AMCC 405EP
PowerPC

Document Issue 1.00

September 2004

PPC405EP Evaluation Board Kit
User's Manual

AMCC
APPLIED MICRO CIRCUITS CORPORATION

AMCC reserves the right to make changes to its products, its datasheets, or related documentation, without notice and warrants its products solely pursuant to its terms and conditions of sale, only to substantially comply with the latest available datasheet. Please consult AMCC's Term and Conditions of Sale for its warranties and other terms, conditions and limitations. AMCC may discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information is current. AMCC does not assume any liability arising out of the application or use of any product or circuit described herein, neither does it convey any license under its patent rights nor the rights of others. AMCC reserves the right to ship devices of higher grade in place of those of lower grade.

AMCC SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

PowerPC® 405EP

Evaluation Board Kit

User's Manual

Preliminary

SA14-2707-00

PowerPC®

First Preliminary Edition (December 2002)

This edition of the *IBM PPC405EP Evaluation Board Kit User's Manual* applies to the IBM PPC405EP 32-bit embedded controller, until otherwise indicated in new versions or application notes.

IBM Microelectronics Division
1580 Route 52, Bldg. 504
Hopewell Junction, NY 12533-6351

The IBM home page can be found at http://www.ibm.com

The IBM Microelectronics Division home page can be found at http://www.ibm.com/chips

Contents

Figures

Tables

About This Book

This book contains the information you need to install and use the IBM® PowerPC PPC405EP Evaluation Board Kit, a hardware and software development tool for the PowerPC PPC405EP 32-bit RISC microprocessor.

The PowerPC PPC405EP Evaluation Board Kit (hereinafter referred to as the PPC405EP evaluation board kit) hardware includes the PowerPC 405EP Evaluation Board (hereinafter referred to as the evaluation board), power supply, and board interface cables. Features of the evaluation board include a PowerPC PPC405EP processor, 128MB SDRAM, four 32-bit PCI slots, built-in Ethernet support, 512KB socketed flash memory, 512KB SRAM, 2 serial ports, and a time-of-day clock with 8KB NVRAM.

The PPC405EP evaluation board kit software includes the ROM Monitor (resident in the flash memory on the board), ROM Monitor source code, IBM's OS Open real time operating system, sample application programs, application development libraries and tools, IBM's High C/C++ compiler, and IBM's RISCWatch, a source-level debugger that runs on the host system.

The PPC405EP evaluation board kit also includes technical specifications and board schematics.

Connection of the evaluation board to a host system is required for the exercises in this book. Supported host systems include:

- An IBM or compatible PC running one of the following:
  - Windows 95/98/ME
  - Windows NT 4.0/Windows 2000/XP

Who Should Use This Book

This book is for hardware and software developers who need to evaluate the PowerPC PPC405EP microprocessor and use the debugging features of the PowerPC PPC405EP Evaluation Board Kit to support software development.

Users should understand hardware and software development tools, concepts, and environments. Specifically, users should understand:

- The host's operating system
- The PowerPC Architecture™ and implementation-specific characteristics of the PowerPC microprocessor being used
- C and Assembler language programming

How to Use This Book

This book contains the following chapters and appendixes:

Chapter 1, "Overview of the Evaluation Board Kit" describes the product, its hardware and software components, and its relationship with the software tools on the host.

Chapter 2, "Host System Requirements" lists the hardware and software requirements of the host system.

Chapter 3, “Installing the Software” describes the software installation on the host system.

Chapter 4, “Host Configuration” describes the steps required to facilitate communications between the host computer and the evaluation board.

Chapter 5, “Hardware” describes the evaluation board, its memory map, its hardware components and their functions.

Chapter 6, “Board Connectors” describes the evaluation board connectors and the procedures for connecting the board to a host system.

Chapter 7, “ROM Monitor” describes the operations of the ROM monitor.

Chapter 8, “Sample Applications” describes how to compile, load, and run the sample applications on the evaluation board.

Chapter 9, “Application Libraries and Tools” describes the application libraries and host tools provided with the evaluation board software.

Chapter 10, “OS Open Function Reference” lists the OS Open functions for the PowerPC 405EP Evaluation Board platform. The function calls are arranged alphabetically by function name.

Appendix A, “Program Trace Calls” describes the messages for interfacing a debugger on the host system to the ROM Monitor on the evaluation board.

Appendix B, “ROM Monitor Load Format” describes the load format requirements supported by the ROM monitor.

Conventions Used in This Book

This book follows the numeric and highlighting notation conventions based on those used in the RISC System/6000 and AIX publications.

Numeric Conventions

In general, numbers are used exactly as shown. Unless noted otherwise, all numbers are in decimal, and, if entered as part of a command, are entered without format information.

In text, binary numbers are preceded by a “B” followed by the number enclosed in single quotes, for example:

B'010'

In commands, binary numbers are preceded by “0b” or “b” followed by the number, which may be enclosed in single quotes, for example:

0b010 or b’010’

In text, hexadecimal numbers are preceded by an “X” followed by the number enclosed in single quotes, for example:

X'1A7'

In commands, hexadecimal numbers are preceded by “0x” or “x” followed by the number, which may be enclosed in single quotes, for example:

0x1a7 or x'1a7'

In text, the hexadecimal digits A through F appear in uppercase. In commands, these digits are typically entered in lowercase.

Highlighting Conventions

This book uses the following highlighting conventions:

The names of invariant objects known to the software appear in bold type. In some text, however, such as in lists, no special typographic treatment is used. Examples of such objects include:

- Function and macro names
- Data types and structures
- Constants and flags

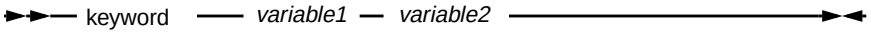
Names of objects known to the software must be entered exactly as shown.

- Variable names supplied by user programs appear in italic type. In some text, however, such as in lists, no special typographic treatment is used. Examples of these objects include arguments and other parameters.
- No highlighting appears in code examples.

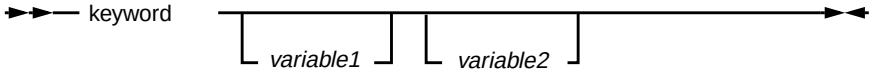
Syntax Diagram Conventions

Throughout this book, diagrams illustrate the syntax for string formats and commands. The following list shows how to read these diagrams:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
- A ►►— symbol begins a diagram.
- A —► symbol indicates continuation of a diagram on the next line.
- A ►— symbol indicates continuation of a diagram from the previous line.
- A —►◄ symbol terminates a diagram.
- Keywords are in regular type, and variables are in italics. Keywords must be typed exactly as shown.
- Keywords or variables on the main path of a diagram are required.

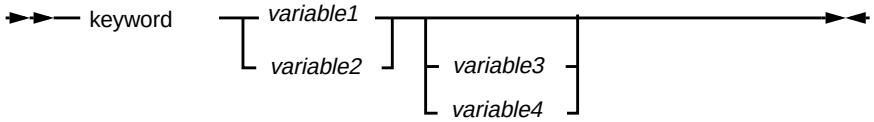


- Keywords or variables shown on branches below the main path are optional.

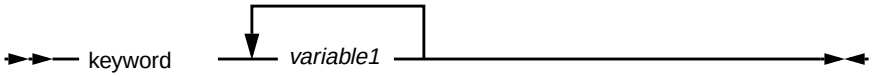


- Keywords or variables can appear in a stack, indicating that only one item in a stack can be chosen. If an item in a stack is on the main path, you must choose an item from the stack. If all items in a stack are below the main path, you may choose an item from the stack.

For example, in the following syntax diagram, you must choose either *variable1* or *variable2.* However, because *variable3* and *variable4* are below the main path, neither is required.



- A repeat separator is a returning arrow that surrounds a syntax element or group and shows that the element or group can be repeated.



Contacting the IBM Embedded Systems Solution Center

For information about the PowerPC PPC405EP Evaluation Board Kit and the IBM family of hardware and software products for embedded system developers, call the IBM Embedded Systems Solution Center at (919) 543-5701, or check out the IBM Microelectronics web site at:

http://www.chips.ibm.com/products/embedded

Please send any comments or questions regarding this product to the following Internet address:

ppcsupp@us.ibm.com

Related Publications

Many of the following publications are included on the CD ROM that comes with the evaluation kit. The others are available from your IBM Microelectronics representative:

- Embedded Application Binary Interface (EABI) Publications

  PowerPC Embedded Application Binary Interface (EABI)

  System V Application Binary Interface, Third Edition, 0-13-0100439-5

  System V Application Binary Interface, PowerPC Processor Supplement

- IBM High C/C++ Publications

  The following list includes the books in the IBM High C/C++ library:

  IBM High C/C++ Programmer's Guide for PowerPC, 92G6920

  IBM High C/C++ Language Reference for PowerPC, 92G6923

  IBM ELF Assembler User's Guide for PowerPC, 92G6921

  IBM ELF Linker User's Guide for PowerPC, 92G6922

- OS Open Publications

  IBM OS Open Programmer's Reference, Volume 1, 92G6911

  IBM OS Open Programmer's Reference, Volume 2, 92G6912

  IBM OS Open User's Guide, 92G6897

- RISCWatch Debugger Publications

  RISCWatch Debugger User's Guide, 13H6964

  RISCWatch Debugger Installation Guide, 13H6984

- PowerPC PPC405EP Publications

  PowerPC 405EP Embedded Processor Data Sheet

  PowerPC 405EP Embedded Processor User's Manual

- Evaluation Board Publications

  PowerPC 405EP Evaluation Board Manual

Chapter 1. Overview of the Evaluation Board Kit

This chapter introduces the hardware and software components included in the PPC405EP evaluation board kit.

1.1 Hardware Components

The PPC405EP evaluation board kit contains the evaluation board, power supply, power supply line cord, serial port and Ethernet cables.

1.1.1 Evaluation Board

Features of the evaluation board include the PowerPC PPC405EP processor, 128MB SDRAM, four 32-bit PCI slots, 2 built-in Ethernet ports (10BaseT/100BaseTX), 512KB socketed flash memory, 512KB SRAM, 2 serial ports, a time-of-day clock with 8KB NVRAM, and a I^2C port.

For a detailed description of the evaluation board, refer to the PowerPC 405EP Evaluation Board Manual.

1.1.2 Cables and Power Supply

The PPC405EP evaluation board kit includes a serial port interface cable for connecting the board's serial port 1 to a terminal or terminal emulator running on the host.

An Ethernet crossover cable is provided in the kit to support direct Ethernet communication with the host system. Standard 10BaseT/100BaseTX RJ45 Ethernet connectors are provided on the evaluation board. The Ethernet crossover cable is for direct connection to a single host and *cannot* be used with a hub or a building's Ethernet network. The crossover cable is only supported for 10Mb/s operation - for 100Mb/s a hub (not supplied) should be used.

A power supply line cord is also provided with the PPC405EP evaluation board kit.

1.2 Software Components

The PPC405EP evaluation board kit software consists of the Board Support Package (BSP), the RISCWatch source level debugger, and the IBM High C/C++ evaluation compiler.

1.2.1 BSP Software

The BSP software includes the ROM Monitor code resident in flash memory, ROM Monitor source code, the IBM OS Open real time operating system, several sample programs (including the Dhrystone benchmark program), and application development libraries and tools.

1.2.1.1 ROM Monitor

The ROM Monitor program for the evaluation board is supplied in the 512KB socketed flash memory module on the evaluation board. This code initializes the 405EP processor and the board for serial and Ethernet communications. By supporting communications with the host computer system, the

ROM Monitor allows applications to be loaded from the host onto the board and debugged using the RISCWatch source level debugger in ROM Monitor mode.

The ROM Monitor is accessed through a terminal (or terminal emulator) attached to serial port 1 on the board. The RISCWatch debugger, when in ROM Monitor mode, runs on the host system, communicating with the ROM Monitor through an Ethernet interface on the board.

The ROM Monitor source code is provided and can be readily used for product development. The availability of the code helps lower software development costs and quicken product time to market. The code is also provided so that debuggers other than RISCWatch may be integrated with the PPC405EP evaluation board kit. Appendix A describes the trace calls that support communication between the RISCWatch debugger on the host and the ROM Monitor running on the board.

1.2.1.2 OS Open Real-Time Operating System

OS Open is a real-time operating system (RTOS) available for the PowerPC 400, 600, and 700 families of processors. OS Open is designed to take full advantage of the power of the IBM PowerPC RISC processors. Also, because the OS Open environment is built in a scalable fashion, it can be configured to meet the functional requirements and memory constraints of a wide variety of embedded systems.

OS Open features:

- Hard real-time support, including deterministic execution, priority inheritance protocols, and priority ceiling protocols
- Board support packages for plug-and-play operation of popular board-level products
- Support for existing American National Standards Institute (ANSI) C and emerging POSIX standards
- Open network interfaces to support embedded systems in heterogeneous environments
- Scalable implementations to meet the requirements and constraints of a variety of embedded systems

The version of OS Open included in the BSP software contains a reduced-function kernel that limits the number of threads that can be in existence at one time. Additional details can be found in the README file following software installation. A full-function OS Open kernel is available from IBM. Contact the IBM Embedded Systems Solutions Center at (919) 543-5701 for additional information.

1.2.1.3 Dhrystone Benchmark Program

The Dhrystone benchmark is a commonly available integer benchmark. It is included as an example program to be built, loaded onto the board, and executed. The results of this benchmark may vary based on compiler options and the system environment in which it is run.

1.2.1.4 Application Tools

Several host-based tools are provided to support ROM and application development on the evaluation board.

1.2.2 RISCWatch Debugger

The RISCWatch source level debugger provides a window-based debugging environment for loading, debugging, and executing application programs on the board. Debugger installation and usage for

ROM Monitor and OS Open (non-JTAG) targets are addressed in the *RISCWatch Debugger Installation Guide* and the *RISCWatch Debugger User's Guide* included on the publications CD-ROM in the kit. A sample debug session is included with the debugger.

1.2.3 IBM High C/C++ Evaluation Compiler

The IBM High C/C++ compiler is a globally optimizing compiler developed for the PowerPC family of processors. It produces executable code in Extended Link Format (ELF) file format. The version included in the kit is a limited capacity version created specifically for the PPC405EP evaluation board kit. It supports the compilation, assembly, and linkage of the sample application programs and the ROM Monitor source code. A full featured version of the IBM High C/C++ compiler is available from IBM. For more information contact the PowerPC Embedded Systems Solutions Center at ppcsupp@us.ibm.com.

Chapter 2. Host System Requirements

This chapter describes the hardware and software requirements of the host system to which the evaluation board is to be connected. Supported host systems include:

- IBM (or compatible) PC running one of the following:
  - Windows 95/98/ME
  - Windows NT 4.0/2000/XP

2.1 PC Host System Requirements

Hardware requirements of the host PC include:

- IBM or compatible system unit. Minimum requirements: x486 DX2 50/66 MHz with 8 MB of RAM
- VGA/SVGA Display Monitor. Minimum requirement: VGA 640 x 480. Recommended: SVGA 1024 x 768
- Approximately 50 MB of free disk space. This space is required for the IBM High C/C++ compiler, the Board Support Package software, and the RISCWatch debugger. When planning disk space usage, consider disk space requirements for Windows and any other software packages.
- At least one available serial port for terminal emulation. Establishing an Ethernet host-to-board connection will most likely require the installation of an Ethernet adapter card on the host (if not already installed) and some additional connectivity hardware. That hardware might include any or all of the following:
  - An Ethernet 10BaseT/100BaseTX network transceiver, a twisted pair cable, and a hub. At a minimum, a point-to-point connection will require the Ethernet crossover cable supplied with the kit. The Ethernet crossover cable is for direct connection to a single host and *cannot* be used with a hub or a building's Ethernet network. The crossover cable is only supported for 10Mb/s operation - for 100Mb/s a hub (not supplied) should be used.

The following software must be installed on the host PC to run the debugger that communicates with the ROM Monitor on the board:

- RISCWatch 5.0 or higher
- Windows 95/98/ME or Windows NT 4.0/2000/XP

Chapter 3. Installing the Software

This chapter describes the procedures for installing the BSP software and the High C/C++ Compiler on the host system. Details of the software, its directories and their contents, are also given. Instructions for installing the RISCWatch Debugger software can be found in the *RISCWatch Debugger Installation Guide*. Please refer to the section corresponding to your host system.

3.1 PC Software Installation

Before beginning the installation, you must have:

- BSP for PC installation CD
- A PC running Windows 95/98/NT 4.0/2000

The following procedure installs the BSP software:

Note: For Windows NT/2000 users, we recommend that you log on as **administrator**.

1. Insert the CD into the CD drive (assumed from here on to be drive “D”). This should automatically run the install program. If it does not, proceed to step 2.
2. Select **Start** from the Windows task bar.
3. Select **Run**.
4. Type **D:\setup** then press **Enter** to run the installation program.
5. Follow the installation program instructions. A typical install will install the BSP Software, the HighC/C++ Compiler, and RISCWatch. The default install directory is **\Program Files\IBM\405EP_EvalKit**.

3.1.1 BSP Software Installation

If the default install directory is accepted, the BSP software is installed in the **osopen** subdirectory tree under install directory.The **osopen** directory tree contains the files and tools that support OS Open application and ROM development. The **osopen** subdirectories and their contents are as follows.

\bin

This directory contains several host based utilities used for application and ROM program development.

- **elf2rom.exe** - creates a ROM image from an ELF executable file
- **eimgbld.exe** - creates a ROM Monitor loadable image from an ELF executable file
- **hbranch.exe** - places an absolute branch in the last address of a ROM image
- **rambuild.exe** - creates an assembler source file that contains the files found in a specified directory
- **gnumake.exe** - supports the use of *makefiles* when building application programs
- **bootpd.exe** - bootp server to support ROM Monitor downloads

- **tftpd.exe** - tftp server to support host-to-board file transfers

\ld

Contains dynamically loadable modules that can be run from OS Open's OpenShell

\m405h_evb

This directory contains the ROM Monitor and OS Open platform specific code for the evaluation board included in the kit.

- **README.TXT** - contains the latest information regarding this release
- **\include** - contains OS Open include files
- **\lib** - contains OS Open libraries
- **\m4** - contains assembler preprocessor include files
- **\openbios** - contains the source code for the ROM Monitor (See Chapter 7, "ROM Monitor")
- **\samples** - contains samples programs that can be compiled and run
- **\make -** contains parameters used by various makefiles

Considerable effort goes into providing a quality product with consistent documentation. To insure that our customers have the advantage of the latest software features and updated information, **README.TXT** contains clarifications and additional information and should be considered essential reading.

Providing Feedback

Please provide any feedback to ppcsupp@us.ibm.com. Your feedback and suggestions will help us to improve our products and technical publications.

3.1.2 High C/C++ Evaluation Compiler Installation

The IBM High C/C++ Compiler is installed in the **highcppc** directory tree under the base installation directory. The **highcppc\bin** directory contains the files required for the IBM High C/C++ Compiler. Those files include:

- **asppc.exe** - assembler for assembler language programs
- **ldppc.exe** - ELF linker/binder to build applications to be run on the board
- **hcppc.exe** - High C/C++ compiler for C programs
- **arppc.exe** - ELF library archiver

The readme file under the **highcppc** directory contains the latest information regarding the compiler and should be considered essential reading.

3.1.3 RISCWatch Debugger Installation

RISCWatch is installed under the RISCWatch directory tree under the base installation directory. Proper environment setup is required before initial usage. Please refer to the *RISCWatch Debugger Installation Guide* for debugger environment setup instructions.

Chapter 4. Host Configuration

Several host configuration steps are required to facilitate communications between the host computer and the board. These steps are outlined in this chapter. Please refer to the section corresponding to your host system.

4.1 PC Host Configuration

The following sections discuss setup of host configuration for PC hosts.

4.1.1 Serial Port Setup - PC

Most PCs include two serial ports to support communications via asynchronous data transfer. These ports are sometimes referred to as communication or COM ports. These ports are usually accessed from the back of the system unit. You should consult your PC literature to determine how many serial ports are available on your unit and where they are located. In this section, S1 and S2 refer to the respective serial ports on the host PC, and SP1 and SP2 to the respective serial ports on the board.

One serial port should be used to connect a terminal emulator running on the host to the ROM Monitor running on the board. This section addresses the proper configuration of the S1 serial ports to support this connection. Users should also refer to the Windows on-line help for “Changing Serial Port Settings”.

The connection of the terminal emulator running on the host to the ROM Monitor running on the board, is made through the S1 serial port on the PC and the SP1 (J7 lower) serial port on the board. The S1 port must be configured for a baud rate of 9600, 8 data bits, 1 stop bit, and no parity. The proper setting of these parameters is discussed later in the section on terminal emulation.

4.1.2 Ethernet Setup - PC

An Ethernet connection can be used for host-to-board communications. The Ethernet connection is made through an Ethernet adapter on the host and the 10BaseT/100BaseTX Ethernet port (J18) on the board. Ethernet is is recommended when downloading large applications on to the board or when using the RISCWatch debugger.

An Ethernet connection may require additional hardware. The evaluation board supports a standard Ethernet, twisted pair (10BaseT/100BaseTX) connection. This connection requires that the host PC be equipped with an appropriate Ethernet adapter. The host adapter is not included in the kit. Please consult your PC and adapter documentation for requirements and installation instructions.

At a minimum, a 10BaseT/100BaseTX connection requires a crossover Ethernet twisted pair cable (included in the kit) for point-to-point communications. The Ethernet crossover cable is for 10Mb/s direct connection to a single host and *cannot* be used with a hub or a building’s Ethernet network. If you want more than two nodes, or 100Mb/s connectivity, you will need a hub and straight-through twisted pair cables.

Other hardware required will depend on the type of Ethernet adapter you have on your PC and whether the board is being connected to an existing Ethernet network. Please consult your system administrator and the documentation included with the adapter hardware for additional instructions.

Establishment of an Ethernet interface requires a host IP address. If the host PC is connected to an existing Ethernet network, the host IP address should already be defined and there is no need to set it again. Consult your network administrator on how to obtain the host's Ethernet IP address and how to add the board to the existing network.

To set the host IP address for the Ethernet connection:

1. Select the My Computer icon from the desktop.
2. Select Control Panel.
3. Select Network.
4. Add the appropriate Adapter network component for the Ethernet adapter being used (if not already added).
5. Add a Protocol network component of Microsoft - TCP/IP' (if not already added). Specify the IP address (*7.1.1.4* is recommended to maintain consistency with this document) and netmask (*255.255.240.0*) to be used.

For the update to take effect, TCP/IP may need to be restarted. This may require a reboot of the system and/or a restart of TCP/IP. Make a note of the host IP address assigned to the Ethernet adapter, as this value will need to be made known to the ROM Monitor on the board.

4.1.3 ROM Monitor-Debugger Communication Setup - PC

Before the RISCWatch Debugger can be used, some additional steps need to be taken to establish ROM Monitor-Debugger communications. These steps involve an update of the TCP/IP **services** file to establish a named communications port and port number for TCP/IP socket communications, and a restart of the TCP/IP package for the update to take effect.

Windows 95/98 places the **services** file under C:\WINDOWS\SERVICES. Windows NT places the **services** file under C:WINDOWS\SYSTEM32\DRIVERS\SERVICES. Users should consult their TCP/IP documentation or system administrator if they can not locate the file. The following lines must be added to the file:

```
osopen-dbg    20044/tcp     # for RISCWatch OS Open debug
osopen-dbg    20044/udp     # for RISCWatch rom_mon debug
```

For the update to take effect, TCP/IP needs to be restarted. This might require a reboot of the system and a restart of the TCP/IP package.

Chapter 5. Hardware

The PPC405EP evaluation board kit includes the evaluation board which contains the following features:

- PowerPC PPC405EP processor, which includes:
  - PowerPC 405 core
  - Two 10BaseT/100 Base TX (RJ45) Ethernet Ports
  - Two 16550-type serial ports
  - IIC (I^2C) port
  - General Purpose Timers
  - Interrupt Controller
  - PC-133 SDRAM Controller
  - DMA controller
  - ROM/Peripheral controller
  - Internal PCI Controller
  - General-purpose I/Os
- Memory
  - 128MB SDRAM, single DIMM socket, support up to 128MB
  - 512KB socketed flash memory
  - 512KB SRAM
- Real-time clock with 8KB NVRAM and battery-backup
- Four 32-bit PCI connectors

For detailed descriptions of the evaluation board specifications, features, and its memory mapping, please refer to the *PowerPC 405EP Evaluation Board Manual*.

Chapter 6. Board Connectors

For detailed descriptions of the connectors and jumpers on the evaluation board, please refer to the *PowerPC 405EP Evaluation Board Manual*.

6.1 Connecting the Evaluation Board to the Host

To establish a working environment, the evaluation board must be connected to a host system. ROM Monitor access requires a connection between the serial port on the board (J7 lower) and the S1 (COM1) serial port on the host. Users must also establish a connection for debug and downloading applications from the host to the board. This connection is made over the Ethernet network established during host configuration.

Included in the PowerPC PPC405EP Evaluation Board Kit is an interface cable supporting either 9-pin or 25-pin serial port connections. Assuming a terminal emulator running on the host is going to be used for ROM Monitor access, connect the 9-pin serial port connector on one end of a cable to the J7 lower serial port connector on the board, and the other end of the same cable to the S1 (COM1) serial port on the host. The host end might require a serial port adapter (not supplied) for connectivity.The Ethernet connection can be made in two ways. If the connection is to be used exclusively between the host and the board, and only 10Mb/s speed is required, the provided crossover cable can be used to directly connect the two nodes. Otherwise, a 10BaseT or 100BaseTx hub (not provided) must be used to connect the nodes together.

Note: The Ethernet 10BaseT crossover cable supplied will not work if plugged into a hub.

Figure 6-1 shows the connections and signal assignments required in a crossover cable:

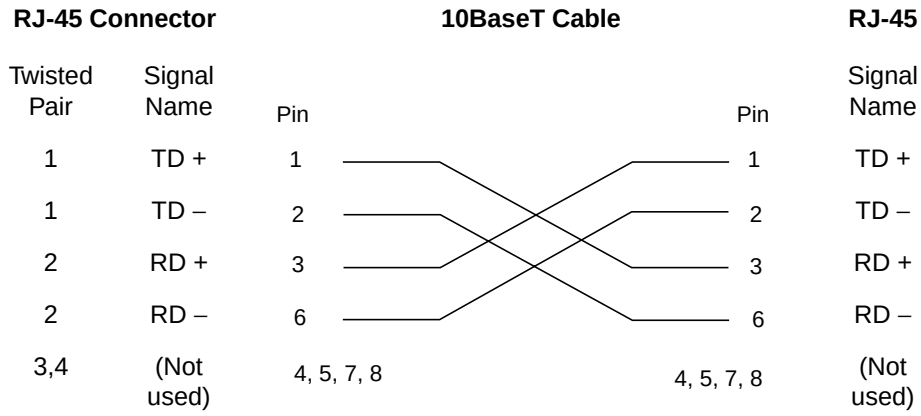


Figure 6-1. Wiring in a Crossover Cable

Figure 6-2 shows a point-to-point Ethernet connection using the provided crossover cable:

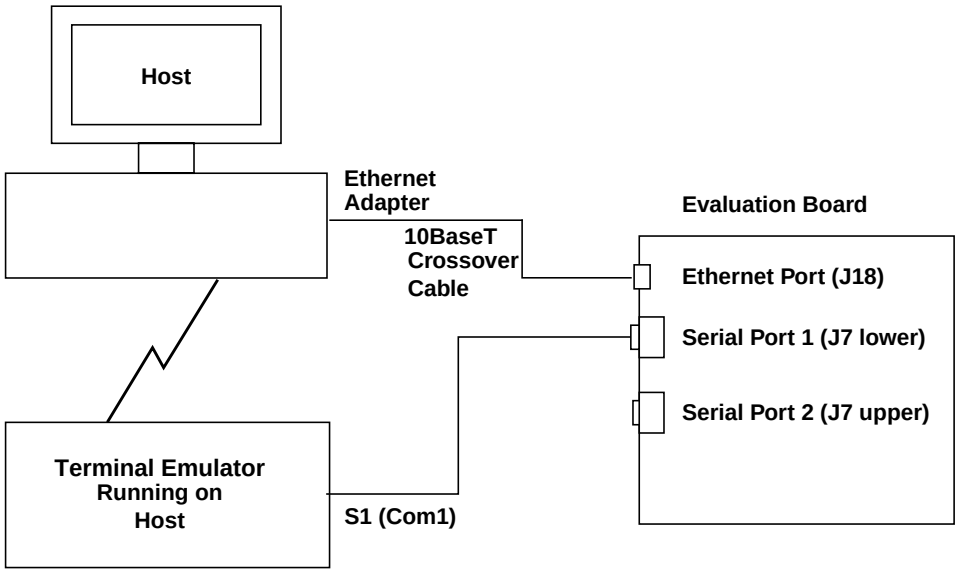


Figure 6-2. Point-to-Point Ethernet Connection

Figure 6-3 shows an Ethernet connection using a hub:

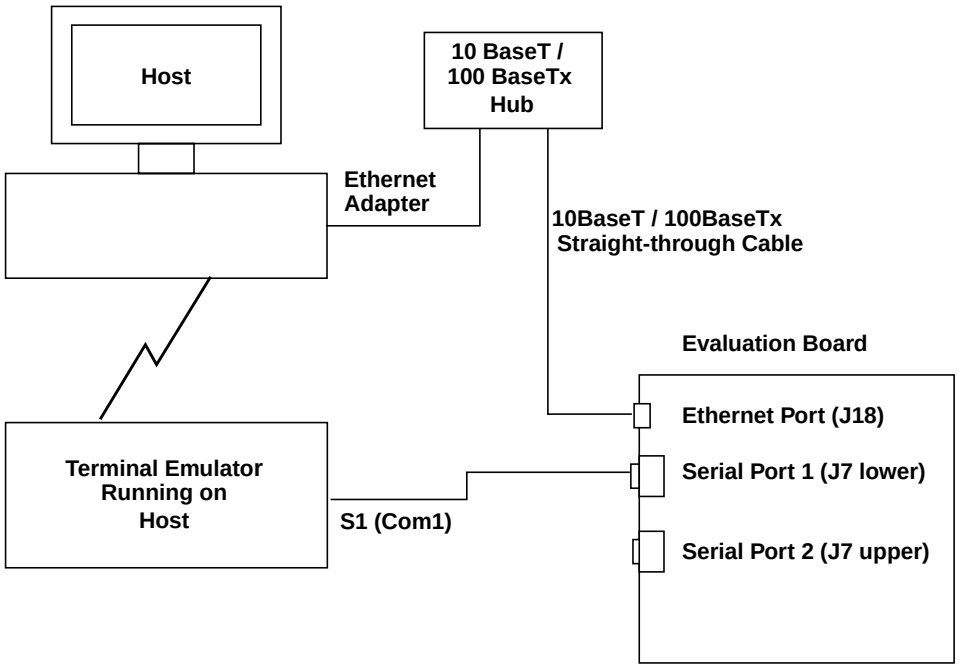


Figure 6-3. Ethernet Connection with Hub

If the connection is to be made to an existing Ethernet network, users should consult their Network Administrator to insure proper connectivity.

6.2 Using a Terminal Emulator

The ROM Monitor transmits/receives data through serial port 1 (J7 lower) on the board. Access to the ROM Monitor can be achieved by connecting a VT100 (or compatible) terminal directly to serial port 1 (J7 lower) on the board or by using a terminal emulator running on the host. When using a terminal emulator, access is obtained via a connection between the serial port 1 connector on the board and an available serial (or COM) port on the host system.

6.2.1 PC Terminal Emulation

Once all the host-to-board connections have been properly made and power has been supplied to the board, the Windows HyperTerminal program can be used as a terminal emulator to support communications with the ROM Monitor. The steps for setting up the terminal emulator connected to COM1 are as follows:

1. Select **Start** from the Windows task bar.
2. Select **Programs**.
3. Select **Accessories**.
4. Select **HyperTerminal**.
5. If you see a window that says "You need to install a modem before you can make a connection. Would you like to do this now?" click on "No". You do *not* need a modem to connect to the board.
6. Select the **Hypertrm** icon.
7. Enter a name, for example "PPCEVB" and select an icon.
8. Select the following:
   Connect using **Direct to Com 1**(default)
   Bits per second – **9600**
   Data bits – **8** (default)
   Parity – **None** (default)
   Stop Bits – **1** (default)
   Flow Control – **Xon/Xoff**
9. Select **OK**.

After resetting the board, the ROM Monitor menu should appear in the HyperTerminal window. If it does not, check your HyperTerminal settings and ensure proper connectivity between the host and the board.

