the resulting executable image exceeds the size of the XP’s IMEM.

To compile a program to run on the XP and produce an executable file, additionally include the following switch and object file specification on the compiler command line:

```
-nostdlib $(CPORT)/bin/Gnu/lib/gcc-lib/mips-cport-elf/3.2/xpcrt0.o
```

(The **xpcrt0.o** object file must precede any other sources, objects, and libraries in the command line.)

When compiling for the XPRC, be aware that:

- The XP's IMEM contains the program's text section. It is limited to 32KB and is located at 0x8000 for C-5 programs, and 0x0000 for C-5e programs. The IMEM is the only place from which the XP can execute instructions.
- The XP's default DMEM (bus ID 25) is limited to 16KB and is located at BD900000h. By default, this area contains all static variables, constants, and so on, and also contains the XP's primary stack. To place aligned data in the XP's default DMEM, use the ALIGNED16, ALIGNED 64, or ALIGNED128 macros.
- The XP's secondary DMEM (bus ID 24) is limited to 16KB and is located at BD800000h. This area contains data declared using the FAR_DATA macro (that is, any program data section whose name begins with the five characters ".far."). To place aligned data in the XP's secondary DMEM, use the FAR_ALIGNED16, FAR_ALIGNED 64, or FAR_ALIGNED128 macros. The compiler does not use the secondary DMEM unless specifically requested.

See the file **dcpRegisterDefsXp.h** for definitions of the XP control registers.

Linker Recommendation

If your XPRC or CPRC program fails to link due to the production of an executable image that is too large for the target IMEM resource, by default the linker produces no map. This can make debugging more difficult.

To force the linker to produce a map when the linkage itself fails, use this line in your Makefile:

```
make BUILD=Release CFLAGS_DBG='-WI,--print-map' name_of_application
```

or some variant combination of switches that includes the '--print-map' switch.

kaging and Loading Application

With its 17 RISC cores (one XPRC and 16 CPRCs) and 17 microcode targets (16 Channel Processor SDPs and one Fabric Processor FDP), the C-Port C-5, C-5e, or C-3e Network Processor (NP) can load many possible combinations of executable images. To support the NP's many possible loadable configurations, the C-Ware Software Toolset provides an object called a *package* that encapsulates the executable images that run on the NP.

A package is a set of binary data that is the concatenation of a set of executable images, typically including:

- The primary boot image
- The user's XPRC executable, the CP application executables (including the CP SDPs executables), the FP FDP executable
- Other software infrastructure

About the dcpPackage Tool, Package Description Files, and Package Files

You build a package using the C-Ware Software Toolset's **dcpPackage** tool. The **dcpPackage** tool assembles a package from a set of executable components specified in a *package description file* then stores the package in a *package file*, which is a binary file with a '.pkg' filename extension. Packaging is described in more detail in "Using the 'dcpPackage' Tool" on page 81.

The package description file serves these purposes:

- Identifies the location of each executable image to be built into the package.
- Determines the configuration of the target NP's Channel Processors, including which CPs will be loaded with software and whether a given cluster is shared.
- Identifies XPRC executable images so that they can be loaded in an overlaid manner as the XPRC runs. See the section "Specifying the XPRC Init and Main Phase Executables" on page 86.
- Identifies additional executable images that can be loaded into a CPRC by the XPRC program based on run-time considerations.

A package file is suitable to be stored in a PROM device or can be loaded from any off-chip source supported by the NP boot process.

anding NP tion Configurations

When the NP boots, executable images are loaded into the Executive Processor RISC Core (XPRC) and the Channel Processor RISC Cores (CPRCs). The possible configurations of these executables are described in the following sections.

XPRC Executable

The XPRC executable is responsible for chip boot, initialization, and run time processing.

Due to its multiple roles, the XPRC executable is typically “partitioned” into one or more *init phase* executables and one *main phase* executable. For information about coding your XPRC and CPRC programs to support partitioning, see “Compiling a Program Targeted for the XPRC” on page 77 and “Compiling a Program Targeted for the CPRC” on page 76.

CPRC Executables

The configuration of the CPRCs is somewhat more complex. Depending on the nature of your application, you might specify that the same executable image be loaded into each CPRC, or you might use different images that run on certain CPRCs or images that differ from CP cluster to CP cluster.

A CP cluster uses one executable image that is loaded into the cluster’s shared IMEM. However, the SDPs for the CPs in that cluster might or might not use the same SDP executable image; you can specify this in the package description file. See “Using the ‘dcpPackage’ Tool” for more information.

Examples of Application Configuration

Examples of some of the possible configurations are found in the C-Ware Reference Library applications. In your C-Ware Software Toolset installation look for the package description files (***.dsc**) in the **apps\***AppName***\\run\** subdirectories.

For example, in the ATM Cell Switch application the same executable is loaded on the CPs configured to transmit as for those configured to receive. This application uses shared IMEM on all four CP clusters in the NP and loads the same CPRC executable in all clusters.

On the other hand, in the Gigabit Ethernet application separate receive and transmit CP executables run in certain CP clusters. The receive executable runs on two of the CP clusters, and the transmit executable runs on the other two CP clusters.

ng the ‘dcpPackage’

You use the **dcpPackage** tool to build a new package and store it to a file, and to report the contents of an existing package file.

Building a Package

The input to **dcpPackage** must be a package description file (default extension ‘.dsc’). The output from **dcpPackage** is a package file (default extension ‘.pkg’).

