INTEGRATED SYSTEMS

pSOSystem

GETTING STARTED

68K Processors

MRI Release

Integrated Systems, Inc. • 3260 Jay Street • Santa Clara, CA 95054-3309
Support: 408-980-1500, x501 or 1-800-458-7767
FAX: 408-980-0400 (corporate); 408-980-1647 (support)
e-mail: psos_support@isi.com • Home Page: http://www.isi.com

Document Title: **pSOSystem Getting Started: 68K Processors MRI Release**
Part Number: **000-5001-004**
Revision Date: **March 1996**

Contents

Contents

About This Manual

1 Introduction to the pSOSystem Environment

8 Application Examples

9 Understanding and Developing Board-Support Packages

A Board-Specific Information

About This Manual

Purpose

This manual is part of a documentation set that describes pSOSystem, the modular, high-performance real-time operating system from Integrated Systems.

This manual provides introductory information about the operation of pSOSystem, including tutorials for pSOSystem, the XRAY Debugger for pSOSystem, and shared memory multiprocessing.

Read this manual to gain an understanding of how to begin using the various software components in pSOSystem.

Audience

This manual is targeted for application developers who have a familiarity with UNIX terms and want to begin using pSOSystem.

Organization

This manual is organized as follows:

Chapter 1, "Introduction to the pSOSystem Environment," introduces you to the organization and operating theory of pSOSystem.

Chapter 2, "pSOSystem Tutorial for Workstation Hosts," guides you through the process of bringing up and starting pSOSystem on a target system.

Chapter 3, "pSOSystem Tutorial for PC Hosts," guides you through the process of bringing up and starting pSOSystem on a target system.

Chapter 4, "XRAY Debugger for pSOSystem Tutorial: Multiple Windows Version," presents a tutorial you can use to learn the XRAY Source-Level Debugger from a SunOS or Solaris host.

Chapter 5, "XRAY Debugger for pSOSystem Tutorial: Viewport Version," presents a tutorial you can use to learn the XRAY Source-Level Debugger on an HP, RS6000, or PC host.

Chapter 6, "Shared Memory Multiprocessing Tutorial," discusses how to use pSOSystem in a multiprocessor configuration running the pSOS+m kernel.

Chapter 7, "Configuration and Startup," gives an overview of the configuration table associated with the corresponding pSOSystem component, tells how to start up each component, contains formulas for changing the starting address of software components, and defines the trap vectors.

Chapter 8, "Application Examples," provides a detailed description of the example applications in the **$PSS_ROOT/apps** directory.

Chapter 9, "Understanding and Developing Board-Support Packages," explains how to develop a board-support package (BSP) for either a custom or unsupported target board.

Appendix A, "Board-Specific Information," gives information for various individual CPU boards that run Integrated Systems products, and information on switch settings, Boot PROM installation, and on serial channel usage.

A glossary provides definitions of several key pSOSystem terms.

Related Documentation

As you read this manual, you may also want to refer to the other manuals in the standard documentation set:

- *pSOSystem Installation Guide* describes the installation of pSOSystem on UNIX and PC hosts.
- *pSOSystem System Concepts* contains detailed descriptions of the pSOS+ real-time kernel, the pSOS+m multiprocessing kernel, network programming, the pHILE+ file system manager, the pREPC ANSI C library, and the input/output system.
- *pSOSystem Programmer's Reference* contains detailed descriptions of system services, interfaces and drivers, configuration tables, and memory usage.
- *pSOSystem System Calls* provides a reference of pSOS+, pHILE+, pREPC+, pNA+, and pRPC+ system calls and error codes.
- *pROBE+ User's Guide* describes how to use the pROBE+ system-level analyzer for monitoring pSOSystem execution.

Based on your software configuration, you may need to refer to one or more of the following manuals:

- *C++ Support Package User's Manual* documents the C++ support services including the pSOSystem C++ Classes (library) and support for the C++ run-time library.
- *ESp User's Manual: PC Hosts* and *ESp User's Guide: Workstation Hosts* document the ESp front-end analyzer, which displays application activities, and the pMONT component, the target resident application monitor.
- *LAP Driver User's Guide* describes the interfaces provided by the LAP (Link Access Protocol) drivers for OpEN product, including the LAPB and LABD frame-level products.
- OpEN*: OSI Lower Layers User's Guide* describes how to use the pSOSystem Open System Interconnections (OSI) product named OPeN: OSI Lower Layers.
- OpEN *User's Manual* describes how to install and use the pSOSystem OpEN (Open Protocol Embedded Networking) product.

- *OSPF User's Guide* describes the Open Shortest Path First (OSPF) pSOSystem protocol driver.
- *Point-to-Point Protocol Driver's User's Guide* describes how to install and use the pSOSystem Point-to-Point Protocol (PPP) product.
- *SNMP User's Manual* describes the internal structure and operation of SNMP, the Simple Network Management Protocol product from Integrated Systems. It also describes how to install and use the SNMP Management Information Base (MIB) Compiler.
- *TCP/IP for OPeN User's Guide* describes how to use the pSOSystem Streams-based TCP/IP for OpEN (Open Protocol Embedded Networking) product.
- *XRAY Debugger for pSOSystem User's Guide* describes how to use the XRAY Debugger to debug pSOSystem applications.
- *X.25 User's Guide* describes the interfaces provided by the X.25 for Open multiplexing driver that implements the packet level protocol defined by the CCITT.

Support

Customers in the United States can contact Integrated Systems using one of the following methods:

- Call 408-980-1500, extension 501, or 1-800-458-7767.
- Send a Fax to 408-980-1647.
- Send e-mail to **psos_support@isi.com**.

International customers can contact their local pSOSystem distributor for assistance or call 408-980-1500, extension 501.

Before contacting Integrated Systems Technical Support, it is helpful to have the following information:

- The exact version of the pSOSystem software you are running. This information is in the **$PSS_ROOT/include/version.h** file.
- Your customer ID. This is a 4-digit customer number assigned by Integrated Systems.

Integrated Systems actively seeks suggestions and comments. Please send your comments by e-mail to **ideas@isi.com**.

Notation Conventions

The following notation conventions are used in this manual:

- Function names (**q_receive**), filenames (**pdefs.h**), pathnames (**usr/psosroot/bsps/**), keywords (**int**), UNIX program names (**tip** and **cu**), and operators (**!**) that you must type exactly as shown are in **bold**. Code examples also appear in **bold.**
- *Italics* indicate that a user-defined value or name (*drive*:*pathname*) can be substituted for the italicized words shown. Italics also indicate emphasis, such as when important terms are introduced.
- Keynames [Enter] are shown within square brackets. Keynames separated by hyphens are typed together. For example, to type [Control-Shift-e], hold down the [Control] and [Shift] keys and type the letter e.
- `Code examples are shown in constant width.`
- Hexadecimal numbers are preceded by 0x or 0X (for example, 0xDEADDEAD).

NOTE: This manual uses the breve symbol (•) to indicate a required space.

1 Introduction to the pSOSystem Environment

This chapter introduces the internal organization and operating theory of the pSOSystem environment. Read this chapter before you attempt to use the tutorials in subsequent chapters.

1.1 Target Architecture

The purpose of the pSOSystem environment is to help you develop an application on a host system, then download and run the application on an embedded computer. The embedded computer is called the *target system*. The description of the pSOSystem environment begins with the target system architecture. The description of the host system starts in Section 1.2, "Host Development System Layout." For an illustration of the relationship between the host and the target system, see Figure 1-1 on page 1-2.

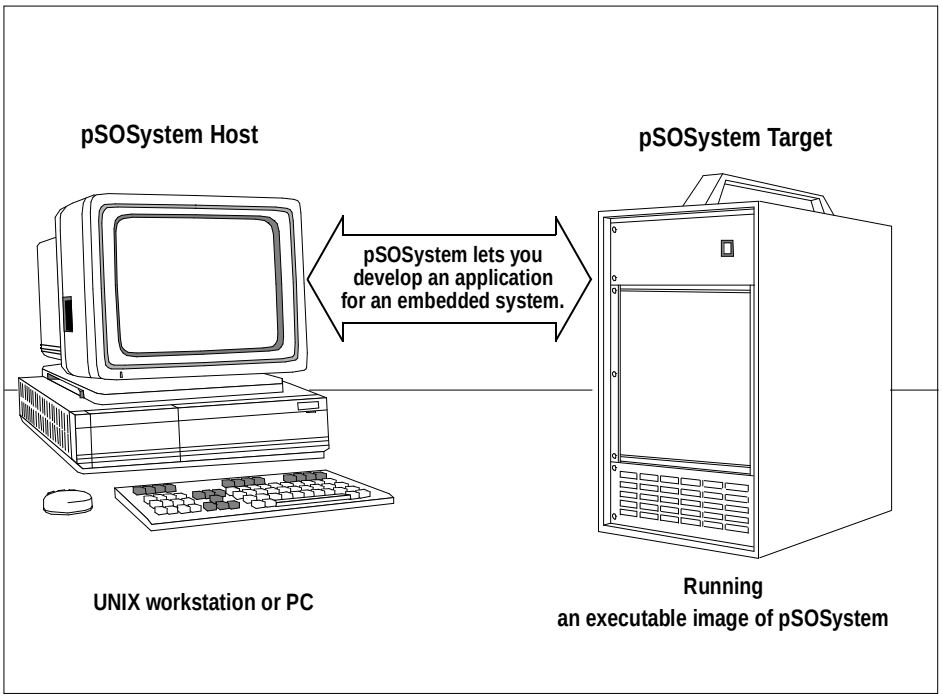


Figure 1-1 Architecture of pSOSystem

In a pSOSystem environment model, the target system software is usually an application that you develop on the host, as shown in Figure 1-1. Two major software elements run on the target hardware: the *pSOSystem* software and the *application code*. You link these elements together on the host system and download this combination to the target. The downloaded software is called an *executable image*.

You do not necessarily have to link the application code and the pSOSystem software before you download them as one executable image. Instead, you can use the pSOSystem Loader at runtime to load application modules to the target. (For a detailed description of the Loader, see the description of system services in the *pSOSystem Programmer's Reference*.) For simplicity, this manual assumes that you use one executable image.

1.1.1 pSOSystem Software

The pSOSystem software provides a standard set of services for the application code and debugging tools. It almost always contains the pSOS+ real-time kernel and frequently contains the following companion software elements:

- pROBE+, pNA+, and pHILE+ components
- Device drivers and interrupt handlers for the target hardware
- Configuration tables used to customize the operating system for a particular target system

The pSOSystem software is a combination of standard components, system configuration code, and hardware-specific environment code. The hardware-specific code is known as a *pSOSystem Board-Support Package*, or BSP. Integrated Systems provides BSPs for a number of commercially available target boards. If you are using one of these boards, you can begin developing pSOSystem application code immediately. If you are using unsupported or custom hardware, you must provide a board-support package for the target system. For detailed information on board-support packages, see Chapter 9, "Understanding and Developing Board-Support Packages."

1.1.2 Application Code

The application code is what makes one target system different from another. It implements the functional behavior of the target system. Normally, application code is very specialized and contains few standard software elements, if any. It is usually developed from scratch, although you can utilize code fragments from the sample applications that come with the pSOSystem software.

1.2 Host Development System Layout

pSOSystem code consists of read-only *object* libraries, *include* files, and *source* files. The code can be kept in a central location on the host system so that multiple users can have access to it. The directory tree that contains this shared code is the *pSOSystem directory tree*, and its root directory is the *pSOSystem root directory*. Within pSOSystem source files,

pathnames generally begin with **PSS_ROOT**. You should set the environment variable PSS_ROOT to the pathname of the pSOSystem root directory, as explained in the *pSOSystem Installation Guide*.

You can create a pSOSystem executable image from any directory in the host system, not just within the pSOSystem root directory tree. A directory where you create an executable image is called a *working directory*. For information on the contents of this directory, see Section 1.2.4, "Working Directory." For information on the executable image and how to create a working directory, see Chapter 2, "pSOSystem Tutorial for Workstation Hosts."

1.2.1 Configuration Files

Source files that control the configuration of the pSOSystem environment are called configuration files. Configuration files exist for all systems built with the pSOSystem software, and these files are compiled and linked into the executable image. A set of the common configuration files resides in the directory **PSS_ROOT/configs/std**. You should not need to make changes to these common files.

The configuration files contain parameters that control the behavior of the pSOSystem software. Examples of these parameters are the baud rate for the serial channels, IP addresses in networked systems, and the operating mode for the pROBE+ debugger. You can change these parameters either when you build the pSOSystem environment or through an interactive startup dialog during run-time startup. For detailed information on configuration parameters, see Chapter 7, "Configuration and Startup."

Configuration parameters are normally specified at system build time by the values you supply in the system configuration file **sys_conf.h**. This system configuration file resides in the working directory. An option in **sys_conf.h** allows you to specify that the operating system try to locate saved versions of these parameters in the target board's nonvolatile storage area. This is useful when you are using the pSOSystem Boot ROMs because the executable image you then download can use the same parameter values you give to the Boot ROMs. You can also enable a special startup dialog that allows you to change the parameters at runtime start-up through an RS-232 connection. Both of these options are enabled by definitions in **sys_conf.h**.

The C source files in **PSS_ROOT/configs/std** contain numerous conditional compilation statements that are controlled by the contents of **sys_conf.h**. The **dialog.c** file contains the source code for the optional system startup dialog. Most of the other files contain the startup code that builds the configuration tables for the various operating system components. These files are provided as read-only source files; you should not need to modify them.

In addition to the source files, **PSS_ROOT/configs/std** contains a file called **config.mk**, which the application's **makefile** must include. The directives in **config.mk** compile the files in the **std** directory.

1.2.2 Board-Support Packages

Directory **PSS_ROOT/bsps** contains a collection of directories that are included in the pSOSystem software. Each directory in **PSS_ROOT/bsps** contains a board-support package (BSP) that corresponds to a specific board. For example, the following path:

PSS_ROOT/bsps/*boardname*

indicates a board-support directory for *boardname*. For a detailed description of BSPs, see Appendix A, "Board-Specific Information."

1.2.3 System Library

The **sys.lib** file in the **PSS_ROOT/sys/os/** directory is the *system library*. It contains the various operating system components and the runtime bindings that the application uses to make system calls to these components. The system library is usually built once as part of the pSOSystem host installation. It needs to be rebuilt only when you receive new or updated software components. For additional information about the system library, see the *pSOSystem Installation Guide*.

1.2.4 Working Directory

The pSOSystem executable image is built from within a *working directory*. The working directory contains the application code. Its location does not depend on the location of the pSOSystem root directory. A working directory must contain the following:

- A system configuration file (**sys_conf.h**)
- A **makefile**

- A driver configuration file (**drv_conf.c**)
- Application code

1.2.4.1 System Configuration File

The system configuration file **sys_conf.h** is a C *include* file that must reside in the working directory. The **sys_conf.h** file has many elements and affects many aspects of the pSOSystem environment. The following list illustrates the range of items that **sys_conf.h** controls, namely:

- Which pSOSystem components are built into the executable image.
- Which peripheral devices are enabled.
- Whether a startup dialog is included.
- How the system initialization code is compiled.
- Various entries in the individual component configuration tables, such as the numbers of tasks, queues, and other objects, for the pSOS+ environment.

For a detailed description of the **sys_conf.h** file, see Chapter 7, "Configuration and Startup."

Each of the pSOSystem sample applications includes an example **sys_conf.h** file tailored to fit the application as described in Section 1.2.5, "Sample Applications." However, you may still need to change some parameters, such as the IP address of the target board.

1.2.4.2 makefile

This section describes the rules for writing a **makefile** to build a pSOSystem application.

The first items in the **makefile** are the following macro definitions:

PSS_BSP — Supplies the pathname of the pSOSystem board-support package you use to build the executable image. This is usually one of the subdirectories of **PSS_ROOT/bsps**.