6.3 Board Reset

When the connectors have been installed and power is applied to the board, you must first press the board's On/off switch to power up the board. Pressing the Reset switch causes the processor and the communications controllers to reset. After the ROM Monitor (resident in flash) initializes the processor and board peripherals, the monitor menu is displayed if a properly configured terminal (or terminal emulator) is attached to serial port 1 (J7 lower) of the board. Details of ROM Monitor operation are provided in Chapter 7, "ROM Monitor."

Chapter 7. ROM Monitor

This chapter describes the ROM Monitor program, also known as OpenBIOS. This ROM resident program provides chip (and board level) initialization and a user interface menu that supports board diagnostics, program downloads, and debug.

7.1 ROM Monitor Source Code

The ROM Monitor source code is provided for ROM development purposes. This code is separate from the OS Open and sample application code described in Chapter 8. The ROM Monitor code is loosely organized by function in the following subdirectories and files within the **405EP\openbios** directory.

align_h.s	Alignment handling code
bootprom.cmd	RISCWatch command script for reprogramming Serial Boot PROM
dbLib/	Ptrace debug interface routines
devTab.c	Handles boot device definitions
enetLib/	Ethernet related code
entry.s	Processor and C environment initialization
flash/	Code to support re-programming the flash memory
include/	C include files
ioLib/	I/O helper functions
lib/	Repository for intermediate libraries
m4/	assembler preprocessor include files
Makefile	Top level makefile to create ROM monitor image
malLib/	Memory Access Layer routines
mapfile1	Mapfile to specify ROM Monitor linkage directives
miscLib/	Miscellaneous routines used for ROM monitor
netLib/	IP and UDP processing functions
ppcLib/	C callable functions to access PowerPC special instructions
rom_*.map**	Load map of the ROM Monitor version *** shipped with the board
s1ldLib/	Code to support S1 serial port downloads
s1Lib/	Serial Port interface routines

7.2 Communications Features

The ROM Monitor runs as part of the boot code in the flash memory on the board. The monitor communicates with an asynchronous terminal (or terminal emulator) attached to serial port 1 (SP1)

on the board, through which the user accesses the monitor menu. The ROM Monitor can download applications and communicate with the host debugger through different Ethernet adapters, depending on which devices are enabled. Ethernet communications use the Internet Protocol (IP) over standard Ethernet. The ROM Monitor also supports the downloading of programs via serial port 1, but not debug. To use this feature, a VT100 terminal emulator that supports binary file transfers (such as kermit) must be used on the host system.

7.3 Configuration of bootp and tftp to Support ROM Monitor Loads

Both the debugger and the ROM Monitor can be used to load applications onto the board. Details on how to use the debugger can be found in the *RISCWatch Debugger User's Guide*. To use the facilities of the ROM Monitor for downloading applications to the board, the host workstation must be configured to support the **bootp** protocol and **tftp** daemons. The configuration consists of two parts. The **bootptab** file on the host must be customized to match system requirements, and the **bootp** and **tftp** daemons (or servers) must be made available.

7.3.1 PC bootp and tftp Configuration

Not all TCP/IP packages include the **bootpd** and **tftpd** servers required for ROM Monitor downloads. For this reason both the **bootpd** and **tftpd** servers have been included in the BSP software package under the **\osopen\bin** directory. These servers can be installed and used in conjunction with Windows Socket compliant TCP/IP packages that come with Windows 95/98 and Windows NT.

Configuration consists of two parts. The **bootptab** and **services** files on the host must be customized to match system requirements, and the **bootpd** and **tftpd** servers must be made available. If you choose to use the **bootpd** and **tftpd** servers provided with this package, you will need to modify your **autoexec.bat** file to specify the location of the **bootptab** and **services** files. This is accomplished by adding a line that sets up an ETC environment variable to specify the directory where the **bootptab** and **services** files are located (e.g., `set etc=c:\windows` for Windows 95/98, `set etc=c:\winnt\system32\drivers\etc`. for Windows NT 4.0). Consult your TCP/IP documentation or contact your system administrator if the **services** file cannot be found.

A sample **bootptab** file, **bin\bootptab**, is included with the BSP software. The **bin\bootptab** file can be copied to the ETC directory set in the **autoexec.bat** file and modified appropriately. Note that the **bootptab** file in the ETC directory must be named **bootptab** with no file extension. Entries describing the board to the host PC must be added to the **bootptab** file.

When creating or modifying the **bootptab** file, the following rules apply:

- Blank lines and lines beginning with "#" are ignored.
- Each entry must be entered on a single line.
- Each entry must start with a host name followed by the legends (see the sample bootptab file for legend descriptions).
- Use ":" to separate each legend and leave no spaces between legends.
- User must supply the host IP address via the "ip" legend.
- If the "hd" (home directory) & "bf" (bootfile) legends are not provided for a particular entry, the first defined "hd" and "bf" legends in the bootptab file will be taken as default.

File entries similar to the one below would be suitable.

```
enetc:ht=ethernet:hd=\osopen\m405h_evb\samples:bf=boot.img:bs:ip=7.1.1.5:
   sm=255.255.255.255:ha=xxxxxxxxxxxx
```

Each of the entry should be entered on a *single* line. The value of the Ethernet hardware address field in the **enetc** entry, ha=xxxxxxxxxxxx, should match the twelve character hardware address listed for the Ethernet Boot Source on the ROM Monitor menu.

The connection uses the file **\osopen\m405h_evb\samples\boot.img** as the source for the application image to be downloaded onto the board. Be sure that the **ht=ethernet** keyword is used for the Ethernet connection entry and that the IP addresses are those of the board. Since a board IP address was not required for Ethernet setup, the IP address used in the enetc entry defines the IP address of the board for the Ethernet connection. If the suggested bootptab entries are used, 7.1.1.5 would be the board's Ethernet IP address. Take note of the board's IP addresses, since they must be made known to the ROM Monitor.

The **services** file (no file extension) must also exist in the ETC directory set in the **autoexec.bat** file. It must be updated with the port and protocol information for the **bootpd** and **tftpd** servers. To use the servers provided with this package, the following entries must be included in the services file:

```
bootps      67/UDP

bootpc      68/UDP

tftp        69/UDP
```

For the update to take effect, TCP/IP needs to be re-started. This may require a reboot of the system and/or a restart of the TCP/IP package. After that, the bootpd and tftpd servers are ready for use.

You may choose to run **bootpd.exe** and **tftpd.exe** automatically every time that WIndows is started or you can run these programs only when needed. To make these program run automatically every time WIndows is started perform the following steps:

1. Select **Start** from the Windows task bar.
2. Select **Settings**.
3. Select **Taskbar**.
4. Select **Start Menu Programs**.
5. Select **Add...**.
6. In the command line field enter the following:

   ```
   BOOTPD -c C -h 7.1.1.4
   ```

   where C is the driver letter containing the boot image and *7.1.1.4* is the host IP address
7. Select **Next**.
8. In the **Select Program Folder** window, select the **Programs/Startup** folder.
9. Select **Next**.
10. Select **Finished**.
11. To start **tftp** follow the above steps, but enter the following in the command line field:

    ```
    TFTPD
    ```

The **bootp** and **tftp** daemons will be started automatically upon the next restart of Windows.

7.4 Accessing the ROM Monitor

The ROM Monitor expects a real or emulated VT100 type ASCII display attached to serial port 1 with line protocol parameters of 9600 bps, eight bits per character, no parity, and one stop bit. Once the terminal connected to SP1 is configured properly, you can access the ROM Monitor menu options, use the ping test, and load an application onto the board.

The ROM Monitor also provides the interface to the RISCWatch debugger. This facility, along with the image download process, is accessed via an IP network connection to the host workstation. Network configuration of the host was discussed earlier in the chapter on host configuration. The actual connection is via Ethernet using the 10BaseT/100BaseTX Ethernet port on the board.

7.5 ROM Monitor Operation

The ROM Monitor requires a block of DRAM for its operation and makes some assumptions about applications loaded on the board. Some of these assumptions may be disregarded if you do not need the ROM Monitor to interface with a debugger or otherwise support communication between the host workstation and the board.

Applications wishing to coexist with the ROM Monitor must observe the following constraints.

- Provide exception vectors for application events starting at address 0x0000 0000. For example, an application's external interrupt handler should be located at 0x0000 0500. This is handled for you when using OS Open.
- Use storage addresses between 0x0010 0000 and the end of DRAM only, except for application vectors.
- Do not alter the EVPR register.
- Do not start applications lower than address 0x0010 0000.

Figure 7-1 shows the address map of the evaluation board under control of the ROM Monitor.

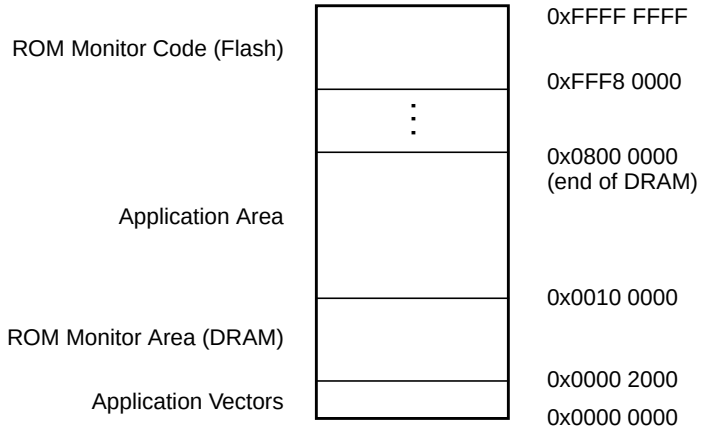


Figure 7-1. ROM Monitor Address Map

7.6 Monitor Selections and Submenus

At this point it is assumed that the host has been properly configured, all board connections have been made, power has been supplied, and the terminal emulator running on the host has been configured and started successfully. The main menu, shown below, is displayed after the board has been reset and the ROM Monitor completes initialization. Note that some of the values you see, in particular the ROM Monitor version, the IP addresses, and the Ethernet controller's hardware address, may differ with those shown below.

Each menu option is described separately in the following sections. "Local" in the context of the ROM Monitor IP addressing means the IP address assigned to the board, while "remote" means the IP address assigned to the host workstation. Using option 8 to save changes made to the configuration will allow the new values to persist beyond subsequent power-on or resets. The ROM Monitor supports this by storing its configuration data in NVRAM.

7.6.1 Initial ROM Monitor Menu

The following menu is displayed after the board has been reset. (Note: The following screens may not exactly match the PPC405EP kit).

```
405EP 1.19 ROM Monitor (09/05/02)

--------------------- System Info ----------------------
 Processor         = 405EP,   PVR: 51210950
 Processor speed   = 200 MHz
 PLB speed         = 100 MHz
 Ext Bus speed     = 50MHz
 PCI Bus speed     = 33 MHz
 Amount of SDRAM   = 128 MBytes
 External PCI arbiter enabled
--------------------------------------------------------

--- Device Configuration ---
 Power-On Test Devices:
   000  Enabled   System Memory [RAM]
   001  Enabled   Ethernet 0 [EMAC0]
----------------------------
 Boot Sources:
   001  Enabled   Ethernet 0 [EMAC0]
                  local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
005  Enabled   Serial Port 1 [S1]
                  Baud=9600
 ----------------------------
 Debugger: Disabled
 ----------------------------
  1 - Enable/disable tests
  2 - Enable/disable boot devices
  3 - Change IP addresses
  4 - Ping test
  5 - Toggle ROM monitor debugger
  6 - Toggle automatic menu
  7 - Display configuration
  8 - Save changes to configuration
  9 - Set baud rate for s1 boot
  A - Enable/disable I cache (Enabled)
  B - Enable/disable D cache (Enabled)
  0 - Exit menu and continue
->
```

7.6.2 Selecting Power-On Tests

Option 1 in the main menu selects power-on tests. These tests are run when the menu exits and before the ROM loader begins the **bootp** processing.

```
  1 - Enable/disable tests
  2 - Enable/disable boot devices
  3 - Change IP addresses
  4 - Ping test
  5 - Toggle ROM monitor debugger
  6 - Toggle automatic menu
  7 - Display configuration
  8 - Save changes to configuration
  9 - Set baud rate for s1 boot
  A - Enable/disable I cache (Enabled )
  B - Enable/disable D cache (Enabled )
  0 - Exit menu and continue
->1
```

When option 1 is selected, the following submenu is displayed.

```
--- ENABLE AND DISABLE POWER-ON TESTS ---
 Power-On Test Devices:
   000   Enabled     System Memory [RAM]
   001   Enabled     Ethernet 0 [EMAC0]
   004   Enabled     Ethernet 1 [EMAC1]
----------------------------
Select device to change ->
```

Selecting a test toggles its testing status. For example, since the System Memory test is enabled in the above menu, selecting 0 at the prompt disables it.

```
Select device to change ->0       [Selects system memory]
```

After the selection has been made, the new setting is displayed, followed by the main menu.

```
Select device to change ->0
   [RAM] test is disabled         [Message describing change]

  --- Device Configuration ---
  Power-On Test Devices:
    000  Disabled   System Memory [RAM]
    001  Enabled    Ethernet 0 [EMAC0]
    004  Enabled    Ethernet 1 [EMAC1]
  ----------------------------
  Boot Sources:
    001  Enabled   Ethernet 0 [EMAC0]
                   local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
    004  Enabled   Ethernet 1 [EMAC1]
                   local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
    005  Enbaled   Serial Port 1 [S1]
                   Baud=9600
```

```
  ---------------------------
  Debugger : Disabled
  ---------------------------
   1 - Enable/disable tests
   2 - Enable/disable boot devices
   3 - Change IP addresses
   4 - Ping test
   5 - Toggle ROM monitor debugger
   6 - Toggle automatic menu
   7 - Display configuration
   8 - Save changes to configuration
   9 - Set baud rate for s1 boot
   A - Enable/disable I cache (Enabled)
   B - Enable/disable D cache (Enabled)
   0 - Exit menu and continue
->
```

Remember to use Option 8 to save any configuration changes that you may have made. If the changes are not saved, they will be lost upon an exit from the menu or upon a board reset.

7.6.3 Selecting Boot Devices

Option 2 in the main menu enables and disables boot devices.

```
 1 - Enable/disable tests
 2 - Enable/disable boot devices
 3 - Change IP addresses
 4 - Ping test
 5 - Toggle ROM monitor debugger
 6 - Toggle automatic menu
 7 - Display configuration
 8 - Save changes to configuration
 9 - Set baud rate for s1 boot
 A - Enable/disable I cache (Enabled)
 B - Enable/disable Dcache (Enabled)
 0 - Exit menu and continue
->2
```

When option 2 is selected, the following submenu is displayed.

```
--- ENABLE AND DISABLE BOOT DEVICES ---
 Boot Sources:
   001   Enabled   Ethernet 0 [EMAC0}
                   local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
---------------------------
Select device to change ->
```

Selecting a device toggles its boot status. Selecting 1, for example, would disable Ethernet Port 1 as a boot device.

```
Select device to change ->1        [Selects ethernet port]
```

After the selection has been made, the new setting is displayed, followed by the main menu.

```
Select device to change ->4
   [EMAC0] boot is disabled       [Message describing change]

  --- Device Configuration ---
  Power-On Test Devices:
    000  Disabled   System Memory [RAM]
    001  Disabled  Ethernet 0 [EMAC0]
    004  Enabled    Ethernet 1 [EMAC1]
  ----------------------------
  Boot Sources:
    001  Disabled Ethernet 0 [EMAC0]
                  local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
    005  Enabled   Serial Port 1 [S1]
                  Baud=9600
  ----------------------------
  Debugger : Disabled
  ----------------------------
   1 - Enable/disable tests
   2 - Enable/disable boot devices
   3 - Change IP addresses
   4 - Ping test
   5 - Toggle ROM monitor debugger
   6 - Toggle automatic menu
   7 - Display configuration
   8 - Save changes to configuration
   9 - Set baud rate for s1 boot
   A - Enable/disable I cache (Enabled)
   B - Enable/disable D cache (Enabled)
   0 - Exit menu and continue
->
```

When the user selects option 0 and exits from the monitor menu, the monitor attempts a boot of the application image on the host using the enabled boot sources in the order they are listed. In the above example, a boot is attempted over Ethernet since it is the first boot source enabled. If more than one boot source is enabled, an attempt to boot over the first enabled device is made. If that attempt fails, a boot over the next enabled device is attempted.

7.6.4 Changing IP Addresses

Option 3 in the main menu allows users to change the IP addresses for the board and the host workstation. These addresses are used for bootp processing, debugger communications, and in the host connectivity “ping” test.

Note: The local IP address is that of the board and the remote IP address is that of the host workstation. The IP addresses must match those set during host configuration.

```
1 - Enable/disable tests
2 - Enable/disable boot devices
3 - Change IP addresses
4 - Ping test
5 - Toggle ROM monitor debugger
```

```
 6 - Toggle automatic menu
 7 - Display configuration
 8 - Save changes to configuration
 9 - Set baud rate for s1 boot
 A - Enable/disable I cache (Enabled)
 B - Enable/disable D cache (Enabled)
 0 - Exit menu and continue
->3
```

When option 3 is selected, the following submenu is displayed:

```
--- CHANGE IP ADDRESS ---
Device List:
    001  Enabled   Ethernet 0 [EMAC0]
                   local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
    004  Disabled  Ethernet 1 [EMAC1]
                   local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
----------------------------
Select device to change ->
```

Select the appropriate device.

```
Select device to change ->1       [Selects Ethernet]
```

When a valid device is selected, the following submenu is displayed.

```
 1 - Change local address
 2 - Change remote address
 0 - Return to main menu
->
```

Make the appropriate selection. To change the board's IP address, you would select option 1, Change local address.

```
->1                               [Selects the local address]
Current IP address = (7.1.1.5)    [Displays the current value]
Enter new IP address ->Enter IP address in dot notation (e.g., 8.1.1.2)
```

Now enter the new IP address in dotted decimal notation.

```
7.1.1.5
```

After the selection has been entered, the new configuration is displayed, followed by the main menu.

```
--- Device Configuration ---
Power-On Test Devices:
   000  Disabled   System Memory [RAM]
   001  Enabled    Ethernet 0 [EMAC0
   004  Enabled    Ethernet 1 [EMAC1
----------------------------
Boot Sources:
  001  Enabled   Ethernet 0[EMAC0]
                 local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
  004  Disabled  Ethernet 1 [EMAC1]
```

```
                  local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
    005  Enabled   Serial Port 1 [S1]
                  Baud=9600
  ----------------------------
  Debugger : Disabled
  ----------------------------
   1 - Enable/disable tests
   2 - Enable/disable boot devices
   3 - Change IP addresses
   4 - Ping test
   5 - Toggle ROM monitor debugger
   6 - Toggle automatic menu
   7 - Display configuration
   8 - Save changes to configuration
   9 - Set baud rate for s1 boot
   A - Enable/disable I cache (Enabled)
   B - Enable/disable D cache (Enabled)
   0 - Exit menu and continue
->
```

This option should be repeated to set all of the IP addresses to their appropriate values. If the suggested IP addresses are being used, the local and remote addresses for both the Ethernet and the Serial Port should match those in the above menu. Remember to save any configuration changes via option 8.

7.6.5 Using the Ping Test

Option 4 in the main menu selects the ping test. The ping test can be used for a basic assurance test of IP connectivity to the host workstation. It should be performed after setting the IP addresses to insure host-to-board communications. If the ping test fails, users can not load applications on to the board. The local and remote addresses for the specified device are used for the source and destination of the ICMP ping packets.

```
 1 - Enable/disable tests
 2 - Enable/disable boot devices
 3 - Change IP addresses
 4 - Ping test
 5 - Toggle ROM monitor debugger
 6 - Toggle automatic menu
 7 - Display configuration
 8 - Save changes to configuration
 9 - Set baud rate for s1 boot
 A - Enable/disable I cache (Enabled)
 B - Enable/disable D cache (Enabled)
 0 - Exit menu and continue
->4
```

When option 4 is selected, the current configuration is displayed, followed by another command prompt.

```
--- PING TEST ---
Device List:
     001  Enabled   Ethernet 0[EMAC0]
```

```
                    local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
     004  Disabled  Ethenet 1 [EMAC1]
                    local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
----------------------------
Select device to ping ->
```

Select the appropriate device to ping (in this case only Ethernet is enabled).

```
Select device to ping ->1         [selects the Ethernet port]
```

If the board is able to successfully ping the host, a message similar to the following should appear:

```
Using [EMAC0] to ping. press any key to stop.
PING 7.1.1.4 56 data bytes
78 bytes from 7.1.1.4: icmp_seq=0 ttl=255 time=2 ms
78 bytes from 7.1.1.4: icmp_seq=2 ttl=255 time=1 ms
```

Pressing any key terminates the ping test. The main menu is redisplayed following the PING status report.

```
--- 7.1.1.4 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
 1 - Enable/disable tests
 2 - Enable/disable boot devices
 3 - Change IP addresses
 4 - Ping test
 5 - Toggle ROM monitor debugger
 6 - Toggle automatic menu
 7 - Display configuration
 8 - Save changes to configuration
 9 - Set baud rate for s1 boot
 A - Enable/disable I cache (Enabled)
 B - Enable/disable D cache (Enabled)
 0 - Exit menu and continue
->
```

If the ping test fails,

- Verify that the local and remote IP addresses are set correctly. The local IP address should be that of the board and the remote IP address should be that of the host. These IP addresses were assigned during host configuration (see earlier chapter).
- Verify that the cables are connected properly.
- Verify TCP/IP is running on the host.

Note: The ROM Monitor will not respond to an inbound ping test from the host unless the ROM Monitor is in Debug mode (via options 5 and 0) or the ROM Monitor ping test is active on the board at the same time (via option 4).

7.6.6 Entering the Debugger

Option 5 toggles the feature of the ROM Monitor that allows communication with the host based source level debugger. Debugging may be enabled/disabled, and saved as part of the configuration

using option 8. The debugger is not actually called by the monitor until after the user exits the main menu by selecting option 0 (exit and continue).

```
   --- Device Configuration ---
   Power-On Test Devices:
      000  Disabled   System Memory [RAM]
      001  Enabled    Ethernet 0 [EMAC0]
      004  Enabled    Ethernet 1 [EMAC1]
   ----------------------------
   Boot Sources:
     001  Enabled   Ethernet 0[EMAC0]
                    local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
     004  Disabled  Ethernet 1[EMAC1]
                    local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
     005  Enbaled   Serial Port 1 [S1]
                    Baud=9600
   ----------------------------
   Debugger : Disabled
   ----------------------------
    1 - Enable/disable tests
    2 - Enable/disable boot devices
    3 - Change IP addresses
    4 - Ping test
    5 - Toggle ROM monitor debugger
    6 - Toggle automatic menu
    7 - Display configuration
    8 - Save changes to configuration
    9 - Set baud rate for s1 boot
    A - Enable/disable I cache (Enabled)
    B - Enable/disable D cache (Enabled)
    0 - Exit menu and continue
->5
ROM monitor debugger will be active on exit
   1 - Enable/disable tests
   2 - Enable/disable boot devices
   3 - Change IP addresses
   4 - Ping test
   5 - Toggle ROM monitor debugger
   6 - Toggle automatic menu
   7 - Display configuration
   8 - Save changes to configuration
   9 - Set baud rate for s1 boot
   A - Enable/disable I cache (Enabled)
   B - Enable/disable D cache (Enabled)
   0 - Exit menu and continue
->7

 --- Device Configuration ---
  Power-On Test Devices:
     000  Disabled   System Memory [RAM]
     001  Enabled    Ethernet 0 [EMAC0]
     004  Enabled    Ethernet 1 [EMAC1]
  ----------------------------
  Boot Sources:
```

```
    001  Enabled   Ethernet 0[EMAC0]
                   local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
    004  Disabled  Ethernet 1[EMAC1]
                   local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
    005  Enbaled   Serial Port 1 [S1]
                   Baud=9600
  ----------------------------
  Debugger : Enabled (on exit)
  ----------------------------
   1 - Enable/disable tests
   2 - Enable/disable boot devices
   3 - Change IP addresses
   4 - Ping test
   5 - Toggle ROM monitor debugger
   6 - Toggle automatic menu
   7 - Display configuration
   8 - Save changes to configuration
   9 - Set baud rate for s1 boot
   A - Enable/disable I cache (Enabled)
   B - Enable/disable D cache (Enabled)
   0 - Exit menu and continue
->0
PowerPC ROM Monitor Debugger

                                  Waiting for debug command...
                                         Press any key to exit
```

Use option 8 to save the state of the ROM Monitor debugger. This option in combination with option 6, “Toggle automatic menu”, can be used to configure the board to automatically wait for the debugger to attach after power-on.

The ROM Monitor debugger only communicates over Ethernet so one of these boot devices must be enabled when using the ROM Monitor debugger. After enabling the ROM Monitor debugger (via option 5) and selecting option 0, the RISCWatch debugger can be started on the host and used to load an application onto the board. This is assuming the RISCWatch environment file has been updated for ROM Monitor communications. Once loaded successfully, the application can be run from the debugger.

The *RISCWatch Debugger User’s Guide* contains more information on how to use the debugger to load and execute files with the ROM Monitor as a non-JTAG target. At this point, it is recommended that users become familiar with the debugging environment by following the “Quick Start” sample debug session in the debugger’s User’s Guide. This session takes a user through the basics, including how to use the debugger to load and run applications on the board.

7.6.7 Disabling the Automatic Display

Option 6 in the main menu disables the automatic monitor display when the board boots up. After option 6 has been selected and the configuration has been saved (via Option 8), the menu display is disabled but continues to function until the user exits from the main menu. Following the next power-on or reset, the menu is no longer automatically displayed. This allows the user’s image to be downloaded automatically with no menu input required. This feature also allows a user to download

an application with no cable connected to the serial port 1 on the board (that is, without a terminal emulator).

After the automatic menu display has been disabled, the main menu can be accessed (assuming a terminal emulator is attached successfully to SP1 on the board) by pressing any key during the first five seconds that the board is booting. Otherwise, application download processing starts without displaying the main menu.

7.6.8 Displaying the Current Configuration

Option 7 displays the current configuration.

```
 1 - Enable/disable tests
 2 - Enable/disable boot devices
 3 - Change IP addresses
 4 - Ping test
 5 - Toggle ROM monitor debugger
 6 - Toggle automatic menu
 7 - Display configuration
 8 - Save changes to configuration
 9 - Set baud rate for s1 boot
 A - Enable/disable I cache (Enabled)
 B - Enable/disable D cache (Enabled)
 0 - Exit menu and continue
->7

 --- Device Configuration ---
  Power-On Test Devices:
     000  Disabled   System Memory [RAM]
     001  Enabled    Ethernet 0 [EMAC0]
     004  Enabled    Ethernet 1 [EMAC1]
  ----------------------------
  Boot Sources:
    001  Enabled   Ethernet 0 [EMAC0]
                   local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
    004  Disabled  Ethernet 1 [EMAC1]
                   local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
    005  Enbaled   Serial Port 1 [S1]
                   Baud=9600
  ----------------------------
  Debugger : Enabled (on exit)
  ----------------------------
   1 - Enable/disable tests
   2 - Enable/disable boot devices
   3 - Change IP addresses
   4 - Ping test
   5 - Toggle ROM monitor debugger
   6 - Toggle automatic menu
   7 - Display configuration
   8 - Save changes to configuration
   9 - Set baud rate for s1 boot
   A - Enable/disable I cache (Enabled)
   B - Enable/disable D cache (Enabled)
   0 - Exit menu and continue
```

```
->
```

When a menu operation is selected to alter configuration settings, the current configuration is automatically redisplayed.

7.6.9 Saving the Current Configuration

Option 8 saves the current configuration for subsequent power-on resets.

```
 1 - Enable/disable tests
 2 - Enable/disable boot devices
 3 - Change IP addresses
 4 - Ping test
 5 - Toggle ROM monitor debugger
 6 - Toggle automatic menu
 7 - Display configuration
 8 - Save changes to configuration
 9 - Set baud rate for s1 boot
 A - Enable/disable I cache (Enabled)
 B - Enable/disable D cache (Enabled)
 0 - Exit menu and continue
->8
Configuration has been saved
 1 - Enable/disable tests
 2 - Enable/disable boot devices
 3 - Change IP addresses
 4 - Ping test
 5 - Toggle ROM monitor debugger
 6 - Toggle automatic menu
 7 - Display configuration
 8 - Save changes to configuration
 9 - Set baud rate for s1 boot
 A - Enable/disable I cache (Enabled)
 B - Enable/disable D cache (Enabled)
 0 - Exit menu and continue
 ->
```

The configuration is saved in the NVRAM on the evaluation board and is retained until a new configuration is subsequently saved.

7.6.10 Setting the Baud Rate for S1 Boots

Option 9 provides a mechanism for setting the baud rate to be used by serial port 1 when it is used as a device to download programs. Downloading over serial port 1 requires the use of a VT100 terminal emulator that supports **kermit** binary file transfer over serial port 1. Windows NT/2000 users can use HyperTerminal to perform kermit file transfers at up to 115200 baud. The kermit terminal emulator, available as shareware from the **http://www.columbia.edu/kermit** Internet site, can be used on any of the supported hosts to download programs over serial port 1 at speeds up to 115200 baud. Note that the ROM Monitor debugger can not operate over serial port 1.

```
--- Device Configuration ---
 Power-On Test Devices:
    000  Disabled   System Memory [RAM]
    001  Enabled    Ethernet 0 [EMAC0]
    004  Enabled    Ethernet 1 [EMAC1]
 ----------------------------
 Boot Sources:
   001  Enabled   Ethernet 0 [EMAC0]
                  local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
   004  Disabled  Ethernet 1 [EMAC1]
                  local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
   005  Enbaled   Serial Port 1 [S1]
                  Baud=9600
 ----------------------------
 Debugger : Enabled (on exit)
 ----------------------------
  1 - Enable/disable tests
  2 - Enable/disable boot devices
  3 - Change IP addresses
  4 - Ping test
  5 - Toggle ROM monitor debugger
  6 - Toggle automatic menu
  7 - Display configuration
  8 - Save changes to configuration
  9 - Set baud rate for s1 boot
  A - Enable/disable I cache (Enabled)
  B - Enable/disable D cache (Enabled)
  0 - Exit menu and continue
->9

Select a baud rate for S1 boot
  1 -        9600
  2 -       19200
  3 -       28800
  4 -       38400
  5 -       57600
  6 -      115200
=>4

--- Device Configuration ---
 Power-On Test Devices:
    000  Disabled   System Memory [RAM]
    001  Enabled    Ethernet 0 [EMAC0]
    004  Enabled    Ethernet 1 [EMAC1]
 ----------------------------
 Boot Sources:
   001  Enabled   Ethernet 0 [EMAC0]
                  local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
   004  Disabled  Ethernet 1 [EMAC1]
                  local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
   005  Enbaled   Serial Port 1 [S1]
                  Baud=38400
```

```
 ---------------------------
 Debugger : Disabled (on exit)
 ---------------------------
  1 - Enable/disable tests
  2 - Enable/disable boot devices
  3 - Change IP addresses
  4 - Ping test
  5 - Toggle ROM monitor debugger
  6 - Toggle automatic menu
  7 - Display configuration
  8 - Save changes to configuration
  9 - Set baud rate for s1 boot
  A - Enable/disable I cache (Enabled)
  B - Enable/disable D cache (Enabled)
  0 - Exit menu and continue
->
```

Use Option 8 to save the selected speed after reset and power-on.

7.6.11 S1 Boot

To perform an S1 boot you must have a terminal emulator which supports **kermit** file transfer. The file must be a valid boot image and must be sent in binary mode. If you have selected to use a baud rate other than 9600, you must set the terminal emulator to run at that speed before loading the file and set the speed back to 9600 after the download is complete. The following example shows loading the **usr_samp.img** file.

```
--- Device Configuration ---
Power-On Test Devices:
  000  Disabled  System Memory [RAM]
  001  Disabled  Ethernet 0[EMAC0]
  004  Disabled  Ethernet 1[EMAC1]
---------------------------
Boot Sources:
   001  Disabled  Ethernet 0 [EMAC0]
                  local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
   004  Disabled  Ethernet 1 [EMAC1]
                  local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
   005  Enbaled   Serial Port 1 [S1]
                  Baud=38400
---------------------------
Debugger: Disabled
---------------------------
 1 - Enable/disable tests
 2 - Enable/disable boot devices
 3 - Change IP addresses
 4 - Ping test
 5 - Toggle ROM monitor debugger
 6 - Toggle automatic menu
 7 - Display configuration
 8 - Save changes to configuration
 9 - Set baud rate for s1 boot
 A - Enable/disable I cache (Enabled)
```

```
  B - Enable/disable D cache (Enabled)
  0 - Exit menu and continue
->0
Booting from [S1] Serial Port 1...

PLEASE NOTE: You must now...

       a. Exit from terminal emulation mode
       b. Modify the baud rate of your host session
       c. Transmit a file to the target in binary mode
       d. Reset the host baud rate to 9600
       e. Reenter terminal emulation mode
       f. Hit enter to execute the downloaded program
```

At this point kermit users must get to the terminal emulator command mode and change the line speed to match what was selected by option 9 and tell the terminal emulator to send the file in binary format.