To build the package file, in your Makefile use define a build rule for the package file (extension ‘.pkg’) that performs a command line such as the following (for the **gbeSwitch** application):

```
...
# define a rule for putting all the pieces together into a package file
$(BINDIR)/$(APPNAME).pkg : $(APPNAME).dsc CPRC XPRC SDP FDP
$(BOOTFILE)
   $(PACKAGE) $(PACKAGE_FLAGS) \
   -DBTFILE=$(BOOTFILE) \
   -DXPINIT=$(BINDIR)/$(APPNAME)XpInit.dcp \
   -DXPFILE=$(BINDIR)/$(APPNAME)Xp.dcp \
   -DRXFILE=$(BINDIR)/rx.dcp \
   -DTXFILE=$(BINDIR)/tx.dcp \
   -DSDPFILE=$(BINDIR)/gbe.sdp \
   -DFDPFILE=$(BACK_TO_BACK_FABRIC) \
   -o $@ $<
...
```

The series of lines beginning with the line “$(BOOTFILE)” specify a “parameterized” **dcpPackage** command line. Each “-D*symbol*” line in the series of lines defines a symbol used in the application’s package description file. Each symbol stands for the path of the specific RC executable (***.dcp**), SDP executable (***.sdp**), or FP executable (***.fdp**) files that should be built into this application’s package file.

Purpose of the Package Description File

The package description file is an ASCII text file whose records can be empty, contain a comment, or contain a *statement* beginning with a *keyword*.

Each keyword statement must be an expression that specifies the location of a file used in building part of the package file. Some statements introduce a block of other statements enclosed in braces, for specifying the executable image files for one or more CPRCs and their respective associated SDPs.

Package description file statements can refer to symbols and macros in the Makefile that calls **dcpPackage**.

In its simplest form, the package description file's statements declare where the **dcpPackage** tool should find the following image files, from which it builds a package file:

- NP boot image (keyword BOOT)
- XPRC executable image(s) (keywords XP, XPINIT)
- RC Interface Code image (keyword INTERFACE)
- (For each Channel Processor) CPRC executable image (keyword CODE)
- (For each Channel Processor) SDP microcode image (keyword SDP)
- FP microcode image (keyword FP)

The **dcpPackage** tool builds information blocks to describe the CPRC and SDP executable images and appends those blocks, along with the text, data, and relocation information from the image files, into the package. This allows the XP Boot Program (also called the *primary boot*) to find the CP information and pass it to the XP's *DCPmain()*.

Sample Package Description File

Figure 4 on page 83 shows a sample package description file. It tells the **dcpPackage** tool where to find the executable images to be built into the package file.

Figure 4 Sample Package Description File, Ethernet Switch Application

```
PACKAGE  "Demonstration Ethernet Switch package description file";

BOOT  $(BTFILE) "C-Port primary bootstrap";
XPINIT  $(DEMO52XPINIT) "Demo Ethernet Switch - XP init program";
Xpmain: XP  $(DEMO52XPFILE) "Demo Ethernet Switch - XP program";

FP = $(DEMO52FPFILE) "Demonstration Ethernet Switch - FP program";

// Use default rcInterface.dcp code, from Makefile.
// INTERFACE = $(INTERFACE);

// Specify statement label, for referencing via 'USES' clause.
SDP_10: SDP=$(ENET10SDPFILE); // Define 10Mbit ucode image file.
SDP_100: SDP=$(ENET100SDPFILE); // Define 100Mbit ucode image file.

CP0-3 shared {  // CP cluster is shared.
     CODE = $(ENET100RCFILE);
     SDP USES SDP_100;  // All SDPs in cluster use same ucode image.
}

CP4-7 shared {  // CP cluster is shared.
     CODE = $(POSOC3CRCFILE);  // Use same image for 4 shared CPs
     SDP4 = $(POSOC3CSDP4FILE);  // Use image in SDP4
     SDP5-7 = $(POSOC3CSDP5FILE);  // Use image in SDP5 to SDP7
}

CP8 shared {  // CP cluster is shared, but only CP8 is loaded.
     CODE = $(ENET1000RXRCFILE);
     SDP = $(ENET1000RXSDP8FILE);
}

CP12 {  // CP cluster is not shared (default); only CP12 is loaded.
     CODE = $(ENET1000TXRCFILE);  // Loads CP12 *only*.
     SDP = $(ENET1000TXSDPFILE);  // Loads SDP12 *only*.
}

// NOTE:
// This package does not configure the Channel Processors CP13, CP14,
// and CP15, so these CPs are not loaded during this package's load
// time. Neither can these three CPs be loaded during the run-time for
// the XPRC program loaded from this package. The entire C-5 chip
// must be reset before these three CPs can be loaded.
```

Package Description Statements

The following sections describe the keywords used to begin each statement in a package description file. Each (non-comment) line in the package description file starts with a keyword.

Any statement in a package description file can be labeled. Each *label* can be up to eight (8) characters long; the case of all label characters is significant.

PACKAGE Keyword

The 'PACKAGE' keyword specifies the following identification string (in quotes). For informational purposes **dcpPackage** embeds this string in the package file.

BOOT Keyword

The 'BOOT' keyword specifies the location of the file containing the primary *bootstrap code image* file. There can be only one line containing the 'BOOT' keyword in a package description file. The quoted comment is included to provide version or other information.

A bootstrap code image file must be an ELF format image file. This file must be linked to a 'text' base in IMEM. It cannot contain any 'data' section. In Figure 4 on page 83, as in most PROM images, the Freescale supplied "primary boot" is referenced. The **dcpPackage** tool extracts the 'text' section from the executable image and places the boot image's 32-byte header and text at the beginning of the package, as the package's first boot-loadable section.