PSS_DRVOBJS — Defines a list of object files and/or libraries for drivers that you have added to the pSOSystem environment. It must include at least **drv_conf.o**. For a detailed description of how to add drivers to

a system, see Chapter 7, "Configuration and Startup."

PSS_APPOBJS — Defines a list of all the object files and/or object libraries that make up the application.

After the preceding macro definitions, the **makefile** must have the following lines:

```
PSS_CONFIG=$(PSS_ROOT)/configs/std
include $(PSS_BSP)/bsp.mk
include $(PSS_CONFIG)/config.mk
```

The remainder of the **makefile** contains the rules that define how to build application modules. The ***.mk** files that you include define several macros. These macros are used in the following **makefile** commands:

CC — Invokes the C compiler

COPTS — Specifies options for the C compiler that are appropriate for building an executable image

AS — Invokes the assembler

AOPTS — Specifies options for the assembler that are appropriate for building an executable image

The following is an example **makefile** for building an application that contains three object modules (root, task 1, and task 2).

```
PSS_BSP=$(PSS_ROOT)/bsps/cvme964
PSS_DRVOBJS=drv_conf.o
PSS_APPOBJS= root.o task1.o task2.o

PSS_CONFIG=$(PSS_ROOT)/configs/std
include $(PSS_BSP)/bsp.mk
include $(PSS_CONFIG)/config.mk

root.o: root.c
            $(CC) $(COPTS) -o root.o root.c
task1.o: task1.c incl.h
            $(CC) $(COPTS) -o task1.o task1.c
task2.o: task2.c incl.h
            $(CC) $(COPTS) -o task2.o task2.c
drv_conf.o: drv_conf.o \
            makefile \
            sys_conf.h \
```

```
$(PSS_ROOT /include/bspfuncs.h \
$(PSS_ROOT /include/configs.h \
$(PSS_ROOT /include/sysvars.h \
$(PSS_ROOT /bsp.h \
$(CC) $(COPTS) -o drv_conf.o drv_conf.c
```

When you invoke **make** to build the pSOSystem executable image, specify one of the following output targets for **make**:

ram.hex An executable image in S-record format for Motorola processors or Intel Extended Hexadecimal format for Intel processors, suitable to download to the target board's RAM.

ram.x An executable image in IEEE-695 format, suitable to load to the target board's RAM with the XRAY Debugger for pSOSystem.

rom.hex An executable image in S-record format for Motorola processors or Intel Extended Hexadecimal format for Intel processors, suitable for placement in ROM.

rom.x An executable image in IEEE-695 format, suitable for placement in ROM. It is seldom useful for producing ROMs unless the PROM programmer accepts IEEE-695 formatted input files.

os.hex An executable image of the pSOSystem software in S-record format for Motorola processors or Intel Extended Hexadecimal format for Intel processors without the application.

os.x An executable image of the pSOSystem software in IEEE-695 format, without the application.

app.hex An executable image of the application (without the operating system) in S-record format for Motorola processors or Intel Extended Hexadecimal format for Intel processors.

app.x An executable image of the application (without the operating system) in IEEE-695 format.

NOTE: If you don't specify a target, the first target found in the application's makefile is built.

To build a system for downloading to the target board's RAM through the pROBE+ debugger, for example, you would enter the following:

```
make ram.hex
```

The pSOSystem build process also produces an ASCII map file. The map file contains a load map and cross-reference listing of symbols. Its name is **ram.map**, **rom.map**, **os.map**, or **app.map**, depending on the output target you specify.

1.2.4.3 Driver Configuration File

The driver configuration file **drv_conf.c** contains two routines that are called during system startup to install pSOSystem drivers in the appropriate tables. Each of the pSOSystem sample applications includes an example driver configuration file. Normally, you don't need to edit this file unless you are adding special or custom drivers to the pSOSystem environment. For a detailed description of how to add drivers to a system, see Chapter 7, "Configuration and Startup."

1.2.5 Sample Applications

Directory **PSS_ROOT/apps** has a number of subdirectories, each subdirectory containing a pSOSystem sample application. The sample applications allow you to build, download, and run an executable image without writing a single line of code if you use a supported target platform. Integrated Systems recommends that you use one of these sample applications as a starting point for developing your own application. The examples in Chapter 2, "pSOSystem Tutorial for Workstation Hosts," show how to do this. Along with source code for the application, each sample application directory includes a properly structured **makefile** and the pSOSystem configuration files.

Each sample application is designed to illustrate a different aspect of pSOSystem. Source code is provided for all samples, and you can use the source code for each sample as a starting point for an application or as a learning tool. Each sample resides in its own subdirectory, and each subdirectory contains a **readme** file with detailed information about the sample. The pSOSystem base package includes the following samples in the **apps** directory:

hello This sample program prints a message to either the target's serial port or the XRAY standard output screen. The **hello** program is used by Chapter 2, "pSOSystem

Tutorial for Workstation Hosts," and Chapter 3, "pSOSystem Tutorial for PC Hosts."

pnabench This sample program runs a benchmark program that measures TCP throughput.

xraydemo This sample program is used by Chapter 4, "XRAY Debugger for pSOSystem Tutorial: Multiple Windows Version," and Chapter 5, "XRAY Debugger for pSOSystem Tutorial: Viewport Version."

proberom This is the application that is used to build pSOSystem Boot ROMs for non-networking systems.

tftp This application is used to build the pSOSystem Boot ROMs for networking systems. It includes a TFTP bootloader program for loading and starting a pSOSystem executable image.

philepna This application demonstrates the use of pSOS+, pHILE+, pNA+, and pREPC+.

nfs This application demonstrates the use of pSOSystem NFS client services.

xray_cxx This application demonstrates the use of the C++ Support Package.

The next chapter explains how to use a sample application by copying it to a local working directory. You can also create new applications by taking code fragments from different samples and combining them. For more information about sample programs, see Chapter 8, "Application Examples."

1.3 Standard Download/Debug Sequence

After an executable image is created, the next steps are to download it to the target system, initiate its execution, and begin debugging. This section provides a brief introduction to this process.

The description in this section assumes that the pROBE+ debugger has been configured into the pSOSystem environment. The pROBE+ debugger is used to debug application code either directly or through a

source-level debugger. You can remove the pROBE+ debugger after fully debugging the executable image.

1.3.1 Requirements for Downloading the Executable Image

An executable image does not depend on the way it is downloaded or its execution initiated. You can use tools such as ROM monitors or in-circuit emulators that can read the following formats:

- Motorola S-record for Motorola processors
- Intel Extended Hexadecimal for Intel processors
- IEEE-695 for source-level debuggers

The pSOSystem software provides ROM monitors called *pSOSystem Boot ROMs* for all supported boards. (For descriptions of supported boards, see Appendix A, "Board-Specific Information.") The ROM monitors are described in "Section 1.4, "pSOSystem Boot ROMs," and in greater detail in the *pSOSystem Programmer's Reference*.

The executable image generated with MRI tools is automatically placed at its final execution address, so no user-supplied relocation address or offsets are required for downloading.

1.3.2 Starting the Executable Image

The executable image is started by passing control to its *start address*. An image's start address depends on the hardware and thus can vary from board to board. You can find the start of the **code** section in the **ram.map** file.

For executable images built with pSOSystem, the starting address is the start of the **code** section + 8. For example, if the **code** section starts at 0xA0060000, the starting address is 0xA0060008.

When control passes to the start address, the pROBE+ debugger gains control and waits for further input. Neither the components nor the application code is yet initialized or running.

The pROBE+ debugger operates in either *stand-alone* or *remote* mode. In stand-alone mode, the pROBE+ debugger is controlled through a terminal connected to the target hardware. In remote mode, the pROBE+ debugger acts as a back-end to a source-level debugger on the host system. They are connected either through a serial channel or a network connection. The operating mode of the pROBE+ debugger and the

communication medium (when using remote mode) are specified in the **sys_conf.h** file. The three possible operating modes are as follows:

- The pROBE+ debugger operating stand-alone
- A source-level debugger and the pROBE+ debugger communicating over a serial channel
- A source-level debugger and the pROBE+ debugger communicating over an Ethernet network

The pROBE+ debugger operating in stand-alone mode may be adequate for your initial needs; however, you may find that a source-level debugger running over a serial channel or an Ethernet network provides additional benefits. To communicate over an Ethernet network, you will need the pNA+ TCP/IP Network Manager.

1.3.3 Debugging the Operating System and Application Code

When the pSOSystem software first begins operation, the pROBE+ debugger comes up and awaits further input from its console. Usually, you initialize the remainder of the pSOSystem software now. The pROBE+ debugger has commands to initialize the pSOS+ kernel and other components. After component initialization, control returns to the pROBE+ debugger. Note that the application code has not yet started.

After the pSOSystem software has been initialized, the pSOS+ ROOT and IDLE tasks have been created, and execution of the ROOT task is pending. Normally, you would also set one or more breakpoints before you start ROOT. The pROBE+ debugger uses the **go** command to pass control to the application code's ROOT task. Descriptions of this and other commands appear in the forthcoming tutorials.

In summary, initiating and debugging an executable image involves the following steps:

1. Loading the executable image into the target system
2. Passing control to the start address of the executable image
3. Starting the pSOS+ kernel
4. Setting desired breakpoints, if any
5. Starting the application

The examples in subsequent chapters demonstrate these steps.

1.4 pSOSystem Boot ROMs

One way to download and start an image is by using the ROM monitors that are available for all supported target boards or can be built using either the **proberom** or **tftp** sample application. The ROM monitors are called *Boot ROMs*. This section provides an overview of how the ROMs are used to download and start an executable image.

1.4.1 Inside the Boot ROMs

A pSOSystem Boot ROM set is actually a pSOSystem executable image burned into ROM. Two types of Boot ROMs are available:

- ROMs that contain just the pROBE+ debugger allow you to download and start an executable image through an RS-232 connection to the host. They use either the stand-alone pROBE+ debugger or the pROBE+ debugger with a source-level debugger.
- ROMs that contain the pNA+ network manager, the pSOS+ kernel, and a TFTP bootloader application in addition to the pROBE+ debugger let you use a network connection between the pROBE+ debugger and a source-level debugger to download and start the executable image. These ROMs can optionally use an automatic bootloader, rather than the pROBE+ debugger or a source-level debugger, to load and start the image. The bootloader uses the TFTP protocol, so the host system must have a running TFTP server.

A target board's capability determines which ROM set is supplied. The full-featured TFTP ROMs are supplied for boards whenever possible. Boards that lack networking capability come with pROBE-only ROMs.

The source code for the ROMs is supplied as a pSOSystem sample application. You can build the ROMs for your target board from the supplied source code.

1.4.2 Using the Boot ROMs

The ROM-resident pSOSystem software is not the same as the version contained in the executable image. The executable image has the following dependencies on the Boot ROMs:

- The downloaded code must not overwrite the RAM that the Boot ROMs use. No memory conflicts must occur between the

Boot ROMs and linker command files for a supported board. See Appendix A for the board-specific memory usage.

- The Boot ROMs and the downloaded system can optionally share certain operating parameters, such as the serial channel baud rate and the board's IP address in a networked system.

To help avoid any confusion, code modules within the ROMs are prefixed with the word *ROM*, and those in the executable image are prefixed with the word *downloaded*. For example, the pROBE+ debugger in the Boot ROMs is referred to as the ROM-pROBE+ debugger and the pSOSystem software in the executable image is called the downloaded operating system.

When you start the downloaded operating system, its operation may seem very similar to the ROM operating system. Consider, for example, the Boot ROMs you configure to run with the pROBE+ debugger from a terminal. Later, by using the ROM-pROBE+ debugger you download and start the executable image. The downloaded pROBE+ debugger is also configured to run from a terminal, so no differences appear. It is important, however, to understand that you are now monitoring a different instance of the pROBE+ debugger. Instructions for these procedures appear in the Chapter 2, "pSOSystem Tutorial for Workstation Hosts." Subsequent chapters contain several examples of Boot ROM usage.

When using Boot ROMs, you typically take the following steps:

1. Reset or power-on the CPU board, bringing control into the Boot ROM's configuration mode.
2. Configure the ROMs, as needed. The ROM-pROBE+ debugger can be configured to operate in stand-alone mode, or with a source-level debugger over a serial channel or network connection.
3. Exit ROM configuration mode, initiating execution of the ROM operating system. This primarily means starting the ROM pROBE+ debugger.
4. Use the ROM pROBE+ debugger (or a source-level debugger) to download the executable image.
5. Use the ROM pROBE+ debugger (or a source-level debugger) to pass control to the start address of the executable image.

6. Continue as described in Section 1.3, "Standard Download/ Debug Sequence."

For more information about Boot ROMs, see Appendix A, "Board-Specific Information."

2 pSOSystem Tutorial for Workstation Hosts

This chapter is for pSOSystem users who are developing applications from UNIX workstations. The tutorial in this chapter helps you gain hands-on experience with the pSOSystem environment by building, downloading, and running an executable image. The executable image you build contains a sample application called **hello**. The **hello** sample resembles the Hello World program familiar to most programmers. It prints a short message and exits.

By following the examples in this chapter, you can learn how to complete the following tasks:

- Configure the pSOSystem software for execution in a particular hardware environment.
- Combine operating system and application code into an executable image.
- Transfer the executable image to the target system and execute it under the control of the pROBE+ debugger.

2.1 Recommended Software and Hardware

This chapter contains explanations and commands you can execute with the following hardware and software:

- This tutorial assumes you have installed the pSOS+ kernel, the pSOSystem standard components, one or more compilers, and a source-level debugger.
- This tutorial assumes you are using the pROBE+ debugger although the pSOSystem software does not require it.
- The examples in this chapter use the pSOSystem Boot ROMs, although the pSOSystem environment does not require them. The Boot ROMs depend on the pROBE+ debugger for downloading and other services.
- Many examples assume a terminal emulation program is running on the host (for example, **cu** or **tip** on a UNIX system). Although the pSOSystem software does not require an emulator, the examples in this tutorial require an emulator.

The examples in this tutorial have target hardware dependencies. The pSOSystem software includes fully operable board-support packages for the supported CPU boards described in Appendix A, "Board-Specific Information." This chapter is based on the use of these supported boards for the following reasons:

- Developing a pSOSystem board-support package for an unsupported board requires development of various board-dependent drivers, interrupt handlers, and so on. Because this is nontrivial, familiarity with the pSOSystem environment *before* you build a custom BSP is helpful. Therefore, you should do the examples in this chapter and, if necessary, create a pSOSystem BSP for the target hardware.
- The pSOSystem software also includes Boot ROMs for all supported boards. Although normal pSOSystem operation does not require Boot ROMs, the examples depend on them.

If the target hardware is not a supported board, you should still try to obtain a supported board for temporary use. The supported boards are widely available, but if you cannot obtain one, you should still read this section and then proceed to Chapter 9, "Understanding and Developing Board-Support Packages."

2.2 Creating a Working Directory

Before you begin this tutorial, create your own copy of the **hello** sample application code in a working directory. The easiest way to do this is with a copy command that copies an entire directory tree, as the following examples show. If you have already set up the PSS_ROOT environment variable, enter the following command:

cp -r $PSS_ROOT/apps/hello *workdir*
cd *workdir*

where *workdir* is the name of a new working directory.

2.3 hello Sample Application

Before you run the **hello** sample, take a few minutes to examine its source code. A listing of the working directory should show the following:

makefile **drv_conf.c** **sys_conf.h**
readme **root.c**

The **root.c** file contains the code for the **hello** sample's ROOT task. This code prints a message to the console, and then suspends itself. Self-suspension causes the IDLE task to run indefinitely.

The source code in **root.c** illustrates some of the basic elements of building pSOSystem application code. As you examine the contents of **root.c**, note the following:

1. First, **root.c** includes the pSOSystem configuration file **sys_conf.h**. For this tutorial, **sys_conf.h** defines the device numbers of the serial driver and periodic tick timer. For a thorough description of the **sys_conf.h** file, see Chapter 7, "Configuration and Startup."