^\c (Cntrl-\c)

```
(Back at waterdeep)
C-Kermit>set speed 38400
/dev/tty0, 38400 bps
C-Kermit>set file type bin
```

You can now load the file.

```
C-Kermit>send usr_samp.img
SF
Type escape character (^\) followed by:
X to cancel file,  CR to resend current packet
Z to cancel group, A for status report
E to send Error packet, Ctrl-C to quit immediately:

Sending: usr_samp.img => USR_SAMP.IMG
Size: 164864, Type: binary
................................................................................
...
................................................................................
....
.... [OK]
ZB
```

When loading is completed, you must change the line speed back to 9600 bps before continuing.

```
C-Kermit>set speed 9600
/dev/tty0, 9600 bps
```

After setting the line speed back to 9600 bps, re-connect to your terminal emulator and press **Enter** to complete the download.

```
C-Kermit>con
Connecting to /dev/tty0, speed 9600.
The escape character is Ctrl-\ (ASCII 28, FS)
Type the escape character followed by C to get back,
or followed by ? to see other options

Loaded successfully ...
Entry point at 0x25f20 ...

Hello 405EP user!

Your ROM Monitor version is : 1.19

Your 405EP Evaluation Board has 16777216 bytes of DRAM installed.

Your Ethernet controller’s network address is :  1000abcdef55

usr_samp done!
```

Assuming the S1 boot baud rate has been set to 38400 and option 0 has been selected to exit the ROM Monitor menu and initiate a load, Windows HyperTerminal users can initiate the kermit binary file transfer by performing the following steps:

1. Select **Call** and then **Disconnect**.
2. Select **File**, **Properties**, **Configure** and set the baud to match the baud rate set via ROM Monitor option 9. In this case, it is 38400.
3. Select **OK** and **OK** again.
4. Select **Call** and then **Connect**.
5. Select **Transfer**, **Send File** and type the file name of the file to load. Set the **Protocol** to Kermit.
6. Select **Send**.

Upon successful completion of the transfer, the baud rate must be changed back to 9600.

7. Select **Call** and then **Disconnect**.
8. Select **File**, **Properties**, **Configure** and set the baud to 9600.
9. Select **OK** and **OK** again.
10. Select **Call** and then **Connect**.
11. Press **Enter** to complete the download sequence.

7.6.12 Exiting the Main Menu

Option 0 exits from the main menu, leaving the monitor active. If the debugger is active prior to selecting option 0, the ROM Monitor waits for the user to start the debugger on the host. In all other cases, option 0 initiates an attempt by the ROM Monitor to load an application from the host to the

board over the enabled boot device(s). When downloading over the Ethernet, the host bootp and tftp configuration must be completed for the ROM Monitor to load an application program successfully. Upon exit of the menu, the ROM Monitor will send a bootp request to the host to obtain the name of the file to download. Once the bootpd server returns the appropriate file name as set in the bootptab file, the ROM Monitor sends a tftp request to the tftpd server on the host to transfer file. Once the file is loaded successfully, it is executed.

When serial port 1 is used, the ROM Monitor requires the user to follow additional instructions to complete the download. The example shown here describes the sequence required when programs are downloaded over serial port 1.

```
--- Device Configuration ---
 Power-On Test Devices:
    000  Disabled   System Memory [RAM]
    001  Disabled   Ethernet 0 [EMAC0]
    004  Disabled   Ethernet 1 [EMAC1]
 ----------------------------
 Boot Sources:
   001  Disabled  Ethernet 0 [EMAC0]
                  local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
   004  Disabled  Ethernet 1 [EMAC1]
                  local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
   005  Enbaled   Serial Port 1 [S1]
                  Baud=38400
 ----------------------------
 Debugger : Enabled (on exit)
 ----------------------------
  1 - Enable/disable tests
  2 - Enable/disable boot devices
  3 - Change IP addresses
  4 - Ping test
  5 - Toggle ROM monitor debugger
  6 - Toggle automatic menu
  7 - Display configuration
  8 - Save changes to configuration
  9 - Set baud rate for s1 boot
  A - Enable/disable I cache (Enabled)
  B - Enable/disable D cache (Enabled)
  0 - Exit menu and continue
 ->0
Booting from [S1] Serial Port 1...

PLEASE NOTE: You must now...

     a. Exit from terminal emulation mode
     b. Modify the baud rate of your host session
     c. Transmit a file to the target in binary mode
     d. Reset the host baud rate to 9600
     e. Re-enter terminal emulation mode
     f. Hit enter to execute the downloaded program
```

The ROM Monitor will now wait for you to follow the above steps. The idea is that you must temporarily modify the terminal emulation session baud rate to match the baud rate expected by the

ROM Monitor for the serial port 1 download. The file must then be transferred to the board from the host. The baud rate is restored to 9600 so that terminal emulation support can function after the program has been downloaded, The ROM Monitor will wait for you to restore the baud rate (9600) and press **Enter** prior to executing the downloaded program. This prevents any program I/O from being lost or incorrectly displayed when it begins execution.

The following is an example of what you might see when the program is allowed to run.

```
Loaded successfully...
Entry point at 0x25130...
.
.
.
```

7.6.13 Cache Options

Options A and B allow the user to enable or disable the processor's instruction and data caches, respectively. These options toggle the status of the caches and take effect immediately upon selection. The current cache status is indicated at the end of each option and remains in effect upon exit from the ROM Monitor menu.

7.7 ROM Monitor User Functions

The ROM Monitor contains several functions that are available to user programs. The prototypes of these functions can be found in the **usr_func.h** file in the directory **osopen\405ep\openbios\include** These functions include:

send_packet_on_bootdev()	Allows an IP packet to be sent over the Ethernet device that was used to load the application program
sh_register()	Used to register a function that will be called when an IP packet is received by the ROM Monitor over the boot device.
get_board_cfg()	Reads the configuration data associated with the board.
enet_send_macframe()	Allows a frame to be sent over the Ethernet.
enet_register()	Allows the user to register an IP address for the Ethernet (an IP address different from that assigned to the ROM Monitor) and to specify a function to be called when a frame arrives for that address.
enetisThere()	Determines if an Ethernet port is connected on the board.
enetInit()	initializes the Ethernet.
getchar()	Reads one character at a time from the keyboard buffer over the first serial port (SP1).
s1putchar()	Writes one character to the first serial port (SP1).

Applications must follow a predefined protocol to access ROM Monitor user functions. An example showing the proper calling procedures are included in the **usr_samp.c** sample program in the

samples directory. This sample program calls the **get_board_cfg()** ROM Monitor function to determine the amount of DRAM installed on the board. This program will be run as a sample program in the next chapter.

7.8 Flash Update Utility

The **openbios/flash** directory contains all the code you need to reprogram the flash memory on the board. This utility takes a binary image file targeted for the ROM as input, and generates a loadable file that will reprogram the flash memory with the data in the binary input file. The file can then be loaded by an existing ROM Monitor version (which will be overwritten upon successful completion of the loaded program) or via RISCWatch JTAG.

IMPORTANT: Please see the **readme.txt** file in the **openbios/flash** directory for important information regarding the use of this tool.

Be aware that if you use the ROM Monitor bootp or the RISCWatch ROM Monitor mode download process to reprogram the flash, and the program loaded contains errors that will not allow you to download images in the same manner, your flash may be corrupted and rendered useless. In this case you will need to use RISCWatch JTAG or a ROM burner to reprogram the flash.

RISCWatch JTAG users will find a RISCWatch command file, **rw_flash.cmd** in the **openbios/flash** directory. This command file can be used to prepare the board, load the flash update program containing the new binary image to program into the ROM, and start it running. This method can be used to program new flash parts, or to reprogram a corrupted flash part when normal ROM Monitor downloads are not possible or inconvenient. When using this command file, RISCWatch must be used in JTAG mode.

7.9 Network Address of the Ethernet Controller

Each of the the evaluation board's Ethernet's ports has been assigned a unique six-byte network address. This address, also known as the media access control or MAC address, may need to be known by customers using the board to develop their own ROM versions.

The easiest way to obtain its value is to hook up a terminal (or terminal emulator) to the serial port 1 (see Chapter 6.1, "Connecting the Evaluation Board to the Host") and bring up the ROM Monitor. After selecting option 7 to display the configuration, the controller's network address is displayed in the Ethernet boot source's *hwaddr* field as twelve hex characters (six bytes).

Another way to obtain the address, is to search the Vital Product Data (VPD) area in ROM where the network address is stored. The VPD fields consist of ASCII strings identifying the type of field, a length byte specifying the length of the associated data, and the data itself. The VPD begins at address 0xFFFFFE00 and is marked by field "*VPD" with 0 bytes of associated data. The network address is marked by "*NA" with six bytes of associated data (the network address). Finally, the end of the VPD is marked with "*END". To extract the network address, a program would typically start at address 0xFFFFFE00, scan for "*NA", verify the next byte is 0x6, and treat the next six bytes as the network address.

Chapter 8. Sample Applications

This chapter describes the steps necessary to build and run the sample programs included in the PPC405EP evaluation board kit software support package. This code includes a limited version of IBM's OS Open real time operating system and is separate from the ROM Monitor code described in Chapter 7.

8.1 Overview

The sample application programs are compiled, assembled, and linked using the IBM High C/C++ compiler, assembler, and linker. OS Open libraries are used during the link step to create an executable file in ELF format. This file includes the OS Open bootstrap code as well as other OS Open functions and is referred to as a boot file. One of the tools provided in the software support package, **eimgbld**, is then used to convert the boot file into the format used by the ROM Monitor to load programs onto the evaluation board (see Appendix B for more information on the ROM Monitor load format). The output of the **eimgbld** step is a file referred to as a boot image file.

There are several ways to load and execute a boot image file. One way is to use the ROM Monitor to load and execute the file. Network loads over Ethernet require that the host contain the bootp and tftp servers and be properly configured to support the bootp and tftp protocols (see the previous chapters on host configuration and ROM Monitor setup). Loads over serial port 1 require a terminal emulator that supports the kermit transfer protocol. A ROM Monitor load is initiated via option 0 from the ROM Monitor main menu.

Another way to load and execute the boot image file is to use the RISCWatch debugger in ROM monitor mode. To bring up RISCWatch in ROM Monitor mode (see the *RISCWatch Debugger User's Guide* for details), you must update the RISCWatch environment file for ROM Monitor communications, enable the ROM Monitor debugger (via option 5), exit the ROM Monitor menu (via option 0) and then start up RISCWatch on the host system. The RISCWatch **load image** command can then be used to load the boot image file onto the board. Once loaded successfully, the program can be debugged and/or executed. At any time the RISCWatch **logoff** command can be issued to execute the program. This command tells the ROM Monitor to exit debug mode and start the execution of the program. After program execution, users should quit and restart RISCWatch before loading another boot image file to run. Without quitting RISCWatch, subsequent boot image execution can not be guaranteed.

Note: RISCWatch also provides the means to load a boot file (as opposed to a boot image file) via its **load file** command. See the "Running Your Programs" section in the *RISCWatch Debugger User's Guide* for additional information. This section also describes the steps required to load and debug boot and boot image files.

8.2 ROM Monitor Flash Image

The flash memory on the board comes preprogrammed with a specific version of the ROM Monitor. This version may not be latest version of the ROM Monitor. To run the samples in the software support package, the latest version should be used. The latest version of the ROM Monitor is included in the software support package in the file:

```
openbios\lib\rom_***.img
```

where *** is equal to the ROM Monitor version. If the *** version number of the ROM Monitor in the software support package does not match the version number displayed by the monitor when it comes up on the board, you can load the more recent version of the monitor provided in the software support package to re-program the flash memory.

The **rom_***.img** file can be loaded using the ROM Monitor or the RISCWatch debugger. For it to load properly upon the selection of ROM Monitor option 0, it must be copied to **boot.img** if the suggested bootptab entry was used (see “Configuration of bootp and tftp to Support ROM Monitor Loads” on page 7-2).

To load using RISCWatch, enable the ROM Monitor debugger (via option 5), exit the ROM Monitor menu (via option 0), start RISCWatch on the host system (make sure the RISCWatch environment file is setup for ROM Monitor communications), then use the following RISCWatch commands to load and execute the **rom_***.img** image file:

```
load image (install dir)\osopen\m405h_evb\openbios\lib\rom_***.img
logoff
```

You will see screen information similar to that shown below. Lines preceded by “$$” are annotation for this example and do not appear on the screen.

```
$$ Standard ROM Monitor load screen below
405EP 1.19 ROM Monitor (12/15/01)
$$ Version 1.4 already installed corresponds to rom_19.img

 --------------------- System Info ----------------------
 Processor         = 405EP,   PVR: 51210950
 Processor speed   = 200 MHz
 PLB Bus speed     = 100MHz
 Ext Bus speed     = 50 MHz
 PCI Bus speed     = 33 MHz
 Amount of SDRAM   = 128 MB
 External PCI arbiter enabled
 ------------------------------------------------------
 --- Device Configuration ---
 Power-On Test Devices:
   000  Disabled  System Memory [RAM]
   001  Disabled  Ethernet 0 [EMAC0]
   004  Disabled  Ethernet 1 [EMAC1]
 ----------------------------------
  Boot Sources:
   001  Enabled   Ethernet 0 [EMAC0]
                  local=7.1.1.5  remote=7.1.1.4  hwaddr=1000abcdef55
   004  Disabled  Ethernet 1 [EMAC1]
                  local=8.1.1.5  remote=8.1.1.4  hwaddr=1000abcdef56
   005  Disabled  Serial Port 1 [S1]
                  Baud=38400
```

```
-----------------------------------
 Debugger: Disabled
-----------------------------------
  1 - Enable/disable tests
  2 - Enable/disable boot devices
  3 - Change IP addresses
  4 - Ping test
  5 - Toggle ROM monitor debugger
  6 - Toggle automatic menu
  7 - Display configuration
  8 - Save changes to configuration
  9 - Set baud rate for s1 boot
  A - Enable/disable I cache (Enabled)
  B - Enable/disable D cache (Enabled)
  0 - Exit menu and continue
->0
$$ Selection of 0 causes evaluation board to be loaded. Previous
$$ arrangements must have been made to place the new ROM Monitor
$$ image (for ex. \osopen\m405h_evb\openbios\lib\rom_13.img) in the
$$ place where bootp expects to find it (for ex. boot.img)
Booting from [ENET] Ethernet...
Sending bootp request ...


Loading file “\osopen\m405h_evb\samples\boot.img” ...
Sending tftp boot request ...
Transfer Complete ...
Loaded successfully ...
Entry point at 0x25028 ...

$$ following information is from the ROM Monitor update program
################ IBM 405EP Evaluation Kit FLASH Update ################
                     ROM Monitor Version 1.3

$$ Heed the following warning. The ROM Monitor image could be
$$ rendered unusable and the board useless until the flash ROM is
$$ replaced.
     WARNING: You are about to re-program your ROM Monitor FLASH
              image.  Do NOT turn off power or press reset
              until this procedure is completed.  Otherwise
              the card may be permanently damaged!!!

Do you wish to continue? (y or n)y

Verifying new FLASH Image...
131072 matches, 0 mismatches

Update complete!
All done!
```

After the update completes, a reset of the board should display the menu of the new ROM Monitor version.

8.3 Using the Software Samples

The sample application programs are in **osopen\m405h_evb\samples** subdirectory. It is recommended that users first build and run the Dhrystone, usr_samp, and timesamp sample programs as detailed below, to become familiar with the working environment. These sample programs use **basic_os.c** to provide a minimal OS Open configuration.

Additional details regarding the sample programs and application development in general can be found in the “Developing OS Open Applications” chapter in the *IBM OS Open User’s Guide*. That chapter should be referenced for instructions on building and running the applprog, benchmk, mailsamp, and cat sample programs.

The sample makefile contains the directives needed to build all the sample programs. It is suggested that this makefile be used as the starting point for building subsequent user applications.

Before attempting to build the samples, ensure the **osopen/bin** directory and the directory that contains the compiler, are part of your execution path (these steps should be modified accordingly based on where the compiler and the software support package were actually installed).

For **PC** hosts:

1. Edit AUTOEXEC.BAT using an editor such as e (you should back this file up before editing).
2. If the following statement is missing, add it to the end of the file.

```
SET PATH=C:\Program Files\IBM\405EP_EvalKit\highcppc\bin;C:\Program
Files\IBM\405EP_EvalKit\osopen\bin;%PATH%;
```

3. Run AUTOEXEC.BAT to update your path.

8.3.1 Building and Running the Dhrystone Benchmark

The Dhrystone benchmark is a commonly available integer benchmark. Since the main loop of this benchmark fits into the caches of many processors, its validity as a predictor of system performance may be suspect. It is included here as an example of an application to be built, loaded onto the evaluation board, and executed.

To build the Dhrystone benchmark, enter the command **gnumake dhry** from the command line while in the **samples** directory. The makefile will compile the Dhrystone source files, link the resulting object files with the support libraries, and produce the boot file, **dhry**, and the boot image file, **dhry.img**.

If the bootptab entry suggested in Chapter 4, “Host Configuration,” was used, then **dhry.img** must be renamed or copied to **boot.img** in order to be selected by the ROM Monitor load process. Select option 0 from the ROM Monitor screen to load and run the image.

To load using RISCWatch, enable the ROM Monitor debugger (via option 5), exit the ROM Monitor menu (via option 0), start RISCWatch on the host system (make sure the RISCWatch environment file is setup for ROM Monitor communications), then use the RISCWatch **load image** command to load the **dhry.img** file. Once successfully loaded, the **logoff** command can be issued to execute the program.

You should see the following messages (or ones like them) appear on the ROM monitor screen. Explanations enclosed by () do not appear on the screen but are added here as clarification.

```
Booting from [ENET] Ethernet...
```

```
Sending bootp request...
```

(This requests the Host workstation to return the name of the boot image.)

```
Loading file "\osopen\m405h_evb\samples\boot.img"...
Sending tftp boot request...
```

(Having obtained the file name, the ROM monitor uses tftp to retrieve the file from the host workstation.)

```
Transfer Complete...
Loaded successfully...
Entry point at 0x25a18...
```

(Having loaded an image, the ROM monitor is now transferring control to the application. Subsequent messages are from the application.)

```
Dhrystone Benchmark, Version 2.1 (Language: C)
Program compiled without 'register' attribute
Please give the number of runs through the benchmark:
```

At this point, enter the number of desired iterations. The test is designed not to give results if the selected iterations completes in less two seconds, so pick a large number (≥ 1000000). After the test completes, a check screen will be displayed, followed by the benchmark results. The results may vary based on the system environment.

8.3.2 Building and Running the usr_samp Program

The **usr_samp.c** program is included as a sample to be built and run on the EVB. It's a simple program that shows how to properly call the get_board_cfg() ROM Monitor user function to determine the ROM Monitor version, the amount of DRAM installed on the board and the Ethernet controller's MAC address. Developers interested in using any of the ROM Monitor user functions should use this program as a guide.

To build the usr_samp program, enter the command **gnumake usr_samp** from the command line while in the **samples** directory. The makefile will compile the **usr_samp.c** file, link the resulting object file with the support libraries, and produce the boot file, **usr_samp**, and the boot image file, **usr_samp.img**.

If the suggested bootptab was used, then **usr_samp.img** must be renamed or copied to **boot.img** in order to be selected by the ROM Monitor load process.Select option 0 from the ROM Monitor screen to load and run the image.

To load using RISCWatch, enable the ROM Monitor debugger (via option 5), exit the ROM Monitor menu (via option 0), start RISCWatch on the host system (make sure the RISCWatch environment file is setup for ROM Monitor communications), then use the RISCWatch **load image** command to load the **usr_samp.img** file. Once successfully loaded, the **logoff** command can be issued to execute the program.

You should see the following messages (or ones like them) appear on the ROM Monitor screen.

```
Booting from [ENET] Ethernet...
Sending bootp request...

Loading file "\osopen\m405h_evb\samples\boot.img"...
Sending tftp boot request...
```

```
Transfer Complete...
Loaded successfully...
Entry point at 0x25e48...

Hello 405EP user!

Your ROM Monitor version is: 1.19

Your 405EP Evaluation Board has 134217728 bytes of DRAM installed.

Your Ethernet controller’s network address is: 1000abcdef55

usr_samp done!
```

The DRAM amount listed should match the amount installed on the board.

8.3.3 Building and Running the timesamp Program

The **timesamp.c** program is included as a sample to be built and run on the EVB. This program is an example of how to properly time a particular function or benchmark. The user must know and define the time base frequency (the number of times the time base register is updated per second) in the timesamp.c to ensure the timing calculations are accurate.

To build the timesamp program, enter the command **gnumake timesamp** from the command line while in the **samples** directory. The makefile will compile the **timesamp.c** file, link the resulting object file with the support libraries, and produce the boot file, **timesamp**, and the boot image file, **timesamp.img**.

If the suggested bootptab was used, then **timesamp.img** must be renamed or copied to **boot.img** in order to be selected by the ROM Monitor load process. Select option 0 from the ROM Monitor screen to load and run the image.

To load using RISCWatch, enable the ROM Monitor debugger (via option 5), exit the ROM Monitor menu (via option 0), start RISCWatch on the host system (make sure the RISCWatch environment file is setup for ROM Monitor communications), then use the RISCWatch **load image** command to load the **timesamp.img** file. Once successfully loaded, the **logoff** command can be issued to execute the program.

You should see the following messages (or ones like them) appear on the ROM Monitor screen.

```
Booting from [ENET] Ethernet...
Sending bootp request...

Loading file “\osopen\m405h_evb\samples\boot.img”...
Sending tftp boot request...
Transfer Complete...
Loaded successfully...
Entry point at 0x25e48...

Please give the number of runs through the benchmark:
```

At this point, enter the desired number of runs through the function or benchmark being timed. In this sample, the function being timed should execute for approximately a second, so a number between 1 and 10 would suffice.

8.3.4 Setting the time in the on-board clock

The battery-backed clock can be synchronised to real (wall-clock) time. A sample function to do this is provided in the samples file **utils.c**. The function *set_time_once_only()* requires that you edit its source code to provide the current time and date infomation. We suggest that you enter a time which is a couple of minutes into the future, to give you time to finish editing the file, recompile, link and download it onto the evaluation board. When the wall-clock time reaches the time that you set in the source code, run the function. This is a one-time only effort, as the battery will ensure that the clock remains set even when power is removed from the board.

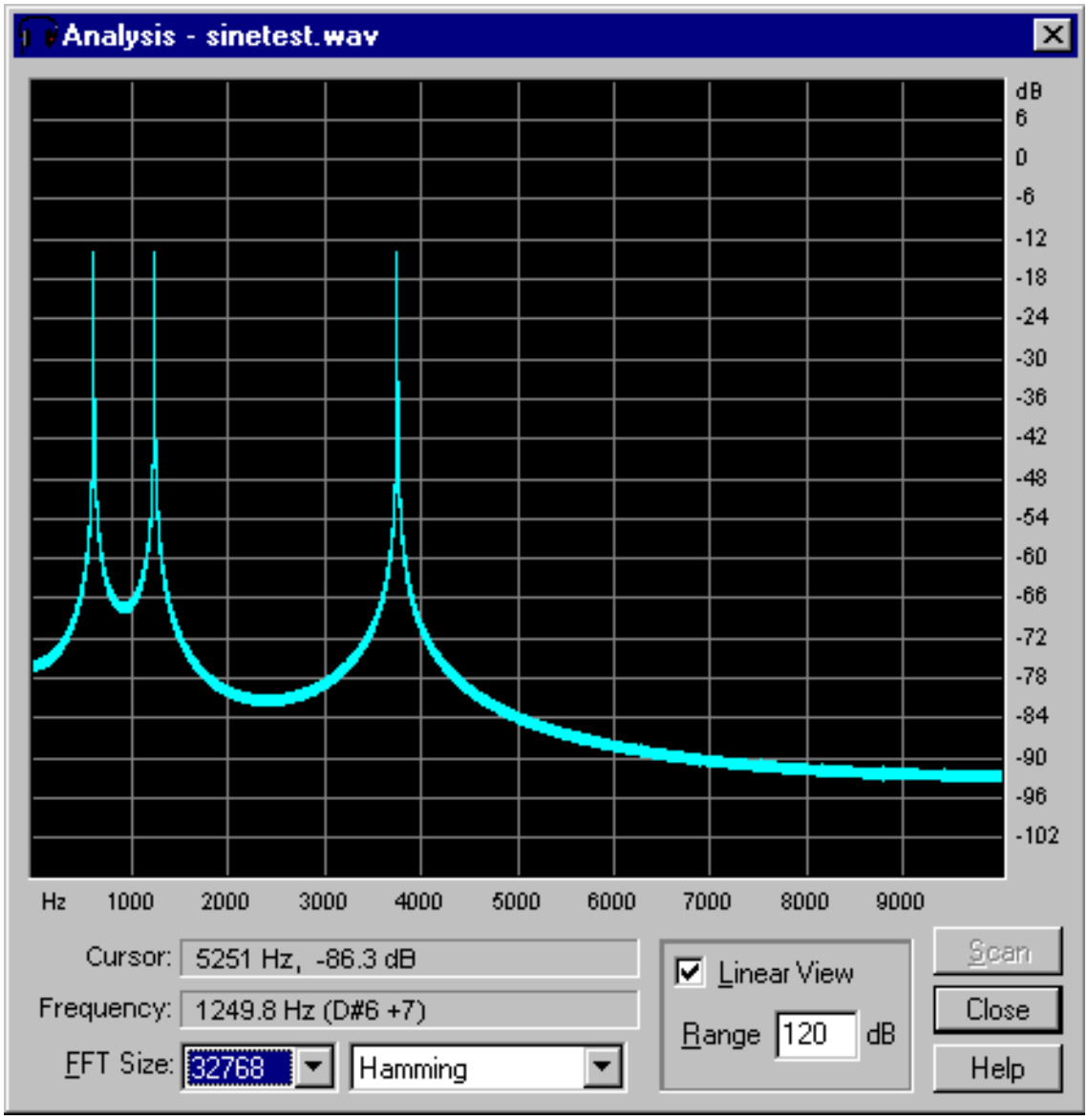
8.3.5 PPC405 MAC instruction sample

This sample program demonstrates the performance advantage of the 405 MAC (multiply/accumulate) instructions for common DSP operations. It is built in to the **applprog** sample image. Refer to the *OS Open User's Guide* for more information on building **applprog**.

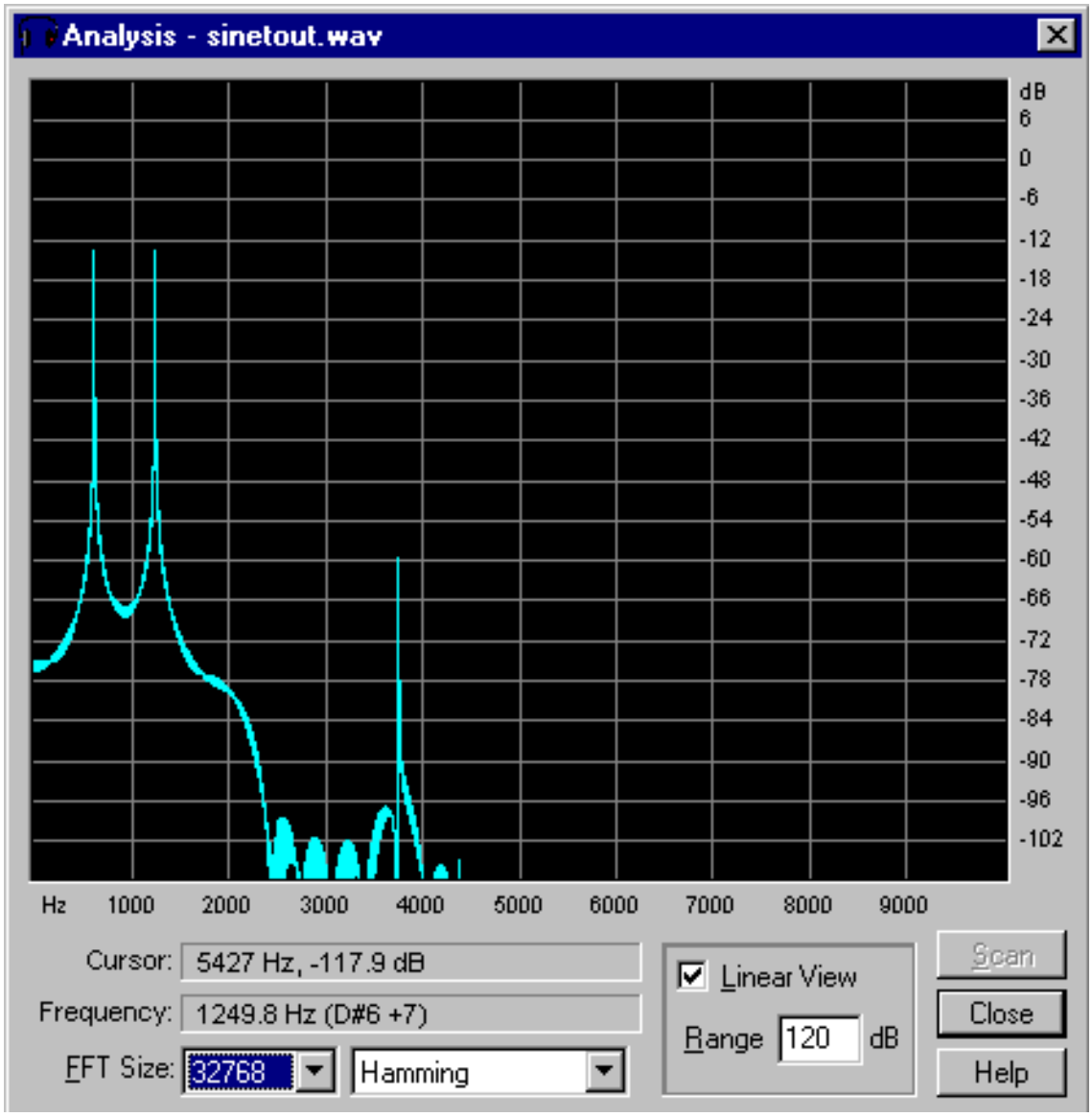
The easiest way to use the program is to call it from the OpenShell prompt as "macsamp()". It will then use standard input and output for the prompts and responses. No file system is required for the basic operation of the program, but if it is desirable to save the outputs, around 250 Kbytes of space is required in the current directory.

First, the program generates a 3 second sample data stream in storage. The sample consists of three sine waves (625, 1250, and 3750 Hz) sampled at 20 KHz using 16-bit signed samples. The program then allows the user to select one of two filter implementations, one using the MAC instructions and another one using the same underlying logic, but implemented using only basic PowerPC instructions. The filter is a 60th order lowpass FIR filter with a stopband gain of -70 db, passband edge at 1.5 KHz, and stopband edge at 3.0 KHz. The filter coefficients were calculated using the programs supplied with "Analog and Digital Filter Design using C" by Les Thede (Prentice Hall ISBN 0-13-352627-5). This book is an excellent reference in understanding the logic of the filter itself.

The cycle count (as derived from timebase values) for the filter operation is displayed, so by running the program twice, selecting each filter, the performance benefit of the MAC instructions is shown. The program also allows the original sample and the filter output to be saved as .WAV files, if a local file system exists. Curious users can transfer the files via FTP to a PC and hear the audible difference the lowpass filter makes. Shown below are frequency domain plots of the generated input sample and the filtered output.



The 3 sine waves are clearly shown in the input sample as being equal amplitude.



The output sample shows the first two sine waves virtually unchanged, but the signal at 3750 Hz has been significantly attenuated, as you would expect for the lowpass roll off beginning at 1.5 KHz. You can also see some amplitude ripples in the transition zone as a result of the filter method used.

8.4 Resolving Execution Problems

Configuration errors in the network or bootp tables cause most of the problems with running the sample applications. This section contains information that will aid users in identifying common problems.