XPINIT and XP Keywords

The C-Ware Software Toolset provides the software infrastructure to support overlays of XPRC executable images. Thus, an XPRC program can have one or more *init phase* executable images and one *main phase* executable image. See the section "Specifying the XPRC Init and Main Phase Executables" on page 86.

FP Keyword

The 'FP' keyword specifies the location of the file containing the Fabric Processor executable image.

INTERFACE Keyword

The 'INTERFACE' keyword specifies the location of the *RC interface image*, which is loaded into the CPRCs so that they can communicate in certain ways with the XPRC. This code provides minimal infrastructure for supporting interrupt-driven operation of the C-Ware Debugger.

By default, if the package description file does not contain an ‘INTERFACE’ statement, the **dcpPackage** *tool looks for an RC Interface Code image file named* **rcInterface.dcp** *in the directory where this package obtained the primary boot descriptor (that is, the executable specified in the ‘BOOT’ statement).*

For more information about how the RC interface code operates, see the section “RC Interface Code” on page 93.

In your C-Ware Software Toolset installation, find sample RC Interface source code in this file:

services\boot\chip\c5\xprc\src\rcInterface.S

CP, CODE, and SDP Keywords

For each configured Channel Processor, or Channel Processor cluster, that should be loaded with an executable image, you include a block of lines that begin with the ‘CP’ keyword. Each block of lines, enclosed in braces, specifies the locations of:

- For one CP, the CPRC program image and SDP microcode image file(s)
- For a CP cluster, the CPRC program to load into the cluster’s shared IMEM and the SDP microcode image file(s) to load into those CPs’ SDPs

In the block of lines that follow the ‘CP’ keyword statement, use the ‘CODE’ keyword to specify the CPRC executable image and use the ‘SDP’ keyword to specify an SDP executable image.

You specify a ‘CP’ statement either for one CP or for a cluster of CPs, as follows:

- For one CP, the ‘CODE’ keyword line that follows specifies that CP’s CPRC executable image file, and the ‘SDP’ line specifies that CP’s SDP executable microcode image file.
- For a cluster of CPs, use the ‘CODE’ keyword line specifies the one CPRC executable file that is loaded into each specified CP (which is not necessarily every CP in that cluster), and use one or more ‘SDP’ keyword lines to specify which executable microcode image file to load in a given SDP in that cluster.

For example, in the package description file shown in Figure 4 on page 83:

- Each of the first three CP clusters (CP0 to CP3, CP4 to CP7, and CP8 to CP11) is configured to be shared; therefore, only one executable image is loaded into each cluster’s shared IMEM. The CPRCs in each respective cluster run the same executable.

- The SDPs in each shared cluster are either configured to use the same executable (as for CP0 to CP3 in the example) or one or more SDPs can be loaded with a different executable (as for CP4 to CP7 in the example).
- CPs 8 to 11 are configured as a shared cluster, but only CP8 is configured to be loaded with an executable.
- CPs 12 to 15 are not configured as a shared cluster (the default), but only CP12 is configured to be loaded with an executable.

In Figure 4 on page 83, as the comment for the sample file's 'CP12' keyword states, by default the CPs are *not* configured as clustered. CPs must be explicitly configured as clustered using the 'shared' keyword.

To demonstrate additional rules for specifying the loading of the CPs, Figure 5 on page 87 shows parts of another sample package description file. In this example, notice the following:

- Lists and ranges of SDPs can be used to specify which executable files to load in a cluster.
- It is illegal to use a label with a statement specifying a range of SDPs.
- It is illegal for more than one 'SDP' statement to specify the same SDP.

Specifying the XPRC Init and Main Phase Executables

If the NP's IMEM resources are insufficient to contain a given XPRC executable, the C-Ware Software Toolset's XPRC run-time infrastructure supports the overlaying of executables at run-time. The infrastructure is implemented to support what are called "initialization phase" (or "init phase") and "main phase" executable images.

An init phase executable would be the first image loaded into the XPRC, and it would perform the work of initializing the NP's Channel Processors, the Fabric Processor, and also perhaps manage the table loading process. After these activities are completed, the init phase executable performs the step of requesting another executable to be loaded and overlaid over itself. This could be another init phase executable that performs additional one-time processing, or could be a main phase executable that remains resident and running in the XPRC until the entire NP is next reset.

Figure 5 Demonstration of Package Description File Rules for Loading Channel Processors

```
PACKAGE  "Demonstration package description file";

BOOT  $(BTFILE) "C-Port primary bootstrap";

...
// Specify statement label, for referencing via ’USES’ clause.
SDP_10: SDP=bar2.sdp;
...

CP0-3 shared {  // CP cluster is shared.
     CODE = foo1.dcp; // One CPRC image shared in cluster’s IMEM.
     SDP0 = bar1.sdp;
     SDP1-3 USES SDP_10; // USES keyword for reusing label statement.
}

CP4-11 shared {  // 2 CP clusters, each is shared.
     CODE = foo2.dcp; // One CPRC image shared in each cluster’s IMEM.
// More than one SDP in a cluster can use the same image file.
     SDP4-6,8-11 = bar3.sdp; // Specify a list or range.
     SDP7 = bar4.sdp;
}

CP12-15 shared {  // 2 CP clusters, each is shared.
     CODE = foo3.dcp; // One CPRC image shared in cluster’s IMEM.
FloatSdp:      SDP12-14 = bar5.sdp; // *ERROR* No labels on ranges.
     SDP14,15 = bar6.sdp; // *ERROR* References SDP 14 again.
     SDP15-16 = bar6.sdp; // *ERROR* Outside cluster’s mask range.
}
```

The XPRC main phase executable would contain code that interacts with the host processor, supporting the overall device’s normal processing activities, as well as interacts with the NP’s CPRC executables to perform the device’s normal data forwarding and control activities.