2. Next, **root.c** includes a standard pSOSystem header file, **psos.h**. Standard pSOSystem header files reside in the **include** directory in the pSOSystem root directory. **psos.h** defines all pSOS+ kernel-related constants such as error codes and options. All source files that call pSOS+ services should include **psos.h**. Similar header files exist for other components.

3. Next, **root.c** includes the **probe.h** file. This header file contains the declaration for the external routine **db_output()**. This routine is used to print messages to the I/O screen of the source-level debugger. Source code for **db_output()** comes in a standard file shipped with the XRAY Debugger for pSOSystem.

4. The next statement defines the OUTPUT_TO_DEBUGGER constant. This constant defines the device to which the **hello** sample directs its output. If OUTPUT_TO_DEBUGGER is 0, output goes to the pSOSystem standard console device. Otherwise, output goes to the I/O screen. The purpose of this parameter is explained later.

5. The tick timer and default console are automatically initialized. This auto-initialization is controlled by the SC_AUTOINIT define statement in the **sys_conf.h** file. The timer device provides periodic interrupts, and its interrupt service routine (ISR) announces these ticks to the pSOS+ kernel so the kernel can perform time-related functions.

6. The **#if** statement controlled by OUTPUT_TO_DEBUGGER implements the output switch described above. If printing goes to the console, **de_write()** is called to print the message. Otherwise, **db_output()** is called.

7. Finally, the task calls **t_suspend()** to suspend itself.

2.4 Building an Executable Image

Before you build an executable image, edit the **makefile** to make sure the definition of PSS_BSP reflects the target board. (Remove the comment mark if one is present. The line defining PSS_BSP becomes the first non-commented line in the **makefile**.) For example, if the name of the target system is *boardname*, the **makefile** should contain the following:

PSS_BSP=$(PSS_ROOT)/bsps/*boardname*

If you have the pSOS+m multiprocessor kernel, the **sys_conf.h** file also requires an edit: search **sys_conf.h** for the following lines:

```
#define SC_PSOS      YES
#define SC_PSOSM     NO
```

Change these two lines to show YES for the presence of the pSOS+m kernel and NO to show the absence of the pSOS+ kernel.

To build an executable image, enter the following:

make ram.hex

The preceding action compiles, assembles, and links all the files required to build an executable image. The executable image contains the **hello** sample application and a version of the pSOSystem software for the current hardware. After the build completes, verify that the two output files, **ram.map** and **ram.hex**, were created. Chapter 1, "Introduction to the pSOSystem Environment," describes these files.

If any errors occurred during the build process, first verify that you performed the copy commands correctly. If you did, then the installed compiler tools may have a problem. If you cannot locate the problem, contact Integrated Systems Technical Support.

2.5 Downloading the Executable Image

To run the executable image created in the previous section, you must first download it to the target system by using the pROBE+ debugger. This section takes you through the download sequence.

2.5.1 Starting the Boot ROMs

For installation instructions for the ROMs, see Appendix A, "Board-Specific Information."

The target board through an RS-232 connection must be connected to an ASCII terminal or a terminal emulation program running on the host. A terminal emulation program is preferable because, with an emulator, you can use the same channel to download to the target.

If a target board has more than one RS-232 port, refer to Appendix A to determine which port to use.

Set the terminal or emulator characteristics to:

- 9600 baud
- 8-bit data
- 1 stop bit

- no parity

When you power up or reset the board, the terminal displays a pSOSystem startup message similar to the one in Figure 2-1.

```
pSOSystem V2.2.0
Copyright (c) 1991 - 1996, Integrated Systems, Inc.
-------------------------------------------------------------------------
START-UP MODE:
  Boot into pROBE+ stand-alone mode
NETWORK INTERFACE PARAMETERS:
  LAN interface is disabled
  Shared memory interface is disabled
MULTIPROCESSING PARAMETERS:
  This board is currently configured as a single processor system
HARDWARE PARAMETERS:
  Serial channels will use a baud rate of 9600
  This board's memory will reside at 0x1000000 on the VME bus
  After board is reset, start-up code will wait 60 seconds
-------------------------------------------------------------------------
To change any of this, press any key within 60 seconds
```

Figure 2-1 pSOSystem Startup Message

If a message like the one in Figure 2-1 on page 2-6 does not appear, check for the following:

- A null modem cable between the target CPU and the terminal may be required.
- Make sure the ROMs are in the correct sockets and have no bent pins.
- Make sure the jumper and switch settings are correct.

If the ROMs still do not operate, contact Integrated Systems technical support.

The pROBE+ startup message shows the default ROM configuration parameters. After the startup message appears, the ROMs wait 60 seconds for keyboard input. If you do not enter a character within this time, the ROMs operate with the configuration shown.

The preceding display shows that the startup mode for the pROBE+ debugger is stand-alone, and the startup code has a wait period of 60 seconds. Use this mode and the 60-second wait for now.

After the wait period has elapsed, a message similar to the following should appear to indicate that the ROM pROBE+ debugger is waiting for input:

```
pROBE+ V2.1.1
COPYRIGHT 1990 - 1996, INTEGRATED SYSTEMS, INC.
ALL RIGHTS RESERVED
pROBE+>
```

The ROM pROBE+ can now be used to download the **ram.hex** file that contains the executable image.

2.5.2 Downloading the Executable Image File

To download the executable image file, **ram.hex**, enter the pROBE+ **dl** (download) command, as follows:

```
pROBE+>dl
```

pROBE+ is now waiting for you to download **ram.hex**. Enter the commands necessary to cause the terminal emulation program to transmit **ram.hex** over the serial channel. For example, on a UNIX system that uses the **cu** terminal emulation program, you can use the following command:

```
~$cat ram.hex
```

If the download is successful, the pROBE+ debugger should display a message similar to the following (the number of records may vary):

```
9840 records read
pROBE+>
```

NOTE: The download takes about five minutes.

Sometimes, the terminal emulation program misses part or all of this message. If so, the results of the most recent download can be obtained by entering the pROBE+ **dl** command again with the **prev** option, as follows:

```
pROBE+>dl prev
9840 records read
```

2.5.3 Starting the Executable Image

The pROBE+ **go** command begins or continues execution. When you enter **go** with the start address, the executable image is started by passing control to its start address. To determine the start address for individual boards, see Appendix A, "Board-Specific Information." An executable image is started by passing control to the image's *load address* + 8.

Once you have the board's start address, enter the command with the start address, as in the following example:

```
pROBE+>go 28008
```

A message similar to the following should appear:

```
pROBE+ V2.1.1
COPYRIGHT 1990 - 1996, INTEGRATED SYSTEMS, INC.
ALL RIGHTS RESERVED

pROBE+>
```

This is the point at which the **hello** sample application can run. Although this message may look similar to the one displayed when the Boot ROMs started, the message comes from the downloaded pROBE+ debugger rather than the ROM pROBE+ debugger. The ROM pROBE+ debugger is no longer functioning and can regain control only if you reset the board.

Like the ROM pROBE+ debugger, the downloaded pROBE+ debugger is running in stand-alone mode. If the parameter SE_DEBUG_MODE is set to STORAGE in **sys_conf.h**, the downloaded pROBE+ debugger operates in the same mode as the ROM pROBE+ debugger. This is the default and is usually the most convenient approach, although ROM and downloaded pROBE+ debuggers do not need to operate in the same mode. If SE_DEBUG_MODE is set to either DBG_SA, DBG_XS, or DBG_XN, the downloaded pROBE+ debugger operates in the mode specified by the SE_DEBUG_MODE setting regardless of the ROM pROBE+ operating mode.

2.5.4 Executing the hello Sample Application

The pROBE+ **gs** (go start pSOS+) command passes control to pSOS+ startup. pSOS+ startup initializes the pSOS+ kernel and any other

operating system components, and then returns control to the pROBE+ debugger. Enter the **gs** command now:

```
pROBE+>gs
```

You should see output similar to that in Figure 2-2.

```
pROBE+> gs

pSOS Initialized.  Running:  'ROOT' -#00020000
--------------------------------------------------------------------------------
SR=3000-tfSM.000...xnzvc USP=FFFFFFFF MSP=003FEF94 ISP=000C342C
VBR=00000000 DFC=0 SFC=0 CACR=80008000
DR=00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
AR=00000000 00000000 00000000 00000000  00000000 00000000 00000000
PC=00029548-00029548  4E56FFE8          LINK    A6,#-24
```

Figure 2-2 Output of pROBE+ gs Command

The pSOS+ kernel is now initialized, and execution of the ROOT task is pending. Because the pSOS+ kernel has just been initialized, only two tasks are in the system. You can verify this by entering the pROBE+ **qt** (query task) command:

```
pROBE+>qt
```

The following information appears as shown in Figure 2-3.

```
pROBE+> gt

  Name      TID    Prio Mode Status    Susp?     Parameters             Ticks
--------------------------------------------------------------------------------
 'IDLE' -#00010000  00  2000   Ready
 'ROOT' -#00020000  E6  2000 Running
```

Figure 2-3 Output of pROBE+ qt Command

You can use the pROBE+ **di** (disassemble memory) command to examine the code that is ready to execute, as follows:

```
pROBE+>di
```

Entering di (**disassemble instruction**) causes the disassembled memory contents to appear as in Figure 2-4.

```
pROBE+> di

   00029548-00029548  4E56FFE8          LINK    A6,#-24
   0002954C-0002954C  2F02              MOVE.L  D2,-(A7)
   0002954E-0002954E  486EFFFC          PEA.L   -4(A6)
   00029552-00029552  486EFFF8          PEA.L   -8(A6)
   00029556-00029556  42A7              CLR.L   -(A7)
   00029558-00029558  2F3C00020000      MOVE.L  #131072,-(A7)
   0002955E-0002955E  4EB900030714      JSR.L   ($00030714).L
   00029564-00029564  486EFFFC          PEA.L   -4(A6)
```

Figure 2-4 Output of the pROBE+ di Command

As explained previously, the **hello** sample's ROOT task prints a message on the console and then suspends itself with a **t_suspend()** call. Once ROOT is suspended, control passes to the IDLE task because no other tasks are in the system.

Because IDLE task executes indefinitely, it is best for control to return to the pROBE+ debugger when the ROOT task blocks. The pROBE+ debugger lets you define a break condition that is triggered when any task makes a specified pSOS+ service call. In this case, define a breakpoint on **t_suspend()** by entering:

```
pROBE+>db se t_suspend * *
```

which produces the console output shown in Figure 2-5.

```
SERVICE_BREAK___SERVICE________BY_TASK____________PARAMETER_____________
      0        T_SUSPEND   ****** ANY ******      Task: ANY
-------------------------------------------------------------------------
```

Figure 2-5 Output of a pSOS+ Service Break Command

Now resume execution with the **go** command. Figure 2-6 shows the resulting display.

```
Hello, world

pSOS  SVC Break.  Service: T_SUSPEND              Task:  'ROOT' -#00020000
                                                Running:  'ROOT' -#00020000
--------------------------------------------------------------------------------
SR=3000-tfSM.000...xnzvc USP=FFFFFFFF MSP=003FEF60 ISP=000C342C
VBR=00000000 DFC=0 SFC=0 CACR=80008000
DR=00000006 00000000 00000000 00000000  00000000 00000000 00000000 00000000
AR=003FEF88 00000000 00000000 00000000  00000000 00000000 003FEF60
PC=0002FFB0-0002FFB0  4E4B              TRAP    #11
```

Figure 2-6 Output From hello Application

Note that the message was printed, and the ROOT task is about to suspend itself.

The **hello** sample application can be restarted without downloading by entering the **gs** command:

```
pROBE+>gs
```

Figure 2-7 shows the resulting display.

```
pROBE+> gs

Kernel Event Break                                Running:  'ROOT' -#00020000
--------------------------------------------------------------------------------
pSOS Initialized Event
--------------------------------------------------------------------------------
SR=3000-tfSM.000...xnzvc USP=FFFFFFFF MSP=003FECCC ISP=0009735C
VBR=00000000 DFC=0 SFC=0 CACR=80008000
ITT0=00FFC000 ITT1=00000000 DTT0=0000C020 DTT1=00FFC040
DR=00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
AR=00000000 00000000 00000000 00000000  00000000 00000000 00000000 003FECCC
PC=0002A5E0-0002A5E0  4E56FFE8          LINK    A6,#-24
```

Figure 2-7 Output of pROBE+ gs Command

You can then continue by entering the following:

```
pROBE+>go
```

Figure 2-8 shows the resulting display.

```
Hello, world

pSOS  SVC Break.  Service: T_SUSPEND              Task:  'ROOT' -#00020000
                                                Running:  'ROOT' -#00020000
--------------------------------------------------------------------------------
SR=3000-tfSM.000...xnzvc USP=FFFFFFFF MSP=003FEF60 ISP=000C342C
VBR=00000000 DFC=0 SFC=0 CACR=80008000
DR=00000006 00000000 00000000 00000000  00000000 00000000 00000000 00000000
AR=003FEF88 00000000 00000000 00000000  00000000 00000000 003FEF60
PC=0002FFB0-0002FFB0  4E4B              TRAP    #11
```

Figure 2-8 Output From hello Application

Note that the break condition on **t_suspend()** was not cleared by **gs**, so you did not need to reenter it.

This concludes the demonstration of the **hello** sample program with the ROM pROBE+ debugger and downloaded pROBE+ debugger operating in stand-alone mode. Refer to the *pROBE+ User's Guide* for more information on using the pROBE+ debugger.

2.6 Printing When Using XRAY Over a Serial Channel

Whether you have a single-channel or multiple-channel board, read and follow the directions in this section.

Some of the boards supported by the pSOSystem software have only a single serial channel. If the communication between the XRAY and the pROBE+ debuggers takes place on this channel, then neither a pSOS+ serial driver nor an application can use the channel. To resolve this conflict, the pSOSystem software provides the service call **db_output()**. When called by the application code, **db_output()** sends text to the XRAY or pSOSystem I/O screen.

An example of how to use **db_output()** is located in the **hello** sample application.

To use **db_output()**, you first need to change a line in the **root.c** file. In **root.c**, look for the following:

```
#define OUTPUT_TO_DEBUGGER 0
```

The preceding line causes the ROOT task to use the standard pSOSystem device driver for output.

To configure the ROOT task to use **db_output()**, change this line as follows:

```
#define OUTPUT_TO_DEBUGGER 1
```

When a system has only a single serial channel, OUTPUT_TO_DEBUGGER must be 1.

2.6.1 Dedicating the Serial Channel to XRAY

After you change OUTPUT_TO_DEBUGGER, the **hello** sample does not print to the serial channel. Nevertheless, the pSOSystem build process still detects a **potential** channel conflict and fails because of it. The two lines in **sys_conf.h** that show the potential conflict are as follows:

```
#define SC_APP_CONSOLE   1
#define SC_PROBE_CONSOLE 1
```

These statements direct the pROBE+ debugger and application output to the same serial channel (channel 1), which is not allowed when the pROBE+ and XRAY debuggers communicate over a serial link. Changing one of these settings can eliminate the conflict. Because the application does not use serial output in the next example, change the definition of SC_APP_CONSOLE as follows:

```
#define SC_APP_CONSOLE NO
```

To see how **db_output()** works in an application, see the appropriate XRAY tutorial for your host. For UNIX hosts, see Chapter 4, "XRAY Debugger for pSOSystem Tutorial: Multiple Windows Version. For PC hosts, see Chapter 5, "XRAY Debugger for pSOSystem Tutorial: Viewport Version."

3 pSOSystem Tutorial for PC Hosts

This chapter is for pSOSystem users who are developing applications from PC hosts. The tutorial in this chapter helps you gain hands-on experience with pSOSystem by building, downloading, and executing an executable image. The executable image you build contains a sample application called **hello**. The **hello** sample resembles the Hello World program known to most programmers. It prints a short message and exits.

