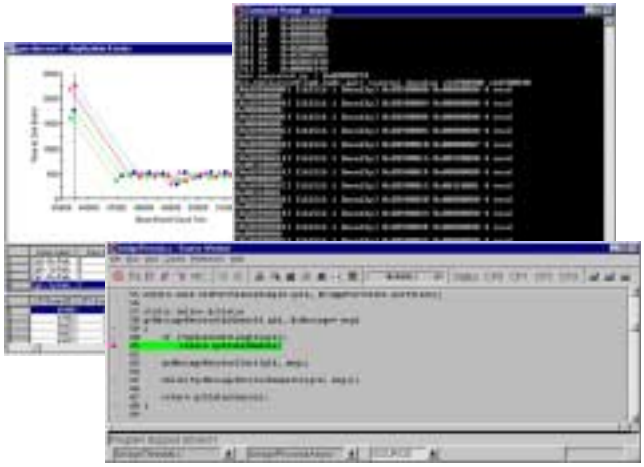
User Guide

C-WARE DEBUGGER

C-WARE SOFTWARE TOOLSET
VERSION 2.4

CSTDBGUG-UG/D
Rev 03

MOTOROLA
intelligence everywhere™



MOTOROLA

C-Ware Debugger User Guide

C-WARE SOFTWARE TOOLSET, VERSION 2.4

CSTDBGUG-UG/D
Rev 03

CONTENTS

FIGURES

TABLES

ABOUT THIS GUIDE

Guide Overview

The *C-Ware Debugger Guide* describes the basic features of the C-Ware Debugger tool and how to use it to accomplish several debugging techniques and strategies.

This guide is primarily intended for software application developers who must produce communications applications that run on a C-Port Network Processor (NP).

This guide assumes a 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 chapters, as follows:

- Chapter 1 discusses the executable software objects that are appropriate targets for debugging, how to build a debuggable executable, and the suitability of optimized code as an object for debugging.
- Chapter 2 presents several topics regarding use of the Debugger for specific tasks, such as interacting with the Debugger's graphical user interface (GUI), setting breakpoints, stepping by instruction, viewing source code, and so on.
- Chapter 3 offers topics relating to troubleshooting a debugging session.

Using PDF Documents

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

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

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

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

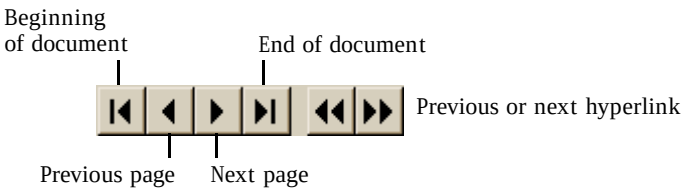


Table 1 summarizes how to navigate within an electronic document.

Table 1 Navigating Within a PDF Document

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

Guide Conventions

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

This icon and text designates information of special note.

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

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

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

References to CST Pathnames

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

- **C:\C-Port\Cst*x.y*** (on Windows 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.

Revision History

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

Note: books in this documentation set began bearing revision numbers in June, 2002. Prior to that date books were identified by CST version number.

Table 2 *C-Ware Debugger Guide* Revision History

REVISION	DATE	CHANGES
03	6/2002	Throughout the document updated references to the C-5 NP to also refer to the C-5e NP and C-3e NP, where appropriate.
CST 2.1	4/2002	In Chapter 2 added the heading "Debugging After a Panic on C-Port NP Hardware".
CST 1.8	7/2001	In Chapter 2 divided the section "Starting the Debugger" into two subsections "Starting the Debugger Outside the Simulator" and "Starting the Debugger Within the Simulator". Added and corrected index entries.
CST 1.7	4/2001	In Chapter 2 added a new major topic "Identifying the Package File". In Chapter 2 added a new topic "Remote Debugging of C-Port NP Devices". In Chapter 2 the major topic "Debugging Overlaid XPRC and CPRC Executables" has been rewritten for clarity. In Chapter 3 added a new topic "Debugging Infrastructure Overhead".

Related Product Documentation

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

Table 3 C-Port Silicon and CST Documentation Set

DOCUMENT SUBJECT	DOCUMENT NAME	PURPOSE	DOCUMENT ID
Processor Information	*C-5 Network Processor Architecture Guide*	Describes the full architecture of the C-5 network processor.	C5NPARCH-RM
	C-5 Network Processor Data Sheet	Describes hardware design specifications for the C-5 network processor.	C5NPDATA-DS
	C-5e/C-3e Network Processor Architecture Guide	Describes the full architecture of the C-5e and C-3e network processors.	C53C3EARCH-RM
	C-5e Network Processor Data Sheet	Describes hardware design specifications for the C-5e network processor.	C5ENPDATA-DS
	M-5 Channel Adapter Architecture Guide	Describes the full architecture of the M-5 channel adapter.	M5CAARCH-RM
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 (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

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

DOCUMENT 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

C-WARE DEBUGGER OVERVIEW

About the C-Ware Debugger

The C-Ware Software Toolset (CST) includes a version of the GNU debugger GDB, called the *C-Ware Debugger*, that has been customized for the C-Ware platform.

The C-Ware Debugger, implemented as the **cport-gdb.exe** tool in the CST, allows you to debug C-Ware applications running either on the C-Ware Simulator or on a C-Port C-5, C-5e, or C-3e Network Processor (NP). The CST includes all necessary software to perform either kind of debugging session.

The C-Ware Debugger incorporates these extensions to the GNU GDB package:

- A graphical user interface (GUI) from Cygnus Solutions.
- A custom C-Ware Debugger target named *dcpsim* that allows the C-Ware Debugger to interact with the C-Ware Simulator as a remote system-under-test.
- A custom C-Ware Debugger target named *dcp* that allows the C-Ware Debugger to interact with NP(s) or the C-Ware Simulator.
- Agent and interface software running on the C-Ware Development System (CDS) Host Application Module that allows the C-Ware Debugger to debug executables running on a NP.