8.4.1 Using the Ping Test on the ROM Monitor to Verify Connectivity

If the ping test fails, verify that TCP/IP is running on the host system and that the IP addresses on the selected interface are correct. The local address refers to the IP address of the evaluation board, and the remote refers to the host workstation address. The host workstation address must match the one selected during configuration of the host network interface. Also consult your TCP/IP documentation to insure proper network configuration.

8.4.2 Setup of bootp and tftp Servers (Daemons) for ROM Monitor Loads

Insure that the bootp and tftp servers are started on the host workstation. If possible, use the **tftp** command from another workstation to retrieve the load image. If this fails, make sure the image exists in the target directory and that it is readable by “others”. If the tftp transfer succeeds, check the bootptab entry in the **bootptab** file to insure that it specifies the correct interface and IP address of the evaluation board.

8.5 Using OS Open Functions

OS Open provides the following major classes of functions for the embedded programming environment:

- Thread management

  The unit of execution context for OS Open is the thread as defined by POSIX standards. Functions are provided to create threads with various scheduling and execution attributes. To manage the execution environment, serialization and synchronization primitives are part of OS Open. The system also provides functions to associate data with specific threads.

- Storage management

  OS Open supports variable block allocations in the form of a heap. Functions are provided to extend the heap, query heap usage, and allocate storage to meet alignment constraints. OS Open also provides an independent storage management mechanism to allocate fixed blocks of storage in constant time.

- Interrupt and fault support

  OS Open provides functions to attach user-written code to any of the processor exceptions and interrupts. Most of the functions of OS Open can be used in these interrupt handlers, except for those functions that suspend execution or are valid only in the context of an executing thread. When the underlying hardware platforms support it, OS Open platform-specific libraries provide additional functions to attach user-written code to external interrupts supported on the platforms.

- Clock and timer management

  OS Open functions provide time-of-day clock support and the ability to create, use, and destroy timers. These timers can be one-time or periodic.

- Device support

  OS Open functions support the installation of user-written device drivers to provide character special files, block special files, and logical file systems. Low-level POSIX I/O (read, write) as well as ANSI C stream (fget, fput) functions are provided for device and regular file access.

- ANSI C library support

OS Open provides a comprehensive set of ANSI C functions, providing support for string manipulation, memory management, string-to-number conversion, input/output, non-local jumps, and variable arguments.

- Pseudo device driver support

  OS Open provides several functions, such as TTY and DOS file system functions, that are installed and managed like device drivers, but they do not manipulate actual hardware nor do they have platform or device dependencies.

  OS Open provides functions that create and manage TCP/IP sockets. Network interface functions for Ethernet are also provided. With the TCP/IP protocol stack and network interfaces, additional functions are provided that implement several popular networking utilities, such as ping, ifconfig, ftp, and telnet.

- Debug functions and kernel abstract data types

  OS Open provides functions that set, clear, and query break points. OS Open features an internal circular trace buffer for operating system and user events. Also, functions are provided that dump kernel data objects in a readable form.

Additional information can be found in the *OS Open's User's Guide*.

Chapter 9. Application Libraries and Tools

This chapter describes some of the application libraries and tools available in the PPC405EP evaluation board kit board support software package. See the OS Open *User's Guide* and *Programmer's Reference* for additional information.

9.1 OS Open Libraries

The OS Open operating system is composed of a real-time executive and optional libraries of functions and macros.

The real-time executive provides a operating system core for embedded applications. Depending on an application's requirements, an embedded application may also incorporate one or more optional libraries.

This modular approach enables embedded system developers to scale an OS Open operating system to match their application requirements. Because unneeded features are not present, an OS Open configuration can provide savings in system hardware, initialization and reset time, and program size.

Table 9-1 summarizes the OS Open libraries, described in the *OS Open User's Guide* and in this user's guide. For detailed descriptions of the OS Open functions and macros, refer to the *OS Open Programmer's Reference*.

Table 9-1. OS Open Libraries

Library	File Name	Platforms
ANSI C Library	cLib.a	Common
ANSI C Math Library	mathLib.a	Common
ANSI C I/O Library	fsLib.a	Common
ROM Monitor Ethernet Library	benetLib.a	PPC405EP
Block Buffer Library	bbuffLib.a	Common
Block Library	blkLib.a	Common
Extended Heap Library	heapLib.a	Common
Boot Library(DRAM)	bootlLib.a	PPC405EP
Boot Library(FLASH)	bootrLib.a	PPC405EP
High C++™runtime support Library	cppLib.a, crti.o, crtn.o,mwdctorl.o	Common
Card Services/enabler software layer for PCMCIA support	csLib.a	Common

Table 9-1. OS Open Libraries (Continued)

Library	File Name	Platforms
Clock Support Library and NV-RAM	clockLib.a	PPC405EP
Debug Support Library	dbLib.a	Common
Device and File Support Library	devLib.a	Common
DOS File System Support Library	fatLib.a	Common
Dynamic Loader Library	ldrLib.a	Common
Ethernet Support Library	enetLib.a	PPC405EP
File Transfer Protocol Support Library	ftpLib.a	Common
Floating Point Library	fpeLib.a	Common
I2C Library	i2cLib.a	PPC405EP
Input/output Support Library	ioLib.a	PPC405EP
Kernel Abstract Data Types Library	kadtLib.a	Common
Network Support Library	netLib.a	Common
NFS Support Library	nfsLib.a	Common
OpenShell	shell.o	Common
PCI Library	pciLib.a	PPC405EP
PCMCIA ATA/IDE Hard disk device driver	pataLib.a	Common
PowerPC Low Level Access Support Library	ppcLib.a	PPC405EP
Queue Library	queLib.a	Common
RAM Disk Library	ramdLib.a	Common
Rate Monotonic Scheduling (RMS) Library	rmsLib.a	Common
Remote Source Level Debug Library	rsldLib.a	Common
Ring Buffer Library	rngLib.a	Common
RPC Support Library	rpcLib.a	Common
Runtime Library	runlib.a	Common
SCSI Support Library	scsiLib.a	Common
Serial Support Library	asyncLib.a	PPC405EP
Socket Services for PCMCIA support	ssLib.a	Common
Symbol Support Library	symLib.a	Common
TCP/IP Protocol Support Library	tcpipLib.a	Common
Telnet Daemon Support Library	tnetdLib.a	Common
Telnet Client Support Library	telnet.o	Common

Table 9-1. OS Open Libraries (Continued)

Library	File Name	Platforms
The Real-time Executive	rtx.o, rtxLib.a	Common
OS Open Minimal Kernel	rtxmin.o	Common
OS Open Kernel Extensions for the minimal kernel	rtxext.o	Common
Timer Tick Support	tickLib.a	PPC405EP
Trivial File Transfer Protocol	tftp.o	Common
TTY Support Library	ttyLib.a	Common

The real-time executive, the only required component in an OS Open operating system, provides a full set of basic operating system services:

- Thread management
- Virtual memory management for OS Open with Virtual Memory
- Storage management
- Signals
- Clocks and timers
- Interrupt and fault handling
- Message queues
- Semaphores
- Trace buffer support
- Miscellaneous services

The C functions for the real-time executive functions are in two libraries, **rtx.o** and **rtxLib.a**. The **rtx.o** library contains the OS Open real-time executive. The **rtxLib.a** library contains interface routines to OS Open functions, and is linked with application programs to resolve calls to the real-time executive.

9.2 Using Libraries and Support Software

The object libraries specific to the evaluation board are described below.

Table 9-2. OS Open Libraries for the PowerPC 405EP Evaluation Board Platform

Library	File Name
Boot Library(RAM)	bootlLib.a
Ethernet Device Driver Support Library	enetLib.a
Memory Access Layer Support Library	malLib.a
I2C Library	i2cLib.a

Table 9-2. OS Open Libraries for the PowerPC 405EP Evaluation Board Platform

Library	File Name
Input/Output Support Library	ioLib.a
PowerPC Low Level Access Support Library	ppcLib.a
Real-time Clock Interface Support Library	clockLib.a
ROM Monitor Ethernet Interface Library	benetLib.a
Serial Support Library	asyncLib.a
PCI Support Library	pciLib.a
On-chip Memory Support Library	ocmLib.a
Software Timer Tick Support Library	tickLib.a

9.2.1 Serial Port Support Library

This library supports the serial ports on the evaluation board. Use in conjunction with the function provided by **devLib.a** and **fsLib.a** to provide a high level I/O interface to application programs. The serial port support functions reside in the **asyncLib.a** library.

9.2.2 Boot Library (RAM)

This library contains the OS Open bootstrap program for the appropriate platform. The boot library performs initial processing to prepare the completed application program for execution on the board. For the evaluation board platform, this processing includes moving the loaded program such that real addresses correspond with addresses assumed by the language development tools. The boot library for the evaluation board platform also dynamically determines available heap space and prepares the symbol table for use by OS Open symbol management routines. The boot library does not export any functions.

9.2.3 Input/Output Support Library

The input/output functions reside in the **ioLib.a** library. To initialize the I/O subsystem, you must call **ioLib_init()** (normal mode) or **dbg_ioLib_init()** (ROM Monitor debug/ethernet) before performing any I/O other function.

9.2.4 I2C Library

This library supports reads and writes to devices on the I^2C bus. It also provides functions to directly access the I^2C registers. The I^2C library functions are in **i2cLib.a**.

9.2.5 PowerPC Low-Level Processor Access Support Library

The low-level access support library contains C-callable versions of the special PowerPC instructions. A few of the sample programs use these functions to manipulate the PPC405EP's special registers. These functions provide access to processor instructions not generated by compilers. For example, device drivers often have a requirement to control data caching, disable interrupts, synchronize I/O,

and other processor and platform-specific operations. The low-level access support functions reside in the **ppcLib.a** library.

9.2.6 ROM Monitor Ethernet IP Interface Library

This library contains routines allowing access to the ROM Monitor's Ethernet IP interface. These functions allow the Ethernet to be simply configured with a unique IP address for use with TCP/IP functions. The ROM Monitor Ethernet IP Interface functions reside in **benetLib.a** library. The **benetLib.a** functions are only available with OS Open without Virtual Memory.

9.2.7 Real-time Clock Interface Support Library

This library contains routines to read and set the evaluation board's battery-backed real-time clock. These functions are not to be confused with the real-time clock functions provided directly by OS Open when the system is running. The real-time clock interface support functions reside in the OS Open's **clockLib.a** library and are available to perform the following features:

- Set the OS Open clock from the real-time clock.
- Set the real-time clock from user-supplied data.
- Read and write NVRAM in the clock chip.

A sample function to set the battery-backed clock to wall-clock time is provide. See "Setting the time in the on-board clock" on page 8-7 for more information.

9.2.8 Ethernet Device Driver Support Library

This library provides support for the ethernet on the PPC405EP. The Ethernet device driver support functions reside in the enetLib.a library.

9.2.9 Software Timer Tick Support Library

The OS Open system requires a periodic call to **timertick_notify()** to maintain internal clocks and timer functions. The **tickLib.a** library contains an implementation of the **timertick_notify()** function for PowerPC architecture machines. Timer tick support functions reside in the **tickLib.a** library.

9.3 Device Drivers Supplied with the Board Support Software

Device drivers provided with the evaluation board support package include:

- Asynchronous
- Ethernet
- I2C

Examples and references are provided where appropriate. Users should also refer to the samples/thread0.c file for driver installation examples. Source code for each of the drivers is included in subdirectories under the samples directory.

For more information about any of the OS Open functions mentioned in this chapter, refer to the *OS Open Programmer's Reference*.

9.3.1 Asynchronous Device Driver

The asynchronous device driver supports the asynchronous communication ports found on the evaluation board. Following is a brief functional description of the device driver:

- Support from 50 baud
- Full duplex modem line control discipline
- Overrun error, parity error, and framing error detection
- BREAK interrupt detection
- Support for data length of 5, 6, 7, and 8 bits
- Support for 1, 1.5 and 2 stop bits
- Support for receive and transmit parity
- Support for odd and even parity
- Support for transmitting BREAK
- Support for 64 byte FIFO in the universal asynchronous receiver transmitter (UART)
- Programmed I/O (PIO) interrupt-driven slave communication
- Interrupt driven input/output
- Polled output functions

Since only full duplex modem line control discipline is supported, connection between the asynchronous port and another device must be made through a NULL modem. A NULL modem is a device that crosses transmitted data and received data pins to enable communication. The only time a NULL modem is not necessary is when connection is made to a real modem device.

Refer to the OS Open sample file thread0.c for an example of installing the asynchronous device driver and to samples/asyncLib for the driver source code.

9.3.1.1 Device Driver Installation

The asynchronous device driver is installed by calling **driver_install()**. Following is an example of asynchronous device driver installation.

```
#include <sys/asyncLib.h>
#include <ppcLib.h>
int devhandle;
rc=driver_install(&devhandle, async_init);
```

async_init() is declared in the file **<sys/asyncLib.h>** as follows.

```
int async_init(driver_t *dsw, va_list vargs)
```

Upon successful installation, **driver_install()** returns 0; otherwise –1 is returned. For more information on **driver_install()**, refer to the *OS Open Programmer's Reference*.

9.3.1.2 Device Installation

After the asynchronous device driver is installed, named devices can be created using **device_install()**. Following is an example of device installation.

```
rc=device_install("/dev/s0", CHRTYPE, devhandle, 1, 1024,
1024,asyncClockRate, UART0_BASE_ADDRESS, CPC0_CR0_UART0_EXTCLOCK_EN,
EXT_IRQ_COM1);
```

For device installation, *devhandle* is the value obtained from the **driver_install()**. Device type CHRTYPE is defined in **<sys/devDrivr.h>**.

Additional parameters passed in the **device_install()** call are as follows.

Table 9-3. Additional Parameters Passed to driver_install()

Parameter	Meaning
Fourth Parameter	Port number to be installed (1 or 2)
Fifth Parameter	Size of write buffer
Sixth Parameter	Size of read buffer
Seventh Parameter	Input clock for the divisor (value defined in ppcLib.h)
Eight Parameter	UART base register address (from ppcLib.h)
Ninth Parameter	UART-relevent bits to be set in the CPC0_CR0 register
Tenth Parameter	Interrupt IRQ_MIN < event <IRQ_MAX (from ioLib.h)
Note 1: These are positional parameters. **Note 2:** Write and read buffer sizes indicate number of characters that can be buffered in the device driver.	

Upon successful installation, **device_install()** returns 0; otherwise –1 is returned. When the device is installed, error reporting for the device is turned off and xon/xoff pacing is enabled. For more information on **device_install()**, refer to the *OS Open Programmer's Reference*.

9.3.1.3 Opening Asynchronous Communication Ports

After the device is installed, the **open()** system call can be used to open a particular device. Following is an example of the **open()** system call used against the asynchronous port.

```
fd1=open("/dev/s0", O_RDWR, asyncParityNone, asyncParityOdd,
         asyncStopBits1, asyncDataBits8, 9600);
```

Additional parameters passed in **open()** are as follows.

Table 9-4. Additional Parameters Passed to open()

Parameter	Meaning
First Parameter	Check/generate parity flag. Valid values are: asyncParityNone and asyncParityGen_Check
Second Parameter	Parity type. Valid values are asyncParityEven and asyncParityOdd. Because parameters are positional, this parameter must be specified even if parity is not used.
Third Parameter	Number of stop bits. Valid values are asyncStopBits1 and asyncStopBits2.
Fourth Parameter	Data length. Valid values are asyncDataBits5, asyncDataBits6, asyncDataBits7, and asyncDataBits8.
Fifth Parameter	Baud rate. Valid values range from 50 baud.
Note: These are positional parameters. All parameter constants can be found in **<sys/ioctl.h>**.	

Note: The *oflag* parameter, O_RDWR in this example, which is passed in the open call, is ignored by the device driver. When successful, **open()** returns a file descriptor, otherwise –1 is returned. **open()** can be called multiple times against the same asynchronous port. Communication parameters passed during the last **open()** call are set in the asynchronous port. For more information on **open()**, refer to the *OS Open Programmer's Reference*.

9.3.1.4 Reading and Writing

After successfully installing and opening the asynchronous port, **read()** and **write()** calls can be issued against that port. Multiple threads can issue **read()** and **write()** calls to the same port at the same time. However, simultaneous **read()** calls issued to the same port may block or be processed in an unexpected order. For these instances, thread scheduling and synchronization must be handled by the application.

Following is an example of **read()** and **write()** calls.

```
rc=write(fd1,"\nOS Open Real-time Executive\n", 29);
rc=read(fd1, buffer, 10);
```

fd1 is the value obtained from the **open()** call.

Note: For more information on **read()** and **write()**, refer to the *OS Open Programmer's Reference*.

9.3.1.5 I/O Control

An **ioctl()** call issued against asynchronous device driver accepts the commands listed in Table 9-5. All parameter constants can be found in **<sys/ioctl.h>**.

Table 9-5. ioctl() Commands for Asynchronous Device Drivers

Command	Parameters	Explanation
ASYNCBAUDSET	Value from 50	Sets baud rate
ASYNCBAUDGET	Pointer to integer	Returns baud rate
ASYNCTRIGSET	asyncFifoTrigger1, asyncFifoTrigger4, asyncFifoTrigger8, asyncFifoTrigger14	Sets FIFO trigger level for asynchronous port
ASYNCTRIGGET	Pointer to integer	Returns current trigger level
ASYNCBREAKSET	None	Starts sending BREAK on port
ASYNCBREAKCLR	None	Stops sending BREAK on port
ASYNCSTICKGET	Pointer to integer	Returns the way the parity bit is interpreted by the port
ASYNCSTICKZERO	None	Disables stick parity
ASYNCSTICKONE	None	Parity interpretation tracks even/odd parity
ASYNCRERRORGET	Pointer to integer	Returns and clears read error conditions. Values are defined in asyncLib.h
ASYNCWERRORGET	Pointer to integer	Returns and clears write error conditions. Values are defined in asyncLib.h
ASYNCERROREN	None	Enables error reporting
ASYNCERRORDIS	None	Disables error reporting. All pending errors are cleared
ASYNCERRORGET	Pointer to integer	Returns error reporting enabled flag
ASYNCDLENGET	Pointer to integer	Returns current data length
ASYNCDLENSET	asyncDataBits5, asyncDataBits6, asyncDataBits7, asyncDataBits8	Sets data length
ASYNCSTOPGET	Pointer to integer	Returns number of stop bits
ASYNCSTOPSET1	None	Sets number of stop bits to 1
ASYNCSTOPSET1_5	None	Sets number of stop bits to 1.5
ASYNCSTOPSET2	None	Sets number of stop bits to 2
ASYNCPARITYNONE	None	Disable parity
ASYNCPARITYGEN	None	Enable parity
ASYNCPARITYSGET	Pointer to integer	Return parity status (enabled/disabled)

Table 9-5. ioctl() Commands for Asynchronous Device Drivers (Continued)

Command	Parameters	Explanation
ASYNCPARITYODD	None	Sets parity to odd
ASYNCPARITYEVEN	None	Sets parity to even
ASYNCPARITYGET	Pointer to integer	Returns parity type
ASYNCXONENABLE	None	Enables XON/XOFF flow control
ASYNCXONDISABLE	None	Disables XON/XOFF flow control
ASYNCXONGET	Pointer to integer	Returns XON/XOFF flow control status
ASYNCMODEMSTAT	Pointer to integer	Returns modem status
ASYNCFLUSHIN	None	Flushes input buffer
ASYNCFLUSHOUT	None	Flushes output buffer
ASYNCDRAIN	None	Blocks until all characters in output buffer have been transmitted
ASYNCIGNBREAK	None	Ignores break interrupts
ASYNCSIGBREAK	None	Sends SIGINT on reception of break condition
ASYNCERRBREAK	None	Returns error from read upon reception of break condition. 0x00 is placed in the receive buffer at the position where break occurred.

Following is an example of an **ioctl()** call issued against an asynchronous device.

```
rc=ioctl(fd1, ASYNCXONDISABLE);
if (rc !=0) printf("ioctl failure\n");
```

fd1 is the value obtained from the **open()** call.

9.3.1.6 Polled Asynchronous I/O

A function is provided for polled output to s1 and s2 serial port.

```
int s1dbprintf(unsigned long uart_clock, unsigned char *base_reg,
unsigned long chcr0_reg, event_t int_level, const char *format, ...)
int s2dbprintf(unsigned long uart_clock, unsigned char *base_reg,
unsigned long chcr0_reg, event_t int_level, const char *format, ...)
```

The parameters passed to these functions are identical to **printf()** except for *uart_clock*, *base_reg*, *chcr0_reg*, and *int_level*. *uart_clock* specifies the clock speed, *base_reg* specifies the address of the base UART register, *chcr0_reg* specifies the bits in the CPC0_CR0 register that are to be set (only the bits relevant to the UART are altered), and *int_level* specifies the external interrupt level. The same values used on the device_install() function may be used. See “Device Installation” on page 9-6.

```
s1bdprintf(asyncClockRate, UART0_BASE_ADDRESS, CPC0_CR0_UART0_EXTCLOCK_EN,
EXT_IRQ_COM1, “hello world\n\r”);
```

Because polled I/O transmits characters synchronously, these functions may be called from first level interrupt handlers (FLIHs) or a user-supplied panic function. Since the function waits until the characters are actually sent before returning, use of this with long strings can significantly affect the timing of calling programs.

9.3.1.7 Flow control

The s1 port is a full 16550-compatible implementation, and supports all 16550 lines, including CTS, RTS, DTR and DSR.

However, the s2 serial port multiplexes the CTS/RTS and DTR/DSR hardware flow control signals onto the same pair of pins, so a choice must be made about which type of hardware flow control is to be used. This is implemented by setting bits in the CPC0_CR0 register. If hardware flow control is desired, it should be set by setting flags in the *chcr0_reg* parameter that is passed to **device_instal()** when installing the s2 port device. The flags available are:

- CPC0_CR0_UART1_CTS_RTS
- CPC0_CR0_UART1_DTR_DSR

One of these flags may be OR'd into any other values specified in the *chcr0_reg* parameter, as shown below:

```
rc=device_install("/dev/s1", CHRTYPE, devhandle, 1, 128, 128,
asyncClockRate, UART1_BASE_ADDRESS,
CPC0_CR0_UART1_EXTCLOCK_EN | CPC0_CR0_UART1_CTS_RTS, EXT_IRQ_COM2);
```

The device driver will automatically make sure that the selected signals appear on the correct pins on the s2 serial port connector, so that a normal serial connection can be made (no special cables required). The pin-switching is done via the on-board FPGA.

If neither hardware flow control option is selected the status of the flow control pins is undefined, and only software flow control (XON/XOFF) should be used.

9.3.2 I2C Device Driver

The I2C driver supports reading and writing to devices attached to the I^2C bus. The nature of the I^2C bus means that support is implemented as I^2C-specific functions, and not through the OS Open device driver model used for other device drivers.

9.3.2.1 Functional Description

- Allows master reads and writes
- Only supports 7 bit addresses
- Only supports slow (100kHz) bus

9.3.2.2 I2C Initialisation

The I2C device is initialised by a call to i2c_setupdriver(), passing in the base address for the memory-mapped I2C registers.

```
#include <sys/i2cLib.h>
#include <ppcLib.h>
rc=i2c_setupdriver(IIC_BASE_ADDRESS);
```

IIC_BASE_ADDRESS is defined by including <ppcLib.h>.

9.3.2.3 I2C read

Data is read from an I2C device by using the i2c_read() funtion. The caller supplies the device address and information about the read. This includes an optional subaddress which is required by some devices. A flags parameter is used to specify whether the subaddress is present or not. Also supplied are a pointer to a place to store the data and a count of how many bytes to read. Between 1 and 4 bytes may be read on each call.

If a subaddress is specified, the device driver first writes the subaddress to the target device, waits for the write to complete, then issues the read.

If the read completes successfully the function returns 0, otherwise it returns -1 if an error occurs, such as no response from the device within a timeout period.

Other flags which may be passed in include the ability to specify the values of the Chaining and Repeated Start bits in the I2C Control register. Constants for the flags values are in <sys/i2cLib.h>.

```
#include <sys/i2cLib.h>
int rc;
unsigned char device, subaddress;
unsigned char data[4];
...
/* Read 4 characters from the device, using the given subaddress */
rc=i2c_read(device, subaddress, 4, data, I2C_FLAGS_SUBADDR);
```

9.3.2.4 I2C write

Data is written to an I2C device with the i2c_write() function. The caller passes the device address and the data to be written, along with other information. This includes an optional device subaddress. The flags parameter specifies whether the subaddress is present. Also passed is the data to be written and the length of the data. A total of 4 bytes can be written on an I2C write, and this number includes the subaddress. So if no subaddress is specified, bewteen 1 and 4 bytes of data may be written. However, if a subaddress is specified, between 0 and 3 bytes of data are allowed. It is possible to only write the subaddress, with no accompanying data, which is why a data length of 0 is allowed only when a subaddress is specified.

As on a read, the flags parameter may specify the value of the Chaining and Repeated Start bits to be used in the I2C Control register.

```
#include <sys/i2cLib.h>
int rc;
unsigned char device, subaddress, device2;
unsigned char data[4];
...
/* Write 3 characters to the device, using the given subaddress */
rc=i2c_write(device, subaddress, 3, data, I2C_FLAGS_SUBADDR);
...
/* Write 4 characters to another device, without a subaddress */
rc=i2c_write(device2, 0, 4, data, 0);
```

9.3.2.5 Accessing I2C Registers

Functions are provided for directly reading and writing the I2C registers. The I2C registers values are specified in <sys/i2cLib.h>.

To read a register, use i2c_read_reg(), passing in the register name and pointer to a place to store the value.

```
#include <sys/i2cLib.h>
unsigned char reg_val;
i2c_read_reg(I2C_STATUS,&regval);
```

To write a value to a register, use i2c_write_reg(), passing in the name of the register and the data to be written to it.

```
#include <sys/i2cLib.h>
i2c_write_reg(I2C_LO_SLAVE_ADDR,0x42);
```

9.3.3 Ethernet Device Driver

The Ethernet device driver is a character device driver supporting packet level read/writes to the Ethernet controller. The driver features the ability to open multiple files. Each file receives packets for a specific standard Ethernet or 802.3 address.

Function highlights are:

- Up to eight receive channels
- Size of receive buffer pool determined by user at driver install time.

Refer to the OS Open sample file thread0.c for an example of installing the ethernet device driver and to samples/enetLib for the driver source code.

9.3.3.1 Device Driver Installation

The Ethernet device driver is installed by calling the driver_install() function. Following is an example of Ethernet device driver installation:

```
rc=driver_install(&devhandle_enet,
enet_init,                        /* device driver init routine */
ENET_RECEIVE_BUFFERS,             /* num_blocks;# of recv buffers*/
NULL,                             /* enet_descriptor pointer */
NULL,                             /* enet_buffer pointer */
board_config_ptr->mac_address);  /* mac_array */
```

num_blocks is the number of receive buffers used by the device driver. This value must be a multiple of 4.

enet_descriptor points to a physically contiguous portion of memory the device driver uses for receive and transmit buffer descriptors. The portion of memory must be at least (8 * *num_blocks*) + 32 bytes in size, and 32 byte aligned. If *enet_descriptor* is NULL, the device driver will attempt to allocate the needed space based on the value of *num_blocks*

enet_buffer points to a physically contiguous portion of memory the device driver uses for receive and transmit buffers. The portion of memory must be at least 296 * *num_blocks* + 1568 bytes, and 32 byte

aligned. If *enet_buffer* is NULL, the device driver will attempt to allocate the needed space based on the value of *num_blocks*.

.

Note: The device driver can not allocate memory that is guaranteed to be physically contiguous in OS Open with Virtual Memory, so in this case *enet_buffer* must point to the buffer to be used.

mac_array points to the 6 byte ethernet hardware address. Typically this value is obtained from the ROM Monitor's **get_board_cfg()** function.

Upon successful installation, **driver_install()** returns 0; otherwise -1 is returned. For more information about the **driver_install()** function, refer to the *OS Open Programmer's Reference* and the OS Open samples thread0.c file.

9.3.3.2 Device Installation

After the Ethernet device driver is installed, Ethernet devices can be installed using the **device_install()** function. Following is an example of device installation.

```
rc=device_install("/dev/en0", CHRTYPE, devhandle);
```

For device installation, *devhandle* is the value obtained from the **driver_install()**. Device type CHRTYPE is defined in **<sys/devDrivr.h>**.

Upon successful installation, **device_install()** returns 0; otherwise -1 is returned. At this point, files may be opened against the Ethernet device.

9.3.3.3 Opening and Closing Ethernet Files

After the device is installed, the **open()** system call can be used to open a particular device. Following is an example of the **open()** system call used to open an Ethernet port.

```
fd1=open("/dev/en0", O_RDWR);
```

When successful, **open()** returns the open file descriptor; otherwise -1 is returned. **open()** can be called multiple times against the same Ethernet device.

When using the **close()** function, the call to the driver-specific **close()** is deferred until all open files on the device are closed. This means that when an Ethernet file is closed, the channel address associated with the file will not be freed if another Ethernet file is open. Be aware that if the Ethernet interface has been connected to the TCP/IP protocol stacks via **enet_attach(), t**here will always be a file open against the Ethernet device, and therefore no channel addresses will be freed even if all the files the application opened are closed. To insure that the channel address will be freed, the ENET_CLEAR_CHANNEL **ioctl()** should always be called for an Ethernet file before closing it.

For more information about the **open()** and **close()** functions, refer to the *OS Open Programmer's Reference*.

9.3.3.4 Reading and Writing

After successfully installing and opening the Ethernet port, the **write()** function can be issued. The write buffer must contain a complete Ethernet packet. The universally administered address that was found in the ISA card read only storage (ROS) passed to **driver_install()** will be copied into the source address field by the device driver. There are prototype Ethernet header structures for both

standard Ethernet and 802.3 Ethernet packets in **<enet.h>**. Note that packets must be between 60 and 1514 byte in length (inclusive).

Before reading from the Ethernet file, an additional step must be performed. The Ethernet device driver supports up to 8 receive channels. What this means is that up to 8 files can be open for read or read/write simultaneously, and files will receive only those packets that have been selected for them. Packet selection is by packet type, in the case of standard Ethernet, and by destination SAP in the case of 802.3 Ethernet. The selection address is set with the ioctl ENET_SET_CHANNEL command, discussed below.

fd1 is the value obtained from the **open()** call.

```
fd1 = open(“/eno”,O_RDWR);
ioctl(fd,ENET_SET_CHANNEL,5,2);
   /* send packet from buffer */
write(fd,buffer,count);
   /* get received packet into buffer */
read(fd,buffer,count);
close(fd);
```

For more information on **read()** and **write()** functions, refer to the *OS Open Programmer’s Reference*.

9.3.3.5 I/O Control

The **ioctl()** call issued against the Ethernet device driver accepts the following commands. In each of these commands, *fd* is the value obtained from the **open()** call.

9.3.3.6 ENET_SET_CHANNEL

This command sets the receive channel address of the file. Once set, a receive channel address cannot be used in a subsequent ioctl ENET_SET_CHANNEL command unless it is first cleared with the ioctl ENET_CLEAR_CHANNEL command.

```
rc = ioctl(fd, ENET_SET_CHANNEL,
     packet_type,/* packet type is an unsigned integer containing
                 the channel address */
     type_length);/* specifies how many of the least sig bytes of
                 the packet type are to be used.Only values 1 and
                 2 are valid. */
```

A word about packet addresses. For standard Ethernet, the packet type is a 2-byte field right after the hardware source address. If *type_length* is 2, the *packet_type* parameter is assumed to refer to a standard Ethernet packet type. For a *type_length* of 1, the *packet_type* is assumed to contain a 1-byte destination SAP.

The incoming packets are differentiated as follows: For 802.3, there is a length field immediately after the source address. By convention, Ethernet packets are 1500 bytes or less, and valid Ethernet types are > 0x600. Hence, if the field after the source address is less than 0x600, the packet is assumed to be an 802.3 packet, and the 1 byte packet_type is compared against the destination SAP. Some reserved type values should not be generally used. They are defined in the file <netinet/if_ether.h>.

9.3.3.7 ENET_CLEAR_CHANNEL

This command clears the receive channel address of the file. This enables the device driver to free up internal resources and return any unread packets on this channel to the receive buffer pool. Once the receive channel address is cleared, it can be used again with the ioctl ENET_SET_CHANNEL command. The file can then be set to another receive channel as well.

```
rc = ioctl(fd, ENET_CLEAR_CHANNEL);
```

9.3.3.8 ENET_QUERY_ADDRESS

This **ioctl** command retrieves the universally administered address that was assigned during device_install.

```
unsigned char ua_address[6];
rc = ioctl(fd, ENET_QUERY_ADDRESS, ua_address);
```

The address is copied into the area supplied as the first data parameter to this **ioctl**.