By following the examples in this chapter, you can learn how to perform the following tasks:

- Configure pSOSystem for execution in a particular hardware environment.
- Combine operating system and application code into an executable image.
- Transfer the executable image to the target system and execute it under the control of both the XRAY for pSOSystem and pROBE+ debuggers.

3.1 Recommended Software and Hardware

This chapter is a tutorial with explanations and commands that you can execute with the following hardware and software:

- You must have installed the Microtec MCC68K tools on the host and added it to your search path. Refer to the Microtec installation instructions for details.
- This tutorial assumes that you are using the pROBE+ debugger, although the pSOSystem software does not require the use of it.
- This chapter has many examples that use the XRAY for pSOSystem source-level cross-debugger. (The pSOSystem software does not require XRAY.) If you have purchased XRAY, install it and add it to your search path before you proceed. Refer to the installation instructions in the *XRAY Debugger for pSOSystem User's Guide* for details.
- Many examples assume a terminal emulation program is running on the host. Although the pSOSystem software does not require an emulator, you will not be able to perform all the examples in this tutorial without an emulator.

The examples demonstrate execution on target hardware. The pSOSystem software includes fully operable board-support packages for the supported CPU boards described in Appendix A, "Board-Specific Information." This chapter is based on the use of supported boards for the following reasons:

- Developing a pSOSystem board-support package for an unsupported board requires the development and debugging of various board-dependent drivers, interrupt handlers, and so on. As this is nontrivial, familiarity with pSOSystem software *before* you build a custom BSP is helpful. Therefore, you should first do the examples in this chapter and, if necessary, create a pSOSystem BSP for the target hardware.
- The pSOSystem software also includes Boot ROMs for all supported boards. Although normal pSOSystem operation does not require Boot ROMs, the examples in this chapter are based on their use.

If the target hardware is not a supported board, you should still try to obtain a supported board for temporary use. The supported boards are widely available, but if you cannot obtain one, you should still read this section and then proceed to Chapter 9, "Understanding and Developing Board-Support Packages."

3.2 Creating a Working Directory

Before you begin this tutorial, create your own copy of the **hello** sample application code in the working directory. The easiest way to do this is with a copy command that copies an entire directory tree, as the following example shows. If you have already set up the PSS_ROOT environment variable, enter the following command:

xcopy \pss\apps\hello *workdir* /s/e

where *workdir* is the name of a working directory that does not already exist. Presumably, **PSS_ROOT** is **\pss** because MS-DOS does not allow expansion of environment variables when entering commands.

3.3 hello Sample Application

Before you run the **hello** sample, take a few minutes to examine its source code. A listing of the working directory should show the following:

makefile drv_conf.c sys_conf.h
README root.c

The **root.c** file contains the code for the **hello** sample's ROOT task. This code prints a message to either the console or XRAY I/O screen and then suspends itself. Self-suspension causes the IDLE task to run indefinitely.

The source code in **root.c** illustrates some of the basic elements of building pSOSystem application code. As you examine the contents of **root.c**, note the following:

1. First, **root.c** includes the pSOSystem configuration file **sys_conf.h**. For this tutorial, **sys_conf.h** defines the device numbers of the serial driver and periodic tick timer. The **sys_conf.h** file is introduced in Section 1.2.1, "Configuration Files."

2. Next, **root.c** includes a standard pSOSystem header file, **psos.h**. Standard pSOSystem header files reside in the **include** directory in the pSOSystem root directory. **psos.h** defines all pSOS+ kernel-related constants such as error codes and options. All source files that call pSOS+ services should include **psos.h**. Similar header files exist for other components.

3. Next, **root.c** includes the **probe.h** file. This header file contains the declaration for the external routine **db_output()**. This routine is used to print messages to the XRAY for pSOSystem I/O screen. Source code for **db_output()** comes in a standard file shipped with the debugger. For convenience, it also exists in the **xp_out.s** file in the **$PSS_ROOT/sys/os** directory.

4. The next statement defines the OUTPUT_TO_DEBUGGER constant. This constant defines the device to which the **hello** sample directs its output. If OUTPUT_TO_DEBUGGER is 0, output goes to the pSOSystem standard console device. Otherwise, output goes to the XRAY Debugger for pSOSystem I/O screen. The purpose of this switch is explained later.

5. The ROOT task initializes the tick timer. Although this can be done at any time by any task, normally the ROOT task does it. This device provides periodic interrupts, and its interrupt service routine (ISR) announces these ticks to the pSOS+ kernel so the kernel can perform time-related functions.

6. The **#if** statement controlled by OUTPUT_TO_DEBUGGER implements the output switch described above. If printing goes to the console, **de_write()** is called to print the message. Otherwise, **db_output()** is called.

7. Finally, the task calls **t_suspend()** to suspend itself.

3.4 Building an Executable Image

Before you build an executable image, edit **makefile** to make sure the definition of PSS_BSP reflects the target board. (Remove the comment mark if one is present. This line defining PSS_BSP becomes the first non-commented line in the **makefile**.) For example, if the target system is a Motorola MVME167, **makefile** should contain the following:

```
PSS_BSP=$(PSS_ROOT)/bsps/m167
```

If you have the pSOS+m multiprocessor kernel, the **sys_conf.h** file also requires an edit. If the kernel is the pSOS+m kernel, search **sys_conf.h** for the following lines:

```
#define SC_PSOS        YES
#define SC_PSOSM       NO
```

Change these lines to show YES for the pSOS+m kernel, **SC_PSOSM**, and NO for the pSOS+ kernel, **SC_PSOS**.

Proceed to building an executable image. Enter the following:

```
make ram.hex
```

The preceding action compiles, assembles, and links all the files required to build an executable image. The executable image contains the **hello** sample application and the pSOSystem software for the selected hardware.

After the build completes, verify that the two output files **ram.map** and **ram.hex** were created. For a description of these files, see Chapter 9, "Understanding and Developing Board-Support Packages."

If any errors occurred during the build process, first check to see if you performed the copy commands correctly. If you did, then the installed Microtec tools probably contain a problem. If you cannot locate the problem, contact Integrated Systems Technical Support.

3.5 Downloading the Executable Image

To run the executable image created in the previous section, you must first download it to the target system. This section takes you through the download sequence.

The examples in this chapter are based on the use of the pSOSystem Boot ROMs, although the pSOSystem does not require them. The Boot ROMs rely on the pROBE+ debugger for downloading and other services. Boot ROMs can operate the pROBE+ debugger in the following modes:

- Stand-alone
- With XRAY over a serial channel
- With XRAY over a network (not available on all boards)

This chapter demonstrates all three modes. Integrated Systems recommends that you perform all the exercises, if possible, or at least read all the sections.

3.5.1 Starting the Boot ROMs

Appendix A, "Board-Specific Information," contains installation instructions for the Boot ROMs.

The target board through an RS-232 connection must be connected to an ASCII terminal or a terminal emulation program running on the host. A terminal emulation program is preferable because, with an emulator, you can use the same channel to download to the target.

If a target board has more than one RS-232 port, refer to Appendix A to determine which port to use.

Set the terminal or emulator characteristics to:

- 9600 baud
- 8-bit data
- 1 stop bit
- no parity

When you power up or reset the board, the terminal should display a pSOSystem startup message similar to the one in Figure 3-1.

```
pSOSystem V2.2.0
Copyright (c) 1991 - 1996 Integrated Systems, Inc.
------------------------------------------------------------------------
START-UP MODE:
  Boot into pROBE+ stand-alone mode
NETWORK INTERFACE PARAMETERS:
  LAN interface is disabled
  Shared memory interface is disabled
MULTIPROCESSING PARAMETERS:
  This board is currently configured as a single processor system
HARDWARE PARAMETERS:
  Serial channels will use a baud rate of 9600
  This board's memory will reside at 0x1000000 on the VME bus
  After board is reset, start-up code will wait 60 seconds
------------------------------------------------------------------------
To change any of this, press any key within 60 seconds
```

Figure 3-1 pSOSystem Startup Message

If a message like the one in Figure 3-1 does not appear, check for the following:

- The terminal baud rate, stop bits, and so on, must be correct.
- A null modem between the target CPU and the terminal may be required.
- Make sure the ROMs are in the correct sockets and have no bent pins.
- Make sure the jumper and switch settings are correct.
- If the ROMs still do not operate, contact technical support.

The preceding message shows the default ROM configuration parameters. After the startup message appears, the ROMs wait 60 seconds for keyboard input. If you do not enter a character within this time, the ROMs operate with the configuration shown.

The preceding display shows that the startup mode for the pROBE+ debugger is stand-alone, and the startup code has a wait period of 60 seconds. Use this mode and the 60-second wait for now.

After the wait period has elapsed, a message similar to the following should appear to indicate that the ROM pROBE+ debugger is waiting for input:

```
pROBE+ V2.1.1 (68040)
COPYRIGHT 1990 - 1996, INTEGRATED SYSTEMS, INC.
ALL RIGHTS RESERVED
pROBE+>
```

The ROM pROBE+ can now be used to download the S-record file that contains the executable image (**ram.hex**).

3.5.2 Downloading the Executable Image

To download the S-record file that contains the executable image, enter the pROBE+ **dl** command, as follows:

```
pROBE+>dl
```

pROBE+ is now waiting for you to download **ram.hex**. Enter the commands necessary to cause the terminal emulation program to transmit **ram.hex** over the serial channel. For a terminal emulation program, for example, you would do the following:

1. Press [Alt-s]
2. Specify anpROBE ASCII transfer protocol
3. Send the **ram.hex** file

If the download is successful, the pROBE+ debugger should display a message similar to the following (the number of records may vary):

```
9840 records read
pROBE+>
```

NOTE: The download takes about five minutes.

Often, the terminal emulation program misses part or all of this message. If so, the results of the most recent download can be obtained by entering the pROBE+ **dl** command again with the **prev** option, as follows:

```
pROBE+>dl prev
9840 records read
```

3.5.3 Starting the Executable Image

The pROBE+ **go** command is used to begin (or continue) execution. When you enter **go** with the start address, the executable image is started by

passing control to its start address. To determine the start address for individual boards, see Appendix A, "Board-Specific Information." (An executable image is started by passing control to the image"s *load address* + 8.)

Once you have the board's start address, enter the command with the start address, as in the following example:

```
pROBE+>go 28008
```

A message similar to the following should appear:

```
pROBE+ V2.1.1 (68040)
COPYRIGHT 1990 - 1996, INTEGRATED SYSTEMS, INC.
ALL RIGHTS RESERVED

pROBE+>
```

This is the point at which the **hello** sample application can run. Although this message may look similar to the one displayed when the Boot ROMs started, the message comes from the downloaded pROBE+ debugger rather than the ROM pROBE+ debugger. The ROM pROBE+ debugger is no longer functioning and can regain control only if you reset the board.

Like the ROM pROBE+ debugger, the downloaded pROBE+ debugger is running in stand-alone mode. If the parameter SE_DEBUG_MODE is set to STORAGE in **sys_conf.h**, the downloaded pROBE+ debugger operates in the same mode as the ROM pROBE+ debugger. This is the default and is usually the most convenient approach, although ROM and downloaded pROBE+ debuggers do not need to operate in the same mode. If SE_DEBUG_MODE is set to either DBG_SA, DBG_XS, or DBG_XN, the downloaded pROBE+ debugger operates in the mode specified by the SE_DEBUG_MODE setting regardless of the ROM pROBE+ operating mode.

3.5.4 Executing the hello Sample Application

The pROBE+ **gs** command passes control to pSOS+ startup. pSOS+ startup initializes the pSOS+ kernel and any other operating system components and then returns control to the pROBE+ debugger. Enter the **gs** command now:

```
pROBE+>gs
```

You should see output similar to that in Figure 3-2 on page 3-10.

```
pROBE+> gs

pSOS Initialized.  Running:  'ROOT' -#00020000
--------------------------------------------------------------------------
SR=3000-tfSM.000...xnzvc USP=FFFFFFFF MSP=003FEF94 ISP=000C342C
VBR=00000000 DFC=0 SFC=0 CACR=80008000
DR=00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
AR=00000000 00000000 00000000 00000000  00000000 00000000 00000000
PC=00029548-00029548  4E56FFE8          LINK    A6,#-24
```

Figure 3-2 Output of pROBE+ gs Command

The pSOS+ kernel is now initialized, and execution of the ROOT task is pending. Because the pSOS+ kernel has just been initialized, only two tasks are in the system. You can verify this by entering the pROBE+ **qt** command:

pROBE+>qt

Figure 3-3 shows the subsequent display:

```
pROBE+> gt

  Name      TID    Prio Mode Status    Susp?     Parameters             Ticks
-------------------------------------------------------------------------------
 'IDLE' -#00010000  00  2000   Ready
 'ROOT' -#00020000  E6  2000 Running
```

Figure 3-3 Output of the qt Command

You can use the pROBE+ **di** command to examine the code that is ready to execute, as follows (the code actually displayed may be different):

pROBE+>di

Entering **di** causes the disassembled memory contents to appear as in Figure 3-4 (the actual displayed code may be different).

```
pROBE+> di

   00029548-00029548  4E56FFE8          LINK    A6,#-24
   0002954C-0002954C  2F02              MOVE.L  D2,-(A7)
   0002954E-0002954E  486EFFFC          PEA.L   -4(A6)
   00029552-00029552  486EFFF8          PEA.L   -8(A6)
   00029556-00029556  42A7              CLR.L   -(A7)
   00029558-00029558  2F3C00020000      MOVE.L  #131072,-(A7)
   0002955E-0002955E  4EB900030714      JSR.L   ($00030714).L
   00029564-00029564  486EFFFC          PEA.L   -4(A6)
```

Figure 3-4 Output of the pROBE+ di Command

As explained previously, the **hello** sample's ROOT task prints a message on the console and then suspends itself with a **t_suspend()** call. Once ROOT is suspended, control passes to the IDLE task because no other tasks are in the system.

Because the IDLE task executes indefinitely, it is best for control to return to the pROBE+ debugger when the ROOT task blocks. The pROBE+ debugger lets you define a break condition that is triggered when any task makes a specified pSOS+ service call. In this case, define a breakpoint on **t_suspend()** by entering:

```
pROBE+>db se t_suspend * *
```

Figure 3-5 shows the subsequent display.

```
SERVICE_BREAK____SERVICE________BY_TASK____________PARAMETER_______________
      0        T_SUSPEND   ****** ANY ******      Task: ANY
--------------------------------------------------------------------------------
```

Figure 3-5 Defined Breakpoint

Now resume execution with the **go** command, as follows:

```
pROBE+>go
```

Figure 3-6 shows the resulting display.

```
Hello, world

pSOS  SVC Break.  Service: T_SUSPEND              Task:  'ROOT' -#00020000
                                                Running:  'ROOT' -#00020000
------------------------------------------------------------------------------
SR=3000-tfSM.000...xnzvc USP=FFFFFFFF MSP=003FEF60 ISP=000C342C
VBR=00000000 DFC=0 SFC=0 CACR=80008000
DR=00000006 00000000 00000000 00000000  00000000 00000000 00000000 00000000
AR=003FEF88 00000000 00000000 00000000  00000000 00000000 003FEF60
PC=0002FFB0-0002FFB0  4E4B              TRAP    #11
```

Figure 3-6 hello Output

Note that the message was printed, and ROOT is about to suspend itself.