The C-Ware Debugger is distributed under the GNU Public License (GPL). Information on using GDB is available on the GNU website.

This chapter cover the following topics:

Supported Targets

When using the C-Ware Debugger, you specify the debugging mode by specifying the target as follows:

- Use *target dcpsim* when the target executable is running under the C-Ware Simulator.
- Use *target dcp* when the target executable is running on an actual C-Port NP-based system.

There is a fundamental difference in the way a C-Ware Debugger session affects the behavior of a simulated NP (target 'dcpsim') and how it affects the behavior of a hardware NP (target 'dcp'). When interacting with the C-Ware Simulator, the C-Ware Debugger allows *all* the simulated NPs to execute concurrently by the same number of simulated clock cycles at a time. When any processor encounters a breakpoint, the entire simulation (all simulated devices) halts instantly. Also, stepping the program hosted on a particular processor by one instruction (whether source-level or machine instruction) causes all other simulated processors to run for exactly the same number of cycles.

For a program running under the C-Ware Simulator, debugging using the C-Ware Debugger and the dcpsim target causes the program to behave identically to a "free" simulator run with no debugging taking place.

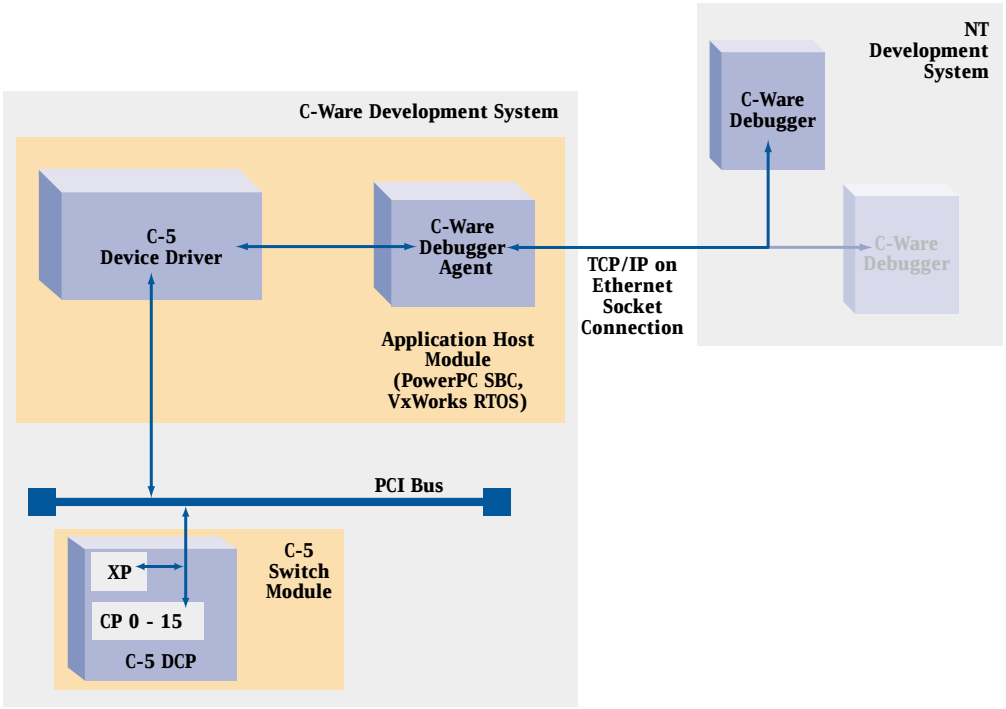
When the Debugger is interacting with the 'dcp' target, user-specified processors on the NP can execute (for example single stepping), while others do not execute at all. Also, when one CPRC processor in a CP cluster encounters a breakpoint, other CPRC processors in the same cluster are also halted, while all other running processors continue to execute. The stop delay between CPRCs in the same cluster cannot be predicted. True simultaneous halting of concurrently executing multiple processors sharing the same cluster is not possible.

C-Ware Development System as a Target Platform

In the C-Ware Development System (CDS), the C-Ware Debugger communicates with the C-Port NP via a TCP/IP network connection to the CDS's Host Application Module, which must be running the appropriate CDS host environment software under a supported host operating system, and within the CDS itself across a PCI bus connection between the Host Application Module and the C-5 Switch Module, where the NP (in this case, a C-5 NP) is physically mounted.

Figure 1 depicts the locations and interconnects among the C-Ware Debugger (running on a workstation where the C-Ware Software Toolset has been installed), the CDS, the CDS Host Application Module, and the C-5 Switch Module within the CDS.

Figure 1 C-Ware Debugger Connection to a C-Port Network Processor



Programs running on the C-Port NP's XP and CPs are valid targets for debugging using the C-Ware Debugger. The debugger does not work with host-side applications (running on the CDS's Host Application Module) or with any communications between host applications and the NP.

For debugging host applications, use the tools that come with Host Application Module's supported operating system's Application Development System. For instance, Tornado for VxWorks contains a complete debugger.

It is possible maintain multiple debugging sessions in parallel when debugging a hardware NP (target 'dcp'). For example, one session might be dedicated to debugging a program running on the XP, while two other debugger sessions are used with two different programs running on two different CPRC clusters. When debugging on a C-Ware Simulator ('target dcpsim'), only a single C-Ware Debugger session is allowed at any given time.