During package loading of the NP, the XPRC executable image specified in the package description file’s first ‘XPINIT’ statement (for an init phase executable) or ‘XP’ statement (for a main phase executable) is loaded first into the XPRC. If an init phase executable is loaded first, at run-time that executable’s code can cause another XPRC init phase executable to be loaded, or can cause the main phase executable to be loaded, by calling the *ksProcLoadXp()* routine and specifying as the second argument the label string of that of ‘XPINIT’ or ‘XP’ statement in the package description file.

Remember that the labeled 'XPINIT' and 'XP' statements themselves do not *cause* the overlaying of executables to occur during the XPRC's run-time. Rather, these statements only associate a label string with each XPRC init phase or main phase executable image that is embedded in the package. That is, in order to code the package description file properly, you must have independent knowledge of the identity and purpose of each XPRC executable image that the package description file references.

The characteristics of and requirements for XPRC init phase and main phase programs are described in the following sections.

Specific Rules for 'XPINIT' and 'XP' Statements

The package description file must specify an 'XPINIT' statement for each XPRC init phase executable. If there is more than one init phase executable specified in the package description file, the second through last 'XPINIT' statements must also be labeled. The statement's label allows that executable image to be referenced in a call to the API routine *ksProcLoadXp()*.

The XPRC init phase executables referenced in labeled 'XPINIT' statements can be loaded (that is, overlaid) at run-time in any order, regardless of the order of those statements' appearance in the package description file.

You can place 'XPINIT' statements anywhere in the package description file.

If the XPRC's main phase executable is not the first executable image loaded when the NP is loaded, then the 'XP' statement must be labeled. This is because the main phase executable will be loaded at run-time of another init phase execute by its calling the *ksProcLoad()* routine, which requires a package description file label as its second argument.

Examples

Figure 6 shows a sample package description file that uses an ‘XPINIT’ line to distinguish the XPRC’s initialization program from its main program. Notice that the ‘XP’ statement is labeled, which allows that executable to referenced by label in a call to *ksProcLoadXp()* in the already loaded overlay executable.

Figure 6 Sample Package Description File, XP Init and Main Phase Programs

```
PACKAGE  "Demonstration of XP Init and XP main executable images";

BOOT  $(BTFILE) "C-Port primary bootstrap";
XPINIT  $(XPINIT) "XP init phase";

CP0-3 {
   CODE = $(CPFILE);
   SDP = t54_0thru3.sdp "SDP contents for CP 0 through 3";
}

XPmain: XP  $(XPFILE) "XP main - 2nd arg to ksProcLoad() is XPmain”;
```

Figure 7 shows a sample package description file that includes more than one ‘XPINIT’ statement. In this case the primary bootstrap loads the XPRC executable specified in the ‘XPINIT’ statement that follows the ‘BOOT’ statement because it is the first ‘XP’ or ‘XPINIT’ statement in the package description file.

Figure 7 Sample Package Description File, Multiple XP Init Phase Programs

```
PACKAGE  "Demonstration of multiple XP Init images";

BOOT  $(BTFILE) "C-Port primary bootstrap";
XPINIT  t54InitXp.dcp "XP Init phase 1, loaded by primary bootstrap";

CP0-3 {
   CODE = $(CPFILE);
   SDP = t54_0thru3.sdp "SDP contents for CP 0 through 3";
}

TestIn2:  XPINIT t54Init2Xp.dcp "XP Init phase 2";
TestMain:  XP t54MainXp.dcp "XP main phase";
```

An executable specified in an ‘XPINIT’ statement can call *ksProcLoadXp()* to chain to another init phase executable specified in another ‘XPINIT’ statement, and so on, until some init phase executable calls *ksProcLoadXp()* and chains to an XPRC main phase executable. For the example shown in Figure 7:

1. The first XPRC init phase executable loads this package description file’s other init phase executable, via a call to *ksProcLoadXp()* with the label ‘TestIn2’ as the second argument passed.

2. The second XPRC init phase executable loads the XPRC main phase executable, via a call to *ksProcLoadXp()* with the label ‘TestMain’ as the second argument passed.

No Overlap of Primary Bootstrap

An executable specified in an ‘XPINIT’ statement is not allowed to overlap the overlay loading code, in anticipation of its later loading an XPRC main phase executable. Thus, its maximum size is less than the maximum possible size of an XPRC executable image.

The primary bootstrap always loads the XPRC executable that is specified in the *first* ‘XPINIT’ or ‘XP’ statement in the package description file.

For more information about XP initialization, see the section “How the XP Starts” on page 94.

Specifying a USES Clause

In a given package description file a statement that includes a *USES clause* can refer to another labeled statement and thereby declare a file indirectly. If any executable image file description statement has a label, that specification can be used elsewhere in the package description file via the USES keyword.

Figure 8 shows a sample package description file that includes more than one USES clause to refer to the same CP executable image file.