The **hello** sample application can be restarted without downloading by entering the **gs** command:

pROBE+>gs

Figure 3-7 shows the resulting display.

```
pROBE+> gs

pSOS Initialized.  Running:  'ROOT' -#00020000
------------------------------------------------------------------------------
SR=3000-tfSM.000...xnzvc USP=FFFFFFFF MSP=003FEF94 ISP=000C342C
VBR=00000000 DFC=0 SFC=0 CACR=80008000
DR=00000000 00000000 00000000 00000000  00000000 00000000 00000000 00000000
AR=00000000 00000000 00000000 00000000  00000000 00000000 00000000
PC=00029548-00029548  4E56FFE8          LINK    A6,#-24
```

Figure 3-7 pSOS+ Status and CPU Register Contents

You can then continue by entering the following:

pROBE+>go

Figure 3-8 shows the resulting display.

```
Hello, world

pSOS  SVC Break.  Service: T_SUSPEND              Task:  'ROOT' -#00020000
                                                Running:  'ROOT' -#00020000
--------------------------------------------------------------------------------
SR=3000-tfSM.000...xnzvc USP=FFFFFFFF MSP=003FEF60 ISP=000C342C
VBR=00000000 DFC=0 SFC=0 CACR=80008000
DR=00000006 00000000 00000000 00000000  00000000 00000000 00000000 00000000
AR=003FEF88 00000000 00000000 00000000  00000000 00000000 003FEF60
PC=0002FFB0-0002FFB0  4E4B              TRAP    #11
```

Figure 3-8 hello Output

Note that the break condition on **t_suspend()** was not cleared by **gs**, so you did not need to reenter it.

This concludes the demonstration of the **hello** sample with the ROM pROBE+ debugger and downloaded pROBE+ debugger operating in stand-alone mode. Refer to the *pROBE+ User's Guide* for more information on using the pROBE+ debugger.

3.6 Printing When Using XRAY Over a Serial Channel

Whether you have a single-channel or multiple-channel board, read and follow the directions in this section.

Some of the boards supported by the pSOSystem software have only a single serial channel. If the communication between the XRAY and the pROBE+ debuggers takes place on this channel, then neither a pSOS+ serial driver nor an application can use the channel. To resolve this conflict, the pSOSystem software provides the service call **db_output()**. When called by the application code, **db_output()** sends text to the XRAY or pSOSystem I/O screen. An example of how to use **db_output()** is in the **hello** sample application.

To use **db_output()**, you first need to change a line in the **root.c** file. In **root.c**, look for the following:

```
#define OUTPUT_TO_DEBUGGER 0
```

The preceding line causes the ROOT task to use the standard pSOSystem device driver for output.

To configure the ROOT task to use **db_output()**, change this line as follows:

```
#define OUTPUT_TO_DEBUGGER 1
```

When a system has only a single serial channel, OUTPUT_TO_DEBUGGER must be 1.

3.6.1 Dedicating the Serial Channel to the XRAY Debugger

After you change OUTPUT_TO_DEBUGGER, the **hello** sample does not print to the serial channel. Nevertheless, the pSOSystem build process still detects a **potential** channel conflict and fails because of it. The two lines in **sys_conf.h** that show the potential conflict are as follows:

```
#define SC_APP_CONSOLE   1
#define SC_PROBE_CONSOLE 1
```

These statements direct the pROBE+ debugger and application output to the same serial channel (channel 1), which is not allowed when the pROBE+ debugger and XRAY debugger communicate over a serial link. Changing one of these settings can eliminate the conflict. Because the application does not use serial output in the next example, change the definition of SC_APP_CONSOLE as follows:

```
#define SC_APP_CONSOLE NO
```

To see how **db_output()** works in an application, see the appropriate XRAY tutorial for your host. For UNIX hosts, see Chapter 4, "XRAY Debugger for pSOSystem Tutorial: Multiple Windows Version." For PC hosts, see Chapter 5, "XRAY Debugger for pSOSystem Tutorial: Viewport Version."

3.7 Using XRAY Debugger for pSOSystem Over Ethernet

This section describes how to run the **hello** sample by using the XRAY Debugger for pSOSystem over an Ethernet connection.

The exercises in this section require the pNA+ network manager in the system. The use of XRAY Debugger for pSOSystem over Ethernet requires extra installation steps on some hosts. The *XRAY Debugger for pSOSystem User's Guide* describes the extra steps. You must take these steps before you proceed.

3.7.1 Connecting to the Network

To use Ethernet, the target and host must connect over Ethernet. This is usually done by adding the target system to the office network but can also be done with a private network.

3.7.2 Adding pNA+ to the Downloaded System

The file **sys_conf.h** now requires editing. First, because communication takes place over Ethernet, the operating system must include the pNA+. network manager. To include the pNA+ network manager in the pSOSystem environment, change the following statement:

```
#define SC_PNA     NO
```

to

```
#define SC_PNA     YES
```

Other changes can be made to **sys_conf.h**. You can change the pROBE+ operating mode to *XRAY Over a Network*, add an Ethernet interface, and assign an IP address to the Ethernet interface. However, as long as SC_SD_PARAMETERS is set to STORAGE, the downloaded system uses the same settings as the pSOSystem ROMs, so don't change these parameters in **sys_conf.h**. For information on how to changes to the ROM parameters, see Chapter 7, "Configuration and Startup."

3.7.3 XRAY Debugger for pSOSystem Output Over Ethernet

Recall that when the pROBE+ debugger runs in stand-alone mode, the **hello** sample output goes to the standard pSOSystem console. When you connected the XRAY debugger to the serial channel, you had to change the **hello** sample source code to use the **db_output()** routine and did so by changing the definition of OUTPUT_TO_DEBUGGER in **root.c**.

Now that XRAY is to connect over Ethernet for this part of the tutorial, either method of output can be used. If you want to use the pSOSystem console driver, connect a terminal to the serial channel to see the output.

4 XRAY Debugger for pSOSystem Tutorial: Multiple Windows Version

This chapter provides a tutorial on the use of the multiple window version of XRAY Debugger for pSOSystem, a multitasking debugger from Integrated Systems. The tutorial shows how to control and monitor multitasking applications from a SunOS or Solaris host. If you are using an HP, RS6000, or PC, see Chapter 5, "XRAY Debugger for pSOSystem Tutorial: Viewport Version."

XRAY Debugger for pSOSystem has the following features:

- A graphical user interface with multiple windows
- Automatic tracking of program execution through source code
- Traces and breaks on high-level language statements
- Breaks on task state changes and operating system calls
- Monitoring of language variables and system-level objects such as tasks, queues, and semaphores
- Profiling for performance tuning and analysis
- Full-featured C++ language support
- Integration with the XRAY MasterWorks Development Environment

- The ability to debug optimized code
- System Debug Mode support

In addition to supporting System Debug Mode, some versions of XRAY also support Task Debug Mode. This tutorial illustrates the use of System Debug Mode only. For more information on Task Debug Mode, refer to the *XRAY Debugger for pSOSystem User's Guide*.

4.1 xraydemo Sample Application

The **apps** directory contains the subdirectory **xraydemo**, which contains a sample application called **xraydemo**. The tutorial in this chapter uses this sample application.

The **xraydemo** sample application is a C program that contains the code for eight tasks. The ROOT task, which automatically receives control from the pSOS+ kernel after startup, creates the seven other tasks and then blocks. The seven other tasks function as follows:

Task	Function
'MEM1'	Requests memory segments of various sizes.
'MEM2'	Receives memory segments from MEM1 and frees them.
'IO1•'	Reads a block from the RAM disk device.
'IO2•'	Writes a block to the RAM disk device.
'SRCE'	Sends messages to the 'SS••' Queue.
'SINK'	Consumes messages from the 'SS••' Queue.
'MSG•'	Consumes messages from Queue CNSL and sends the contents to the XRAY terminal.

NOTE: The 'IO1•' and 'IO2•' tasks consist of 4-character names where the fourth character is a required space. Similarly, the 'SS••' queue name includes two required spaces; the breve symbol (•) indicates a required space.

A task name entered with a command sometimes requires single quotes around the task name. The examples in this manual show when the quotes are required.

NOTE: Before invoking the XRAY debugger, either serially or over Ethernet, you must set several environment variables. For a detailed description of XRAY environment variables, see the *XRAY Debugger for pSOSystem User's Guide* or your Microtec Research documentation.

4.2 Creating an Executable Image

This section guides you through the steps needed to build an executable image that contains an operating system for the target board and the **xraydemo** sample application. It is assumed you have performed the steps in Chapter 2, "pSOSystem Tutorial for Workstation Hosts," or Chapter 3, "pSOSystem Tutorial for PC Hosts," so only a brief description appears here.

First, build a new working directory, named **xd**, containing **xraydemo**, and then switch to that directory. For example, you can enter the appropriate command sequence for your environment.

```
mkdir xd
cp -r $PSS_ROOT/apps/xraydemo/* ./xd
cd xd
```

NOTE: The xraydemo application prints by using db_output(). Therefore, the constant OUTPUT_TO_DEBUGGER used in the hello application does not exist in xraydemo. Also, the value of SC_APP_CONSOLE in sys_conf.h is not relevant.

4.2.1 Customizing The Operating System

You may need to customize the operating system to work with XRAY Debugger for pSOSystem. Customizing the operating system requires one or more of the following steps:

- Set PSS_BSP in **Makefile** to an appropriate value. A PSS_BSP value has the form **$(PSS_ROOT)/bsps/*directory_name***, where *directory_name* is the name of a BSP.
- If you use XRAY Debugger for pSOSystem over a serial channel, follow the directions in Section 4.3, "Using XRAY Over a Serial Channel."

- If you use the XRAY debugger over Ethernet, set SC_PNA in **sys_conf.h** to YES. For more information, see Section 4.4, "Using XRAY Debugger for pSOSystem Over Ethernet."
- If you use the pSOS+ kernel instead of the pSOS+m kernel, set the **sys_conf.h** parameters as follows: SC_PSOS to NO and SC_PSOSM to YES.

4.2.2 Building The Executable Image

After you make the needed changes, build the executable image with the following command:

make ram.x

As noted above, you can download the **ram.x** file to the target system using a serial channel (as described in section 4.3) or an Ethernet port (as described in section 4.4). Follow the directions in the section appropriate for your environment.

4.3 Using XRAY Over a Serial Channel

This section describes how to run the **xraydemo** sample program by using XRAY Debugger for pSOSystem connected to the target over a serial channel as shown in Figure 4-1. In this section, both the ROM and downloaded pROBE+ debugger operate in remote mode communicating with XRAY Debugger for pSOSystem over a serial channel. The operating modes of the ROM pROBE+ debugger and downloaded pROBE+ debugger do not need to be the same, but in most cases they are.

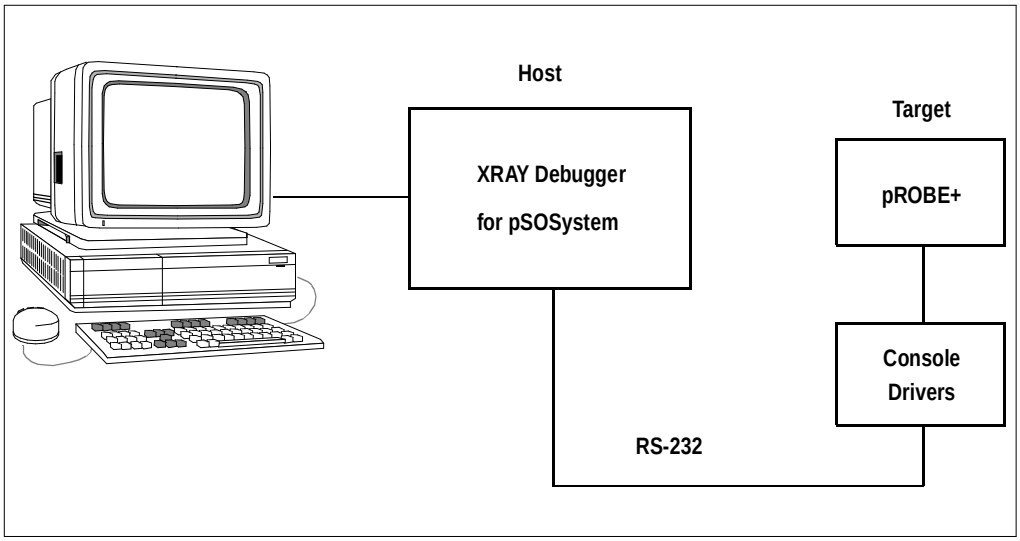


Figure 4-1 Using XRAY Debugger for pSOSystem Over a Serial Channel

4.3.1 Reconfiguring the ROMs for a Serial Channel

To reconfigure the ROMs to communicate with XRAY Debugger for pSOSystem, reset the target CPU board and enter a character before the specified time elapses. When you power up or reset your board, a message similar to the following appears as part of the pSOSystem startup display (as shown in Figure 2-1 on page 2-6):

```
To change any of this, press any key within 5 seconds
```

Press any key within 5 seconds, and a message similar to the one in Figure 4-2 appears.

```
For each of the following questions, you can press <Return> to select the
value shown in braces, or you can enter a new value.

How should the board boot?
  1. pROBE+ stand-alone mode
  2. pROBE+ waiting for host debugger via serial connection
  3. pROBE+ waiting for host debugger via a network connection
  4. Run the TFTP Bootloader

Which one do you want? [1]
```

Figure 4-2 Reconfiguring the ROMs

To use XRAY Debugger for pSOSystem over a serial channel, enter a **2**.

You do not need to change the remaining parameters, so you can bypass each of the remaining questions by pressing [Return] until the following question appears:

How long (in seconds) should CPU delay before starting up? [60]

In most cases, 60 seconds is unnecessarily long. Enter a more suitable value, such as 3. When you power up or reset the board, the terminal displays a pSOSystem startup message similar to the one in Figure 4-3.

```
------------------------------------------------------------------------
START-UP MODE:
  Boot into pROBE+ and wait for host debugger via a serial connection
NETWORK INTERFACE PARAMETERS:
  LAN interface is disabled
  Shared memory interface is disabled
MULTIPROCESSING PARAMETERS:
  This board is currently configured as a single processor system
HARDWARE PARAMETERS:
  Serial channels will use a baud rate of 9600
  This board's memory will reside at 0x1000000 on the VME bus
  Processor Type :: MC68040 operating at 25 Mhz
  RAM configuration :: Parity DRAM 4 Mb
                    :: SRAM 128 Kb
  After board is reset, start-up code will wait 3 seconds
------------------------------------------------------------------------
(M)odify any of this or (C)ontinue? [M]
```

Figure 4-3 pSOSystem Startup Message for a Serial Channel

Press the letter c followed by [Return] to accept the changes. pROBE+ will indicate that it is ready to talk to the debugger over the serial port.

If a terminal emulator is running, exit it now because the next section requires the use of XRAY Debugger for pSOSystem over a serial channel.

NOTE: Resetting a board does not cause the ROMs to revert to their default configuration. Furthermore, because configuration data is stored in battery-backed or nonvolatile RAM when available, powering down a board causes a return to default values only if the board lacks nonvolatile storage capability.

4.3.2 Invoking the XRAY Debugger for a Serial Channel

Because XRAY Debugger for pSOSystem is a source-level debugger, it must locate the source code for the object code it is debugging. It first searches the current directory and then searches as directed by the environment variable **XRAY**. If you set **XRAY** to **@**, XRAY Debugger for pSOSystem uses the source file pathnames embedded in the executable image. Set the **XRAY** environment variable for use with **csh** with the following command:

```
setenv XRAY @
```

You can now invoke XRAY Debugger for pSOSystem on the host. To invoke XRAY for use over a serial channel connecting to a 68K target, use the following syntax:

```
xp68k -e r<dev> ram.x &
```

where ***dev*** specifies the serial device. For example:

```
xp68k -e rttyb ram.x &
```

For Sun-4 hosts running the SunOS or Solaris operating system, valid device names are **ttya** and **ttyb** (or similar serial device names).

For other ways of invoking XRAY Debugger for pSOSystem, refer to the *XRAY Debugger for pSOSystem User's Guide*.