Ensuring Your Program is Debuggable by the C-Ware Debugger

The following conditions must be met in order for the C-Ware Debugger to debug your program:

- The program must be compiled with the -g option to enable the generation of symbol table debugging information.
- For dcpsim targets, make sure the program is already loaded as a result of either loading a package or performing a C-Ware Simulator 'load' command (or the equivalent operation specified in a Simulator configuration file).
- For dcpsim targets, ensure that the C-Ware Simulator command line is waiting for user input (not executing another command).
- For 'dcp' targets, the program must not disable interrupts.

Optimized Code and the C-Ware Debugger

Using the C-Ware Debugger with optimized code can lead to unexpected results. Optimizing the code may, for example, eliminate the code where you want to set a breakpoint or a watchpoint. Motorola recommends the following techniques to help minimize the problem:

- Remove code optimization (compile with the '-O0' option) where possible.
- Reduce code optimization (compile with a lower optimization level) where possible.
- Remove code optimization from only the code module to be debugged.

Chapter 2

USING THE C-WARE DEBUGGER

Starting the Debugger

1. Open a DOS command window.
2. Change directory to the **bin** directory under the C-Ware Software Toolset's installation directory.
3. Run the **sv** script, as described in the *C-Ware Software Toolset Getting Started Guide*.
4. Change directory to a subdirectory from which it is convenient to enter debugger commands that refer to the path(s) of your debuggable program images.
5. Enter the command: **cport-gdb**

   Alternatively, to start the C-Ware Debugger so that it presents only the conventional GDB command-line interface, enter the command: **cport-gdb --nw**

Debugger's GUI Windows

The C-Ware Debugger supports these GUI windows:

- Console, for command entry
- Source, for C-language source and assembly language
- Registers, for viewing register values
- Stack
- Memory
- Local Variables
- Watch Expressions
- Breakpoints

By default, when the Debugger starts, it opens a Source window, as shown in Figure 2 on page 27. To open a Console window, press the Console button (small "terminal" icon) on the Source window's button tool bar.

Figure 3 on page 28 shows the default Console window. You can interact fully with the Debugger using the console window. However, not all Debugger commands are fully supported via the interface elements shown in the source window.

Figure 2 C-Ware Debugger's Source Window

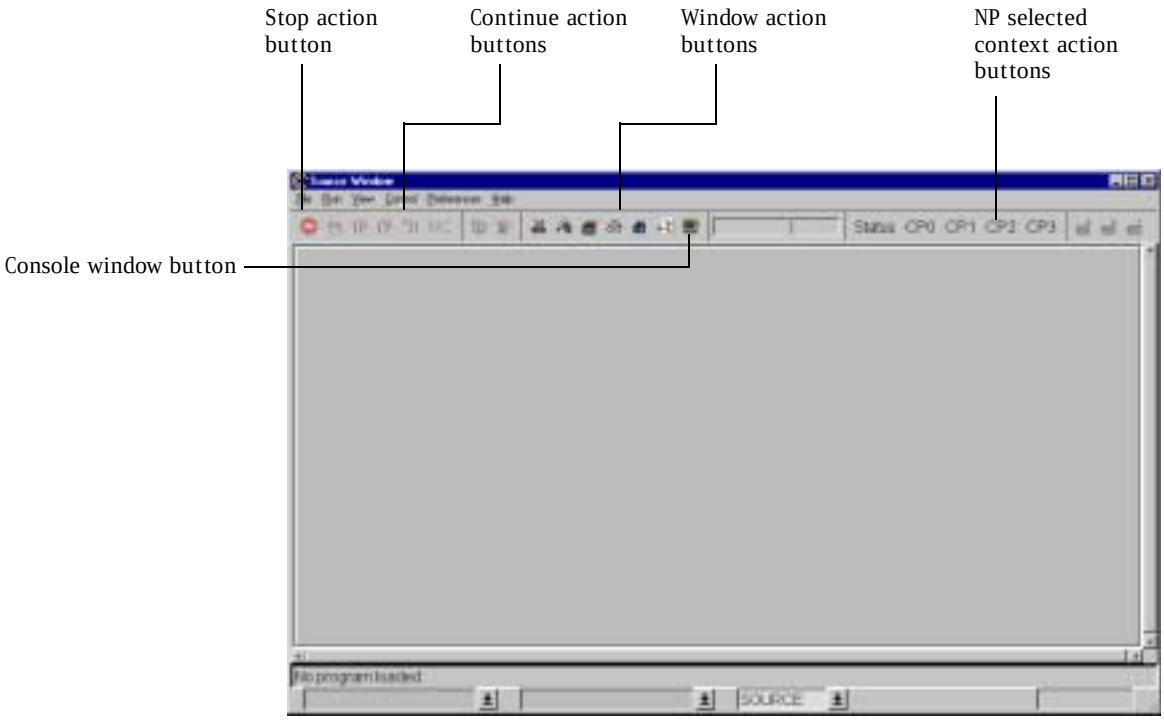
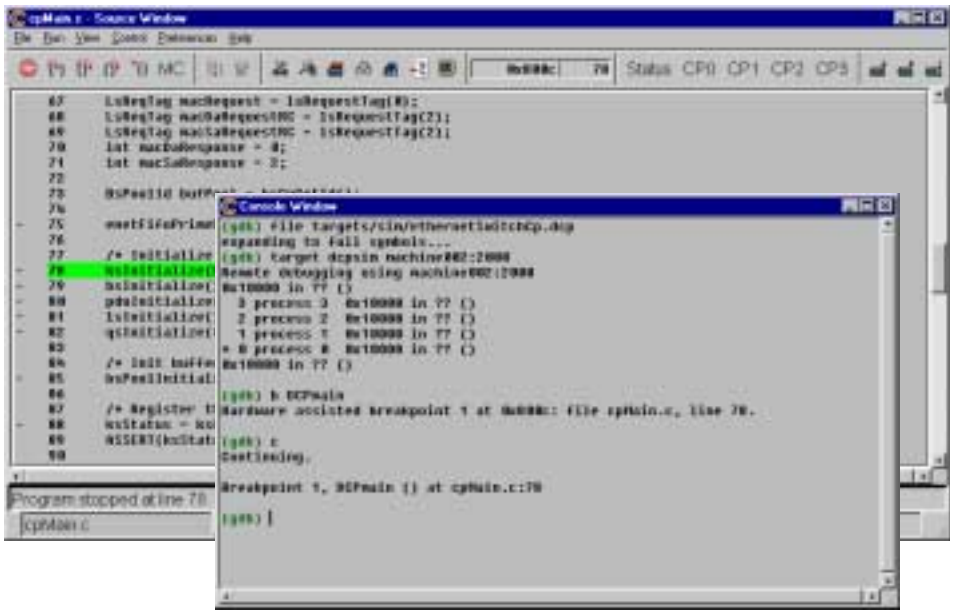