Figure 8 Sample Package Description File, USES Clause and Identification Strings

```
PACKAGE  "Demonstration of USES clauses with identification strings";

BOOT  $(BTFILE) "C-Port primary bootstrap";
XP  $(XPFILE) "Ethernet switch - XP program";

FP = t52_fp.sdp "Fabric port program";

CpFile:  CODE = $(CPFILE) "Common CP program";

CP0 {
   CODE USES CpFile;
   SDP = t52_0.sdp "SDP contents for CP 0";
}

CP1 {
   CODE USES CpFile;
   SDP = t52_1.sdp "SDP contents for CP 1";
}

CP2 {
   CODE USES CpFile;
   SDP = t52_2.sdp "SDP contents for CP 2";
}
```

ing the Contents of a Package File

Use the **dcpPackage** tool's '-l' switch to produce a summary listing of a package file's contents. Enter a command line like this:

```
C:\C-Port\CST2.2\bin> dcpPackage -l myPackageFile.pkg
```

Figure 9 on page 92 shows the output for a package whose contents define a boot image, two XPRC init phase executables, one XPRC main phase executable, the RC Interface Code executable, and one shared Channel Processor cluster's (CP0 to CP3) single CPRC executable (but no SDP executable).

Figure 9 Sample Output from 'dcpPackage -l' Command Line

```
$ dcpPackage -l test54.pkg
// test54.pkg - dump of package - Created Fri Jun 01 14:14:47 2002

// RCs active: 0-3
// RCs primary (1/shared cluster + nonshared): 0
// RCs in shared clusters: 0-3

PACKAGE "XP Init phase test";

BOOT "../../services/boot/lib/c5-d0-sim-debug/xprc/xpPrimaryBoot.dcp"
     "C-Port primary bootstrap";
  XPINIT "test54InitXp.dcp" "XP init phase";
TestIn2:    XPINIT "test54Init2Xp.dcp" "XP 2nd phase init code";
TestMain:   XP "test54MainXp.dcp" "XP main phase";
INTERFACE
"../../services/boot/lib/c5-d0-sim-debug/xprc/rcInterfaceSequence.dc
p";

  // File offset of header = 0x1ec0

RC0-3 SHARED {   // Primary RCs = 0;  Secondary RCs = 1-3
    CODE  "test54Cp.dcp";
}
```

Listing of Command Line Switches

Use the **dcpPackage** tool's '-h' switch to produce a listing of all command line switches and their arguments.

nterface Code

RC Interface Code is infrastructure software that allows C-Port NP CPRC programs to communicate with the XPRC program in two situations:

- When loading and reloading of CPRC takes place
- When CPRC program debugging is taking place

To work with the C-Ware Software Toolset as provided, you need not edit the RC Interface Code source code, nor attempt to manipulate this executable's behavior at run-time. It is described here only because the **dcpPackage** tool always embeds it in every NP package.

Inclusion in a NP Package File

An RC Interface Code image must be included in each package file that you create using the **dcpPackage** tool. In a package description file you can specify an 'INTERFACE' statement to identify where **dcpPackage** should look for an RC Interface Code image file.

If the package description file does not specify an 'INTERFACE' statement, **dcpPackage** expects to find the file **rcInterface.dcp** in the directory that contains the primary boot descriptor file that is also incorporated into the same package. If **dcpPackage**'s attempt to find this file is not successful, **dcpPackage** produces an error message and exits.

Run-Time Activity

After the NP leaves reset and as the CPRCs are being loaded, the RC Interface Code image is loaded into each CPRC's high IMEM and reserves memory in each CPRC's high DMEM. The RC Interface Code is the first code executed on the CPRC after its IMEM is loaded.

Use the C-Ware Software Toolset's **cport-objdump** *tool to identify the IMEM and DMEM resources that the RC Interface Code image uses.*

The RC Interface Code enters its *interface loop*, which continues executing until the CP is stopped or the NP is reset. In this loop the code polls its DMEM region for "command" notifications received from the XP. For the details see the source file:

services\boot\chip\c5\xprc\src\rcInterface.S

When you use the C-Ware Debugger to conduct a debugging session on a given CPRC program, the act of setting a breakpoint in that program image causes the XPRC's infrastructure software to send a "command" to that CPRC that, in turn, causes that CPRC's RC Interface Code to insert a 'break' instruction in the program image.

When the target CPRC program again executes and encounters that breakpoint, the interrupt is triggered and causes a jump to an RC Interface Code interrupt handler. By default, this handler causes the RC Interface Code to notify the XP program that a breakpoint has been encountered and also causes the RC Interface Code to enter its “interface loop” to await the next “command” from the XP.

As the Debugger user issues GDB commands in response to reaching the breakpoint, additional sets of interactions occur between the XPRC’s infrastructure code and the target CPRC. In this manner, the RC Interface Code interacts with the XPRC program’s infrastructure to make possible a C-Ware Debugger session targeted at a CPRC program image.

How the XP Starts

When the NP is released from reset by some outside agent, only the XP begins executing. The CPs stay in reset until they are individually released.

The XP (more specifically, the XPRC) begins execution in its IROM at location 0x10000, the XP start address. The contents of the XP IROM are different from the CPRC IROMs, and they cause code to be copied into memory from somewhere. The pointer to this “somewhere” is in location 0xBD808300 (the XP Configuration Register for Debug Counter 0 Start Value). The contents of this location at reset time are 0xBFC00000, which is a pointer to the boot PROM.

It is possible for an outside agent to change the value in location 0xBD808300 (using the PCI interface) before releasing reset on the NP, thus causing the XP to boot from some other location (presumably mapped onto the PCI bus). Whatever the source, we hereafter refer to the area pointed to by the contents of location 0xBD808300 as “the boot PROM.”

To model this behavior, the C-Ware Simulator supports this configuration variable:

```
XpBootFileName filename
```

If an ‘XpBootFileName’ entry is present in the Simulator configuration file, it names a disk file that must contain an exact image of the boot PROM. The contents of this file are mapped to memory starting at location 0xBFC00000. The XP IROM will load from the PROM.