4.3.3 Changing the Baud Rate for a Serial Channel

Unless you are using XRAY Debugger for pSOSystem with Ethernet, you may want to increase the baud rate of the serial channel. You can increase it to either 19200 or 38400 baud if the host and target both

support the higher rate. The baud rate must be changed in several places:

- The Boot ROMs should be reconfigured for the higher rate.
- The XRAY debugger uses a default baud rate of 9600. Therefore, when you invoke XRAY Debugger for pSOSystem, you must explicitly specify a higher rate. You do this by appending the baud rate to the name of the serial device, separated by a comma. For example, the following command changes the baud rate to 19200:

  xp68k -e rttya,19200 ram.x

- The terminal or terminal emulation program must be reconfigured to operate at the higher rate.

This concludes the description of XRAY Debugger for pSOSystem over a serial channel.

4.4 Using XRAY Debugger for pSOSystem Over Ethernet

This section describes how to run the **xraydemo** sample program by using XRAY Debugger for pSOSystem over an Ethernet connection as shown in Figure 4-4.

The exercises in this section require the pNA+ network manager in the system. The use of the XRAY debugger over Ethernet requires extra installation steps on some hosts (as described in the following sections).

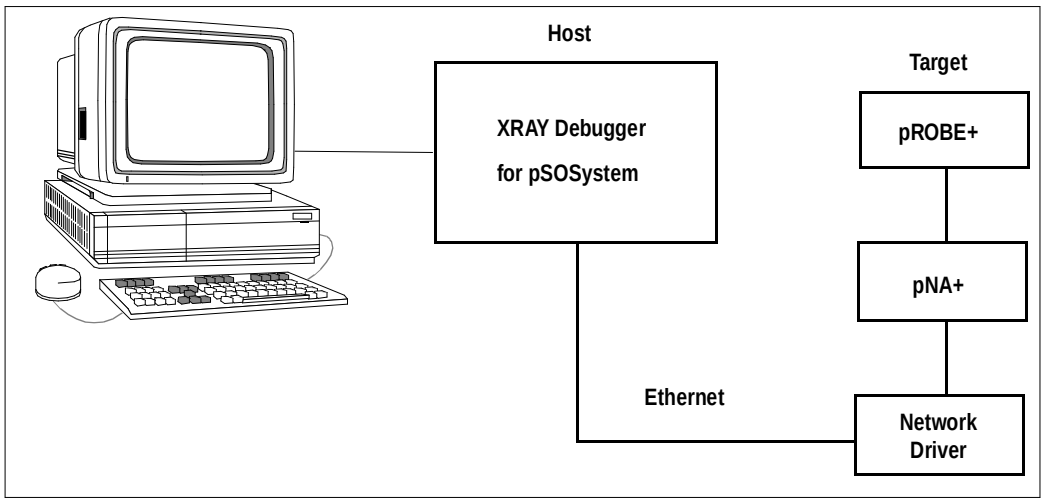


Figure 4-4 Using XRAY Debugger for pSOSystem Over Ethernet

4.4.1 Connecting to the Network

To use Ethernet, the target and host must connect over Ethernet. This is usually done by adding the target system to the office network but can also be done with a private network.

4.4.2 Adding pNA+ to the Downloaded System

Because communication takes place over Ethernet, the operating system must include the pNA+ network manager. To include the pNA+ network

manager in the pSOSystem environment, change the following statement in your **sys_conf.h** file:

```
#define SC_PNA     NO
```

to

```
#define SC_PNA     YES
```

You can make other changes to **sys_conf.h** such as changing the pROBE+ operating mode to XRAY Over a Network, adding an Ethernet interface, and assigning an IP address to the Ethernet interface. However, as long as the SC_SD_PARAMETERS value is set to STORAGE, the downloaded system uses the same settings as the pSOSystem ROMs, so you don't need to change these parameters in **sys_conf.h**.

4.4.3 Reconfiguring the ROMs for Ethernet

To reconfigure the ROMs for communication with XRAY over Ethernet, reset the target CPU board and enter a character at the pSOSystem startup screen before the specified time elapses, as indicated in the following prompt:

```
To change any of this, press any key within 5 seconds
```

Press any key within the time limit, and a display similar to the one shown in Figure 4-5 appears.

```
For each of the following questions, you can press <Return> to select the
value shown in braces, or you can enter a new value.

How should the board boot?
  1. pROBE+ stand-alone mode
  2. pROBE+ waiting for host debugger via serial connection
  3. pROBE+ waiting for host debugger via a network connection
  4. Run the TFTP Bootloader

Which one do you want? [1]
```

Figure 4-5 Reconfiguring the ROMs for Ethernet

To use XRAY with pROBE+ over Ethernet, enter a **3**.

NOTE: The steps described in this section are done through an ASCII terminal, regardless of the subsequent operating mode of the ROMs.

By entering a **3**, you also get the following network questions:

```
NETWORK INTERFACE PARAMETERS:
Do you want a LAN interface?  [N] y
```

Enter a **y**. (**n** applies to multiprocessor systems where some boards communicate through shared memory rather than Ethernet.) The next prompt displays your IP address as follows:

```
This board's LAN IP address? (0.0.0.0 = RARP)?
[199.99.99.99]
```

Enter a **y** to accept the displayed IP address or correct the displayed IP address and then enter a **y**. Enter the IP address of the CPU board using standard dot notation for internet addresses (four numeric fields, each separated by a period). Answer **n** to the next two questions, because they enable features that are useful only in multiprocessor target systems:

```
Use a subnet mask for the LAN interface? [N]
Do you want a shared memory network interface? [N]
```

The next question is as follows:

```
Should there be a default gateway for packet routing? [N]
```

The usual answer is **n** unless the target and host systems are on different networks connected through a gateway. If you are not sure how to answer this, ask your system administrator.

Accept the indicated default values for the remaining questions as shown below:

```
MULTIPROCESSING PARAMETERS:

Do you want to configure a multiprocessing pSOS+m system?
[N]

HARDWARE PARAMETERS:

Baud rate for serial channels [9600]

Bus address of this board’s dual-ported memory [4000000]

How long (in seconds) should CPU delay before starting up?
[5]
```

When you power up or reset the board, the terminal displays a pSOSystem startup message similar to the one in Figure 4-6.

```
START-UP MODE:
  Boot into pROBE+ and wait for host debugger via a network connection
NETWORK INTERFACE PARAMETERS:
  IP address on LAN is 199.99.99.99
  Shared memory interface is disabled
MULTIPROCESSING PARAMETERS:
  This board is currently configured as a single processor system
HARDWARE PARAMETERS:
  Serial channels will use a baud rate of 9600
  This board's memory will reside at 0x1000000 on the VME bus
  After board is reset, start-up code will wait 3 seconds
------------------------------------------------------------------------
(M)odify any of this or (C)ontinue? [M]
```

Figure 4-6 pSOSystem Startup Message for Ethernet

Press any key within 5 seconds to change any of the settings shown in Figure 4-6. No pROBE+ prompt appears, because pROBE+ is now waiting for a connection from XRAY over the network.

4.4.4 Invoking the XRAY Debugger Over Ethernet

Invoking the XRAY debugger for use over Ethernet is similar to invoking the XRAY debugger over a serial channel, as described in section 4.3.2, except that, following the **-e** option, you specify a **p** rather than an **r** and then enter the host IP address. For example, if the target system IP address is 199.99.99.99 as in Figure 4-6, the form of the command is as follows:

```
xp68k -e p199.99.99.99 &
```

where **68k** indicates a 68K target.

4.5 Initializing XRAY Debugger for pSOSystem

A sign-on screen appears while XRAY Debugger for pSOSystem is initializing. On some hosts you can define an alias name to represent the IP address of the target. You can then type this name in place of the actual IP address.

First, the MasterWorks Control Panel appears as shown in Figure 4-7 if XRAY is not already executing. The icons that appear in the MasterWorks Control Panel depend on the tools you have installed.

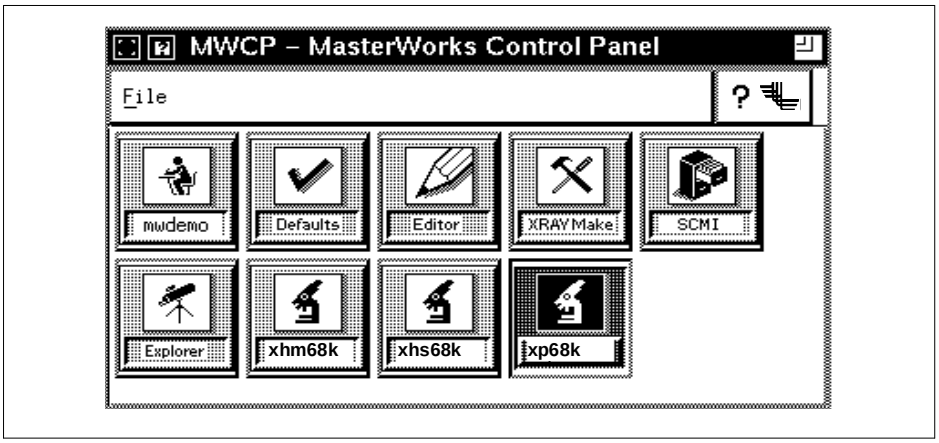


Figure 4-7 MasterWorks Control Panel

You can invoke the XRAY debugger by double-clicking on the **xp68k** icon (on the bottom right). For additional information about XRAY, see the *XRAY Debugger for pSOSystem User's Guide*.

When you invoke XRAY, the Code and Command windows appear as shown in Figure 4-8 on page 4-14.

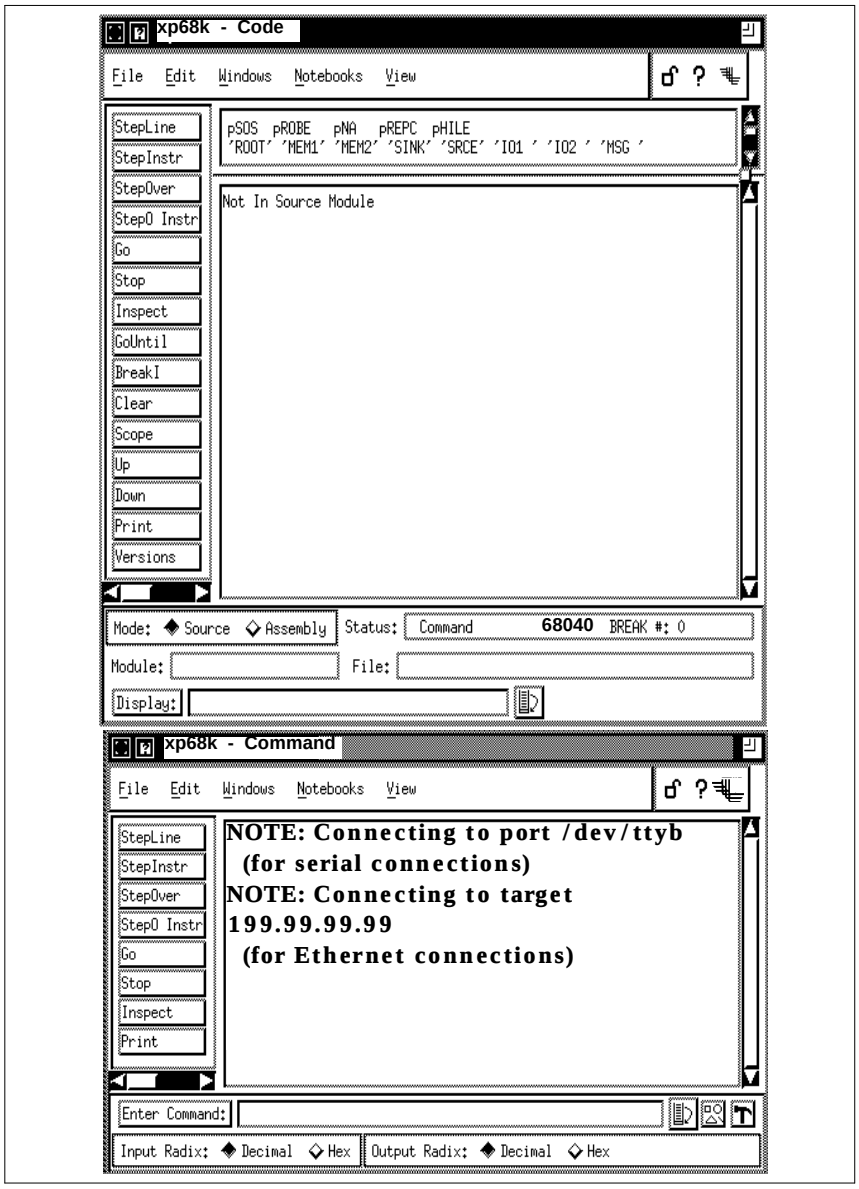


Figure 4-8 Code and Command Windows

To download an application, follow these steps:

1. Select File in the menu bar.
2. Select Load in the File menu. This opens the **Command Dialog** window, shown in Figure 4-9.

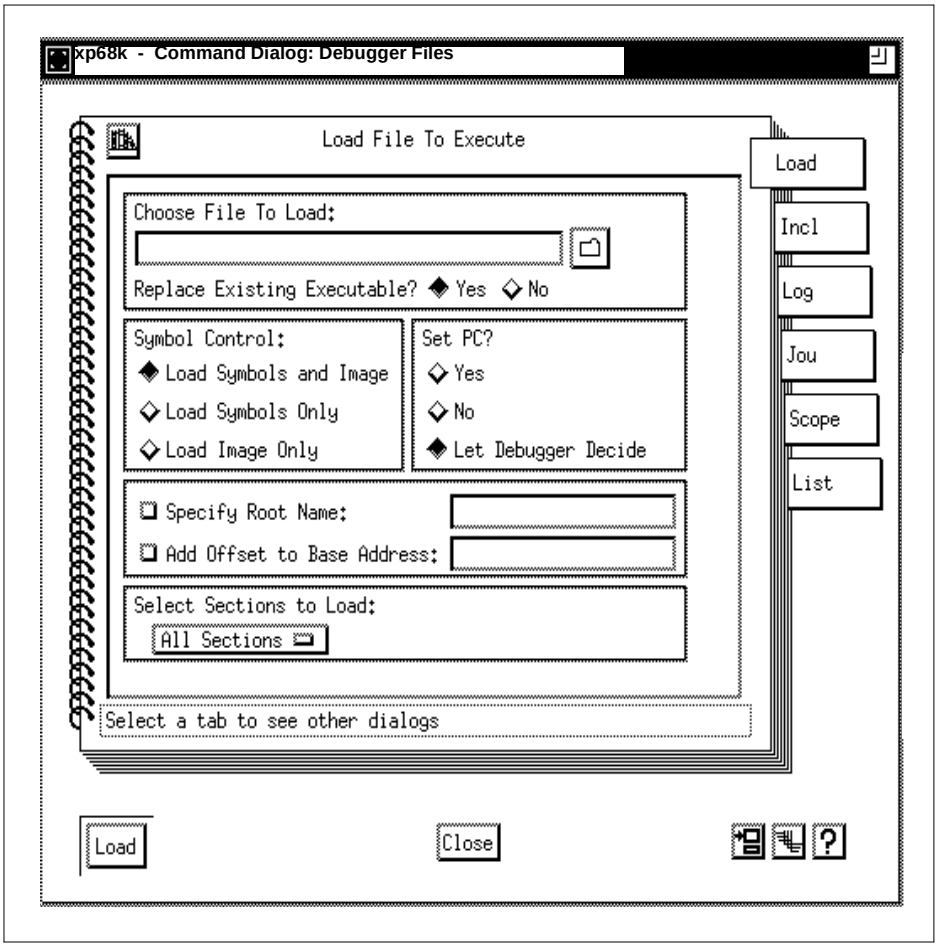


Figure 4-9 Command Dialog Window

3. Select the Folder icon in the Command Dialog window. This icon is at the right of the Choose File To Load text box.

4. Select File Chooser and the dialog shown in Figure 4-10 appears. The default in the **File Chooser** dialog is files with a .**x** extension .
5. Select the application to download (for example, **ram.x**).
6. Click **OK** in the File Chooser.
7. Click **Load** in the Command Dialog window (shown in Figure 4-9 on page 4-15). The downloading process begins. It can last several minutes. The display shows that downloading is taking place by showing “Reading” in the Status field. The Command window shows processing of the **load** command.