Figure 3 C-Ware Debugger's Console Window

Figure 4 on page 29 shows the appearance of the Debugger's source and console windows after reaching a breakpoint in the **gbeSwitch** application (included with the C-Ware Software Toolset).

Figure 4 Debugger Windows After Reaching a Breakpoint in gbeSwitch Application



More information on using the Debugger's GUI interface is available from the interface's help menu.

Connecting to the Simulator target

When the C-Ware Simulator is the target of your debugging session, there is more than one way to start the Debugger.

Starting the Debugger Outside the Simulator

Start the C-Ware Simulator in a separate DOS window as described in the *C-Ware Simulation Environment User Guide*. Issue the debug request command from the C-Ware Simulator's command line:

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

where:

- **Imem-spec** is the name of the Simulator's memory object connected with the CP or cluster of CPs to debug. For example, 'imemxp' designates the XP as the CPU to debug, while 'imem0' designates the cluster containing CP 0 as the target.
- **hostname:TCP port number** designates the host and TCP port to be used in the connection to the C-Ware Debugger.

For example, this Simulator command opens a debugging session on CP1 across a particular TCP/IP connection:

```
dcpsim> appdebug imem1 mozart:32000
```

Connect the C-Ware Debugger to the Simulator at the C-Ware Debugger command prompt:

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

Where *hostname:TCP port number* designates the host where the Simulator is running and must match the values from the 'appdebug' command above.

Once the connection is established, use the C-Ware Debugger's file command in the C-Ware Debugger console window to inform the Debugger which NP executable will be running on the target:

```
(gdb) file ethernetSwitchCp.dcp
```

Start debugging.

Starting the Debugger Within the Simulator

Start the C-Ware Simulator in a separate DOS window as described in the *C-Ware Simulation Environment User Guide*.

Issue a Simulator command to move to the XPRC or to the CPRC whose program you want to debug. For example:

```
dcpsim> m cprc0
```

Use a Simulator command to start the C-Ware Debugger:

```
dcpsim> gdb
```

To open a console window in the Debugger, press Ctrl+N or press the Console button on the button bar.

In the Debugger's console window add a breakpoint at the start of the program:

```
(gdb) b DCPmain
```

In the Debugger's console window start the target program:

```
(gdb) cont
```

Start debugging.

Connecting to a CDS Target

Start the C-Ware Development System (CDS). Load the application to debug from the DCP Shell command prompt:

```
DCP> packload DCP_name package_name boot_flags
```

Where:

- *DCP_name* indicates the C-Port NP to use in the CDS. For example, 'dcp0', or 'dcp1'.
- *package_name* is the C-Port NP package to load.
- *boot_flags* specifies the type of debugger session:
  - 0x200 for XP
  - 0x400 for CPRC cluster
  - 0x600 for both

For example:

```
DCP> packload dcp0 ethernetswitch.pkg 0x600
```

Using boot flags forces a breakpoint upon entry to the program's DCPmain() *function. You can load programs without a debug boot flag to avoid the initial breakpoint and still use the C-Ware Debugger. Simply connect the debugger to the program as described below. When you connect this way the program will already be running, and may have passed your area of interest.*

Now connect the C-Ware Debugger to the program. Enter a target command at the C-Ware Debugger prompt:

```
(gdb) target dcp IP_address CP_ID:NP_number
```

Where:

- *IP_address* specifies the IP address of the Host Application Module in the CDS
- *CP_ID:NP_number* is the ID of the target CPU, and, optionally, the NP number. For example, for the CPID use 25 for the XP or use 0 for the cluster containing CPRC0. *NP_number* is only needed if there is more than one C-Port NP in the CDS. *NP_number* defaults to 0 if it is not specified.

For example:

```
(gdb) target dcp 182.168.154.1 25:0
```

Table 4 lists the valid CP IDs.

Table 4 CPU IDs

ID	CPU
0	CP0
1	CP1
2	CP2
3	CP3
4	CP4
5	CP5
6	CP6
7	CP7
8	CP8
9	CP9
10	CP10
11	CP11
12	CP12
13	CP13
14	CP14
15	CP15
25	XP

Once the connection is established, use the C-Ware Debugger's file command to inform the debugger which NP executable will be running on the target:

```
(gdb) file ethernetSwitchXp.dcp
```

Start debugging.

See the *C-Ware Development System User Guide* for more information on using the debugger with the CDS.

Program Interruption

You can interrupt running code under control of the C-Ware Debugger. Press Ctrl+c in command line mode, or press the stop button in the GUI interface.

Viewing C-Ware Debugger Help

To view help about C-Ware Debugger command line options, enter the command: **cport-gdb -h**

To view help about C-Ware Debugger commands in the console window, enter the GDB command 'help'.