9.3.3.9 MIB Functions

The ethernet device driver supports gathering of certain statistical information, as specified in IEEE 802.3 and RFC 1757: Remote Network Monitoring Managment Information Base. These functions each return a count of the statistic that they are measuring. The counts are not reset at any time - in order to determine how many evants have occurred since the last time the count was obtained, you must record the previous count and subtract it from the current value. You must handle counters which may wrap when they reach 2**32, such as aOctetsReceived, and be sure to sample at a frequent enough interval to avoid the counter cycling completely between samples.

For example, on a 100Mbps connection, a maximum of 100M bits can be processed per second, which is 12.5 M octets per second. The aOctetsReceived counter will count 2**32, or about 4.2 billion, octets before wrapping. Therefore you should sample the counter at an interval no greater than: (4,200,000,000/12,500,000) = 336 seconds, or just over 5 minutes.

The functions provided are listed below. More infomation about each one is available in “OS Open Function Reference” on page 10-1.

enet_get_aAlignmentErrors()
enet_get_aFrameCheckSequenceErrors()
enet_get_aFramesReceivedOK()
enet_get_aFramesTransmittedOK()
enet_get_aMultipleCollisionFrames()
enet_get_aOctetsReceived()
enet_get_aOctetsTransmitted()
enet_get_aSingleCollisionFrames()

9.3.4 ROM Monitor Ethernet Device Driver

The OS Open ROM Monitor Ethernet device driver provides network access to the applications running on the board while still allowing the ROM Monitor to access the RISCWatch debugger over the ethernet.

This device driver uses code resident in the ROM monitor to send and receive ethernet packets. A different IP address must be specified to distinguish the packets from ROM Monitor and OS Open. I/O initialization should be done by calling **dbg_ioLib_init()** rather than **ioLib_init()**.

The ROM Monitor Ethernet device driver is installed by calling **biosenet_attach()**. Following is a prototype of this function.

```
#include <benetLib.h>
int biosenet_attach(unsigned long ipaddr, int init_flag);
```

Upon successful installation, **biosenet_attach()** returns 0; otherwise -1 is returned. The IP address for OS Open is specified in the *ipaddr* parameter. The *init_flag* specifies whether the Ethernet controller needs to be initialized. If *init_flag* is set to 0 then the Ethernet controller is not initialized. If *init_flag* is set to a non-0 value, initialization of the Ethernet controller is performed. Please see samples/thread0.c for example code.

9.4 Environment Startup and Initialization

The following section describes the processing that occurs when the evaluation board environment is initialized.

Upon power-up or reset the ROM Monitor initializes the processor and other peripherals on the board. If a ROM Monitor load is attempted (via option 0), all enabled power-on tests are executed and, following their completion, a bootp request is sent to the host. This request involves an exchange of UDP packets corresponding to the bootp protocol. In essence, the ROM Monitor asks for and is supplied with the name of the boot image file on the host workstation. **tftp** (Trivial File Transfer Protocol) is then initiated by the ROM Monitor to transfer the boot image to the evaluation board.

Once the file has been transferred, two simple checks are made. A “magic number” in the boot image’s 32-byte header verifies that the image is one that can be loaded by the ROM Monitor (i.e., a file created by the eimgbld tool - see appendix B for details of the load format). The ROM Monitor also checks that the supplied boot image’s start address does not overlay sections of reserved DRAM. After the load is complete, control is transferred to the specified entry point in the boot image, which is in the bootstrap program.

When using RISCWatch’s *load image* command to load a boot image file, the debugger strips off the file’s 32-byte header and loads the remaining bytes of the file onto the board. The start address of the load is designated in bytes 4-8 of the header. Once loaded, the IAR register is set to the boot image’s entry point as defined in bytes 16-19 of the header. This entry point is in the bootstrap code. See the “Running Your Programs” section in the *RISCWatch Debugger User’s Guide* for additional information on loading files.

9.4.1 Board Bootstrap

The source for OS Open’s bootstrap code is included in the **samples\bootLib** directory. The bootstrap program performs the following functions:

1. Unpacks the boot image format, placing the **.text** and **.data** sections in the addresses specified at link time.
2. Modifies the kernel configuration block with new heap size and start address.
3. Sets the **.bss** section to zeros, in accordance with ANSI C requirements.

9.4.2 Environment Initialization

OS Open requires information about the system environment at initialization. The following source files, which are included with the samples, are used to supply that information and to establish the working environment.

basic_os.c	Contains pieces of config.c, io_init.c, panic.c, thread0.c, and utils.c to provide a minimal OS Open configuration
config.c	Configures the OS Open kernel
io_init.c	Initializes OS Open's I/O subsystem
network.c	Configures the host names and addresses for your environment
panic.c	Provides a sample panic function
thread0.c	Configures various features of OS Open (networking, remote debugger, etc.)
utils.c	Provides some useful utilities such as dir() to produce a directory listing

Additional information can be found in the "Configuring the OS Open Operating System" and "Developing OS Open Applications" chapters in the *IBM OS Open User's Guide*.

9.5 Tools

Several host based tools are provided to assist you in creating your own applications for the board. The tools can also be used for ROM program development.

9.5.1 elf2rom

elf2rom takes an ELF format executable file (output from the linker/binder), extracts the text and data sections, and writes them to a binary file. The resulting binary file can be programmed into ROM using a ROM programmer or the flash update utility included with board support software.

Syntax:

```
elf2rom [-v] [-d] [-p] [-s size] [-i offset] [-o output_file] input_elf
```

Description:

The program takes the input file *input_elf,* assumed to be an ELF file output from the linker, extracts the text and data sections, and writes them to the file*, output_file*. There are several optional flags that can affect **elf2rom** processing. They are described below.

-v	The verbose flag causes information about the generated output file to be written to stderr at the completion of the utility. This information includes the sizes and origins of the various sections and entry point.
-d	The debug flag will cause the symbol information from the input ELF file to be included after the data section in the output binary file.

-p	The promotion flag causes the data section to be aligned on a full word boundary if possible. This alignment facilitates full word moves of data to the appropriate target address without causing alignment exceptions.
-s	The size flag causes the output binary file to be padded to a particular size. This option is useful if it is necessary to create binary files that are the same size as a target ROM device. Error messages are generated if the generated image exceeds the specified size.
-i offset	The info flag places an information block into the output binary file at the specified offset. Since this info block overlays what is currently in the file at the specified offset, space should be reserved for its placement. The info block contains the following fields.
long block_id	Magic Number 0xBFAB0030
long entry_point	Entry point of image
long toc_ptr	Used for XCOFF; not used for ELF
long text_size	Size of text section in bytes also offset from beginning of image to data section
long text_p_addr	Text origin address as generated in ELF module
long data_size	Size of data section
long data_p_addr	Data origin as specified in generated ELF module
long bss_size	Size of bss section
long bss_p_addr	bss origin as specified in generated ELF module
long num_syms	Number of symbols from symbol section only valid if debug flag is set)
long sym_p_addr	Address of symbol table. Calculated as text origin + offset of symbols with created ROM image
long text_offset	Offset of text section from beginning of original ELF file. This information is required by certain debuggers
-o output_file	Allows the specification of an output file name. The default name is input_elf.img.
input_elf	This is simply the ELF binary input file. (elf2rom only)

Figure 9-1 shows the relationship of the various sections in the produced output file. The figure assumes that the info block flag [-i] was specified with an offset of 0x00.

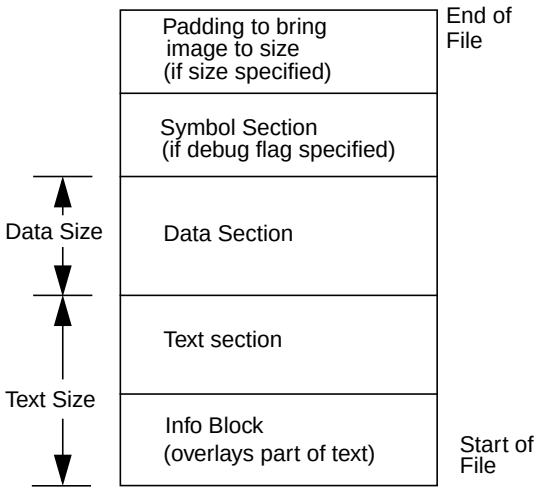


Figure 9-1. elf2rom Output File

Users can find an example of using **elf2rom** in the ROM Monitor's Makefile under **osopen/m405h_evb/openbios**.

9.5.2 hbranch14

hbranch14 places a branch at the end of a ROM image. **hbranch14** can also be used to store a communication device's network address in the ROM's Vital Product Data (VPD) area. **hbranch14** allows up to 4 network addresses.

Syntax:

hbranch14 [-v] [-s size] [-n net_addr] input_image

Description:

The program takes the input file *input_image* (which must be the output of **elf2rom** or eimgbld with an information block at 0x0 relative) pads it to size *size* and writes a relative branch to the entry point recorded in the end of the image. The entry point must be a label, not a function descriptor. There are several optional flags that can affect **hbranch14** processing. They are described below.

-v	The verbose flag causes information about the generated output image to be written to *stderr* at the completion of the utility. This information includes entry point information.
-s size	The size flag causes the image to be padded to a particular size. This facility is useful if it is necessary to create binary images that are the same size as a target ROM device.
-n net_addr	The network address flag stores net_addr, a 12 hex character network address (the media access control (MAC) address), in the VPD area in ROM. The ROM Monitor uses this option to store the EVB's ethernet controller's network address in its VPD. **hbranch14** allows up to 4 differrent network addresses

-p patch_file The patch file flag causes the file *patch_file* to be placed into the image just before the final branch and logically inserted into the instruction stream between the branch at the end of the file and the entry point. The patch file is inserted into the image “as is” and will usually contain the binary representation of position independent executable instructions. See Figure 9-2 for the details as to how normal hbranch processing is changed by a patch file.

input_image This is simply the source image file. The output is written to *stdout*.

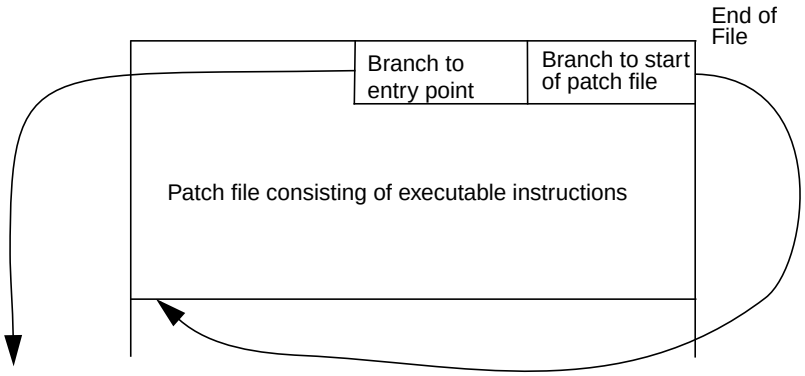


Figure 9-2. Detail of Patch File Placement

Figure 9-3 shows the relationship of the various sections in the produced output image.

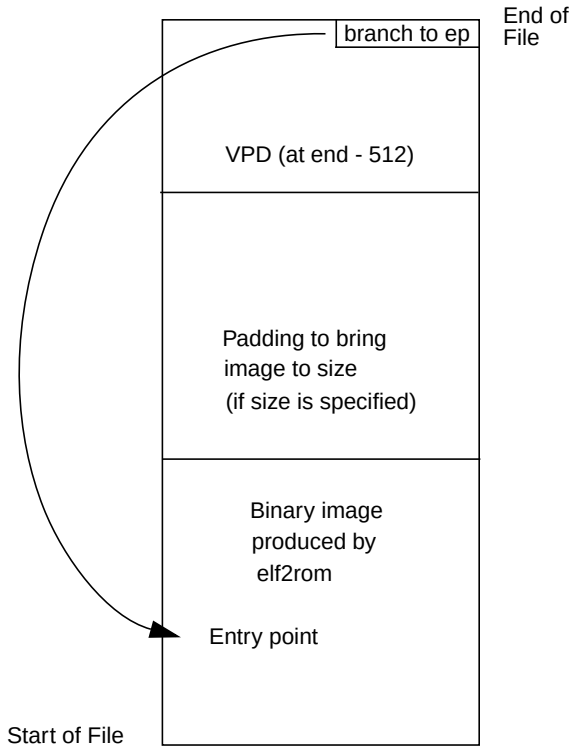


Figure 9-3. hbranch Output Image

Users can find an example of using hbranch in the ROM Monitor's Makefile under **osopen/m405h_evb/openbios**.

9.5.3 eimgbld

The **eimgbld** tool converts an output file from the linker/binder into the format used by the ROM Monitor to load programs from the host onto the evaluation board. The ELF file must be an otherwise executable file (with the text and data addresses bound at link time) and have space reserved after the entry point for the load information block (see "ROM Monitor Load Format" on page B-1 for more details). **eimgbld** sets the fields within the load information block so the application's bootstrap code can perform relocation.

Since the ROM loader does no relocation and simply transfers control to the application's entry point after a successful download, the application's entry point must point to suitable bootstrap code. It is the bootstrap code that relocates the application based on the data placed in the load information block by **eimgbld**.

Syntax:

eimgbld: [-D -P -S -v -b addr -m m_file -o o_file -s s_file -x x_file] input_elf

Description:

The program takes the input file *input_elf* (which must be the final ELF executable file produced from the build process) and converts it into the load format used by the ROM Monitor. There are several optional flags that can affect **eimbgld** processing:

-D	Set debug flag. A flag is set in the image causing the ROM Monitor debugger to be invoked immediately after the image is loaded.
-P	Creates output image in PReP format. PReP format is used by some PowerPC platforms.
-S	Suppress symbol information. Specifying this flag will prevent the symbol table from being included in the image.
-v	Verbose option. Directs information about the produced image to stderr.
-b addr	Set the symbol start location to address, addr.
-m m_file	Specify the ROM address map file. The format of this file is two addresses on each line (start address and ending address separated by a ",").
-o o_file	Allows the specification of an output file name. The default name is input_elf.img.
--s s_file	Restrict symbol table to names in specified file, s_name. The format of this file is one symbol on each line.
-x x_file	Suppress section names listed in specified file, x_name. The format of this file is one section name on each line.

Users can find an example of using **eimgbld** in the sample Makefile under **osopen/m405h_evb/samples**.

Chapter 10. OS Open Function Reference

This chapter describes the OS Open functions for the evaluation board. The function calls and macros are arranged alphabetically by name. For information about the effective use of some of these functions, refer to the microprocessor's user's manual.

All descriptions contain the following sections:

- Synopsis
- Library
- Description
- Errors
- Attributes

Examples and references are provided or referenced where appropriate.

10.1 Attributes and Threads

Functions and macros have attributes that affect thread execution. Depending on their behavior, functions may or may not be "async safe," "cancel safe," and "interrupt handler safe."

10.1.1 Async Safe Functions

An async safe function may be entered by two or more concurrently executing threads, with each thread getting the correct results.

Functions that operate only on disjoint or local data objects are reentrant, and are therefore async safe. For example, **ppcCntlzw()** operates only on its arguments, making it reentrant and therefore async safe.

Functions that operate on common or global data objects may use serialization techniques, such as mutexes and semaphores, within the functions to ensure async safe operation. **enet_send_packet()** uses the functions **semwait()** and **sempost()** to force serialization. Refer to the *OS Open User's Guide* for more information about the use of mutexes and semaphores.

10.1.2 Cancel Safe Functions

The cancel safe attribute is important only to threads executing in deferred cancelability mode (the cancel state is enabled; the cancel type is deferred).

A thread executing in deferred cancelability mode can execute a cancel safe function without being canceled. If the same thread executes a non-cancel safe function, the thread may or may not be canceled during execution of the function.

10.1.3 Interrupt Handler Safe Functions

An interrupt handler safe function may be called by a first level interrupt handler (FLIH).

10.1.4 Callable from Application Thread Group Functions

This attribute is only a concern when running OS Open with Virtual Memory. A function that is callable from an application thread group may be called from all thread groups. A function not callable from an application thread group will cause an exception if called from any thread group other than the kernel thread group.

10.2 Functions

Descriptions of the functions provided specifically to support the PPC405EP evaluation board kit are listed in alphabetical order in Table 10-1:

Table 10-1. Functions Specific to the PPC405EP Design Kit

Function or Macro	Description	Page
async_init()	Installs the asynchronous device driver	10-8
biosenet_attach()	Attaches the Ethernet to an IP address	10-9
clock_set()	Sets the OS Open POSIX clock to the value obtained from the battery operated real time clock	10-11
clockchip_get()	Reads the real-time clock	10-12
clockchip_nvram_read()	Reads bytes from the clock chip's NVRAM	10-13
clockchip_nvram_write()	Writes bytes to the clock chip's NVRAM	10-14
clockchip_start()	Starts the real-time clock	10-16
clockchip_stop()	Stops the real-time clock	10-17
clockLib_init()	Initializes the clockLib library routines	10-18
dbg_ioLib_init()	Initializes the I/O library when using ROM Monitor debugger or benetLib.a	10-19
dcache_flush()	Flushes cache lines, beginning at the effective address and continuing for a specified number of bytes	10-20
dcache_invalidate()	Invalidates cache lines beginning at the effective address and continuing for a specified number of bytes	10-21
dma_disable()	Disable a DMA channel	10-22
dma_setup()	Initialise a DMA channel for a transfer	10-23
dma_status()	Return status information for a DMA channel	10-24
enet_get_aAlignmentErrors()	Return MIB AlignmentErrors statistic	10-25
enet_get_aFrameCheckSequenceErrors()	Return MIB FrameCheckSequenceErrors statistic	10-26
enet_get_aFramesReceivedOK()	Return MIB FramesReceivedOK statistic	10-27

Table 10-1. Functions Specific to the PPC405EP Design Kit (Continued)

Function or Macro	Description	Page
enet_get_aFramesTransmittedOK()	Return MIB FramesTransmittedOK statistic	10-28
enet_get_aMultipleCollisionFrames()	Return MIB MultipleCollisionFrames statistic	10-29
enet_get_aOctetsReceived()	Return MIB OctetsReceived statistic	10-30
enet_get_OctetsTransmitteda()	Return MIB OctetsTransmitted statistic	10-31
enet_get_aSingleCollisionFrames()	Return MIB SingleCollisionFrames statistic	10-32
enet_init()	The Ethernet device driver initialization function	10-33
ext_int_config()	Configures the interrupt level specified by an eventl	10-34
ext_int_disable()	Disables the interrupt level specified by an event	10-35
ext_int_enable()	Enables the interrupt level specified by an event	10-36
ext_int_install()	Installs a first level interrupt handler (FLIH) for an event	10-37
ext_int_query()	Returns information about the FLIH	10-38
i2c_read()	Read data from an I^2C device	10-39
i2c_read_reg()	Read an I^2C register	10-40
i2c_setupdriver()	Initialise the I^2C device driver	10-41
i2c_write()	Write data to an I^2C device	10-42
i2c_write_reg()	Write to an I^2C register	10-43
inshort_swap()	Reads in a byte-swapped halfword	10-44
inword_swap()	Reads in a byte-swapped word	10-47
ioLib_init()	Initializes I/O library	10-48
malChannelActivate	Activates a MAL channel	10-49
malChannelDelete	Deletes a MAL channel	10-50
malChannelDescTblPtrGet	Retrieves the data pointer for a MAL Channel’s Descriptor Table	10-51
malChannelInit	Initialize a specific MAL channel for operation	10-52
malChannelIntMaskGet	Retrieves the MAL Channel’s interrupt mask	10-53
malChannelIntMaskSet	Sets the MAL Channel’s interrupt mask	10-54
malChannelStop	Stops a MAL channel	10-55
malInit	Initializes the MAL driver	10-56
malReset	Resets the MAL controller	10-57

Table 10-1. Functions Specific to the PPC405EP Design Kit (Continued)

Function or Macro	Description	Page
memcpy_io()	memcpy() for I/O areas	10-58
outshort_swap()	Writes out a byte-swapped halfword	10-59
outword_swap()	Write out a byte-swapped word	10-60
pci_find_device()	Finds a specified PCI device	10-61
pci_find_device_type()	Finds a specified type of PCI device	10-62
pci_get_io_base()	Returns a PCI I/O base address	10-63
pci_get_memory_base()	Returns a PCI memory base address	10-64
pci_init()	PCI initialization	10-65
pci_master_abort()	Looks for and clears a PCI master abort condition	10-66
pci_read_config_reg()	Reads from a PCI configuration register	10-67
pci_write_config_reg()	Writes to a PCI configuration register	10-68
ppcAbend()	Executes an invalid opcode forcing a program check interrupt	10-69
ppcAndMsr()	ANDs a value with the contents of the MSR	10-70
ppcCntlzw()	Counts consecutive leading zeros in a value	10-71
ppcDcbf()	Copies the cache block back to main storage (if the block resides in cache and has been modified with respect to main storage) and then invalidates the cache block	10-72
ppcDcbi()	Invalidates a cache block, discarding any modified contents if the block is valid in cache	10-73
ppcDcbst()	Copies a cache block, discarding any modified contents if the block is valid in cache	10-74
ppcDcbz()	Sets a cache block to 0	10-75
ppcDflush()	Flush and invalidate the data cache	10-76
ppcEieio()	Ensures that all storage references before the call finish before any storage references after the call start	10-77
ppcHalt()	Is a one instruction spin loop, effectively putting the processor in an enabled wait at the point of invocation	10-78
ppcIcbi()	Invalidates an instruction cache block	10-79
ppcIsync()	Causes the processor to discard any instructions that may have been prefetched	10-80
ppcMfccr0()	Returns the value of the CCR0 register	10-81
ppcMfdac1() - ppcMfdac2()	Returns the value of the DAC1 or DAC2 regsiter	10-82
ppcMfdbcr0() - ppcMfdbcr1()	Returns the value of the DBCR0 or DBCR1 regsiter	10-83

Table 10-1. Functions Specific to the PPC405EP Design Kit (Continued)

Function or Macro	Description	Page
ppcMfdbsr()	Returns the value of the DBSR register	10-84
ppcMfdccr()	Returns the value of the DCCR register	10-85
ppcMfdcr_any()	Returns the value of any DCR register	10-86
ppcMfdcwr()	Returns the value of the DCWR register	10-87
ppcMfdear()	Returns the value of the DEAR register	10-88
ppcMfdvc1() - ppcMfdvc2()	Returns the value of the DVC1 or DVC2 register	10-89
ppcMfesr()	Returns the value of the ESR register	10-90
ppcMfevpr()	Returns the value of the EVPR register	10-91
ppcMfgpr1()	Returns the value of GPR(1)	10-92
ppcMfgpr2()	Returns the value of GPR(2)	10-93
ppcMfiac1() - ppcMfiac4()	Returns the value of the IAC1 through IAC4 register	10-94
ppcMficcr()	Returns the value of the ICCR register	10-95
ppcMficdbdr()	Returns the value of the ICDBDR register	10-96
ppcMfmsr()	Returns the value of the MSR register	10-97
ppcMfpid()	Returns the value of the PID register	10-98
ppcMfpit()	Returns the value of the PIT register	10-99
ppcMfpvr()	Returns the value of the processor version register	10-100
ppcMfsgr()	Returns the value of the SGR register	10-101
ppcMfsler()	Returns the value of the SLER register	10-102
ppcMfsprg0()- ppcMfsprg7()	Returns the value of the special purpose register generals (SPRG0-SPRG7)	10-103
ppcMfsrr0()	Returns the value of SRR0	10-104
ppcMfsrr1()	Returns the current value of SRR1	10-105
ppcMfsrr2()	Returns the current value of SRR2	10-106
ppcMfsrr3()	Returns the current value of SRR3	10-107
ppcMfsu0r()	Returns the value of the SU0R register	10-108
ppcMftb()	Returns the current time base data	10-109
ppcMftcr()	Returns the value of the TCR register	10-110
ppcMftsr()	Returns the value of the TSR register	10-111
ppcMfzpr()	Returns the value of the ZPR register	10-112
ppcMtccr0()	Sets the value of the CCR0 register	10-113

Table 10-1. Functions Specific to the PPC405EP Design Kit (Continued)

Function or Macro	Description	Page
ppcMtdac1() - ppcMtdac2()	Sets the value of the DAC1 or DAC2 regsiter	10-114
ppcMtdbcr0() - ppcMtdbcr1()	Sets the value of the DBCR0 or DBCR1 regsiter	10-115
ppcMtdbsr()	Sets the value of the DBSR register	10-116
ppcMtdccr()	Sets the value of the DCCR register	10-117
ppcMtdcr_any()	Sets the value of any DCR register	10-120
ppcMtdcwr()	Sets the value of the DCWR register	10-121
ppcMtdear()	Sets the value of the DEAR register	10-122
ppcMtdvc1() - ppcMtdvc2()	Sets the value of the DVC1 or DVC2 register	10-123
ppcMtesr()	Sets the value of the ESR register	10-124
ppcMtevpr()	Sets the value of the EVPR register	10-125
ppcMtiac1() - ppcMtiac4()	Sets the value of the IAC1 through IAC4 register	10-126
ppcMticcr()	Sets the value of the ICCR register	10-127
ppcMtmsr()	Sets the MSR	10-128
ppcMtpid()	Sets the value of the PID register	10-129
ppcMtpit()	Sets the value of the PIT register	10-130
ppcMtsgr()	Sets the value of the SGR register	10-131
ppcMtsler()	Sets the value of the SLER register	10-132
ppcMtsprg0() - ppcMtsprg7()	Sets the special purpose register generals (SPRG0 - SPRG7)	10-133
ppcMtsrr0()	Sets the SRR0	10-134
ppcMtsrr1()	Sets the SRR1	10-135
ppcMtsrr2()	Sets the SRR2	10-136
ppcMtsrr3()	Sets the SRR3	10-137
ppcMtsu0r()	Sets the value of the SU0R register	10-138
ppcMttb()	Sets the current time base data	10-139
ppcMttcr()	Sets the value of the TCR register	10-140
ppcMttsr()	Sets the value of the TSR register	10-141
ppcMtzpr()	Sets the value of the ZPR register	10-142
ppcOrMsr()	Performs the OR of a value and the current MSR, updating the MSR	10-143
ppcSync()	Causes the processor to wait until all data cache lines scheduled to be written to main storage have actually been written	10-144

Table 10-1. Functions Specific to the PPC405EP Design Kit (Continued)

Function or Macro	Description	Page
s1dbprintf()	A version of **printf()** that may be used before I/O has been established	10-145
s2dbprintf()	A version of **printf()** that may be used before I/O has been established for serial port 2	10-147
timebase_speed()	Returns the speed of the timebase	10-148
timertick_install()	Installs and starts the timer tick handler	10-149
timertick_remove()	Removes the timer tick handler	10-150
vs1dbprintf()	A version of printf() that uses polled writes (no interrupts), and may be used before I/O has been established and accepts a va_list as a parameter instead of a variable number of parameters	10-151

Synopsis

```
#include <sys/asyncLib.h>
int driver_install(int *devhandle,async_init);
```

Library

asyncLib.a

Description

asyncLib.a is the asynchronous device driver that supports the asynchronous communication port on the PPC405EP evaluation board kit platform. **asyncLib.a** is installed by calling **driver_install()** with *devhandle* as the first parameter and **async_init** as the second parameter.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from Application Thread Group	No

References

driver_install(): *OS Open Programmer's Reference*

"Device Drivers Supplied with the Board Support Software" on page 9-5

Synopsis

```
#include <benetLib.h>
int biosenet_attach( unsigned long ipaaddr , int init_flag);
```

Library

benetLib.a

Description

biosenet_attach() attaches the TCP/IP protocol stack to the Ethernet device. The IP address should be different from the IP address defined to the 403 EVB ROM Monitor. init_flag determines if **biosenet_attach()** should initialize the Ethernet interface. The Ethernet device should be initialized only if OS Open was loaded through an interface other than Ethernet. A non-zero value will cause **biosenet_attach()** to initialize the Ethernet and a 0 value causes **biosenet_attach()** not to initialize the Ethernet interface. **biosenet_attach()** returns 0 if successful and -1 if it is unsuccessful.

Note 1: When using **biosenet_attach()** the I/O should be initialized by calling dbg_ioLib_init() rather than ioLib_init().

Note 2: **biosenet_attach()** is unavailable for OS Open with Virtual Memory.

Errors

None

Example

Initialize TCP/IP and define an IP address to biosenet_attach().

```
#include<sys/tcpipLib.h>

int rc;
rc=tcpip_init(ìmyhostnameî, 1 , 100);
if (rc!=0) {
return(-1);}
if (net_init() ) return(-1);
return(biosenet_attatch(0x07010104,0)); /* specify the IP addr. and the
init flag*/
```

Attributes

Async Safe	No
Cancel Safe	No
Interrupt Handler Safe	No
Callable from Application Thread Group	No

Processors

PowerPC 403GA	Yes
PowerPC 403GC	Yes
PowerPC 403GCX	Yes

References

“Ethernet Device Driver” on page 9-13

Synopsis

```
#include <clockLib.h>
int clock_set(void);
```

Library

clockLib.a

Description

clock_set() sets the OS Open POSIX clock to the value obtained from the battery operated real-time clock. The clockLib must be initialized by calling clockLib_init() prior to calling this function.

Errors

[EIO]	Real-time clock not running.

Attributes

Async Safe	Yes/No[1]
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

1. Not Async Safe in OS Open with Virtual Memory

References

“clockchip_set()” on page 10-15

“clockLib_init()” on page 10-18

Synopsis

```
#include <clockLib.h>
int clockchip_get( time_t *timeval );
```

Library

clockLib.a

Description

clockchip_get() reads the battery-backed real-time clock into the *timeval* structure supplied by the user. The clockLib library must be initialized by calling **clockLib_init()** prior to calling this function.

Errors

[EINVAL]	Library not initialized.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

“clockchip_set()” on page 10-15

“clockLib_init()” on page 10-18

clockchip_nvram_read()

Synopsis

```
#include <clockLib.h>
int clockchip_nvram_read( int index, unsigned char *buffer, int length );
```

Library

clockLib.a

Description

clockchip_nvram_read() reads non-volatile RAM from the clock chip. *index* specifies the starting byte of NVRAM, *buffer* points to the location where the bytes will be copied to and *length* specifies the maximum number of bytes to read. **clockchip_nvram_read()** returns the actual number of bytes read. The clockLib library must be initialized by calling **clockLib_init()** prior to calling this function.

Note: *index* must be within the range specified during **clockLib_init()**

Errors

[EINVAL]	Library not initialized or *index* out of range.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

“clockchip_nvram_write()” on page 10-14

“clockLib_init()” on page 10-18

clockchip_nvram_write()

Synopsis

```
#include <clockLib.h>
int clockchip_nvram_write( int index, unsigned char *buffer, int length );
```

Library

clockLib.a

Description

clockchip_nvram_write() writes non-volatile RAM in the clock chip. *index* specifies the starting byte of NVRAM, *buffer* points to the location where the bytes will be copied from and *length* specifies the maximum number of bytes to write. **clockchip_nvram_write()** returns the actual number of bytes written. The clockLib library must be initialized by calling **clockLib_init()** prior to calling this function.

Note: *index* must be within the range specified during **clockLib_init()**

Errors

[EINVAL]	Library not initialized or *index* out of range.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

“clockchip_nvram_read()” on page 10-13

“clockLib_init()” on page 10-18

clockchip_set()

Synopsis

```
#include <clockLib.h>
int clockchip_set( time_t timeval );
```

Library

clockLib.a

Description

clockchip_set() sets the battery-backed real-time clock to *timeval*, which should contain the number of seconds since January 1st, 1970 UTC.

Errors

[EIO]	Real-time clock not running.
[EINVAL]	Library not initialized.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“clock_set()” on page 10-11

clockchip_start()

Synopsis

```
#include <clockLib.h>
int clockchip_start( void );
```

Library

clockLib.a

Description

clockchip_start() starts the real-time clock. The clockLib library must be initialized by calling **clockLib_init()** prior to calling this function.

Errors

[EINVAL]	Library not initialized.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

“clockchip_stop()” on page 10-17

“clockLib_init()” on page 10-18

clockchip_stop()

Synopsis

```
#include <clockLib.h>
int clockchip_stop( void );
```

Library

clockLib.a

Description

clockchip_stop() stops the real-time clock. The clockLib library must be initialized by calling **clockLib_init()** prior to calling this function.