If there is no ‘XpBootFileName’ entry, nothing is mapped into memory at location `0xBFC00000`. In this case, the Simulator supplies a dummy IROM image that consists of a small infinite loop. In this case, when the Simulator ‘go’ command is given, the XP executes but does no useful (and no damaging) work. This allows CP-only utilization under the Simulator.

If the Simulator user issues a 'load' command directed to the XP, the specified 'XpBootFileName' entry argument's file is loaded, and the XP starts executing at the entry point of the file rather than at the XP start address. This effectively disables the IROM. Thus, if there is an 'XpBootFileName' entry in the **config** file, you probably should not explicitly load the XP (that is, you should not include an 'XpIROMFileName' entry in the **config** file).

The XP IROM expects that the boot PROM will contain one or more boot-loadable sections. A *boot-loadable section* consists of:

- A 32-byte *package header*, whose structure is defined in the header file named **dcpPackage.h** (See Table 14.)
- Data to load (length rounded up to a multiple of four)
- Text to load (rounded up)
- String (null-terminated) with the original file name that contained the text and data
- Identification string (null-terminated) that is supplied to the **dcpPackage** tool
- Filler if needed to make the section a multiple of four bytes long

Table 14 Contents of the Package Header

OFFSET	CONTENTS
0	"XPU1"
4	Size of text section
8	Address of text section in IMEM
12	Number of data sections
16	Total size of data
20	Starting address
24	Total size of header, text, data, and ID information
28	Unused

The XP IROM copies the text from the boot-loadable section into the address specified in the header, then transfers to the routine at the start address. It passes the address of the boot PROM as a parameter to that routine. The IROM requires that the boot-loadable section it loads have a zero-length data section. (The **dcpPackage** tool enforces this restriction.)

The code loaded by the IROM is usually a routine called the *primary boot sequence*, supplied by the CST in the **XpPrimaryBoot.dcp** file. It must be coded to coordinate with the XP IROM.

XpPrimaryBoot.dcp sets up the stack pointer, copies the next boot-loadable section from the PROM, and executes it. This section may contain both code and data, and is usually the full XPRC application.

The primary boot passes a pointer to the next section in the PROM as the first parameter to the XPRC application. This is the location in the boot PROM just after the section it loaded. Usually this is a package header pointer, suitable for passing to the loader services.

At this point, the XPRC application starts executing. The application should use the C-Ware API Kernel Services routines to enable and initialize the respective CPs.

Loading and Reloading the CPRCs

CPRC executables cannot be overlaid in the same manner as XPRC executables. However, just as t he XPRC program causes the CPRCs to be loaded with software and to start, the XPRC program can also direct any CPRC to stop and to be reloaded with another executable image.

The XPRC program calls the API routine *ksProcStop()* to stop one CPRC (or *ksProcStopCps()* to stop one or more CPRCs). The XPRC program calls the API routine *ksLoadCps()* to load one or more CPRCs with a given executable.

To support this functionality, it is possible to include in a NP package any number of CPRC executable images in addition to those that are first loaded into the CPs. We recommend that you declare these executables in uniquely labeled 'CODE' statements at or near the end of the package description file. These statements must be labeled so that the XPRC can specify a given executable in the package by its label string.

CPRC Life Cycle

Consult the topic "CPRC Program Life Cycle," which is presented in the *C-Ware API User Guide* document in its "Kernel Services" chapter and within the major topic "Processor Operations."

State of Channel Processor DMEM After Reloading the CPRC

Consult the topics “Preserving Data For Use After a Channel Processor Reload” and “Preserving Data For Use After Multiple Channel Processor Reloads,” which are presented in the *C-Ware API User Guide* document in its “Kernel Services” chapter and within the major topic “Processor Operations.”

Appendix A

OFFLINE TABLE BUILDING LIBRARIES

Using Offline Table Building Libraries

The C-Ware Software Toolset (CST) provides Offline Table Building Libraries (OLTBLs) targeted for the C-Port Network Processors (NP). Library files are provided for each of the C-5/C-5e/C-3e NPs on Windows 2000/XP, Sun SPARC Solaris, and Linux platforms. The libraries support building code for simulated and actual systems; most commonly, they are used for testing/debugging an application before the target hardware is available.

The OLBTLs allow the initial state of the NP's Table Lookup Unit (TLU) and of C-Ware API Table Services' data structures to be generated and captured once. Those results can subsequently be loaded into the (actual or simulated) C-5/C-5e/C-3e for each new run of the application. This significantly reduces the simulated application's cycle count between the point of the application's load and startup and the point of starting payload throughput.

The OLTBLs serve these main purposes:

- For a hostless C-5/C-5e/C-3e application or system:
  - These libraries provide a way to create and populate the C-Ware complex table types (for example, Index/Vp-Trie/Data) that cannot be created and populated by C-5/C-5e/C-3e XPRC and/or CPRC programs.
  - These libraries allow the initial state of the TLU and of the data structures used internally by the C-Ware API Table Services to be generated and captured, then subsequently loaded into the C-Ware Simulator (or C-5/C-5e/C-3e hardware) on a hostless system.
- For a C-Ware application developer working with the C-Ware Simulator, by partitioning the application's table-building code into a distinct program that can be run once and whose results can be saved for subsequent loading at application startup time, the developer can load and run the table-reliant portion of a simulated C-5/C-5e/C-3e application more conveniently and efficiently.

The CST provides the OLTBLs in source code form (with the exception of source code for the TLU reference model) along with the Makefiles for building those libraries.