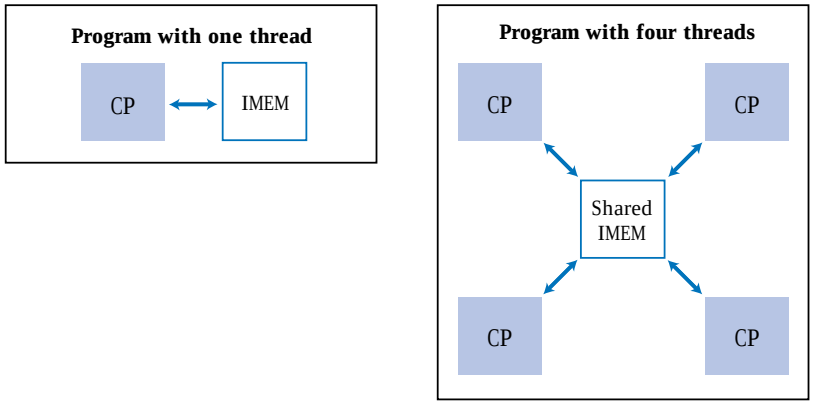
Identifying Threads

Regardless of the target type, a single C-Ware Debugger session controls the execution of the program on a single cluster. Four CPRCs sharing IMEM count as one cluster, while the XP also counts as one cluster. When controlling four CPRCs in a cluster, the C-Ware Debugger treats each of them as a separate thread of execution.

Do not confuse this concept with CPRC's or XPRC's register contexts.

The C-Ware Debugger identifies the XP and each CP by its thread number. For example, the executable running on the XP appears as thread 25. Also, if the IMEM is shared among CP0 through CP3, debugging information about "thread0" through "thread3" actually refers to CP0 through CP3, respectively. See Figure 5.

Figure 5 How the C-Ware Debugger Identifies Threads by CP



To view information about the threads and their associated CPs (or XP), enter this GDB command:

```
(gdb) info threads
```

In the output from this command notice the standard GDB convention of using an asterisk (*) to denote the current thread. Any standard GDB command that reads/writes memory or registers pertains only to the current thread.

To change the current thread enter the following GDB command:

```
(gdb) thread 2
```

The thread number must refer to a thread shown in the output of a GDB `info threads` command.

Examining Processor Register Contexts

The XP and each CP has four sets of 32 general-purpose registers, a program counter (PC) pseudo-register, a context register (CR), and a frame pointer (FP) pseudo-register.

The contents of a CP's context register (values 0 to 3) indicates which of the CP's four sets of general-purpose registers is currently in use by the CPRC. To set the contents of a CP's context register, enter this command:

```
(gdb) set $ctxt=n
```

where *n* can be 0 through 3.

To display the contents of the current CP's current context of 32 general-purpose registers as well as its PC, CR, and FP pseudo-register, enter this GDB command:

```
(gdb) info registers
```

Examining Program Memory

The executables that run in a cluster of CPs share an IMEM resource but not their respective DMEM resources. Thus, if the same executable is running on more than one CPRC in a cluster, the same variable in DMEM has different addresses as the current thread changes.

For example:

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

Using Breakpoints

You can set breakpoints on any source line by interacting with the source window of the C-Ware Debugger's GUI interface or by entering commands in the command line window.

In the Source window a small red square in the left margin of the source listing indicates the location of a breakpoint.

When you use the C-Ware Debugger's GUI interface to set a breakpoint in a source file, that breakpoint pertains to all CPs that execute that code.

Regular Breakpoints

If a cluster of four CPs share an IMEM resource, then any regular breakpoint pertains to all CPs and stops any of them if encountered during execution.

To set a regular breakpoint in a source file, use a GDB command like this:

```
(gdb) b 45
```

This command sets a breakpoint at line 45 of the current source file.

Thread-specific Breakpoints

You can also set breakpoints that are thread-specific. That means they only pertain to a particular CP.

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

This command sets a breakpoint at line 45 of the current source file, pertaining only to CP2.

Temporary Breakpoints

You can set a breakpoint that will be encountered only once. For instance, you can set a breakpoint in code that resides in a shared IMEM. After the first CPRC encounters the breakpoint, the C-Ware Debugger automatically clears that breakpoint.

To set a temporary breakpoint at line 45 of the current source file, enter this command:

```
(gdb) tb 45
```

Viewing Breakpoints

GDB consecutively numbers all breakpoints and watchpoints. After you have set a breakpoint, you can view the set of all breakpoints, regardless of the thread in which each breakpoint is set, by entering the 'info breakpoints' command (or the command synonym 'info watchpoints') in the console window.

```
(gdb) info break
```

For example, GDB produces this console window output in response to the 'info breakpoints' command:

```
(gdb) info breakpoints
Num Type           Disp Enb Address    What
1   hw breakpoint  keep y   0x00008010 in DCPmain at cpMain.c:78
   breakpoint already hit 1 time
2   watchpoint     keep y   bufPool
```

Using Watchpoints

Setting a *watchpoint* on a location (or range of locations) tells the debugger to stop program execution and notify you when the value in that location *changes*. More accurately, this is called a *write watchpoint*. (This version of the debugger does not support setting a *read watchpoint*, which stops program execution when the specified location is read during program execution.)

For example, the debugger can set a watchpoint at any location in the XPRC's DMEM, CPRC's DMEM, or any other globally addressable location.