Errors

[EINVAL]	Library not initialized.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

“clockchip_start()” on page 10-16

“clockLib_init()” on page 10-18

Synopsis

```
#include <clockLib.h>
int clockLib_init( unsigned char *regbase, int reg_delta, int first_index, int
last_index);
```

Library

clockLib.a

Description

clockLib_init() initializes the clockLib library routines. *regbase* specifies the base address of the clock/nvram chip, *reg_delta* specifies the distance (in bytes) between each addressable byte in the chip. *first_index* and *last_index* indicate the range of bytes in the NVRAM that can be accessed by **clockchip_nvram_read()** and **clockchip_nvram_write()**. The range is specified using starting and ending index values (inclusive). **clockLib_init()** returns 0 if successful.

A constant defining the base address of the clock_nvram chip, RTC_NVRAM_BASE_ADDRESS, is specified by including **<ppcLIb.h>**.

Note: clockLib_init() should be called once at system initialization.

Errors

[EINVAL]	Already initialized or index out of range.

Example

clockLib_init(RTC_NVRAM_BASE_ADDRESS, 1 ,0 ,0x1ff7);

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes

References

“clock_set()” on page 10-11

“clockchip_get()” on page 10-12

“clockchip_nvram_read()” on page 10-13

“clockchip_nvram_write()” on page 10-14

“clockchip_set()” on page 10-15

“clockchip_start()” on page 10-16

“clockchip_stop()” on page 10-17

dbg_ioLib_init()

Synopsis

```
#include <ioLib.h>
int dbg_ioLib_init( void );
```

Library

ioLib.a

Description

dbg_ioLib_init() initializes the I/O library. Unlike **ioLib_init()**, this function allows external I/O interrupts to be screened by the ROM monitor, enabling debug to be performed from outside of the OS Open environment. Only external I/O through IRQ's other than those used by the ROM Monitor are available to OS Open.

If successful, **dbg_ioLib_init()** returns 0. Otherwise, **dbg_ioLib_init()** returns –1.

Errors

[ENOMEM]	Insufficient memory to allocate first level interrupt handler control areas.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

"ioLib_init()" on page 10-48

dcache_flush()

Synopsis

```
#include <ioLib.h>
void dcache_flush( void *address, unsigned int count );
```

Library

ioLib.a

Description

dcache_flush() flushes data cache lines, beginning at the effective address and continuing for *count* bytes.

A cache line flush forces the current contents of the cache line to main storage (if the line is valid and marked as modified) and then invalidates the line.

Note: Since cache flushes occur on cache line boundaries, the operation can occur outside of the bounds specified by the function call. For example, if *address* is X'216' and *count* is X'12', two cache lines, spanning addresses from X'200' to X'23F', would be flushed.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“dcache_invalidate()” on page 10-21

dcache_invalidate()

Synopsis

```
#include <ioLib.h >
void dcache_invalidate( void *address, unsigned int count );
```

Library

ioLib.a

Description

dcache_invalidate() invalidates data cache lines beginning at the effective address given by *address* and continuing for *count* bytes.

Note: Since cache invalidation occurs on cache line boundaries, invalidation can occur outside of the bounds implied by this command. For example, if *address* is X'104' and *count* is 16, the cache line spanning the addresses from X'100' to X'120' would be invalidated.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“dcache_flush()” on page 10-20

dma_disable()

Synopsis

```
#include <ioLib.h>
int dma_disable( unsigned int channel);
```

Library

ioLib.a

Description

dma_disable() disables the specified *channel* (0-3).

The **dma_disable()** function returns 0 if successful or -1 if *channel* is invalid.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“dma_setup()” on page 10-23

“dma_status()” on page 10-24

Synopsis

```
#include <ioLib.h>
int dma_setup( unsigned int channel, unsigned long dmacr, unsigned long count,
void* dst_address, void* src_address, struct dma_sg_t *dmasb);
```

Library

ioLib.a

Description

dma_setup() initialises a DMA channel for the specified transfer. *channel* specifies the DMA channel, *dst_address* the destination address, *src_address* the source address, *count* the length of the data transfer, *dmacr* the value to be written to the DMACRn register. *channel* must be a value 0-3, *count* must be greater than 0 and less than or equal to 65536. Note that the PW field in the *dmacr* register may affect the transfer size, so for memory-to-memory transfers the total data sent is the size specified by the PW field multiplied by the *count* value.

If *dmasb* is non-0, a scatter/gather transfer is used, and *dmasb* is the address of the first descriptor table element, which must have been initialised before calling **dma_setup()**. In this case the only other parameter that is used is *channel*, the others are ignored. Note that if you set an enable interrupt bit in a descriptor table element, you should install an interrupt handler to process the interrupt, using *ext_interrupt_install()*.

The **dma_setup()** function returns 0 if successful or -1 if *channel* is invalid.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

"dma_disable()" on page 10-22

"dma_status()" on page 10-24

"ext_int_install()" on page 10-37

dma_status()

Synopsis

```
#include <ioLib.h>
int dma_status( unsigned int channel, struct dma_stat * dstat);
```

Library

ioLib.a

Description

dma_status() returns status information from the specified *channel* (0-3). The structure pointed to by *dstat* is filled with status information. struct dma_stat is defined in **<ioLib.h>**.

The **dma_status()** function returns 0 if successful or -1 if *channel* is invalid.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“dma_disable()” on page 10-22

“dma_setup()” on page 10-23

enet_get_aAlignmentErrors()

Synopsis

```
#include <enet.h>
unsigned int enet_get_aAlignmentErrors( void);
```

Library

enetLib.a

Description

enet_get_aAlignmentErrors() returns the current value of the IEEE 802.3 clause 30 aAlignmentErrors counter. This is a 32 bit, non-resettable counter that contains the number of non-integral frames that are received and do not pass the frame check sequence (FCS) check.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

IEEE Std 802.3, 1998 Edition

“MIB Functions” on page 9-24

enet_get_aFrameCheckSequenceErrors()

Synopsis

```
#include <enet.h>
unsigned int enet_get_aFrameCheckSequenceErrors( void);
```

Library

enetLib.a

Description

enet_get_aFrameCheckSequenceErrors() returns the current value of the IEEE 802.3 clause 30 aFrameCheckSequenceErrors counter. This is a 32 bit, non-resettable counter that contains the number of integral frames that are received, but do not pass the frame check sequence (FCS) check.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

IEEE Std 802.3, 1998 Edition

“MIB Functions” on page 9-24

enet_get_aFramesReceivedOK()

Synopsis

```
#include <enet.h>
unsigned int enet_get_aFramesReceivedOK( void);
```

Library

enetLib.a

Description

enet_get_aFramesReceivedOK() returns the current value of the IEEE 802.3 clause 30 aFramesReceivedOK counter. This is a 32 bit, non-resettable counter that contains the number of frames that are successfully received. This counter does not include frames that are received with errors.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

IEEE Std 802.3, 1998 Edition

“MIB Functions” on page 9-24

enet_get_aFramesTransmittedOK()

Synopsis

```
#include <enet.h>
unsigned int enet_get_aFramesTransmittedOK( void);
```

Library

enetLib.a

Description

enet_get_aFramesTransmittedOK() returns the current value of the IEEE 802.3 clause 30 aFramesTransmittedOK counter. This is a 32 bit, non-resettable counter that contains the number of frames that are successfully transmitted.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

IEEE Std 802.3, 1998 Edition

“MIB Functions” on page 9-24

enet_get_aMultipleCollisionFrames()

Synopsis

```
#include <enet.h>
unsigned int enet_get_aMultipleCollisionFrames( void);
```

Library

enetLib.a

Description

enet_get_aMultipleCollisionFrames() returns the current value of the IEEE 802.3 clause 30 aMultipleCollisionFrames counter. This is a 32 bit, non-resettable counter that contains the number of frames that are involved in multiple collisions, but are subsequently transmitted successfully.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

IEEE Std 802.3, 1998 Edition

“MIB Functions” on page 9-24

enet_get_aOctetsReceived()

Synopsis

```
#include <enet.h>
unsigned int enet_get_aOctetsReceived( void);
```

Library

enetLib.a

Description

enet_get_aOctetsReceived() returns the total number of octets received, modulo 2**32. This is a 32 bit, non-resettable counter that contains the number of octets received, including those in bad packets. This is equivalent to the etherStatsOctets counter defined in RFC 1757.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

RFC 1757, Remote Network Monitoring Management Information Base, 1995

“MIB Functions” on page 9-24

enet_get_aOctetsTransmitted()

Synopsis

```
#include <enet.h>
unsigned int enet_get_aOctetsTransmitted( void);
```

Library

enetLib.a

Description

enet_get_aOctetsTransmitted() returns the total number of octets transmitted, modulo 2**32. This is a 32 bit, non-resettable counter that contains the number of octets transmitted, including retransmission of packets with collisions.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

IEEE Std 802.3, 1998 Edition

“MIB Functions” on page 9-24

enet_get_aSingleCollisionFrames()

Synopsis

```
#include <enet.h>
unsigned int enet_get_aSingleCollisionFrames( void);
```

Library

enetLib.a

Description

enet_get_aSingleCollisionFrames() returns the current value of the IEEE 802.3 clause 30 aSingleCollisionFrames counter. This is a 32 bit, non-resettable counter that contains the number of frames that are involved in a single collision, but are subsequently transmitted successfully.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

IEEE Std 802.3, 1998 Edition

“MIB Functions” on page 9-16

enet_init()

Synopsis

```
#include <enet.h>
int driver_install( int devhandle, enet_init, int num_blocks,
void *enet_descriptor, void *enet_buffer, char *mac_array);
```

Library

enetLib.a

Description

enetLib.a is the Ethernet device driver supporting packet level read/writes to the intergrated Ethernet controller. enetLib.a is installed by calling **driver_install()** with six parameters. The first parameter is the device handle, **devhandle**. The second parameter is the device driver initialization function, **enet_init**. The third parameter is the number of 256 byte buffers allocated for the Ethernet driver's use, **num_blocks**. The fourth parameter is the address of memory to use for buffer descriptors, **enet_descriptor**. The fifth parameter is the address of memory to use for buffers, **enet_buffer**. The sixth parameter is the location of the universal MAC address assigned to the Ethernet controller, **mac_array**.

Please see “Ethernet Device Driver” on page 9-13 for additional information.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No

References

driver_install() : OS Open Programmer’s Reference

“Ethernet Device Driver” on page 9-13

Synopsis

```
#include <ioLib.h>
void ext_int_config( int event , int flags);
```

Library

ioLib.a

Description

ext_int_config() configures the interrupt level specified by *event.* The items that can be configured are the polarity, trigger setting and whether the event is critical. These are specified by the *flags* parameter. **ioLib.h** defines the interrupt levels that can be configured.

The *flags* parameter can take any of the following values, which may be OR'd together:

EXTINT_NEG_ACTIVE or EXT_INT_POS_ACTIVE
EXTINT_LEVEL or EXT_INT_EDGE_TRIG
EXTINT_NON_CRITICAL or EXTINT_CRITICAL

The **ext_int_config()** function returns 0 if successful or -1 if *event* is invalid.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

"ext_int_enable()" on page 10-36

"ext_int_install()" on page 10-37

"ext_int_query()" on page 10-38

"ioLib_init()" on page 10-48

ext_int_disable()

Synopsis

```
#include <ioLib.h>
void ext_int_disable( int event );
```

Library

ioLib.a

Description

ext_int_disable() disables the interrupt level specified by *event.* **ioLib.h** defines the interrupt levels that can be disabled.

The **ext_int_disable()** function returns nothing.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“ext_int_enable()” on page 10-36

“ext_int_install()” on page 10-37

“ext_int_query()” on page 10-38

“ioLib_init()” on page 10-48

ext_int_enable()

Synopsis

```
#include <ioLib.h>
void ext_int_enable( int event );
```

Library

ioLib.a

Description

ext_int_enable() enables the interrupt level specified by *event.* **ioLib.h** defines the interrupt levels that can be enabled.

ext_int_enable() returns nothing.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“ext_int_install()” on page 10-37

“ext_int_query()” on page 10-38

“ioLib_init()” on page 10-48

ext_int_install()

Synopsis

```
#include <flih.h>
#include <ioLib.h>
int ext_int_install( int event, flih_t *new_flih, flih_t *old_flih );
```

Library

ioLib.a

Description

ext_int_install() installs a first level interrupt handler (FLIH) for the external interrupt *event*. **ioLib.h** defines the interrupt levels that can be set.

If *new_flih* is NULL, the current interrupt handler is removed for the specified event. If *new_flih* is non-NULL, it points to a **flih_t** structure containing the following fields:

flih_stack	Pointer to the first stack location; obtained by allocating memory and adding the size of the stack. flih_stack must be 16 byte aligned.
flih_function	Pointer to a function invoked when *event* occurs.
arg	A user-defined (void *) value passed to *flih_function*.

If *old_flih* is not NULL, the previous values of *flih_function*, *flih_stack*, and *arg* are stored in the structure pointed to by *old_flih*.

Note: to install an interrupt handler for other, non-external, interrupts, see **int_install()**.

If successful, **ext_int_install()** returns 0. Otherwise, **ext_int_install()** returns –1.

Errors

[EINVAL]	*event* does not refer to a valid event.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“ext_int_enable()” on page 10-36

“ext_int_query()” on page 10-38

“ioLib_init()” on page 10-48

“int_install()” on page 10-45

Synopsis

```
#include <ioLib.h>
#include <flih.h>
int ext_int_query( int event, flih_t *flih );
```

Library

ioLib.a

Description

ext_int_query() returns information about the first level interrupt handler (FLIH), if any, for *event*.

ioLib.h defines the events for which FLIHs can query.

The *flih* argument points to a **flih_t** structure containing the following fields:

flih_stack	Pointer to the first stack location; obtained by allocating memory and adding the size of the stack.
flih_function	Pointer to a function invoked when *event* occurs.
arg	A user-defined (void *) value passed to *flih_function*. If no FLIH is installed for the specified level, each field in the **flih_t** structure is assigned NULL.

If successful, **ext_int_query()** returns 0. Otherwise, **ext_int_query()** returns –1.

Errors

[EINVAL]	*event* does not refer to a valid event.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“ext_int_enable()” on page 10-36

“ext_int_install()” on page 10-37

“ioLib_init()” on page 10-48

Synopsis

```
#include <sys/i2cLib.h>
int i2c_read(unsigned char dev_addr, unsigned char dev_subaddr, unsigned char
count, unsigned char *data, unsigned char flags)
```

Library

i2cLib.a

Description

i2c_read() reads *count* (1 to 4) characters from the i2c device specified by *dev_addr*, and places them in the buffer pointed to by *data*. The optional subaddress specified by *dev_subaddr* may also be used by the device to determine which data to send.

flags may contain any of the following vlaues, OR'd together:

I2C_FLAGS_SUB_ADDR: a device subaddress is to be used, and is in *dev_subaddr*
I2C_FLAGS_CH: chaining - sets the Chain bit in the I2C Control register
I2C_FLAGS_REP_ST: repeated start - sets the Repeated Start bit in the I2C Control register.

The I2C driver must have been initialised with a call to i2c_setupdriver() before this function is called.

Returns 0 if successful and -1 otherwise.

Errors

None

Example

Read 2 bytes from the specified device, using the given subaddress.

```
#include <sys/i2cLib.h>
...
int rc;
unsigned char dev_addr, dev_subaddr;
unsigned char data[4];
rc=i2c_read(dev_addr,dev_subaddr,2,data,I2C_FLAGS_SUB_ADDR);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from Application Thread Group	Yes

References

"i2c_setupdriver()" on page 10-41

PPC405EP Embedded Processor User's Manual

i2c_read_reg()

Synopsis

```
#include <sys/i2cLib.h>
int i2c_read_reg(unsigned char reg, unsigned char *value)
```

Library

i2cLib.a

Description

i2c_read_reg() reads the contents of I2C register *reg*, and returns it in the character pointed to by *value*. *reg* must be in the range 0 to 15. Constants defining the register names are in <sys/i2cLib.h>.

Returns 0 if successful and -1 otherwise.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

i2c_setupdriver()

Synopsis

```
#include <sys/i2cLib.h>
int i2c_setupdriver(char * base_address)
```

Library

i2cLib.a

Description

i2c_setupdriver() initialises the I2C device driver. This function should be called before any other I2C functions are attempted. The *base_address* parameter contains the starting address of the memory-mapped IIC registers. This infomration may be obtained from processor documentation, and is also contained in the symbol IIC_BASE_ADDRESS, obtained by including file <ppcLib.h>.

Returns 0 if successful and -1 otherwise.

Errors

ENOSPC, ENOMEM: Insufficient memory

Example

Initialise the I2C driver.

```
#include <sys/i2cLib.h>
#include <ppcLib.h>
...
int rc;
rc=i2c_setupdriver(IIC_BASE_ADDRESS);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <sys/i2cLib.h>
int i2c_write(unsigned char dev_addr, unsigned char dev_subaddr, unsigned char
count, unsigned char *data, unsigned char flags)
```

Library

i2cLib.a

Description

i2c_write() writes *count* characters to the i2c device specified by *dev_addr*, from the buffer pointed to by *data*. The optional subaddress specified by *dev_subaddr* may be used by the device to determine where to place the data. If no device subaddress is used, the value of *count* may be 1 to 4. If a device subaddress is used, *count* may be 0 to 3. When *count* is 0, only the device subaddress is written.

flags may contain any of the following vlaues, OR'd together:

I2C_FLAGS_SUB_ADDR: a device subaddress is to be used, and is in *dev_subaddr*
I2C_FLAGS_CH: chaining - sets the Chain bit in the I2C Control register
I2C_FLAGS_REP_ST: repeated start - sets the Repeated Start bit in the I2C Control register.

The I2C driver must have been initialised with a call to i2c_setupdriver() before this function is called.

Returns 0 if successful and -1 otherwise.

Errors

None

Example

Write 2 bytes to the specified device, using the given subaddress.

```
#include <sys/i2cLib.h>
int rc;
unsigned char dev_addr, dev_subaddr;
unsigned char data[4];
rc=i2c_write(dev_addr,dev_subaddr,2,data,I2C_FLAGS_SUB_ADDR);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from Application Thread Group	Yes

References

"i2c_setupdriver()" on page 10-41

PPC405EP Embedded Processor User's Manual

i2c_write_reg()

Synopsis

```
#include <sys/i2cLib.h>
int i2c_write_reg(unsigned char reg, unsigned char value)
```

Library

i2cLib.a

Description

i2c_write_reg() writes *value* to I2C register *reg*. *reg* must be in the range 0 to 15. Constants defining the register names are in <sys/i2cLib.h>.

Returns 0 if successful and -1 otherwise.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	No
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ioLib.h>
unsigned short inshort_swap(unsigned short * address)
```

Library

ioLib.a

Description

inshort_swap() returns a halfword read from the I/O port specified by *address*. The halfword is byte-reversed, by using the **lhbrx** instruction.

After the halfword is read, the PowerPC **eieio** instruction is issued to enforce in-order execution of I/O.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“outshort_swap()” on page 10-59

“inword_swap()” on page 10-47

inshort(): *OS Open Programmer’s Reference*

lhbrx instruction in *PPC405EP Embedded Processor User’s Manual*

Synopsis

```
#include <flih.h>
int int_install( int event, flih_t *new_flih, flih_t *old_flih );
```

Library

rtxLib.a

Description

int_install() installs a first level interrupt handler (FLIH) for *event*. **flih.h** defines the interrupt levels that can be set.

If *new_flih* is NULL, the current interrupt handler is removed for the specified event. If *new_flih* is non-NULL, it points to a **flih_t** structure containing the following fields:

flih_stack	Pointer to the first stack location; obtained by allocating memory and adding the size of the stack.
flih_function	Pointer to a function invoked when *event* occurs.
arg	A user-defined (void *) value passed to *flih_function*.

If *old_flih* is not NULL, the previous values of *flih_function*, *flih_stack*, and *arg* are stored in the structure pointed to by *old_flih*.

Note: to install an interrupt handler for a device which generates an external interrupt (one handled by the Universal Interrupt Controller, UIC) use the function **ext_int_install()**.

If successful, **int_install()** returns 0. Otherwise, **int_install()** returns –1.

Errors

[EINVAL]	*event* does not refer to a valid event.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“int_query()” on page 10-46

“ioLib_init()” on page 10-48

“ext_int_install()” on page 10-37

Synopsis

```
#include <flih.h>
int int_query( int event, flih_t *flih );
```

Library

rtxLib.a

Description

int_query() returns information about the first level interrupt handler (FLIH), if any, for *event*.

flih.h defines the events for which FLIHs can query.

The *flih* argument points to a **flih_t** structure containing the following fields:

flih_stack	Pointer to the first stack location; obtained by allocating memory and adding the size of the stack.
flih_function	Pointer to a function invoked when *event* occurs.
arg	A user-defined (void *) value passed to *flih_function*. If no FLIH is installed for the specified level, each field in the **flih_t** structure is assigned NULL.

If successful, **int_query()** returns 0. Otherwise, **int_query()** returns –1.

Errors

EINVAL]	*event* does not refer to a valid event.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“int_install()” on page 10-45

“ioLib_init()” on page 10-48

Synopsis

```
#include <ioLib.h>
unsigned long inword_swap(unsigned long * address)
```

Library

ioLib.a

Description

inword_swap() returns a word read from the I/O port specified by *address*. The word is byte-reversed, by using the **lwbrx** instruction.

After the word is read, the PowerPC **eieio** instruction is issued to enforce in-order execution of I/O.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“outword_swap()” on page 10-60

“inshort_swap()” on page 10-44

inword(): *OS Open Programmer’s Reference*

lwbrx instruction in *PPC405EP Embedded Processor User’s Manual*

Synopsis

```
#include <ioLib.h>
int ioLib_init( void );
```

Library

ioLib.a

Description

ioLib_init() initializes the I/O library.

If successful, **ioLib_init()** returns 0. Otherwise, **ioLib_init()** returns –1.

ioLib_init() should not be used when using the ROM Monitor Ethernet interface or the ROM monitor debugger. **dbg_ioLib_init()** should be used instead.

Errors

[ENOMEM]	Insufficient memory to allocate first level interrupt handler control areas.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“dbg_ioLib_init()” on page 10-19

malChannelActivate()

Synopsis

```
#include <malLib.h>
int malChannelActivate(MAL_DATA * mdata, UINT channel_type, UINT
channel_number);
```

Library

malLib.a

Description

malChannelActivate() activates a MAL channel. The channel must have been previously initialized with the malChannelInit() function.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <malLib.h>
int malChannelDelete(MAL_DATA * mdata, UINT channel_type, UINT
channel_number);
```

Library

malLib.a

Description

malChannelDelete() removes a MAL channel from the set of channels MAL is currently managing. The channel must not be active when this function is called.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

malChannelDescPtrGet()

Synopsis

```
#include <malLib.h>
int malChannelDescPtrGet(MAL_DATA * mdata, UINT channel_type, UINT
channel_number, MAL_BD ** descTblAdrs);
```

Library

malLib.a

Description

malChannelDescPtrGet() returns the address of the descriptor table array that was allocated for the channel in the malInit() function.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <malLib.h>
int malChannelInit(MAL_DATA * mdata, MAL_CHANNEL *mchannel);
```

Library

malLib.a

Description

malChannelInit() initializes a MAL channel and readies it for operation. It does not start the channel.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

malChannelIntMaskGet()

Synopsis

```
#include <malLib.h>
int malChannelIntMaskGet(MAL_DATA * mdata, UINT channel_type, UINT
channel_number, UINT * intmask);
```

Library

malLib.a

Description

malChannelIntMaskGet() returns the interrupt mask for a channel.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <malLib.h>
int malChannelIntMaskSet(MAL_DATA * mdata, UINT channel_type, UINT
channel_number, UINT * intmask);
```

Library

malLib.a

Description

malChannelIntMaskGet() sets the interrupt mask for a channel.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

malChannelStop()

Synopsis

```
#include <malLib.h>
int malChannelStop(MAL_DATA * mdata, UINT channel_type, UINT channel_number);
```

Library

malLib.a

Description

malChannelStop() stops an active MAL channel.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <malLib.h>
int malInit(MAL_DATA * mdata);
```

Library

malLib.a

Description

malInit() initializes the Memory Access Layer (MAL) core.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

malReset()

Synopsis

```
#include <malLib.h>
int malReset(MAL_DATA * mdata);
```

Library

malLib.a

Description

malReset() performs a software reset on the Memory Access Layer (MAL) core.

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

memcpy_io()

Synopsis

```
#include <ioLib.h>
int memcpy_io(void * target, void * source, size_t length);
```

Library

ioLib.a

Description

memcpy_io() is provided for compatiblity with some other paltforms whcih require special handling for copying which involves I/O space. In this platform **memcpy_io()** behaves the same as **memcpy().**

Errors

None

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

memcpy(): *OS Open Programmer's Reference*

outshort_swap()

Synopsis

```
#include <ioLib.h>
void outshort_swap(unsigned short * address, unsigned short data)
```

Library

ioLib.a

Description

outshort_swap() writes the halfword containing *data* to the I/O port specified by *address*. The halfword is byte-reversed, by using the **sthbrx** instruction.

After the halfword is written, the PowerPC **eieio** instruction is issued to enforce in-order execution of I/O.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“inshort_swap()” on page 10-44

“outword_swap()” on page 10-60

outshort(): *OS Open Programmer’s Reference*

sthbrx instruction in *PPC405EP Embedded Processor User’s Manual*

Synopsis

```
#include <ioLib.h>
void outword_swap(unsigned long* address, unsigned long data)
```

Library

ioLib.a

Description

outword_swap() writes the word containing *data* to the I/O port specified by *address*. The word is byte-reversed, by using the **stwbrx** instruction.

After the word is written, the PowerPC **eieio** instruction is issued to enforce in-order execution of I/O.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“inword_swap()” on page 10-47

“outshort_swap()” on page 10-59

outword(): *OS Open Programmer’s Reference*

stwbrx instruction in *PPC405EP Embedded Processor User’s Manual*

pci_find_device()

Synopsis

```
#include <sys/pciLib.h>
int pci_find_device( unsigned short vendorid, unsigned short deviceid,
unsigned int *nextp);
```

Library

pciLib.a

Description

pci_find_device() searches the PCI devices on the system looking for one which matches the *vendorid* and *deviceid*. The *vendorid* is compared to the Vendor ID field on each PCI device on the system, and the *deviceid* is compared to the Device ID field.

The value pointed to by *nextp* determines the where the search starts. To find the first device on the system that matches, **nextp* should be PCI_NEXT_INIT. When the search completes successfully, **nextp* is updated with the location of the device. On a subsequent call to this function using the same **nextp*, the search starts at the device after the last one that was found. In this way, all devices which match the search criteria may be found. When no device is found, *nextp is set to PCI_NEXT_INIT.

If successful, returns an integer containing the bus and device numbers of the found device. For the format of this integer, see the description of *bus_dev_func* in **pci_read_config_reg()**.

Errors

Returns -1 if device is not found.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PCI Local Bus Specification, Revision 2.1

“pci_find_device_type()” on page 10-62

“pci_read_config_reg()” on page 10-67

Synopsis

```
#include <sys/pciLib.h>
int pci_find_device_type( int class_code, unsigned int *nextp);
```

Library

pciLib.a

Description

pci_find_device_type() searches the PCI devices on the system looking for one which matches the *class_code*. The class code consists of 3 bytes, as defined in the PCI specification. The two most significant, the base class and sub-class, must exactly match the corresponding fields in the class code field on the PCI device. The least significant field, interface, is a bit map. In order for the device to completely match the *class_code*, it must have at least the interface bits set that are specified in the *class_code* interface map.

In the four-byte *class_code* variable, the most significant byte is unused, the next byte contains the base class, next is the sub-class, and the least significant byte contains the interface byte.

The value pointed to by *nextp* determines the where the search starts. To find the first device on the system that matches, **nextp* should be PCI_NEXT_INIT. When the search completes successfully, **nextp* is updated with the location of the device. On a subsequent call to this function using the same **nextp*, the search starts at the device after the last one that was found. In this way, all devices which match the search criteria may be found. When no device is found, *nextp is set to PCI_NEXT_INIT.

If successful, returns an integer containing the bus and device numbers of the found device. For the format of this integer, see the description of *bus_dev_func* in **pci_read_config_reg()**.

Errors

Returns -1 if device is not found.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PCI Local Bus Specification, Revision 2.1

“pci_find_device()” on page 10-61

“pci_read_config_reg()” on page 10-67

pci_get_io_base()

Synopsis

```
#include <sys/pciLib.h>
int pci_get_io_base( int base_addr );
```

Library

pciLib.a

Description

pci_get_io_base() returns the base I/O address for the PCI address specified by *base_addr*. Typically this is used to determine where I/O space starting at address 0 appears in the CPU memory map.

Errors

Returns -1 if no base address matches *base_addr*.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“pci_get_memory_base()” on page 10-64

Synopsis

```
#include <sys/pciLib.h>
int pci_get_memory_base( int base_addr );
```

Library

pciLib.a

Description

pci_get_memory_base() returns the base CPU (PLB) memory address for the PCI address specified by *base_addr*. A typical use for this is used to determine where PCI memory space starting at address 0 appears in the CPU memory map.

Errors

Returns -1 if no base address matching *base_addr* is mapped.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“pci_get_io_base()” on page 10-63

pci_init()

Synopsis

```
#include <sys/pciLib.h>
int pci_init( void );
```

Library

pciLib.a

Description

pci_init() initialises the PCI controller as a master.

Errors

None.

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

pci_master_abort()

Synopsis

```
#include <sys/pciLib.h>
int pci_master_abort( void );
```

Library

pciLib.a

Description

pci_master_abort() tests if a master abort happened during a previous PCI master access, and clears the error if so. Returns 0 if there was no master abort, returns -1 if there was.

Errors

None.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

pci_read_config_reg()

Synopsis

```
#include <sys/pciLib.h>
unsigned int pci_read_config_reg( int bus_dev_func, int reg, int width );
```

Library

pciLib.a

Description

pci_read_config_reg() reads a register, *reg*, from the device specified by *bus_dev_func*. The amount of data read is specified by *width*, and may be 1, 2, or 4 bytes. For 2 or 4 byte reads, *reg* must be appropriately aligned.

bus_dev_func contains the identifier for a device, consisting of the bus and device numbers. They are placed within the word so as to be able to be used directly by the PCI Configuration Address Register. The bus number is placed in bits 23:16, the device number in bits 15:11 (using PCI bit notation where bit 31 is most significant).

Returns the value of the specified register.

Errors

Returns -1 if *width* is not 1, 2, or 4.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“pci_write_config_reg()” on page 10-68

Synopsis

```
#include <sys/pciLib.h>
int pci_write_config_reg( int bus_dev_func, int reg, unsigned int value, int
width );
```

Library

pciLib.a

Description

pci_write_config_reg() writes *value* to a register, *reg*, in the device specified by *bus_dev_func*. The amount of data read is specified by *width*, and may be 1, 2, or 4 bytes. For 2 or 4 byte writes, *reg* must be appropriately aligned. For the format of *bus_dev_func*, see the description in **pci_read_config_reg()**. Returns 0 if successful.

Errors

Returns -1 if *width* is not 1, 2, or 4.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“pci_read_config_reg()” on page 10-67

ppcAbend()

Synopsis

```
#include <ppcLib.h>
void ppcAbend(void)
```

Library

ppcLib.a

Description

ppcAbend() executes an invalid opcode forcing a Program Check interrupt.

Errors

None

Example

Force an illegal instruction exception.

```
ppcAbend()
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcAndMsr(unsigned long value);
```

Library

ppcLib.a

Description

ppcAndMsr() ANDs *value* with the contents of the MSR.

The MSR is updated with the result of the AND operation.

ppcAndMsr() returns the previous contents of the MSR.

Refer to the **<ppcLib.h>** file for the defines of the MSR constants.

Errors

None

Example

Disable external interrupts.

```
unsigned long orig_msr = ppcAndMsr(~ppcMsrEE);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“ppcOrMsr()” on page 10-143

“ppcMtmsr()” on page 10-128

PPC405EP Embedded Processor User’s Manual

ppcCntlzw()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcCntlzw(unsigned long value);
```

Library

ppcLib.a

Description

ppcCntlzw() counts consecutive leading zeros in *value.*

ppcCntlzw() returns the count, which ranges from 0 through 32, inclusive.

Errors

None

Example

Return count of leading zeros in variable k.

```
int k;
 unsigned long k = ppcCntlzw(0x0700AA55); /* k = 5 */
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcDcbf(void *addr);
```

Library

ppcLib.a

Description

ppcDcbf() copies the cache block at the effective address specified by *addr* back to main storage (if the block resides in cache and has been modified with respect to main storage) and then invalidates the cache block.

Effectively, this function acts like **ppcDcbst()** followed by **ppcDcbi()**.

Errors

None

Example

Flush the cache line at the effective address X'1000' to main storage and then invalidate the cache line. You might do this in preparation for a DMA slave transfer.