The Offline Table Building Libraries continue to undergo as table building functionality is integrated into the C-Ware Software Toolset's host application development environment. These tools are provided to give developers another option for building and analyzing TLU tables.

Workflow to Build a Program that Uses the OLTBLs

1 Write a C language program, called the "client program," that creates and populates a set of C-5/C-5e/C-3e resident tables that your C-5/C-5e/C-3e application requires. This program calls the C-Ware API Table Services routines.

2 Compile the client program on your CST development workstation and link it with the OLTBL version for your combination of C-5/C-5e/C-3e and workstation platform (that is, Windows or Solaris).

3 Run the client program, which generates up to three output files with the following default filenames:

- **tle_restore.h** — This C language header file contains a 'static inline' function that initializes Table Services internal data structures to their state at the end of the execution of the client program. This function must be called from the last phase of a multi-phase XPRC program. It must be called only if the last phase of the multi-phase XPRC program subsequently makes Table Services calls. Use this file to support a "hostless" C-5/C-5e/C-3e application targeted to the C-Ware Simulator or to actual hardware.
- **Tlu.State** — This binary file captures the state of the C-5/C-5e/C-3e TLU SRAM at the end of the execution of the client program. Use this file when running the C-5/C-5e/C-3e application under the C-Ware Simulator to load the TLU SRAM and configuration registers.
- **tle_writes.h** — This C language header file defines the "captured" C-5/C-5e/C-3e Ring Bus transactions that initialized the TLU's SRAM and config registers as the client program was run. This file is most commonly used to support a hostless C-5/C-5e/C-3e application running on actual hardware.

4 If running the C-5/C-5e/C-3e application under simulation:

- Insert a call to *tsRestoreTLU()* (an inline function defined in the generated **tle_restore.h**) in the initialization section of the application's XPRC program. This call should be conditionalized so that it is executed only when the XPRC program is run under simulation.
- Rebuild the application targeted for the C-Ware Simulator.
- When the application next runs under the C-Ware Simulator, direct the Simulator to perform these two commands:

```
c5sim> cd tlu
c5sim> restore ./Tlu.State
```

  Doing so updates the state of the simulated TLU SRAM to that at the end of execution of the client program.

If running the C-5/C-5e/C-3e application on actual hardware:

- Insert a call to *offlineTableConfig()* (an inline function defined in the file **apps/components/tableUtils/chip/np/xprc/tableConfig.h** file), which sends the C-5/C-5e/C-3e Ring Bus messages "captured" (that is, encoded) in the generated **tle_writes.h** file to the Ring Bus in the application.
- Rebuild the application targeted to the appropriate hardware environment.

Hardware-Targeted NP Applications That Use OLTBL

For a hardware-targeted C-5/C-5e/C-3e application whose XPRC program uses the OLTBL techniques, be aware that use of these techniques consumes some XP IMEM and XP DMEM resource and that more of the saved TLU state resides in XP DMEM than IMEM.

Depending on how your application's XPRC program utilizes XP IMEM and DMEM to support functionality distinct from table building operations, you might have the need to take additional care in partitioning and building the XPRC program's executable(s).

ucture of an OLTBL Client Program

You can code the OLTBL client program any way that you choose. However, the sample applications provided in the CST use OLTBL client programs that have a common structure, as described in this section.

Source Files

The client programs for the CST 's sample applications use two C .language source files:

- **tables.c** — Contains calls to C-Ware API Table Services routines to initialize, create, and populate the tables required by the NP application. Typically, this code is identical to code that might run in a production C-5/C-5e/C-3e application or C-Ware host application.
- **offline.c** — Contains code to parse the command-line arguments passed to the client program and calls to OLTBL functions that cause those files to be opened, written to, and closed. The command-line arguments specify which OLTBL output files will be generated by the client program.)

Command Line Parsing

The syntax of the client program's command line can be fully determined by the application developer. However, the sample applications provided in the CST use the following command-line syntax:

client_program_name [*rb_record_file* [*ts_state_file*] [*tlu_sram_file*]]]

Where:

- *rb_record_file* defaults to "tle_writes.h"
- *ts_state_file* defaults to "tle_restore.h"
- *tlu_sram_file* defaults to "Tlu.State"

Routines Defined in the OLTBL Library

Table 15 lists the routines defined in the OLTBL library file, where each is defined, and the purpose of each routine.

Table 15 Offline Table Building Library Routines

DEFINED IN THIS HEADER FILE	ROUTINE AND SIGNATURE	PURPOSE
services\table\host\offline\inc\tsOfflineSvcs.h	*tsOfflineRingBusRecordStart()* void tsOfflineRingBusRecordStart(char* infile)	Begins recording NP Ring Bus traffic to the specified *infile*.
	tsOfflineRingBusRecordEnd() void tsOfflineRingBusRecordEnd(void)	Ends recording of NP Ring Bus traffic.
	tsOfflineTableSvcsSave() TsStatus tsOfflineTableSvcsSave(char* infile)	Saves the internal state of the Table Services data structures to the specified *infile*.
services\table\host\offline\inc\tsOfflineTlu.h	*tsOfflineTluInit()* void tsOfflineTluInit(void)	Initializes the TLU model. Must be called before any TLU operations are performed in the client program.
	tsOfflineTluStateSave() void tsOfflineTluStateSave(char* filename)	Saves the state of the NP TLU SRAM to the specified *filename*.

These functions are found in the libraries:

services\table\offline\$(CPORT_BUILDENV)\tsOffline.a

where “$(CPORT_BUILDENV)” is "c5-winnt" (or “cxe-winnt”) or "c5-unix" (or “cxe-unix”).