In a debugger console window, use the **watch** command to set a watchpoint. For example:

```
(gdb) watch my_var
Watchpoint 2: my_var
```

Watchpoints are available only when using the **dcpsim** *target. Watchpoints are not available when debugging an application on a C-Port NP.*

Do not confuse a watchpoint with GDB's watch expressions*. Open the Watch Expressions window by pressing the Watch Expressions (eyeglasses icon) button on the button panel in a debugger source window. Expressions and values monitored in the Watch Expressions window are distinct from the variables associated with watchpoints.*

Depending on the level of optimization selected when your program was built, it is possible that certain variables coded in the program do not exist in the resulting image and therefore cannot be the object of a watchpoint.

Watchpoints on Arrays and Structures

You can set a watchpoint on a variable that names an array or structure. For example:

```
(gdb) whatis my_var
type = int[100]
(gdb) watch my_var
Hardware watchpoint 2: my_var
```

Watching arrays can cause the debugger to produce additional output in the console window regarding “read before write” warnings. This is because the debugger might read the entire array or structure at least once before enabling the watchpoint and before the program itself can inherit its values.

Using Watchpoints with a CP Cluster's Shared Program

When the members of a Channel Processor cluster are running the same program image, each CPRC has its own private DMEM, and there is a separate instance of each program variable stored in those DMEMs. Because the debugger views each CPRC as, in effect, running a distinct thread based on the program image, each watchpoint set for this program image must have its CPRC's thread ID associated with it.

For example, the debugger produces this output for a watchpoint set for a variable (bufPool) used in a program image that is shared among the members of a CP cluster:

```
(gdb) info watchpoints
Num Type           Disp Enb Address    What
1   hw breakpoint  keep y   0x00008010 in DCPmain at cpMain.c:78
   breakpoint already hit 1 time
2   hw watchpoint     keep y   bufPool
   stop only in thread 0
```

Stepping and Continuing Execution for a Simulator Target

You can use standard GDB commands or the C-Ware Debugger's GUI interface to examine the pertinent registers, stack frame, and objects in the current thread's DMEM resource, and so on.

When the debugger is interacting with a **dcpsim** target, if you issue standard GDB execution commands (*stepi*, *nexti*, *step*, *continue*, and so on), each causes all processors on the NP to execute. For instance, the *step* command causes the "current thread" to execute forward to the next source instruction (or until some other thread on the XP or a CP reports an event), but also causes *all other processors* (that have an executable image loaded) to execute *concurrently* for the same number of clock cycles.

The C-Ware Debugger's multicontinue *command behaves identically to the* continue *command, when the target is* **dcpsim**.

You should carefully watch the debugger's Console window while stepping in a given processor in Channel Processor cluster, because it might be interrupted and cause the debugger's current thread to be switched to another CP in that cluster.

Stepping and Continuing Execution for CDS Target

You can use standard GDB commands or the C-Ware Debugger's GUI interface to examine the pertinent registers, stack frame, and objects in the current thread's DMEM resource, and so on.

When the debugger is interacting with a **dcp** target, issuing standard GDB execution commands (*stepi*, *nexti*, *step*, *continue*, and so on) causes only the 'current thread' to execute. In contrast, the *multicontinue* command causes all threads on the targeted XP or CP, or cluster of Channel Processors, to execute.

After the Simulator has granted debugging control to the C-Ware Debugger and after you have set at least one breakpoint, you are ready to let the C-Port NP's XP and CPs execute their respective programs. To do so, enter this command:

```
(gdb) continue
```

When the targeted processor encounters a breakpoint, that processor stops its execution *and becomes the current thread*. However, the C-Ware Debugger also causes *all* other processors from the target processor's cluster to stop. The other executables halt after a minimal delay. This is consistent with GDB's standard behavior when debugging a multithreaded program.

As a result of hitting a breakpoint, the current thread changes, and the C-Ware Debugger displays this message:

```
(gdb) ...
Continuing.
[Switching to process 3]     // This refers to thread 3/CP3
```

At this point, you can issue standard GDB execution commands (stepi, nexti, step, continue, and so on) to continue execution of the current thread only. You can use standard GDB commands or the C-Ware Debugger's GUI interface to examine the pertinent registers, stack frame, and objects in the current thread's (that is, current CP's) DMEM resource, and so on.

Multicontinue starts all processors if there are no pending events. Pending events (like breakpoints) are handled sequentially.

Thus, after all CPs have stopped due to one of them encountering a breakpoint, if you use the *multicontinue* command with the intention of causing all CPs in that cluster to resume running, the C-Ware Debugger first checks whether any additional events are present in its incoming event queue. If so, the C-Ware Debugger causes all CPs in that cluster to remain stopped and immediately processes the next queued event. If the next queued event is the notice that a CP in the same cluster has encountered a breakpoint, then the C-Ware Debugger processes that breakpoint by updating the display in its source window (and other relevant windows) and makes that CP the current thread.

If there are no pending events when you issue the *multicontinue* command, the C-Ware Debugger allows all threads to continue executing concurrently.

If there are no pending events against the current thread when you issue the *continue* command, the C-Ware Debugger allows that thread to continue executing.

A thread (a CP in a CP cluster that shares IMEM) cannot continue if it has an event pending, and all threads (CPs in that cluster) cannot continue from all being stopped if any thread has an event pending.

Identifying the Package File

The XPRC or any CPRC can execute a *multi-phase* program. This is a program that has been partitioned into more than one executables, to conserve IMEM resource. That is, one executable uses a special calling interface to call another executable. One executable might perform initialization and other one-time operations and perform data forwarding operations in another executable.

The called executable is overlaid in IMEM over the calling executable. Global data structures that provide overall program context can be defined so that they persist across calls to subsequent executables for the same IMEM.

To allow proper operation of a debugging session for a multi-phase XPRC or CPRC program, the Debugger must access the package (**.pkg**) file as the next execuable is loaded. Thus, early in that debugging session the user must specify the path of the package file.