```
ppcDcbf((void *)0x1000);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“ppcDcbst()” on page 10-74

“ppcDcbi()” on page 10-73

“ppcDcbz()” on page 10-75

“ppcDflush()” on page 10-76

PPC405EP Embedded Processor User’s Manual

ppcDcbi()

Synopsis

```
#include <ppcLib.h>
void ppcDcbi(void *addr);
```

Library

ppcLib.a

Description

ppcDcbi() invalidates the cache block containing *addr*, discarding any modified contents if the block is valid in cache.

Errors

None

Example

Invalidate the cache line beginning with 0x3000. This might be done before reading an area of storage updated by a DMA transfer.

```
ppcDcbi((void *)0x3000);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“ppcDcbf()” on page 10-72

“ppcDcbst()” on page 10-74

“ppcDcbz()” on page 10-75

“ppcDflush()” on page 10-76

PPC405EP Embedded Processor User’s Manual

Synopsis

```
#include <ppcLib.h>
void ppcDcbst(void *addr);
```

Library

ppcLib.a

Description

ppcDcbst() copies the cache block containing *addr* to main storage, if the block is valid in cache and has been modified with respect to main storage.

Errors

None

Example

Force the cache line beginning with 0x4000 to memory if the block is valid and out of sync with storage. This would be done to synchronize the cache and storage without invalidating the cache line.

```
ppcDcbst((void *)0x4000);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“ppcDcbf()” on page 10-72

“ppcDcbi()” on page 10-73

“ppcDcbz()” on page 10-75

“ppcDflush()” on page 10-76

PPC405EP Embedded Processor User’s Manual

Synopsis

```
#include <ppcLib.h>
void ppcDcbz(void *addr);
```

Library

ppcLib.a

Description

ppcDcbz() sets the cache block containing the byte referenced by *addr* to 0.

The line is established, if necessary, without fetching the line from main storage.

Note: If an invalid real address is specified, problems could occur when a subsequent attempt is made by the cache unit to store that line to main storage.

Errors

None

Example

Assume buffer is 16 cache lines long and cache aligned. To quickly set it to 0, set to first buffer address.

```
char *bpt = buffer;
for(j = 0; j < 16; j++)
  {
   ppcDcbz((void *)bpt);
   bpt += cache_line_size;
 }
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“ppcDcbf()” on page 10-72

“ppcDcbi()” on page 10-73

“ppcDcbst()” on page 10-74

“ppcDflush()” on page 10-76

PPC405EP Embedded Processor User’s Manual

Synopsis

```
#include <ppcLib.h>
void ppcDflush(void);
```

Library

ppcLib.a

Description

ppcDflush() flushes the existing data in the data cache back into memory, invalidating all of the lines in the data cache, then turns off the data caches by writing 0s into the Data Cache Cacheability Register (DCCR).

Errors

None

Example

Force data reads from memory instead of the data cache.

```
ppcDflush();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

“ppcDcbf()” on page 10-72

“ppcDcbi()” on page 10-73

“ppcDcbst()” on page 10-74

“ppcDcbz()” on page 10-75

PPC405EP Embedded Processor User’s Manual

ppcEieio()

Synopsis

```
#include <ppcLib.h>
void ppcEieio(void);
```

Library

ppcLib.a

Description

ppcEieio() ensures that all storage references before the call finish before any storage references after the call start.

The PPC405EP may internally reorder operations to storage. In the case of memory mapped I/O, such reordering can be undesirable and can be prevented by appropriate use of **ppcEieio()**.

Errors

None

Example

Ensure storage references are done in order.

```
char *one_loc = (char *)0x202;
char *two_loc = (char *)0x204;

*one_loc = 0xAA; /* write a 0xAA to 0x202 */
ppcEieio(); /* insure the store completes before setting two_loc */
*two_loc = 0x55;
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcHalt(void);
```

Library

ppcLib.a

Description

ppcHalt() is a one instruction spin loop, effectively putting the processor in an enabled wait at the point of invocation.

Errors

None

Example

Wait at the point of invocation.

ppcHalt();

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcIcbi(void *addr);
```

Library

ppcLib.a

Description

ppcIcbi() invalidates the Instruction Cache Block pointed to by the address passed. This may be done after updating an instruction.

Errors

None

Example

Write a trap into location 0x3000.

unsigned in * i_addr = (int *) 0x3000;

```
*i_addr = 0x7c800008; /* tw instruction */
ppcDbcst((void *) 0x3000);
ppcIcbi((void *) 0x3000);
ppcIsync();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcIsync(void);
```

Library

ppcLib.a

Description

ppcIsync() causes the processor to discard any instructions that may have been prefetched before **ppcIsync()**. This call must be used after modifying instruction storage.

Errors

None

Example

Place a trap into a given address.

```
*trap_address = 0x7F000008;
 ppcIsync();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

ppcMfccr0()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfccr0(void);
```

Library

ppcLib.a

Description

ppcMfccr0() returns the value of the processor ccr0 register (Core Configuration Register 0).

Errors

None

Example

Retrieve the value of ccr0 register.

```
unsigned long current_ccr0=ppcMfccr0();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfdac1(void)
unsigned long ppcMfdac2(void)
```

Library

ppcLib.a

Description

ppcMfdac1() - ppcMfdac2() returns the current value of the specified register.

The Data Address Compare registers 1 and 2 contain addresses for which debug events may be taken, depending on the values set in the DBCR1 register.

Errors

None

Example

Retrieve the current value of the DAC2 register.

```
unsigned long dac2_value= ppcMfdac2();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMfdbcr0() - ppcMfdbcr1()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfdbcr0(void)
unsigned long ppcMfdbcr1(void)
```

Library

ppcLib.a

Description

ppcMfdbcr0() - ppcMfdbcr1() returns the current value of the specified register.

Dedug Control Registers 0 and 1 are used to enable debug events, reset the processor and set the debug mode of the processor.

WARNING: Enabling bits 0 and 1 can cause unexpected results. Enabling bits 2 and 3 will cause a processor reset to occur. DBCR0 and DBCR1 are designed to be used by development tools, not applications.

Refer to the file <ppc405.h> for defined constants for the DBCR0 and DBCR1 registers.

Errors

None

Example

Retrieve the current value of the DBCR1 register.

```
unsigned long dbcr1_value= ppcMfdbcr1();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfdbsr(void);
```

Library

ppcLib.a

Description

ppcMfdbsr() returns the value of the processor DBSR register.

The Debug Status Register contains the status of debug events and the most recent reset.

The file <ppc405.h> defines constants that can be used when referring to the DBSR.

Errors

None

Example

Retrieve the value of DBSR register.

```
unsigned long current_DBSR=ppcMfdbsr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfdccr(void);
```

Library

ppcLib.a

Description

ppcMfdccr() returns the value of the processor DCCR (Data Cache Cacheability Register).

Errors

None

Example

Retrieve the value of DCCR register.

```
unsigned long current_DCCR=ppcMfdccr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfdcr_any(unsigned long dcr_num);
```

Library

ppcLib.a

Description

ppcMfdcr_any() returns the value of the DCR specified by dcr_num.

Errors

None

Example

Retrieve the value of CPC0_PLLMR register.

```
unsigned long current_CPC0_PLLMR=ppcMfdcr_any(CPC0_PLLMR);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMfdcwr()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfdcwr(void);
```

Library

ppcLib.a

Description

ppcMfdcwr() returns the value of the processor DCWR (Data Cache Write-through Register).

Errors

None

Example

Retrieve the value of DCWR register.

```
unsigned long current_DCWR=ppcMfdcwr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfdear(void);
```

Library

ppcLib.a

Description

ppcMfdear() returns the value of the processor DEAR (Data Exception Address Register).

Errors

None

Example

Retrieve the value of DEAR register.

```
unsigned long current_DEAR=ppcMfdear();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMfdvc1() - ppcMfdvc2()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfdvc1(void)
unsigned long ppcMfdvc2(void)
```

Library

ppcLib.a

Description

ppcMfdvc1() - ppcMfdvc2() returns the current value of the specified Data Value Compare Register.

Errors

None

Example

Retrieve the current value of the DVC2 register.

```
unsigned long dvc2_value= ppcMfdvc2();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfesr(void);
```

Library

ppcLib.a

Description

ppcMfesr() returns the value of the processor ESR (Exception Syndrome Register).

Errors

None

Example

Retrieve the value of ESR register.

```
unsigned long current_ESR=ppcMfesr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMfevpr()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfevpr(void);
```

Library

ppcLib.a

Description

ppcMfevpr() returns the value of the processor EVPR (Exception Vector Prefix Register). Bits 0 to 15 contain the prefix of the address of the exception processing routines. Bits 15 to 31 are reserved.

Errors

None

Example

Retrieve the value of EVPR register.

```
unsigned long current_EVPR=ppcMfevpr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfgpr1(void);
```

Library

ppcLib.a

Description

ppcMfgpr1() returns the current value of GPR(1).

Typically, this is the value of the current stack frame.

Errors

None

Example

See “ppcMfgpr2()” on page 10-93.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User’s Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfgpr2(void)
```

Library

ppcLib.a

Description

ppcMfgpr2() returns the current value of GPR(2).

For XCOFF-based OS Open this is typically the value of the table of contents (TOC) pointer for the current execution context.

Errors

None

Example

Retrieve TOC and stack frame base from current context.

```
toc = ppcMfgpr2();
unsigned long stack_base = ppcMfgpr1();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	Yes

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfiac1(void)
unsigned long ppcMfiac2(void)
unsigned long ppcMfiac3(void)
unsigned long ppcMfiac4(void)
```

Library

ppcLib.a

Description

ppcMfiac1() - ppcMfiac4() returns the current value of the specified Instruction Address Compare Register. The IAC contains the address of the instruction that the debug event will be based on. The Debug Control Register 0 (DBCR0) controls the instruction address debug event. Bits 30 and 31 of the IAC are reserved, since the address must be word aligned.

Errors

None

Example

Retrieve the current value of the IAC4 register.

```
unsigned long iac4_value= ppcMfiac4();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMficcr(void);
```

Library

ppcLib.a

Description

ppcMficcr() returns the value of the processor ICCR (Instruction Cache Cacheability Register).

Errors

None

Example

Retrieve the value of ICCR register.

```
unsigned long current_ICCR=ppcMficcr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMficdbdr(void);
```

Library

ppcLib.a

Description

ppcMficdbdr() returns the value of the processor ICDBDR (Instruction Cache Debug Data Register).

<ppc405.h> has constants defined for use with the ICDBDR register.

Errors

None

Example

Retrieve the value of ICDBDR register.

```
unsigned long current_ICDBDR=ppcMficdbdr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfmsr(void);
```

Library

ppcLib.a

Description

ppcMfmsr() returns the value of the Machine State Register(MSR).

Refer to the **<ppc_arch.h>** file for the defines of constants that can be used as masks with the MSR value.

Errors

None

Example

See “ppcMtmsr()” on page 10-128.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User’s Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfpid(void);
```

Library

ppcLib.a

Description

ppcMfpid() returns the value of the processor PID (Process ID) register.

Errors

None

Example

Retrieve the value of PID register.

```
unsigned long current_PID=ppcMfpid();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMfpit()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfpit(void);
```

Library

ppcLib.a

Description

ppcMfpit() returns the value of the processor PIT (Programmable Interval Timer) register.

Errors

None

Example

Retrieve the value of PIT register.

```
unsigned long current_PIT=ppcMfpit();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfpvr(void);
```

Library

ppcLib.a

Description

ppcMfpvr() returns the value of the processor version register, which indicates the version and revision of the PowerPC processor.

Errors

None

Example

Retrieve the current value of the processor version register. Processor version-specific code may require this value.

```
printf("This is processor version %x\n", ppcMfpvr());
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMfsgr()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfsgr(void);
```

Library

ppcLib.a

Description

ppcMfsgr() returns the value of the processor SGR (Storage Guarded Register).

Errors

None

Example

Retrieve the value of SGR register.

```
unsigned long current_SGR=ppcMfsgr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfsler(void);
```

Library

ppcLib.a

Description

ppcMfsler() returns the value of the processor SLER (Storage Little-Endian Register).

Errors

None

Example

Retrieve the value of SLER register.

```
unsigned long current_SLER=ppcMfsler();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMfsprg0() - ppcMfsprg7()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfsprg0(void);
unsigned long ppcMfsprg1(void);
unsigned long ppcMfsprg2(void);
unsigned long ppcMfsprg3(void);
unsigned long ppcMfsprg4(void);
unsigned long ppcMfsprg5(void);
unsigned long ppcMfsprg6(void);
unsigned long ppcMfsprg7(void);
```

Library

ppcLib.a

Description

ppcMfsprg0() - ppcMfsprg7() returns the current value of the special purpose register generals (SPRG0 - SPRG7).

Typically, the SPRGs provide temporary storage at the operating system level.

NOTE: OS Open reserves SPRG0-3 for its own use.

Errors

None

Example

Read value of SPRG0.

```
unsigned long sprg0_value = ppcMfsprg0();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfsrr0(void);
```

Library

ppcLib.a

Description

ppcMfsrr0() returns the value of SRR0.

Typically, SRR0 is used in interrupt handlers, as it usually contains the address of the next instruction to be executed at the time of the interrupt. SRR0 and SRR1 are set when a noncritical interrupt occurs.

Errors

None

Example

Retrieve the current value of the SRR0. An exception handler may use this value to determine the point of exception.

```
unsigned long current_srr0=ppcMfsrr0();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“ppcMfsrr1()” on page 10-105

PPC405EP Embedded Processor User’s Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfsrr1(void);
```

Library

ppcLib.a

Description

ppcMfsrr1() returns the current value of SRR1.

Typically, SRR1 is used in interrupt handlers, as it contains the old MSR value as well as information bits specific to the interrupt. SRR0 and SRR1 are set when a noncritical interrupt occurs.

Errors

None

Example

Retrieve the current value of SRR1. This register contains the saved MSR, which may be needed by an exception handler.

```
unsigned long current_srr1=ppcMfsrr1();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfsrr2(void);
```

Library

ppcLib.a

Description

ppcMfsrr2() returns the current value of SRR2.

Typically, SRR2 is used in interrupt handlers, as it usually contains the address of the next instruction to be executed at the time of the interrupt. SRR2 and SRR3 are set when a critical interrupt occurs.

Errors

None

Example

Retrieve the current value of SRR2. An exception handler may use this value to determine the point of exception.

```
unsigned long current_srr2=ppcMfsrr2();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfsrr3(void);
```

Library

ppcLib.a

Description

ppcMfsrr3() returns the current value of SRR3.

Typically, SRR3 is used in interrupt handlers, as it contains the old MSR value as well as information bits specific to the interrupt. SRR2 and SRR3 are set when a critical interrupt occurs.

Errors

None

Example

Retrieve the current value of SRR3. This register contains the saved MSR, which may be needed by an exception handler.

```
unsigned long current_srr3=ppcMfsrr3();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfsu0r(void);
```

Library

ppcLib.a

Description

ppcMfsu0r() returns the value of the processor SU0R (Storage User-Defined 0 Register).

On the PPC405EP, SU0R is used to hold the U0 bits indicating storage compression.

Errors

None

Example

Retrieve the value of SU0R register.

```
unsigned long current_SU0R=ppcMfsu0r();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMftb(tb_t *clock_data);
```

Library

ppcLib.a

Description

ppcMftb() returns the current time base data.

Typically, the time base registers are used to determine the number of clock cycles that have passed.

Errors

None

Example

Retrieve the current value of time base high and low registers.

```
tb_t clock_data;
ppcMftb(&clock_data);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMftcr(void);
```

Library

ppcLib.a

Description

ppcMftcr() returns the value of the processor TCR (Timer Control Register).

File <ppc405.h> defines several constants for the TCR.

Errors

None

Example

Retrieve the value of TCR register.

```
unsigned long current_TCR=ppcMftcr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMftsr(void);
```

Library

ppcLib.a

Description

ppcMftsr() returns the value of the processor TSR (Timer Status Register).

File <ppc405.h> defines several constants for the TSR.

Errors

None

Example

Retrieve the value of TSR register.

```
unsigned long current_TSR=ppcMftsr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMfzpr(void);
```

Library

ppcLib.a

Description

ppcMfzpr() returns the value of the processor ZPR (Zone Protection Register).

Errors

None

Example

Retrieve the value of ZPR register.

```
unsigned long current_ZPR=ppcMfzpr();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMtccr0()

Synopsis

```
#include <ppcLib.h>
void ppcMtccr0(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtccr0() sets the value of the processor ccr0 register (Core Configuration Register 0).

Errors

None

Example

Set the value of ccr0 register.

```
ppcMtccr0(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtdac1(unsigned long value)
void ppcMtdac2(unsigned long value)
```

Library

ppcLib.a

Description

ppcMtdac1() - ppcMtdac2() sets the value of the specified register.

The Data Address Compare registers 1 and 2 contain addresses for which debug events may be taken, depending on the values set in the DBCR1 register.

Errors

None

Example

Set the value of the DAC2 register.

ppcMtdac2(value);

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMtdbcr0() - ppcMtdbcr1()

Synopsis

```
#include <ppcLib.h>
void ppcMtdbcr0(unsigned long value)
void ppcMtdbcr1(unsigned long value)
```

Library

ppcLib.a

Description

ppcMtdbcr0() - ppcMtdbcr1() sets the value of the specified register.

Dedug Control Registers 0 and 1 are used to enable debug events, reset the processor and set the debug mode of the processor.

WARNING: Enabling bits 0 and 1 can cause unexpected results. Enabling bits 2 and 3 will cause a processor reset to occur. DBCR0 and DBCR1 are designed to be used by development tools, not applications.

Refer to the file <ppc405.h> for defined constants for the DBCR0 and DBCR1 registers.

Errors

None

Example

Set the value of the DBCR1 register.

ppcMtdbcr1(value);

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtdbsr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtdbsr() sets the value of the processor DBSR register.

The Debug Status Register contains the status of debug events and the most recent reset.

The file <ppc405.h> defines constants that can be used when referring to the DBSR.

Errors

None

Example

Set the value of DBSR register.

```
ppcMtdbsr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMtdccr()

Synopsis

```
#include <ppcLib.h>
void ppcMtdccr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtdccr() sets the value of the processor DCCR (Data Cache Cacheability Register).

Errors

None

Example

Set the value of DCCR register.

```
ppcMtdccr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

Synopsis

```
#include <ppcLib.h>
unsigned long ppcMtdcr_any(unsigned long dcr_num, unsigned long value);
```

Library

ppcLib.a

Description

ppcMtdcr_any() sets the DCR specified by *dcr_num* to *value*.

Errors

None

Example

Disable all interrupts in UIC0 by writing 0's to the enable register.

```
ppcMtdcr_any(UIC0_ER, 0x00000000);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMtdcwr()

Synopsis

```
#include <ppcLib.h>
void ppcMtdcwr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtdcwr() sets the value of the processor DCWR (Data Cache Write-through Register).

Errors

None

Example

Set the value of DCWR register.

```
ppcMtdcwr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtdear(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtdear() sets the value of the processor DEAR (Data Exception Address Register).

Errors

None

Example

Set the value of DEAR register.

```
ppcMtdear(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMtdvc1() - ppcMtdvc2()

Synopsis

```
#include <ppcLib.h>
void ppcMtdvc1(unsigned long value)
void ppcMtdvc2(unsigned long value)
```

Library

ppcLib.a

Description

ppcMtdvc1() - ppcMtdvc2() sets the value of the specified Data Value Compare Register.

Errors

None

Example

Set the value of the DVC2 register.

ppcMtdvc2(value);

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtesr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtesr() sets the value of the processor ESR (Exception Syndrome Register).

Errors

None

Example

Set the value of ESR register.

```
ppcMtesr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtevpr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtevpr() sets the value of the processor EVPR (Exception Vector Prefix Register). Bits 0 to 15 contain the prefix of the address of the exception processing routines. Bits 15 to 31 are reserved.

Errors

None

Example

Set the value of EVPR register.

```
ppcMtevpr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtiac1(unsigned long value)
void ppcMtiac2(unsigned long value)
void ppcMtiac3(unsigned long value)
void ppcMtiac4(unsigned long value)
```

Library

ppcLib.a

Description

ppcMtiac1() - ppcMtiac4() sets the value of the specified Instruction Address Compare Register. The IAC contains the address of the instruction that the debug event will be based on. The Debug Control Register 0 (DBCR0) controls the instruction address debug event. Bits 30 and 31 of the IAC are reserved, since the address must be word aligned.

Errors

None

Example

Set the value of the IAC4 register.

ppcMtiac4(value);

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMticcr()

Synopsis

```
#include <ppcLib.h>
void ppcMticcr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMticcr() sets the value of the processor ICCR (Instruction Cache Cacheability Register).

Errors

None

Example

Set the value of ICCR register.

```
ppcMticcr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtmsr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtmsr() sets the value of the Machine State Register(MSR).

Refer to the **<ppc_arch.h>** file for the defines of constants that can be used as masks with the MSR value.

Errors

None

Example

Enable external interrupts:

```
unsigned long msr=ppcMfmsr()
ppcMtmsr(msr | ppcMsrEE);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMtpid()

Synopsis

```
#include <ppcLib.h>
void ppcMtpid(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtpid() sets the value of the processor PID (Process ID) register.

Errors

None

Example

Set the value of PID register.

```
ppcMtpid(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtpit(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtpit() sets the value of the processor PIT (Programmable Interval Timer) register.

Errors

None

Example

Set the value of PIT register.

```
ppcMtpit(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMtsgr()

Synopsis

```
#include <ppcLib.h>
void ppcMtsgr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtsgr() sets the value of the processor SGR (Storage Guarded Register).

Errors

None

Example

Set the value of SGR register.

```
ppcMtsgr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtsler(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtsler() sets the value of the processor SLER (Storage Little-Endian Register).

Errors

None

Example

Set the value of SLER register.

```
ppcMtsler(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMtsprg0() - ppcMtsprg7()

Synopsis

```
#include <ppcLib.h>
void ppcMtsprg0(unsigned long value);
void ppcMtsprg1(unsigned long value);
void ppcMtsprg2(unsigned long value);
void ppcMtsprg3(unsigned long value);
void ppcMtsprg4(unsigned long value);
void ppcMtsprg5(unsigned long value);
void ppcMtsprg6(unsigned long value);
void ppcMtsprg7(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtsprg0() - ppcMtsprg7() sets the value of the special purpose register generals (SPRG0 - SPRG7).

Typically, the SPRGs provide temporary storage at the operating system level.

NOTE: OS Open reserves SPRG0-3 for its own use.

Errors

None

Example

Set value of SPRG0.

```
ppcMtsprg0(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtsrr0(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtsrr0() sets the value of SRR0.

Typically, SRR0 is used in interrupt handlers, as it usually contains the address of the next instruction to be executed at the time of the interrupt. SRR0 and SRR1 are set when a noncritical interrupt occurs.

Errors

None

Example

Set the value of the SRR0.

ppcMtsrr0(value);

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“ppcMfsrr1()” on page 10-105

PPC405EP Embedded Processor User’s Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtsrr1(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtsrr1() sets the value of SRR1.

Typically, SRR1 is used in interrupt handlers, as it contains the old MSR value as well as information bits specific to the interrupt. SRR0 and SRR1 are set when a noncritical interrupt occurs.

Errors

None

Example

Set the value of SRR1.

```
ppcMtsrr1(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtsrr2(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtsrr2() sets the value of SRR2.

Typically, SRR2 is used in interrupt handlers, as it usually contains the address of the next instruction to be executed at the time of the interrupt. SRR2 and SRR3 are set when a critical interrupt occurs.

Errors

None

Example

Set the value of SRR2.

```
ppcMtsrr2(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMtsrr3()

Synopsis

```
#include <ppcLib.h>
void ppcMtsrr3(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtsrr3() sets the value of SRR3.

Typically, SRR3 is used in interrupt handlers, as it contains the old MSR value as well as information bits specific to the interrupt. SRR2 and SRR3 are set when a critical interrupt occurs.

Errors

None

Example

Set the value of SRR3.

```
ppcMtsrr3(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtsu0r(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtsu0r() sets the value of the processor SU0R (Storage User-Defined 0 Register).

On the PPC405EP, SU0R is used to hold the U0 bits indicating storage compression.

Errors

None

Example

Set the value of SU0R register.

```
ppcMtsu0r(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMttb(tb_t *clock_data);
```

Library

ppcLib.a

Description

ppcMttb() sets the time base data.

Typically, the time base registers are used to determine the number of clock cycles that have passed.

Errors

None

Example

Set the value of time base high and low registers.

```
tb_t clock_data;
ppcMttb(&clock_data);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMttcr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMttcr() sets the value of the processor TCR (Timer Control Register).

File <ppc405.h> defines several constants for the TCR.

Errors

None

Example

Set the value of TCR register.

```
ppcMttcr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcMttsr()

Synopsis

```
#include <ppcLib.h>
void ppcMttsr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMttsr() sets the value of the processor TSR (Timer Status Register).

File <ppc405.h> defines several constants for the TSR.

Errors

None

Example

Set the value of TSR register.

```
ppcMttsr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <ppcLib.h>
void ppcMtzpr(unsigned long value);
```

Library

ppcLib.a

Description

ppcMtzpr() sets the value of the processor ZPR (Zone Protection Register).

Errors

None

Example

Set the value of ZPR register.

```
ppcMtzpr(value);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

ppcOrMsr()

Synopsis

```
#include <ppcLib.h>
unsigned long ppcOrMsr(unsigned long value);
```

Library

ppcLib.a

Description

ppcOrMsr() performs the OR of *value* and the current MSR, updating the MSR.

The previous value of the MSR is returned.

The file **<ppcLib.h>** defines several constants for the MSR that can be used as masks.

Errors

None

Example

Enable instruction address translation.

```
unsigned long old_val = ppcOrMsr(ppcMsrIR);
```

Attributes

Async Safe	No
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“ppcAndMsr()” on page 10-70

PPC405EP Embedded Processor User’s Manual

Synopsis

```
#include <ppcLib.h>
void ppcSync(void);
```

Library

ppcLib.a

Description

ppcSync() causes the processor to wait until all data cache lines scheduled to be written to main storage have actually been written.

Errors

None

Example

Ensure a **ppcDbci()** completes before using the values.

```
char *memptr = (char *)0x2000;
char new_value;
ppcDcbi((void *)memptr)
ppcSync();
new_value = *memptr;
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

Synopsis

```
#include <sys/asyncLib.h>
int s1dbprintf(unsigned long uart_clock, unsigned char *base_reg, unsigned
long cpc0_cr0_reg, event_t int_level, const char *format,...);
```

Library

asyncLib.a

Description

s1dbprintf() is a version of **printf()** that uses polled writes (no interrupts) to serial port 1, and may be used before I/O has been established. **s1dbprintf()** may be called before the async device driver is installed. **uart_clock** is the clock frequency of the serial port. **base_reg** specifies the address of the base UART register. **cpc0_cr0_reg** specifies the fields in the Chip Control 0 register that will be set. Only the fields relevant to UART 0 should be specified. **int_level** specifies the interrupt level associated with serial port 1. The default communication values are 9600 baud, 8 bit data, no parity, 1 stop bit.

Manifest constants for common values for the parameters are supplied in **<sys/asyncLib.h>, <ioLib.h> and <ppcLib.h>.** To use the external UART clock, **uart_clock** must be **asyncClockRate, base_reg** must be **UART0_BASE_ADDRESS, cpc0_cr0_reg** must be **CPC0_CR0_UART0_EXTCLOCK_EN, int_level** must be **EXT_IRQ_COM1.**

Errors

None

Example

Print “Hello World” on serial port 1 before I/O has been initialized.

```
#include <sys/asyncLib.h>
#include <ioLib.h>
#include <ppcLib.h>
#define S1DB_PARMS asyncClockRate, UART0_BASE_ADDRESS,
CPC0_CR0_UART0_EXTCLOCK_EN, EXT_IRQ_COM1
s1dbprintf(S1DB_PARMS, ”Hello World\n\r”);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“vs1dbprintf()” on page 10-151

PPC405EP Embedded Processor User’s Manual

Synopsis

```
#include <sys/asyncLib.h>
int s2dbprintf(unsigned long uart_clock, unsigned char *base_reg, unsigned
long cpc0_cr0_reg, event_t int_level, const char *format,...);
```

Library

asyncLib.a

Description

s2dbprintf() is a version of **printf()** that uses polled writes (no interrupts) to serial port 2, and may be used before I/O has been established. **s2dbprintf()** may be called before the async device driver is installed. **uart_clock** is the clock frequency of the serial port. **base_reg** specifies the address of the base UART register. **cpc0_cr0_reg** specifies the fields in the Chip Control 0 register that will be set. Only the fields relevant to UART 1 should be specified. **int_level** specifies the interrupt level associated with serial port 2. The default communication values are 9600 baud, 8 bit data, no parity, 1 stop bit.

Manifest constants for common values for the parameters are supplied in **<sys/asyncLib.h>, <ioLIb.h>** and **<ppcLib.h>**. To use the external UART clock, **uart_clock** must be **asyncClockRate, base_reg** must be **UART1_BASE_ADDRESS, cpc0_cr0_reg** must be **CPC0_CR0_UART1_EXTCLOCK_EN, int_level** must be **EXT_IRQ_COM2.**

Errors

None

Example

Print “Hello World” on serial port 2 before I/O has been initialized.

```
#include <sys/asyncLib.h>
#include <ioLib.h>
#include <ppcLib.h>
#define S2DB_PARMS asyncClockRate, UART1_BASE_ADDRESS,
CPC0_CR0_UART1_EXTCLOCK_EN, EXT_IRQ_COM2
s2dbprintf(S2DB_PARMS, ”Hello World\n\r”);
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User’s Manual

PowerPC 405EP Evaluation Board Manual

Synopsis

```
#include <tickLib.h>
unsigned long timebase_speed(void);
```

Library

tickLib.a

Description

timebase_speed() returns the timebase frequency, in Hz. This is done by setting serial port 2 to a known speed (9600 bps) in loopback mode and sending a character out to it. This takes a known amount of time for the character to be received. By determining how many increments to the timebase registers occurred during this known time, the timebase frequency can be determined.

Errors

None

Example

Get the timebase speed.

```
unsigned long tb_speed=timebase_speed();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

PPC405EP Embedded Processor User's Manual

timertick_install()

Synopsis

```
#include <tickLib.h>
int timertick_install(void);
```

Library

tickLib.a

Description

timertick_install() installs and starts the timer tick handler to maintain time-of-day in the OS Open real-time executive.

Errors

[ENOMEM]	Insufficient memory to install the timer tick handler.

Example

Do a **timertick_install()**.

```
timertick_install();
```

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“timertick_remove()” on page 10-150

Synopsis

```
#include <tickLib.h>
int timertick_remove( void );
```

Library

tickLib.a

Description

timertick_remove() removes the timer tick handler installed by **timertick_install()**.

Errors

[EINVAL]	Internal error involving tick handler level.

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“timertick_install()” on page 10-149

Synopsis

```
#include <sys/asyncLib.h>
int vs1dbprintf(unsigned long uart_clock, unsigned char *base_reg, unsigned
long cpc0_cr0_reg, event_t int_level, const char *format, va_list arg_list);
```

Library

asyncLib.a

Description

vs1dbprintf() is a version of **s1dbprintf()** that accepts a **va_list** as a parameter instead of a variable number of parameters. **vs1dbprintf()** may be called before the async device driver is installed. **uart_clock** is the clock frequency of the serial port. **base_reg** specifies the address of the base UART register. **cpc0_cr0_reg** specifies the fields in the Chip Control 0 register that will be set. Only the fields relevant to UART 0 should be specified. **int_level** specifies the interrupt level associated with serial port 1. **arg_list** is a list of variable arguments that has been created by a call to **va_start().** The default communication values are 9600 baud, 8 bit data, no parity, 1 stop bit.

Errors

None

Attributes

Async Safe	Yes
Cancel Safe	Yes
Interrupt Handler Safe	Yes
Callable from Application Thread Group	No

References

“s1dbprintf()” on page 10-145

PPC405EP Embedded Processor User’s Manual

PowerPC 405EP Evaluation Board Manual

Appendix A. Program Trace Calls

This appendix describes the remote debugging interface provided by the ROM monitor. These calls may be used by remote debuggers other than the RISCWatch debugger provided with the kit.

A.1 Overview

The following section describes the message (ptrace) protocol that has been implemented in the ROM monitor to support debug. If you want to interface your own debugger to the ROM monitor or modify the ROM monitor to interface with your debugger, you will need to understand the existing message protocol associated with the various debugging functions.