Sample Offline Table Building Programs

Most of the applications provided with the CST use offline table building. Examples can be found in this directory:

apps\*AppName***\offline\c5\**

or:

apps\*AppName***\offline\cxe\**

The **gbeOc12Sar** application is a good example of building data, H-T-K, and LPM tables offline. One caveat is that, because of machine endianness dependencies, you must take care when using the C-Ware API Table Services to insert entries in the table. For best results, structure the data as an array of 32-bit words. This is especially important on little-endian machines.

Structure of OLTBL Client Program

This is the general structure for an OLTBL client program :

```
#include "tsOfflineSvcs.h"
#include "tsOfflineTlu.h"

void main()
{
    tsOfflineTluInit();
    tsOfflineRingBusRecordStart("tle_writes.h");

    /* Table creation/initialization/entry insertion code here. */
    ...

    tsOfflineRingBusRecordEnd();
    tsOfflineTableSvcsSave("tle_restore.h");
    tsOfflineTluStateSave("Tlu.State");
}
```

The code shown in Figure 10 on page 105 is typical for performing table building, and is suitable to be included in an OLTBL client program.

Figure 10 Typical Code to Perform Table Building

```
    ABLE_DATA(myTable);
#define NUM_ENTRIES 4
static int32u tableData[NUM_ENTRIES][8] = {
    { 1, 0x12345678, 0x11111111, 0x55555555, 0, 0, 0, 0},
    { 2, 0x12345678, 0x22222222, 0x55555555, 0, 0, 0, 0},
    { 3, 0x12345678, 0x33333333, 0x55555555, 0, 0, 0, 0},
    { 4, 0x12345678, 0x44444444, 0x55555555, 0, 0, 0, 0}
};

void tableCreate()
{
    /* Init table structure: table, type, numEnt, entrySize, keySize */
    tsTableInit(&myTable, tsTableData, 64 * 1024, 32, 4);
    tsTableCreate(&myTable, &myTableId);
}

void tableSetup()
{
    int i;

    for (i = 0; i < NUM_ENTRIES; i++) {
        tableInsert(myTableId, &tableData[i][0], &tableData[i][0], 32, 0);
    }
}

TsStatus tableInsert(TsTableId tableId, void* keyData, void* data, int length, int maskLen)
{
    TsEntry entry;
    TsKey key;

    key.keyData = (int8u*) keyData;
    key.maskLenBits = maskLen;
    entry.data = data;
    entry.length = length;
    entry.offset = 0;
    entry.reqTag = 0;
    entry.reqTag = tsRequestTag(0);
    entry.respTag = 0;

    while (!tsLookupRequestReady(entry.reqTag));
    return tsEntryInsert(tableId, &key, &entry);
}
```

Preempting Automatic Assignment

To support compatibility with existing C-Port NP microcode, the offline Table Services allow you to specifiy the table ID, preempting the API's automatic assignment. For efficiency, some of the microcode hardcodes the table ID used in the Ring Bus launch pads. Placing code similar to the following (immediately before calling *tsTableInit()*) allows the user to choose a table ID:

```
#ifdef BUILD_OFFLINE
   myTable.tableId = 8;
   myTable.table[2].tableId = myTable.tableId;
#endif
```

Note that choosing an invalid table ID results in the default action by the API of auto-assignment.

Calling tsRestoreTLU()

The C-Ware API function *tsRestoreTLU()* should be called in the last phase of the C-5/C-5e/C-3e application's multi-phase XPRC program after a call to *tsTableInitialize()*, but before any other C-Ware API Table Services calls. The header file **tle_restore.h** must be included to declare *tsRestoreTLU()*.

Calling this function is appropriate for both C-Ware Simulator and hardware targets and is only necessary if further Table Services calls are going to be made in the XPRC program.

Package Components

The OLTBL package includes:

- For the C-5, on the CST's Windows and Sun SPARC Solaris platforms:
  - **services\table\offline\c5-***cst-platform***\tsOffline.a** — A library that contains routines used to create an offline table-building program for the C-5 NP chip

    Where *cst-platform* can be either "winnt" or "unix"
- For the C-5e or C-3e, on the CST's Windows and Sun SPARC Solaris platforms:
  - **services\table\offline\cxe-***cst_platform***\tsOffline.a** — A library that contains routines used to create an offline table-building program for the C-5e or C-3e NP chip

    Where *cst-platform* can be either "winnt" or "unix"

Table 16 summarizes the location of other files in the OLTBL package.

Table 16 Locations of Files in OLTBL Package

COMPONENT	LOCATION IN THE C-WARE SOFTWARE TOOLSET
Library files	**services\table\host\offline\c5-{unix,winnt}\tsOffline.a** **services\table\host\offline\cxe-{unix,winnt}\tsOffline.a**
Header files	**services\table\host\offline\inc\**
Sample OLTBL client program	**apps\gbeSwitch\offline\c5\** (C-5 NP version) **apps\gbeSwitch\offline\cxe\** (C-5e/C-3e NP version)

When building an OLTBL client program, this directory:

services\table\host\offline\inc\

must appear before other include directories in your search path. This directory contains several files (**dcpTableSvcs.h** and others) that, when building an OLTBL client program, must be used in place of the “standard” C-Ware include file of the same name.

INDEX

How to Reach Us:

USA/Europe/Locations not listed:
Freescale Semiconductor Literature Distribution
P.O. Box 5405, Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

Japan:
Freescale Semiconductor Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu
Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor H.K. Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

Learn More:
For more information about Freescale
Semiconductor products, please visit
http://www.freescale.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

CSTADBG-UG/D

REV 01, 9/2004