Use the Debugger's 'dcppkg' command to specify to the Debugger the location of the target program's package file. This command takes only one argument, which is a valid file path. For example:

```
(gdb) dcppkg foo.pkg
```

During a debugging session the 'dcppkg" command can be issued before the 'target' command is issued, or afterwards.

Debugging Overlaid XPRC and CPRC Executables

As described in the *C-Ware Application Building* document, the XPRC or any CPRC can execute a *multi-phase* program. This is a program that has been partitioned into more than one executables. That is, one executable uses a special calling interface to call another executable. One executable might perform initialization and other one-time operations and perform data forwarding operations in another executable. This allows the overall program to conserve its use of IMEM for any one executable.

The called executable is overlaid in IMEM over the calling executable. Global data structures that provide overall program context can be defined so that they persist across calls to subsequent executables for the same IMEM.

To allow proper operation of a debugging session for a multi-phase XPRC or CPRC program, the Debugger must access the package (**.pkg**) file as the next execuable is loaded. Thus, early in that debugging session the user must specify the path of the package file.

'dcppkg' Command

Use the 'dcppkg *package_file_path*' command to allow the C-Ware Debugger to switch between the executable images as they are loaded. For example:

```
(gdb) dcppkg ethernetSwitch.pkg
```

In this example **ethernetSwitch.pkg** is the name of a C-Port NP package file that contains all the relevant executable images.

Performing the 'dcppkg' command causes the Debugger to be notified each time a new executable image is loaded into XPRC's IMEM or into any CPRC cluster's IMEM, for the remainder of the same debugging session. The Debugger automatically stops execution of the XPRC and CPRC programs at the entry to each executable's *DCPmain()* routine, including for each subsequently loaded executable.

During a debugging session the 'dcppkg" command can be issued before the 'target' command is issued, or afterwards.

Qualifications for Debugging Multi-Phase Programs on CPRCs

When debugging multi-phase programs on CPRCs, observe the following qualifications:

- The Debugger assumes that all four CPRCs n a cluster are started and running. If the XP does not cause all CPRCs in the same cluster to start, the Debugger session will hang.
- There is no support in this CST release for debugging a CPRC cluster that started as four CPRCs running but at some point any of the CPRCs exited its program or was stopped by the XP (such as via the *ksProcStopCps()* routine).

Therefore, we recommended the following steps:

1. Set a breakpoint at the end of any CPRC executable image that causes another executable to be loaded.
2. When one of the CPRCs in a cluster hits that breakpoint (that is, reaches a point just before program exit), *remove all other breakpoints* from the program (using the Debugger's 'del' command) then continue program execution. The result is that the Debugger can next encounter only one breakpoint: the breakpoint set automatically by the Debugger at the beginning of *DCPmain()* in the next overlaid executable.

Debuggin a CPRC that is stopped by direction of the XP is a more difficult scenario. In this case the user should ensure that the XP does not, in fact, cause the CPRCs to stop during the debugging session, or the user must prepare the debugging session to prevent the program's execution from allowing the XP to stop the CPRCs.

Debugging on Multiple C-Port NP Devices Concurrently

It is possible to have C-Ware Debugger sessions underway simultaneously to each of multiple C-Port Network Processor (NP) targets, for instance in a C-Ware Development System (CDS). That is, the user can establish distinct debugging sessions on the XP and/or any CPRC cluster on one C-Port NP as well as on the XP and/or any CPRC cluster on one or more other NPs in the same (CDS or other user system) chassis.

To accomplish this, qualify the CP ID (third argument) of the Debugger's 'target' command with the DCP ID number, as follows:

```
// Connect to the XP on dcp0 (implicit)
   (gdb) target dcp 192.168.254.1 25

// Connect to the XP on dcp0 (explicit)
   (gdb) target dcp 192.168.254.1 25:0

// Connect to the XP on dcp1 (explicit)
   (gdb) target dcp 192.168.254.1 25:1

// Connect to the CPRC cluster for CP0 on dcp1 (explicit)
   (gdb) target dcp 192.168.254.1 0:1
```

As shown in these examples, if you omit the DCP ID qualification in the third argument, the Debugger defaults to a target on DCP0.

Remote Debugging of C-Port NP Devices

Remote debugging of the C-Port NP devices in a C-Ware Development System (CDS) is described in the *C-Ware Development System User Guide* document.

Debugging After a Panic on C-Port NP Hardware

Debugging after the occurence of a panic state in a program running on C-Port NP hardware in a C-Ware Development System (CDS) is described in the *C-Ware Development System User Guide* document.

Debugging in this scenario requires the use of the C-Ware Debugger's 'ic' and 'uc' commands:

- 'ic' command — When debugging a C-Ware application running on C-Port NP hardware and after a "panic" condition (that is, a call to the C-Ware API *ksPanic()* routine) has occurred in that application, use this command to access the register data, including the stack contents, of the offending XPRC's or CPRC's context 0 (the "interrupt context" by C-Ware convention).
- 'uc' command — When debugging a C-Ware application running on C-Port NP hardware and after a "panic" condition (that is, a call to the C-Ware API *ksPanic()* routine) has occurred in that application, use this command to access the register data, including the stack contents, of the offending XPRC's or CPRC's contexts 1 to 3 (the "user contexts" by C-Ware convention).

Debugging of XpBootIROM Code

Debugging XpBootROM code is a somewhat special case.