The ptrace interface to the ROM monitor can best be understood by reviewing the information below along with the debug-specific ROM monitor source code (dbLib/ptrace.c).

A.2 MSGDATA Structure

In the interface descriptions shown below, several references are made to a “process id.” The concept of process ids does not apply to the ROM monitor, so any nonzero value can be used. The ROM monitor uses the value 42.

Data structure MSGDATA is defined in dbg.h. New register definitions and new error messages are also defined in dbg.h.

The dbg.h file is shown below:

```
/* @(#)dbg.h4.3 5/9/95 09:12:14 */
/*--------------------------------------------------------------------+
|       COPYRIGHT   I B M   CORPORATION 1994
|       LICENSED MATERIAL  -  PROGRAM PROPERTY OF I B M
|       REFER TO COPYRIGHT INSTRUCTIONS: FORM G120-2083
|      US Government Users Restricted Rights - Use, duplication or |
disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
+--------------------------------------------------------------------*/
#if !defined(DBG_H)
#define DBG_H
#define BREAKPT 0x7D821008
#ifndef MIN
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))
#endif
/*ptrace definitions based on AIX ptrace                              */
#define RD_TRACE_ME       0     /* used ONLY by target task to be traced*/
#define RD_READ_I         1     /* read target instruction addr space   */
#define RD_READ_D         2     /* read target data address space       */
#define RD_READ_U         3     /* read offset from the user structure  */
#define RD_WRITE_I        4     /* write target instruction addr space  */
#define RD_WRITE_D        5     /* write target data address space      */
#define RD_WRITE_U        6     /* write offset to the user structure   */
```

```
#define RD_CONTINUE       7     /* continue execution                    */
#define RD_KILL           8     /* terminate execution                   */
#define RD_STEP           9     /**execute one or more instructions*** !*/
#define RD_READ_GPR      11     /* read general purpose register         */
#define RD_READ_FPR      12     /* read floating point register          */
#define RD_WRITE_GPR     14     /* write general purpose register        */
#define RD_WRITE_FPR     15     /* write floating point register         */
#define RD_READ_BLOCK    17     /* read block of data                    */
#define RD_WRITE_BLOCK   19     /* write block of data                   */
#define RD_ATTACH        30     /* attach to a process                   */
#define RD_DETACH        31     /* detach a proc to let it keep running */
#define RD_REGSET        32     /* return entire register set to caller */
#define RD_REATT         33     /* reattach debugger to proc             */
#define RD_LDINFO        34     /* return loaded program info            */
#define RD_MULTI         35     /* set/clear multi-processing            */
#define RD_READ_I_MULT   70     /* Read multiple inst words              */
#define RD_READ_GPR_MULT 71     /* Read multiple registers               */
#define RD_SINGLE_STEP  100     /**source line single step************ !*/
#define RD_LOAD         101     /* load a task                          !*/
#define RD_LOGIN        103     /*ptrace for login                      !*/
#define RD_LOGON        103     /*ptrace for logon                      !*/
#define RD_LOGOFF       104     /*ptrace for logoff                     !*/
#define RD_FILL         105     /*ptrace for fill memory                !*/
#define RD_PASS         106     /*ptrace for pass                       !*/
#define RD_SEARCH       107     /*ptrace for search memory              !*/
#define RD_WAIT         108     /*ptrace for wait status information   !*/
/* Added to support ADEPT */
#define RD_READ_DCR     110     /*ptrace for reading DCR’s              */
#define RD_WRITE_SPR    111     /*ptrace for writing SPR’s              */
#define RD_WRITE_DCR    112     /*ptrace for writing DCR’s              */
#define RD_STOP_APPL    113     /*ptrace for stopping the application   */
#define RD_STATUS       114     /*ptrace for getting run status         */
#define RD_READ_SPR     115     /*ptrace for reading SPR’s              */
/* Added to support 403GC */
#define RD_READ_TLB     116     /*ptrace for readingTLB(403GC )         */
#define RD_WRITE_TLB    117     /*ptrace for writing TLB(403GC )        */
/* Added to support 602 */
#define RD_READ_SR      118     /*ptrace for reading SR’s               */
#define RD_WRITE_SR     119     /*ptrace for writing SR’s               */
#define MAX_PTRACE      119     /*last ptrace number                    */
#define RL_LOAD_REQ     180     /* Remote Loader - Load Request         */
#define RL_LDINFO       181     /* Remote Loader - Load Information     */
/*TCP/IP services for all sorts of remote debug                         */
#define OSOPEN_SERVNAME “osopen-dbg” /* OS/Open debug service           */
#define OSOPEN_MON_SERVNAME “osopen-mon” /* OS/Open debug monitor svc   */
/*new register definition                                              */
#define DAR   137               /* Data Address Register ($dar)        */
#define DSISR 138               /* Data St Int Status Reg ($dsisr)     */
#define SRR0  139               /* Save and Restore Register 0 ($srr0) */
#define SRR1  140               /* Save and Restore Register 0 ($srr1) */
#define SR0   141               /* Segment Register ($sr0)             */
#define SR1   142               /* Segment Register ($sr1)             */
```

```
#define SR2   143              /* Segment Register ($sr2)              */
#define SR3   144              /* Segment Register ($sr3)              */
#define SR4   145              /* Segment Register ($sr4)              */
#define SR5   146              /* Segment Register ($sr5)              */
#define SR6   147              /* Segment Register ($sr6)              */
#define SR7   148              /* Segment Register ($sr7)              */
#define SR8   149              /* Segment Register ($sr8)              */
#define SR9   150              /* Segment Register ($sr9)              */
#define SR10  151              /* Segment Register ($sr10)             */
#define SR11  152              /* Segment Register ($sr11)             */
#define SR12  153              /* Segment Register ($sr12)             */
#define SR13  154              /* Segment Register ($sr13)             */
#define SR14  155              /* Segment Register ($sr14)             */
#define SR15  156              /* Segment Register ($sr15)             */
#define DEC   157              /* Decrementer ($dec)                   */
#define RTCU  158              /* Real Time Clock Upper ($rtcu)        */
#define RTCL  159              /* Real Time Clock Lower ($rtcl)        */
#define SDR0  160              /* Storage Description Reg ($sdr0)      */
#define SDR1  161              /* Storage Description Reg ($sdr1)      */
#define EIS0  162              /* External Int Summary Reg1($eis1)     */
#define EIS1  163              /* External Int Summary Reg2($eis2)     */
#define EIM0  164              /* External Int Mask Reg1($eim1)        */
#define EIM1  165              /* External Int Mask Reg2($eim2)        */
#define SRR2  166              /* Save and Restore Register 2 ($srr2)  */
#define SRR3  167               /* Save and Restore Register 3 ($srr3)  */
/*other definitions needed for remote debug                           */
#define RD_MAXDATA   1800       /* Total no of DWORDS in a MSGDATA     */
#define RD_MINLENGTH 6          /* Min no of dwords in msg             */
#define RD_MINBYTES (RD_MINLENGTH*sizeof(unsigned long))
#define RD_MAXBUFFER (RD_MAXDATA - RD_MINLENGTH)
#define RD_MAXPACKET 1000000    /* Max bytes in TCP/IP packet          */
#define RD_REGBYTES  (32+8)*4   /* No of bytes for all registers       */
#define NO_KILL            1    /*do not kill any users processes      */
#define KILL_PROC          0    /*kill user process upon logoff        */
#define MAX_ERROR        1014   /*last error for rptrace               */
#define MIN_ERROR        1000   /*first error for rptrace              */
#define MIN_PACKET_SIZE    24
#define DBG_SPORT   20044
#define DBG_DPORT   20050
/*new error codes                                                     */
#define RD_NOLOAD_ERR   1000    /*no loader info available             */
#define RD_COM_ERR      1001    /*communication error occured          */
#define RD_SIZE_ERR     1002    /*not enough room to pass all info    */
#define RD_NOTSUPP      1003    /*call not supported                   */
#define RD_REG_ERR      1004    /*invalid register number requested    */
#define RD_NOTAVAIL     1005    /*call not implemented at this time    */
#define RD_NOFILE_ERR   1006    /*file could not be loaded, no file    */
#define RD_NOSCAN_ERR   1008    /*could not locate scan string file    */
#define RD_NOPERM       1010    /*no permission to log on              */
#define RD_INVALID_SEQ  1011    /*invalid rptrace sequence             */
#define RD_BUSY_ERR     1012    /*some users is already logged on      */
#define RD_PTRACE_ERR   1014    /*internal ptrace error                */
```

```
#define RD_OK              0    /*rptrace completed ok                 */
#define ARCH_403              0x34000000           /* 403 architecture */
#define ARCH_601              0x36000000           /* 601 architecture */
#define ARCH_602              0x36303200           /* 602 architecture */
#define ARCH_603              0x36303300           /* 603 architecture */
#define ARCH_604               0x36303400           /* 604 architecture */
typedef struct msgdata          /* message data structure              */
{   unsigned long data_len;     /* optional data length    }         */
unsigned long retcode;          /* return code             }MIN      */
unsigned long request;          /* request type            }PART     */
unsigned long address;          /*   function parameter    }=        */
unsigned long data              /*   function parameter    }6*DWORD */
struct {      unsigned f1:1;
              unsigned f2:1;
              unsigned f3:1;
              unsigned padd:21;
              unsigned f25:8;
   } flags;
#define printmsg flags.f1
#define breakpt flags.f2
#define dbg_seqno flags.f25
union {      unsigned long trace_buffer[RD_MAXBUFFER];
             unsigned long processid;
   } parameter;
#define buffer parameter.trace_buffer  /* buffer for data, in any     */
#define rpid parameter.processid       /* process id                   */
} MSGDATA;
#endif
```

A.3 Ptrace Definitions

The following section presents the application programming interface (API) for rptrace messages. One field that is not shown here, because it is common to every call, is the msg.printmsg flag. This may be set in an rptrace response where msg.retcode does not equal RD_OK. When the msg.printmsg flag is set it indicates that a text string is contained in msg.buffer and that this message should be displayed to the user. Typically this is an error message that provides more detail as to why the rptrace call failed to return RD_OK.

Another field that is not shown is the dbg_seqno field. The field provides a mechanism for recovering from lost requests and responses. If a request has the dbg_seqno field as not zero, it is compared with the value from the previous request. If it matches, the action is not performed and instead, the previous response is sent. This allows the debugger to time-out and retry requests without danger of performing the same function twice.

A.3.1 RD_ATTACH (30)

Attaches debugger to running process in target environment.

	Parameters	Description
Request	msg.request= RD_ATTACH	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system.(Any non zero value)
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.retcode= EIO (5)	One of the parameters is incorrect
	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_NOTSUPP (1003)	Call not supported for this interface
	msg.retcode= RD_OK (0)	Successful completion
	msg.data_len=0	No additional data

A.3.2 RD_CONTINUE (7)

This request causes the process to resume execution. If the dbg_seqno field of the request is zero, the response is not returned until the process stops due to a break point or error. Otherwise, an immediate response is sent from the RD_CONTINUE request and the debugger should send the RD_STATUS request to see if the process has stopped.

	Parameters	Description
Request	msg.request= RD_CONTINUE	Requested API function
	msg.address= address	This field is ignored by ROM monitor
	msg.data= signal	0
	msg.rpid= process_id	Numeric process ID on the target system
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.data= 0	

A.3.3 RD_DETACH (31)

Detaches debugger from running process in target environment. Debugged process is restarted and execution continues without debugger control.

	Parameters	Description
Request	msg.request= RD_DETACH	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.data= 0	Ignored by ROM monitor
	msg.address=1	Ignored by ROM monitor
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist, or a process that is currently not being debugged
	msg.retcode= RD_COM_ERR (1001)	Communications error occurred
	msg.retcode= RD_NOTSUPP (1003)	Call not supported for this interface
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	One of the parameters is incorrect
	msg.data_len= 0	No additional data is being sent

A.3.4 RD_FILL (105)

Fills memory with zeroes at the location specified by address for the number of bytes specified by data.

	Parameters	Description
Request	msg.request= RD_FILL	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.address= address	Address of memory to fill with zeroes
	msg.data= count	Number of bytes to fill with zeroes
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communications error occurred.
	msg.retcode= RD_NOTSUPP (1003)	Call not supported for this interface
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	One of the parameters is incorrect
	msg.data_len= 0	No additional data is being sent

A.3.5 RD_KILL (8)

This request causes the process to terminate the same way it would with an exit routine. The ROM monitor does not implement this function but simply returns an RD_OK response for compatibility with older debuggers.

	Parameters	Description
Request	msg.request= RD_KILL	Requested API function.
	msg.rpid= process_id	Process ID of the process to be killed.
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.data_len= 0	Length of additional data being sent

A.3.6 RD_LDINFO (34)

Request loader information from target environment. This information is provided to the ROM monitor in the boot header or by the RL_LDINFO request. Refer to ROM Monitor Load Format section for more information.

	Parameters	Description
Request	msg.request= RD_LDINFO	Requested API function
	msg.rpid= process_id	Process ID from which the loader information is requested
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent

	Parameters	Description
Response	msg.retcode= RD_NOLOAD_ERR (1000)	No loader information is available
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_SIZE_ERR (1002)	Not enough room in the buffer to fit all load information
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	One of the parameters is incorrect
	msg.buffer[0]= ldinfo_next	Offset to next loader information segment. See note below
	msg.buffer[1]= fd	File descriptor for loaded object. In remote debug 0xFFFF FFFF should be returned (this is a space filler)
	msg.buffer[2]= textorig	Starting text address
	msg.buffer[3]= textsize	Size of text
	msg.buffer[4]= dataorig	Starting data address.
	msg.buffer[5]= datasize	Size of data
	msg.buffer[6]= (char *)pathname	Fully qualified filename of the object file.
	msg.buffer[X]= (char *)membername	Membername (used for shared library objects). X does not represent position on word boundary. A NULL has to be returned for the membername even if the debugged file has no member name
	msg.buffer[ldinfo_next]= ldinfo_next	Next loader block (notice "ldinfo_next")
	msg.data_len= "variable"	Set to length of data sent in msg.buffer. Data length will vary depending on the amount of information passed. Remember to count all the NULL characters
	Note: dinfo_next=0 indicates that no further loader blocks are present, otherwise ldinfo_next contains the offset of the next loader block in the buffer. This is actually the length of the current block. For example, if the buffer contains three blocks of lengths 38, 40 and 41 bytes, the ldinfo_next fields would be 38, 40 and 0, respectively. Note also that the blocks do not have to be contiguous - it is possible that the end of one block may not directly abut the following block. This may occur if additional information or word-aligning padding is placed after the end of the member name string. Pathname and membername are strings terminated with a null character.	

A.3.7 RD_LOAD (101)

Loads executable program. Full path name of the file to be loaded is passed in this message. The ROM monitor will respond by sending an RL_LOAD_REQ to the remote loader daemon port.

	Parameters	Description
Request	msg.request= RD_LOAD	Requested API function
	msg.buffer= filename	Name of file to load. A NULL character terminates filename. filename contains a fully qualified path to that file
	msg.data_len= strlen(filename)+1	String length of filename plus NULL character
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= RD_NOFILE_ERR (1006)	Could not locate/load the file
	msg.rpid= process_id	Process_id of the newly loaded file. This number (integer) can not be equal to -1 (0xFFFF FFFF) or 0
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent.

A.3.8 RD_LOGIN (103)

Initializes users LOGIN. This request must be the first rptrace request issued by the debugger or results will be unpredictable.

	Parameters	Description
Request	msg.request= RD_LOGIN	Requested API function.
	msg.buffer[0]= host_name	This field is ignored by ROM monitor.
	msg.buffer[strlen(host_name)+1]= user_name	This field is ignored by ROM monitor.
	msg.data_len= strlen(host_name)+strlen(user_name)+2	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.data_len= 0	Length of additional data being sent

A.3.9 RD_LOGOFF (104)

Performs user LOGOFF function. This is used when the debugger performs normal termination using quit or detach.

	Parameters	Description
Request	msg.request= RD_LOGOFF	Requested API function
	msg.data= NO_KILL	This field is ignored by ROM monitor
	msg.data_len= 0	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= RD_INVALID_SEQ (1011)	Not logged on.
	msg.data_len= 0	Length of additional data being sent

A.3.10 RD_READ_D (2)

This request returns the integer in the debugged process address space at the location pointed to by the *address* parameter. If the value of address is not in a valid address space, unpredictable results will occur.

	Parameters	Description
Request	msg.request= RD_READ_D	Requested API function
	msg.address= address	Address of memory to read data from
	msg.rpid= process_id	Numeric process ID on the target system
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Debugged process can not access given address.
	msg.retcode= ESRCH (3)	The *msg.pid* parameter identifies a process that does not exist
	msg.data= data	Data read at location pointed to by address. -1 if error
	msg.data_len= 0	Length of additional data being sent

A.3.11 RD_READ_FPR (12)

This request returns the content of one of the floating-point registers.

	Parameters	Description
Request	msg.request= RD_READ_FPR	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.address= register	Name of the register to be read
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Register is not defined
	msg.retcode= RD_REG_ERR (1004)	Unable to access given register
	msg.data= value	Value read from register. 0xFFFFFFFF if error occurred
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.data_len= 0	Length of additional data being sent

A.3.12 RD_READ_GPR (11)

This request returns the content of one of the general-purpose or special-purpose registers of the debugged process. Valid registers are defined in "dbg.h" and "sys/reg.h". Not all defined registers are supported for all environments.

	Parameters	Description
Request	msg.request= RD_READ_GPR	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.address= register	Name of the register to be read
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occur
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Register is not define
	msg.retcode= RD_REG_ERR (1004)	Unable to access given register
	msg.data= value	Value read from register. 0xFFFFFFFF if error occurred
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.data_len= 0	Length of additional data being sent

A.3.13 RD_READ_GPR_MULT(71)

This request returns the contents of general-purpose registers 0 to 18, inclusive, of the debugged process.

	Parameters	Description
Request	msg.request= RD_READ_GPR_MULT	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= RD_NOTSUPP (1003)	Call not supported by this interface
	msg.retcode= RD_REG_ERR (1004)	Unable to access given register
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.data_len= 76 (0x4C)	Length of additional data being sent
	msg.buffer[0-18]	Values read from GPR0 to GPR18. Undefined if error

A.3.14 RD_READ_I (1)

This request returns the integer in the debugged process address space at the location pointed to by the address parameter. If the value of address is not in a valid address space, unpredictable results will occur.

	Parameters	Description
Request	msg.request= RD_READ_I	Requested API function
	msg.address= address	Address of memory to read data from
	msg.rpid= process_id	Numeric process ID on the target system
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion.
	msg.retcode= EIO (5)	Debugged process can not access given address
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.data= data	Data read at location pointed to by address. -1 if error (retcode should also be set to EIO)
	msg.data_len= 0	Length of additional data being sent

A.3.15 RD_READ_I_MULT (71)

This request returns the 32 integers in the debugged process address space at the location pointed to by the address parameter. If the value of address is not in a valid address space, unpredictable results will occur.

	Parameters	Description
Request	msg.request= RD_READ_I_MULT	Requested API function
	msg.address= address	Address of memory to read data from
	msg.rpid= process_id	Numeric process ID on the target system
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Debugged process can not access given address
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.retcode= RD_NOTSUPP (1003)	Call not supported by this interface
	msg.buffer[0-0x1F]	Contents of addresses from location pointed to by address to address + 0x1F
	msg.data_len= 128 (0x80)	Length of additional data being sent

A.3.16 RD_READ_SPR (115)

This request reads data directly from one of the SPRs (not the process's copy). All SPR registers are accessible through this message request. The sender is responsible for supplying valid SPR values, no error checking is performed on this field.

	Parameters	Description
Request	msg.request= RD_READ_SPR	Requested API function
	msg.address= SPR number	SPR number to read
	msg.data_len= 0	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.data= value	Value read from register
	msg.data_len= 0	Length of additional data being sent

A.3.17 RD_READ_SR (118)

This request returns the content of one of the segment registers.

	Parameters	Description
Request	msg.request= RD_READ_SR	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.address= register	Name of the register to be read
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Register is not defined
	msg.retcode= RD_REG_ERR (1004)	Unable to access given register
	msg.data= value	Value read from register. 0xFFFFFFFF if error occurred
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.data_len= 0	Length of additional data being sent

A.3.18 RD_STATUS (114)

This request is used to get program execution status and to determine if a previous RD_CONTINUE request was received.

	Parameters	Description
Request	msg.request= RD_STATUS	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.address= execution status	Status is 1 if program is running and 0 if stopped. In the case of an error, this field will be -1 (0xFFFFFFFF)
	msg.data= sequence number	Sequence number of the last RD_CONTINUE request that was received
	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= RD_ESRCH (3)	The msg.pid field identifies a process that does not exist

A.3.19 RD_STOP_APPL (113)

This request is used to interrupt program execution.

	Parameters	Description
Request	msg.request= RD_STOP_APPL	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= RD_ESRCH (3)	The msg.pid field identifies a process that does not exist

A.3.20 RD_WAIT (108)

This call allows the debugger to determine the current status of the debugged process after it is stopped. The first (least significant) byte of the process status indicates the reason for stoppage: this is always 0x7f. The second byte contains the signal number that caused the stop. Valid signals are:

- AIX_SIGILL (4) - illegal instruction
- AIX_SIGTRAP (5) - hit a trap instruction (breakpoint)
- AIX_SIGFPE (8) - floating point error
- AIX_SIGSEGV (11) - storage violation

For example after hitting a breakpoint, the status of 0x57f is returned to the debugger. After the program terminates, the first byte contains 0x00 and the rest of the status holds the program exit code. After RD_KILL call wait status of 0x57f should be returned.

	Parameters	Description
Request	msg.request= RD_WAIT	Requested API function
	msg.data_len= 0	Length of data in msg.buffer
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.data= status	Process status
	msg.address= pid	Process id
	msg.data_len= strlen(message_string)	The ROM monitor always returns 0 in this field
	msg.buffer= message_string	Formatted message string text (NULL terminated)

A.3.21 RD_WRITE_BLOCK (19)

This request writes a block of data into the address space of the debugged process at the address pointed to by the msg.address field. The number of bytes to write is contained in the msg.data field and the data is in the msg.buffer field. Unpredictable results occur if the msg.address parameter points to a location that can not be accessed by the debugged process.

	Parameters	Description
Request	msg.request= RD_WRITE_BLOCK	Requested API function
	msg.address= address	Address of memory to write data to
	msg.data= count	Number of bytes of buffer area to be written
	msg.buffer	Data to be written
	msg.data_len= count	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred.
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Debugged process can not access given address.
	msg.data_len= 0	Length of additional data being sent

A.3.22 RD_WRITE_D (5)

This request writes the value of the msg.data parameter into the address space of the debugged process at the address pointed to by the msg.address parameter. Unpredictable results occur if the msg.address parameter points to a location that can not be accessed by the debugged process.

	Parameters	Description
Request	msg.request= RD_WRITE_D	Requested API function.
	msg.address= address	Address of memory to write data to
	msg.data= data	Data to write to memory.
	msg.rpid= process_id	Numeric process ID on the target system
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Debugged process can not access given address
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist.
	msg.data= data	Data written at location pointed to by address. -1 if error (retcode should also be set to EIO or ESRCH).
	msg.data_len= 0	Length of additional data being sent

A.3.23 RD_WRITE_FPR (15)

This request writes data to one of the floating-point registers:

	Parameters	Description
Request	msg.request= RD_WRITE_FPR	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.address= register	Name of the register to be written
	msg.data= value	Value to be written to the register
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Register is not defined
	msg.retcode= RD_REG_ERR (1004)	Unable to access given register
	msg.data= value	Value written to register. 0xFFFFFFFF if error occurred
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.retcode= RD_COM_ERR (1001)	Communication error occurred

A.3.24 RD_WRITE_GPR (14)

This request writes data to one of the general-purpose or special-purpose registers of the debugged process. Valid registers are defined in dbg.h and sys/reg.h. Not all defined registers are supported for all environments.

	Parameters	Description
Request	msg.request= RD_WRITE_GPR	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.address= register	Name of the register to be written
	msg.data= value	Value to be written to the register
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred.
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Register is not defined
	msg.retcode= RD_REG_ERR (1004)	Unable to access given register
	msg.data= value	Value written to register. 0xFFFFFFFF if error occurred
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.data_len= 0	Length of additional data being sent

A.3.25 RD_WRITE_I (4)

This request writes the value of the msg.data parameter into the address space of the debugged process at the address pointed to by the msg.address parameter. This request fails if the msg.address parameter points to a location that can not be accessed by debugged process. This call sets break points in the debugged process by writing TRAP (0x7D821008) instructions.

	Parameters	Description
Request	msg.request= RD_WRITE_I	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.address= address	Address of memory to write data to
	msg.data= data	Data to write to memory
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Debugged process can not access given address
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist.
	msg.data= data	Data written at location pointed to by address. -1 if error (retcode should also be set to EIO or ESRCH)
	msg.data_len= 0	Length of additional data being sent

A.3.26 RD_WRITE_SPR (112)

This request writes data directly to one of the SPRs (not the process's copy). All SPR registers are accessible through this request. The requester is responsible for supplying valid SPR values. No error checking is performed on this field.

	Parameters	Description
Request	msg.request= RD_WRITE_SPR	Requested API function
	msg.address= SPR number	SPR number to be written
	msg.data= value	Data to write to register
	msg.data_len= 0	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.data_len= 0	Length of additional data being sent

A.3.27 RD_WRITE_SR (119)

This request writes data to one of the segment registers.

	Parameters	Description
Request	msg.request= RD_WRITE_SR	Requested API function
	msg.rpid= process_id	Numeric process ID on the target system
	msg.address= register	Name of the register to be written
	msg.data= value	Value to be written to the register
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= EIO (5)	Register is not defined
	msg.retcode= RD_REG_ERR (1004)	Unable to access given register
	msg.data= value	Value written to register. 0xFFFFFFFF if error occurred
	msg.retcode= ESRCH (3)	The msg.pid parameter identifies a process that does not exist
	msg.data_len= 0	Length of additional data being sent

A.3.28 RL_LDINFO (181)

This request provides load information from the host to the ROM monitor. This request is used when the target is loaded by a process other than the debugger. The information specified on the this request will be returned on subsequent RD_LDINFO requests.

	Parameters	Description
Request	msg.request= RL_LDINFO	Requested API function
	msg.data_len= sizeof(struct ldinfo) + strlen(pathname)	Length of additional data being sent
	msg.buffer= load information	See description of RD_LDINFO request
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.data_len= 0	Length of additional data being sent

A.3.29 RL_LOAD_REQ(180)

This request flows from the ROM monitor to the host when a RD_LOAD request is received. The port of the request is for the remote loader daemon (20050) to accommodate loading by a process independent from the debugger.

	Parameters	Description
Request	msg.request= RL_LOAD_REQ	Requested API function
	msg.buffer= filename	NULL terminated string containing fully qualified name of file to be loaded
	msg.data_len= strlen(filename)	Length of additional data being sent
Response	msg.retcode= RD_COM_ERR (1001)	Communication error occurred
	msg.retcode= RD_OK (0)	Successful completion
	msg.retcode= RD_NOFILE_ERR (1006)	Can’t open file or file is incorrect format
	msg.retcode= RD_PTRACE_ERR (1014)	Error reading file
	msg.rpid= process_id	Process ID of newly loaded file. This number (integer) can not be equal to -1 (0xFFFF FFFF) or 0
	msg.data_len= sizeof(msg.rpid)	Length of additional data being sent

Appendix B. ROM Monitor Load Format

This appendix presents the ROM Monitor load format requirements.

B.1 Overview

The ROM Monitor load format is designed to permit the specification of multiple text and data sections. The format consists of a linked list of sections of specified types prefixed by a small boot header, *boot_block*, that specifies the initial target of the image and the entry point. The *boot_block* header is placed at the front of the image by **eimgbld** or **nimgbld**. The ROM Monitor does no relocation. It is assumed that the destination addresses for the individual sections are the same ones specified during the application's linkage. The *info_block* structure is reserved in the bootstrap program, bootlLib.s. **eimgbld** or **nimgbld** patch in the values within the *info_block* structure for bootLib to use at run time. The bootstrap program processes the sections back to front, that is, from the end of the image to the beginning. This is to avoid destructive overlap during the processing of typical images.

The sections are preceded by header blocks which identify the section types. The headers are linked together in a doubly linked list.

B.2 Section Types

There are three basic section types. Generally, they can occur in the image in any order, but are usually arranged in ascending address order. The section header block has the following format:

```
/*-------------------------------------------------------------------------+
| Relocation block structure.
+-------------------------------------------------------------------------*/
typedef struct rel_block {
   unsigned long        type;
   unsigned long        dest_addr;
   unsigned long        size;
   union {
        struct data_info {
                unsigned long   size_to_fill;
                unsigned long   char_to_fill;
        } data_info_str;
        struct text_info {
               unsigned long   toc_pointer;  /* used for XCOFF; not used for ELF */
                unsigned long   entry_pt;
        } text_info_str;
        unsigned long   number_symbols;
   } section_info;
   struct rel_block     *next;
   struct rel_block     *bptr;
} rel_block_t;
```

The **type** field is one of the following manifest constants:

```
#define TEXT_SECT       0x00000001
#define DATA_SECT       0x00000002
#define SYMB_SECT       0x00000004
```

The **dest_addr** specifies the target for the block, while **size** is the extent of the block, not counting the header. The bootstrap program uses this information to move the block to the destination specified at link time. **next** and **bptr** are the section header forward and backward pointers, respectively.

B.2.1 First Section

The first section is a text section. The ROM loader places the entire image at the address specified in the *boot_block* header. The entry point specified in the *boot_block* header is assumed to be a branch, followed by the first section header, *info_block*. This is to allow the bootstrap to easily gain immediate addressability to the first section block.

The format of the first section block is shown below:

```
/*-----------------------------------------------------------------------+
| First section header
+-----------------------------------------------------------------------*/
struct info_block {
        long magic_num;                 /* magic number */
        long text_start;                /* addr of text section from section header  */
        long text_size;                 /* size of text section from section header
*/
        long data_start;                /* addr of data section from section header */
        long data_size;                 /* size of data section from section header */
        long elf_hdr_size;              /* size of ELF headr */
        long sym_start;                 /* addr of symbol table */
        long num_syms;                  /* number of symbols */
        long toc_ptr;                   /* used for XCOFF; not used for ELF         */
        struct rel_block * next;        /* pointer to next boot section header
*/
};
```

magic_num is used for verification purposes and must be X’004D 5054’.

text_start is the physical address value from the object text header.

text_size is the size in bytes from the object text header.

data_start is the physical address from the object data header.

data_size is the size in bytes from the object data header.

elf_hdr_size is the size of the object header. The debugger requires this information.

sym_start is the address of the symbol table in storage.

num_syms is the number of symbol entries.

next points to the next section header.

B.2.2 Text Section

For a text section, the union **section_info** contains the structure **text_info**, specifying the entry point of the text section.

B.2.3 Data Section

For a data section, the union **section_info** contain the structure **data_info**, specifying **size_to_fill** and **char_to_fill**. These parameters are used to optionally fill a region past the size extent specified in the base rel_block with a character. It is most often used to zero bss by specifying the size of the bss in **size_to_fill** and 0x0 for **char_to_fill**.

B.2.4 Symbol Section

For symbols, the union **section_info** contains the number of symbols in the section. The data in this section consists of the symbol table from the original object file.

B.3 Boot Header

The entire image is preceded by the boot header that was added by **nimgbld** or **eimgbld**. The ROM loader uses this information to verify that it is a ROM Monitor load image, determine where to place the image, and whether to invoke the ROM Monitor debugger before transferring control to the entry point. The boot header is stripped off by the ROM Monitor loader and does not appear at the load address.

The boot header has the following format:

```
/*-------------------------------------------------------------------------+
| Boot header.
+-------------------------------------------------------------------------*/
typedef struct boot_block {
   unsigned long        magic;
   unsigned long        dest;
   unsigned long        num_512blocks;
   unsigned long        debug_flag;
   unsigned long        entry_point;
   unsigned long        reserved[3];
} boot_block_t;
```

magic identifies this image as a legitimate ROM Monitor image and must have the value X’0052 504F’.

dest is the target address for the image (after the boot header is stripped off).

num_512blocks - Boot images are padded to a multiple of 512 byte blocks. This field specifies the number of blocks.

debug_flag controls whether the ROM Monitor debugger gets control before the loaded image starts. If the value is 0x0, the image runs immediately. If 0x01, the debugger gains control as soon as the load is complete.

entry_point specifies the address where the image will receive control.

Index