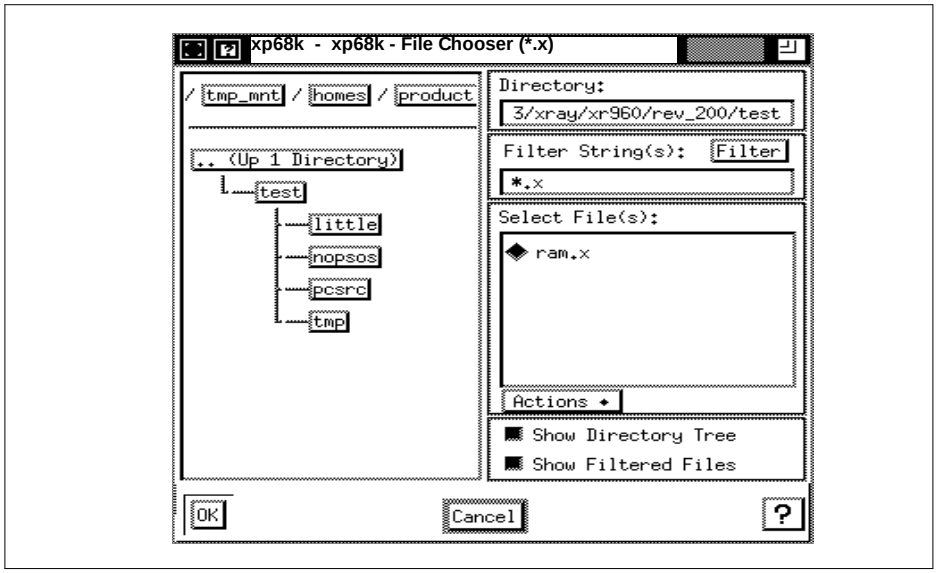


Figure 4-10 File Chooser

Downloading is complete when the Command window displays the following message (see Figure 4-13 on page 4-23 for a sample Command window):

```
(Use RESTART to initialize or re-initialize pSOS+.)
```

The Code and Command windows each can operate in one of two modes: *high-level source mode* and *assembly mode*. After the steps you just completed, the Code window is in high-level source mode, and the

Command window is in assembly mode. To change the Command window to high-level mode, enter the following:

```
>mode high
```

4.6 Starting the Downloaded Operating System

Before starting the pSOS+ kernel, consider the following:

- The executable image has been downloaded to the target but has not been started, as described in Section 4.2, "Creating an Executable Image."
- The Code window may show the message "Not in Source Program" because the current PC is not within the executable image. Once you begin execution, this window normally contains the source code for the currently executing task.
- XRAY Debugger for pSOSystem is communicating with the ROM pROBE+ debugger.

Now you pass control to the operating system in the executable image, by using the XRAY **osboot** command. The **osboot** command causes XRAY to do the following:

- Pass control to the specified address.
- Terminate the connection to the pROBE+ debugger in ROM.
- Establish a new connection with the pROBE+ debugger in the downloaded code.

If, for example, the start address for the downloaded operating system is 0x28008, enter the following:

```
osboot 28008
```

After a short delay, the following message appears:

```
BOOTING COMPLETE
```

This indicates that the downloaded operating system has successfully started, and the downloaded pROBE+ debugger is connected to the XRAY debugger.

When you enter **restart**, the Code window displays the source shown in Figure 4-11.

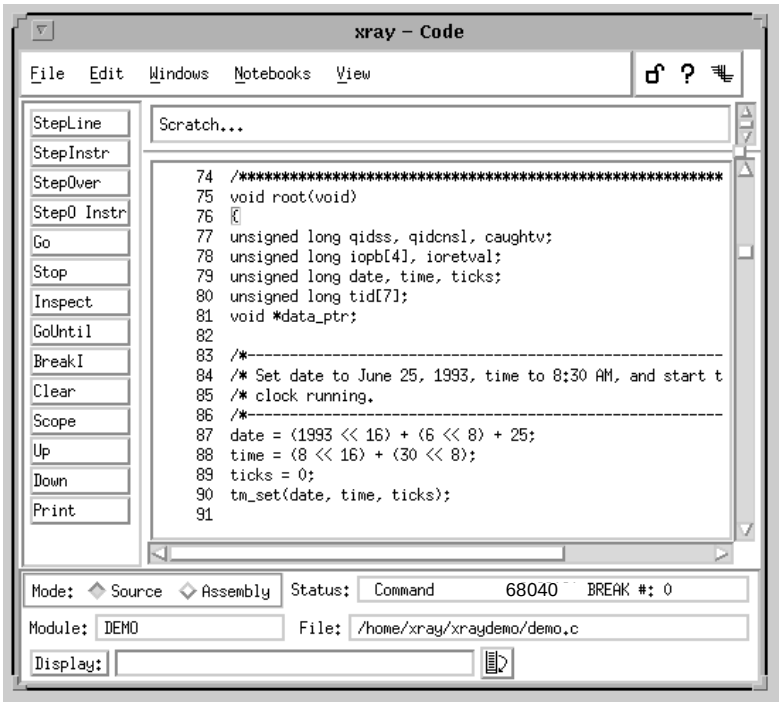


Figure 4-11 xraydemo Source Code

4.7 XRAY Debugger for pSOSystem Product Description

This section describes the following:

- Command Conventions
- Online Help

For additional product information, see the *XRAY Debugger for pSOSystem User's Guide*.

4.7.1 Command Conventions

The following information applies to XRAY debugger commands:

- Commands can be in either uppercase or lowercase.
- Arguments such as task names, routine names, and module names are case-sensitive and must be entered as shown.
- The Stop button aborts the current command and resumes command mode.
- A # identifies a line number within the source code.
- The default interpretation of integer constants is decimal. You can specify hexadecimal by preceding each with a 0x or 0X.
- All commands are terminated by pressing [Return].
- Some commands have a corresponding menu or button. The debugger documentation from Microtec Research lists them.

4.7.2 Online Help

XRAY Debugger for pSOSystem provides context-sensitive online help for all debugger commands, command arguments, and keypad keys. Look at the help system by doing the following:

1. Select the **?** button in the Code window or Command window.
2. Select the "On HELP" menu.
3. Select the Search button as shown in Figure 4-12 on page 4-20.
4. To select a command (such as **BreakI**):

   Scroll to **BreakI** in the **Search_popup** dialog.

-or-

Enter **BreakI** in the Selection text box of the **Search_popup** dialog.

5. Click the **Apply** button.
6. Select the **OK** button.

The help utility displays the command name, its abbreviation, the command syntax, and the command description.

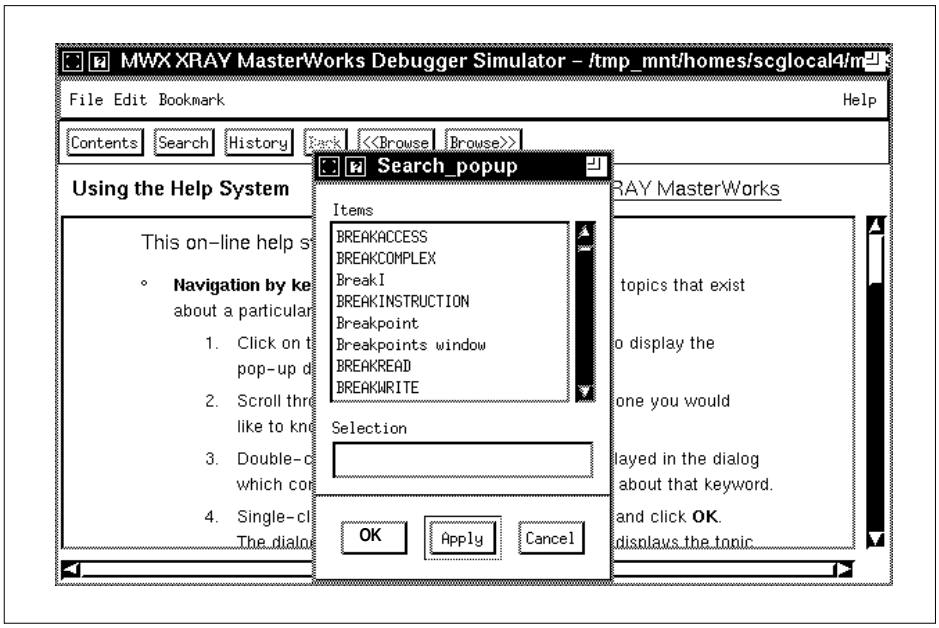


Figure 4-12 Help and Search Windows

4.8 Tutorial Output

The actual program output for this tutorial may differ slightly in this section because:

- The board may have a different memory map, and/or the executable image may load at a different address.

- Using a serial channel for host-to-target communication instead of the pNA+ network manager may cause a difference.
- Different series CPUs have different register sets.

Also, note that the memory-related command examples in this section are based on an on-board RAM starting address of 0, and this is true of most boards. For a few boards, on-board RAM begins at another address, so you must adjust the addresses in the commands. For example, if the on-board RAM begins at 0x4000000 and a tutorial command contains the address 0x1FF00, enter 0x401FF00. For the start addresses of on-board RAM, see Appendix A, "Board-Specific Information."

4.9 Running the System Debug Mode Tutorial

At this point, the **xraydemo** executable image should be running on the target and communicating with XRAY Debugger for pSOSystem on the host.

4.9.1 Memory Manipulation and Windows

To examine memory, enter the following command from the Command window:

dump/l 0..0xFF

The **dump** command requests a hex display of memory. The **/l** requests memory to be displayed in long words. **0..0xFF** specifies the first 256 bytes of target memory.

Fill an area of memory with 0x12, as follows:

fill /b 0x1FF00..0x1FFFF=0x12

The **/b** directs the **fill** command to operate on byte (8-bit) elements (see also the Microtec Research debugger documentation). Each byte in the range of 0x1FF00 through 0x1FFFF is now set to 0x12.

The address range may be unique to each board. Check for a valid address range in Appendix A, "Board-Specific Information."

Now set one byte in this range to a different value:

setmem 0x1FFF0=0

This sets the byte at location 0x1FFF0 to 0. Search for it by entering:

search 0x1FF00..0x1FFFF=0

The XRAY debugger should report:

```
Matched: 001FFF0
```

4.9.2 Starting the pSOS+ Kernel

The XRAY Debugger for pSOSystem **restart** command sets a breakpoint at the first instruction of the pSOS+ ROOT task and passes control to the pSOS+ startup entry point. To initialize the pSOS+ kernel, enter:

```
restart
```

The Command window, shown in the lower portion of Figure 4-13 on page 4-23, displays the following message:

```
pSOS initialized   Running ROOT - #00020000
```

This identifies ROOT as the current running task and 0x00020000 as the task ID.

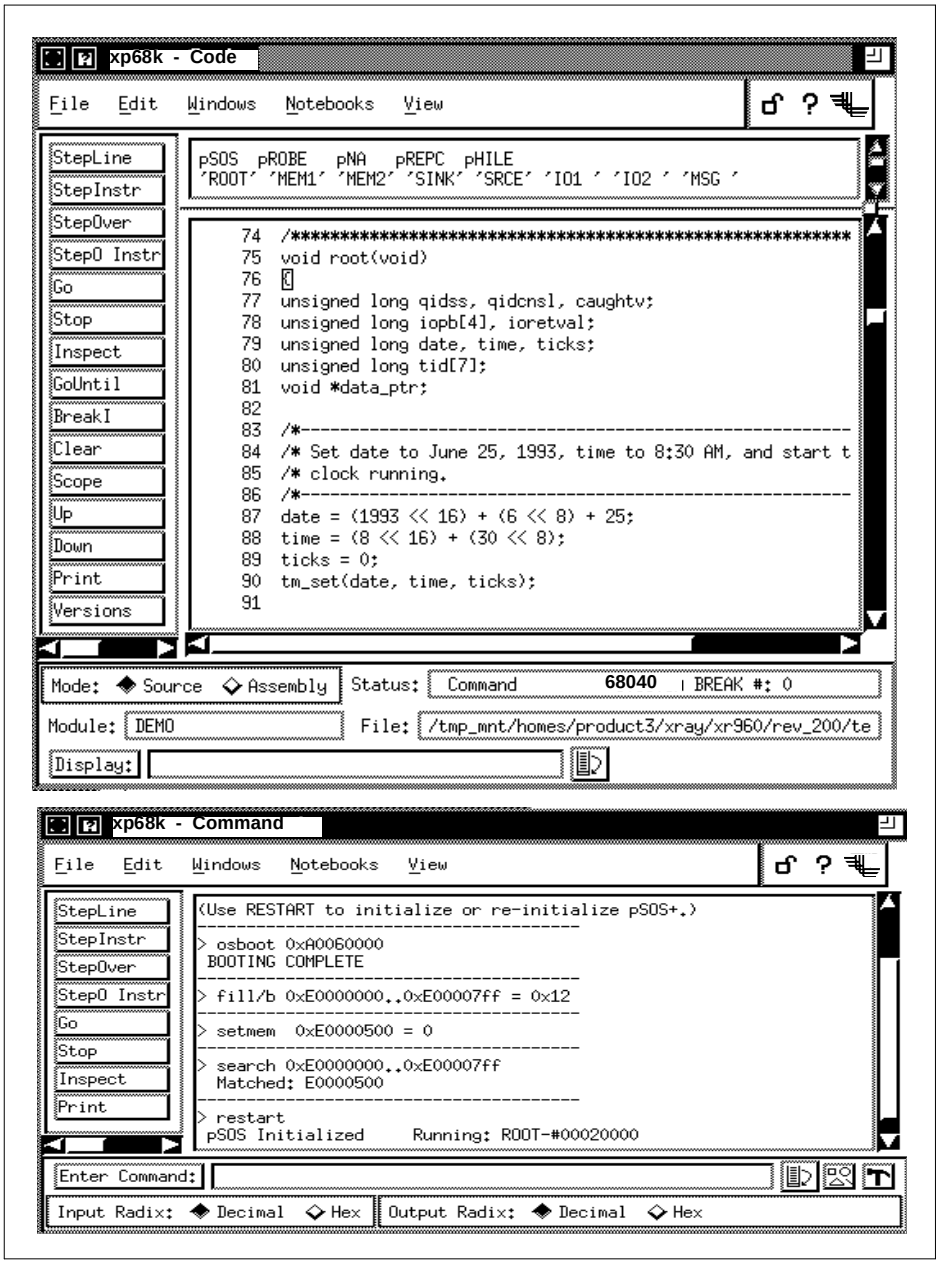


Figure 4-13 Command Window and Code Window

The Code window is now displaying source code for the ROOT task. The XRAY debugger has highlighted the opening brace of the ROOT task, which is the current point of execution. (When control is entering a procedure, XRAY highlights the opening brace.)

4.9.3 High-Level Mode and Assembly-Language Mode

XRAY Debugger for pSOSystem supports two modes of language debugging: the high-level mode and the assembly-language mode. In high-level mode, you debug code at the C/C++ language level, so the Code window shows the C/C++ language source code. In assembly-language mode, you debug at the assembly-language level, so the Code window shows assembly-language code. In general, the command syntax in the two modes are similar, but the behavior of the commands is different.

When in high-level mode, a single step command lets you execute one or more C/C++ language statements.

4.9.4 Stepping C/C++ Statements

Execute C/C++ statements one at a time by selecting the **StepLine** button in the button panel.

Notice that the highlighted line moves down. This is because you are single-stepping lines of executable code. The complexity of the code determines whether the XRAY debugger requires more than one step to complete a single line. In some cases, XRAY may appear to execute several lines of C or C++ code with a single StepLine. This is a result of compiler optimizations.