Disassembling the In-Memory Image Code

Remember that you will not have access to any symbolic information, unless you provide it explicitly to the C-Ware Debugger. Similarly, the C-Ware Debugger cannot provide any source code display for the in-memory image; rather, you must explicitly disassemble the executable code. For example:

```
(gdb) disassemble 0xbfff0000 0xbfff0010
Dump of assembler code from 0xbfff0000 to 0xbfff0010:
0xbfff0000:     lui     $a3,0xbd81
0xbfff0004:     lw      $a3,-32000($a3)
0xbfff0008:     move    $zero,$zero
0xbfff000c:     lw      $a0,8($a3)
End of assembler dump.
```

To automatically track the current location in the assembly code, use the display command; for example:

```
(gdb) display/i $pc
1: x/i $pc  0xbfff0000: lui     $a3,0xbd81
```

Stepping Through the Code

The output allows you to track (in a disassembled form) where the program counter (PC) is pointing. Next, you can single-step through the code; for example:

```
(gdb) stepi
0x0 in ?? ()
1: x/i $pc  0x0:        nop    # Pipeline needs to be filled
(gdb) stepi
warning: Hit heuristic-fence-post without finding
warning: enclosing function for address 0xbfff0000
This warning occurs if you are debugging a function without any symbols
(for example, in a stripped executable).  In that case, you may wish to
increase the size of the search with the ‘set heuristic-fence-post’
command.

Otherwise, you told GDB there was a function where there isn’t one, or
(more likely) you have encountered a bug in GDB.
0xbfff0000 in ?? ()
1: x/i $pc  0xbfff0000: lui     $a3,0xbd81
(gdb) stepi
warning: Hit heuristic-fence-post without finding
warning: enclosing function for address 0xbfff0004
0xbfff0004 in ?? ()
1: x/i $pc  0xbfff0004: lw      $a3,-32000($a3)
(gdb) stepi
warning: Hit heuristic-fence-post without finding
warning: enclosing function for address 0xbfff0008
0xbfff0008 in ?? ()
1: x/i $pc  0xbfff0008: move    $zero,$zero
(gdb)
```

Setting a Breakpoint

You can also set a breakpoint in the in-memory image code. For example:

```
(gdb) b *0xbfff0038
Hardware assisted breakpoint 1 at 0xbfff0038
(gdb) c
Continuing.
warning: Hit heuristic-fence-post without finding
warning: enclosing function for address 0xbfff0038
This warning occurs if you are debugging a function without any symbols
(for example, in a stripped executable). In that case, you may wish to
increase the size of the search with the ‘set heuristic-fence-post’
command.

Otherwise, you told GDB there was a function where there isn’t one, or
(more likely) you have encountered a bug in GDB.
Breakpoint 1, 0xbfff0038 in ?? ()
(gdb)
```

Notice the warnings in the sample output above. These warnings appear when stepping through code that does not have associated debugging information. XpBootIROM code does not have debugging information.

Chapter 3

TROUBLESHOOTING WHILE DEBUGGING

Debugging Infrastructure Overhead

To enable debugging of executables running on the XPRC and the CPRCs, the Debugger builds a certain amount of “infrastructure” (or “overhead”) code into a debugging executable image.

In each CST version this overhead is kept to a minimum, but can change from CST version to version. See the *C-Ware Software Toolset Release Notes* document, which part of the CST documentation set that is included with your CST package, for any statement of any change in the size of this overhead from CST version to version.

This debugging infrastruction code exists in these forms:

- In any XPRC or CPRC executable, initialization code that is executed at the beginning of any debuggable executable image
- In any CPRC executable, the *RC Interface* code that allows the debug state of a CPRC program to be communicated to the XPRC and, in turn, to the user (for an actual C-Port Network Processor device, across the NP’s PCI interface to the host processor and then to the user).

Effects of Optimization

Motorola recommends using optimization level -O0 for all code to be debugged. -O0 optimization may not be usable in all cases, however, due to memory limitations. In that case compile the module to debug with -O0 optimization, and all others with -O3 optimization.

If this is impossible or you prefer not to build special versions of your program just for debugging, keep in mind the following:

- In your executable, variables are passed in registers whenever possible, and those registers might later be used in various expressions that can change their value and effectively "lose" (from the debugger's point of view) the original contents.
- Entire blocks of code, including functions (and not only in-line functions), might be "optimized out" of your executable file. That is, the generated code might show little trace of its source code form to the debugger.

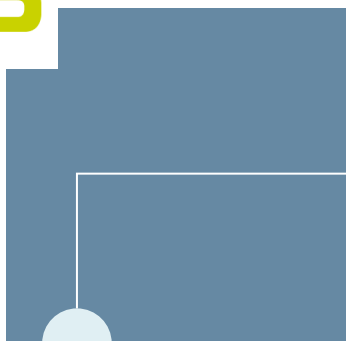
Avoid Stepping Over or Into Context Switches

If you direct the C-Ware Debugger to step over a call to *ksContextSwitch()* API routine, it begins single-stepping. At the point the code steps into this routine, the debugger realizes that it is dealing with a function call. Thus, it sets a breakpoint where it expects the program to return and issues a "continue" request. However, the program itself has switched register contexts and has jumped to the code for that context.

What happens next depends on many things. In its new context if the program starts looping while waiting on a piece of data to arrive or for some other event, and that event does not yet occur, then the program will not (if ever) return to the point that the C-Ware Debugger was expecting. Thus, the C-Ware Debugger waits for the "return" breakpoint to be encountered, but this event doesn't occur.

Similarly, when stepping over function calls that loop while waiting for data, the C-Ware Debugger encounters a similar scenario. This scenario should be considered commonplace in a C-Port NP's multiprocessor run-time environment, where CPs work together to perform particular tasks.

The answer: Single-step by assembly instructions through the call to *ksContextSwitch()*, and your debugging session can emerge in the destination context where you can resume debugging in source mode.



INDEX

Motorola, Inc. C-Port Family of Network Processors
120 Water Street, No. Andover, MA 01845 Voice: (978) 773-2300 FAX: (978) 773-2301