Change the Command window to high-level mode by entering:

mode high

Repeat the **StepLine** command until the line containing the subroutine call **tm_set()** is highlighted (which may require only one StepLine). In high-level mode, the **StepLine** command ordinarily steps into a called subroutine. The **StepLine** command does not step into an assembly-language instruction in high-level mode. Instead, it treats the assembly subroutine as an atomic operation and highlights only the high-level call.

Single step again and repeat until you are on **de_init()**. This is another assembly routine.

Before you change to assembly-language mode, make the current line the first line in the Code window. Do this with a **scope** command; for example:

scope #92

Other uses of the **scope** command appear later in this tutorial.

To switch to assembly mode, select the **Assembly** button in the Mode section of the Code window, and then enter the following in the Command window:

mode assembly

The Code window displays intermixed source and assembly code. Numbered lines show high-level code corresponding to the assembly code.

The highlighted instruction is preparation for a function call. Single step some lines by repeatedly clicking the **StepInstr** button (see Figure 4-14).

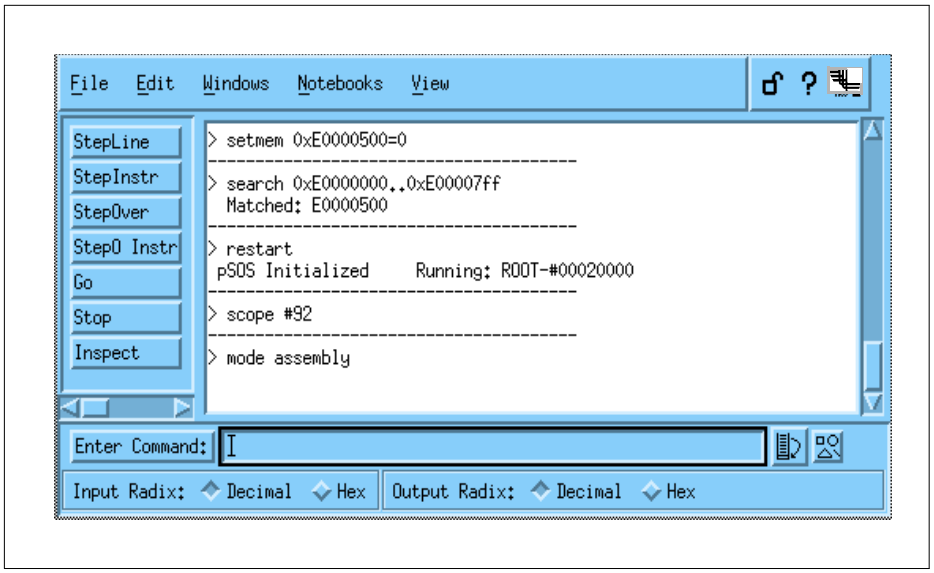


Figure 4-14 Command Window

To step into the routine, select **StepInstr** until the instruction to call a subroutine is highlighted (for example, **jsr** for 68K processors). Now,

select **StepInstr** one more time. Notice that the **StepInstr** command causes a *step into* the routine. Clicking the **StepInstr** button single steps one microprocessor instruction at a time—even in high-level mode. Similarly, selecting the **StepLine** button single steps one source line at a time—even in assembly-language mode.

The **StepO Instr** command, when it encounters a subroutine call, executes the call, and halts when the called subroutine returns. Selecting **StepO Instr** single steps one microprocessor instruction at a time—even in high-level mode. Similarly, selecting **StepOver** single steps one source line at a time (even in assembly-language mode).

Now return to high-level mode by taking the following steps:

1. In the Code window, click the **Source** button.
2. In the Command window, enter **mode high.**

The **go** instruction resumes execution of the application and optionally allows you to set a temporary breakpoint. The application continues until it encounters a break condition (if specified). For example, to resume execution with a temporary breakpoint at line #102, enter:

go #102

The XRAY debugger should report a break at line #102.

4.9.5 Queries and Breakpoints

XRAY Debugger for pSOSystem offers direct access to a subset of pROBE+ commands. The majority of these commands are query commands. Refer to the *XRAY Debugger for pSOSystem User's Guide* for a complete list. Query the status of all active tasks by entering:

qt

The View window with sample output appears as shown in Figure 4-15.

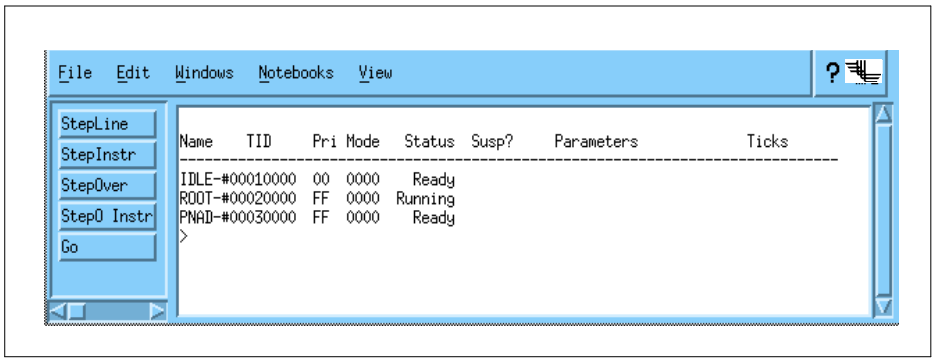


Figure 4-15 Output of the View Window

Because you have just begun execution of the ROOT task, you have only the three tasks automatically created by the pSOS+ kernel and the pNA+ network manager: IDLE, ROOT, and PNAD (pNA+ creates the PNAD task).

If you allow ROOT to execute further, it creates many more tasks. You could stop ROOT by setting a breakpoint at a statement further along in the code, but XRAY Debugger for pSOSystem supports more sophisticated types of breaks. For example, the XRAY debugger allows you to halt execution when a task makes a particular pSOS+ service call. Note that ROOT calls **ev_receive()** after creating all the other tasks and queues. You can use the **breakcomplex** command (**bc**) to halt execution when ROOT calls **ev_receive()**. The **bc** command accepts options and prompts for parameters as required. Enter **bc** with the option **se** to indicate a break on a service call:

bc se

XRAY displays all the service calls and prompts for a pSOS+ function name. You can enter a particular pSOS+ function, or enter * to indicate any function. Enter **ev_receive** when XRAY Debugger for pSOSystem prompts for a function, as follows.

Function: ev_receive

XRAY Debugger for pSOSystem now prompts you to identify the call origin, which is either a task or an interrupt service routine (ISR). You can enter a particular task name or task ID, ISR, or * to indicate any task. In this case, enter ROOT when the XRAY debugger prompts for an origin (the single quotes around ROOT are required):

Origin: 'ROOT'

XRAY confirms the break definition by displaying a complete list of all active breakpoints in the Breakpoints window. In this case, only the one service break is set (which is ROOT making the **ev_receive()** call).

Click the **Go** button. Execution stops because ROOT made an **ev_receive()** call. XRAY reports it encountered a service break, the type of call made, and the address from which the call was made. Display the Traceback window to see that the call originated in the C interface file **psos.s**. To display the Traceback window, do the following:

1. Select Windows in the menu bar.
2. Select Traceback Window in the Windows menu.

At this point, ROOT has completed spawning and activating the other tasks. You can use the **query task** command (**qt**) to view these tasks:

qt

Figure 4-16 shows the resulting **qt** display in the View window.

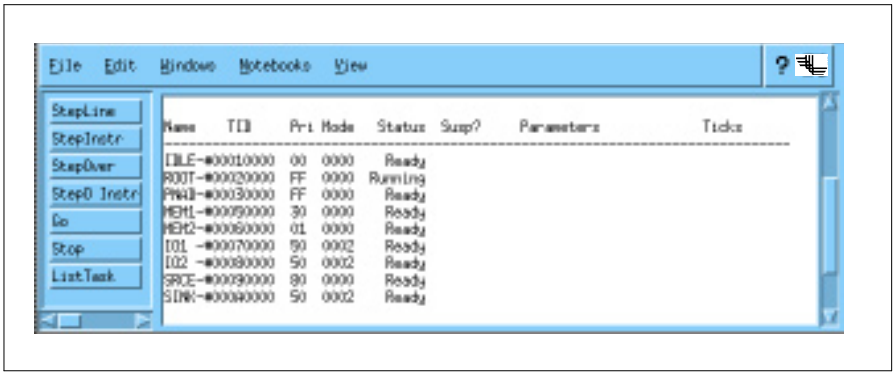


Figure 4-16 Output of the qt Command

ROOT is about to block, and all other tasks are ready to run. Halting execution each time a new task executes is instructive. You can halt on new task execution with the *dispatch break*. The dispatch break halts execution whenever a specified task begins executing. You can also specify a wild card, which causes a break upon any context switch.

Define a dispatch break on any task and resume by entering:

bc di *

Because the application is not using the pNA+ network manager, you should lower the priority of PNAD. The XRAY Debugger for pSOSystem command **SYSCALL** (**sy**) lets you make pSOS+ calls directly from the XRAY debugger, so you can use **t_setpri()** to lower the priority of PNAD to 1, for example:

sy t_setpri 'PNAD' 1

NOTE: The sy command is necessary if you are running the pNA+ networking component.

Resume execution by clicking **go**. The display shows the last running task, the reason it stopped running, and the task that is about to run. Click **go** three more times and observe which tasks execute.

4.9.6 Clearing Breakpoints

The **clear** command is used to clear breakpoints. To see how many breakpoints are set, do the following:

1. Select the Windows menu on the menu bar.
2. Select the Breakpoints option. The Breakpoints window appears as shown in Figure 4-17.

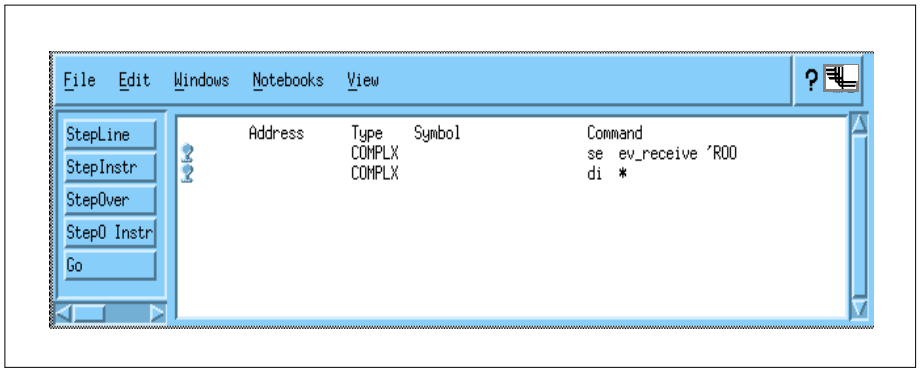


Figure 4-17 Breakpoints Window

Currently, the service break and dispatch break are set. You can clear the breakpoints by doing the following:

1. Click on a breakpoint in the Breakpoints window to select one.
2. Click the **Clear** button as shown in Figure 4-13 on page 4-23.

Examining the system after most tasks have been blocked is instructive. Because the IDLE task has the lowest priority (0), it runs only when all other tasks are blocked. Set a dispatch breakpoint to stop execution when IDLE executes by entering:

bc di 'IDLE'

Click the **Go** button. Execution stops because task IDLE is about to run. The Code window displays the following message because the code for task IDLE is part of the pSOS+ kernel and is therefore unknown to the XRAY debugger:

Not in Source module.

Check the state of other tasks by entering:

qt

The View window appears with the **qt** output as shown in Figure 4-18.

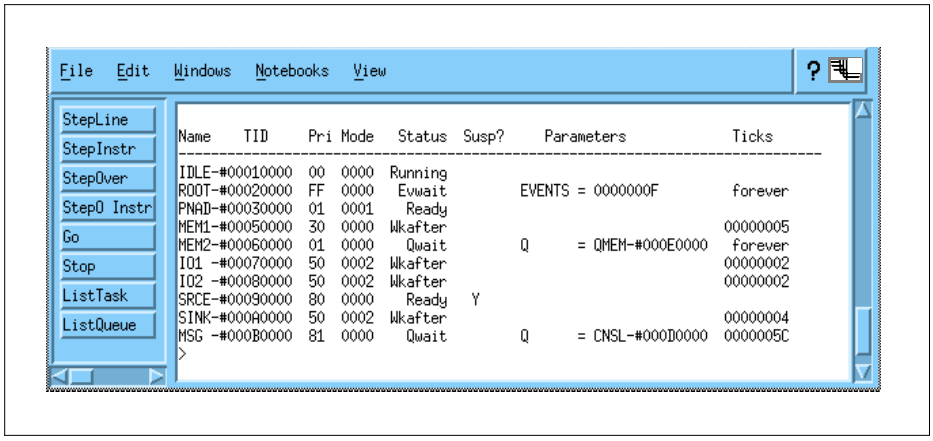


Figure 4-18 View Window Showing qt Output

The Status column (displayed in Figure 4-18) shows that IDLE is running. All other tasks besides PNAD are either paused, suspended, waiting for events, or waiting for messages. PNAD was blocked when the break occurred, but use of the network communication link since that time has changed it to the ready state.

4.9.7 Querying Queues

The query task command (**qt**) is used to examine tasks. Other query commands are available to examine other pSOS+ objects. For example, examine all queues in the system by entering:

qq

The View window appears with the **qq** output as shown in Figure 4-19.

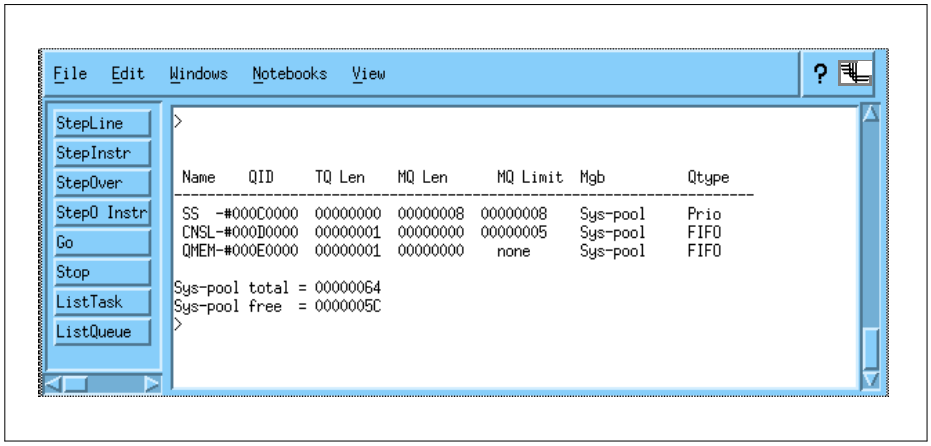


Figure 4-19 View Window Showing qq Output

The system has four queues. The first queue is 'SS••'. It uses priority queuing and currently has eight messages pending. The second and third queues are CNSL and QMEM. They use FIFO queuing, and each has one task waiting. The fourth queue is for each open serial port for the DITI.

In addition to halting at a line number, you can also set a breakpoint on a procedure name. To see an example of this, do the following:

1. Clear the dispatch break
2. Enter the following in the Command window:

   go process_data

Execution stops because **process_data** has been called.

4.9.8 Symbols and Variables

XRAY Debugger for pSOSystem commands let you examine and modify variables and symbols that the program uses. The **printvalue** command (**p**) is used to print the value of a variable. The **process_data** routine uses a global variable **Index**. To examine a local variable enter the following:

```
p Index
```

The XRAY debugger may display the following error message:

```
At start of procedure, no local variables yet.
```

If it does, step into the procedure and try again:

```
step
p Index
```

The Command window displays the value of this variable. The **mem1** routine has a local variable **ticks**, and you can examine it by entering:

```
p ticks
```

When you try to examine **ticks**, the following error message appears because you are currently within the scope of the routine **process_data**:

```
Symbol not available from this scope without a qualifier
```

To examine variables in another routine, specify the routine's name:

```
p mem1\ticks
```

Now the following error message is displayed because local variables are allocated on the stack when the routine is entered:

```
Local variable not alive
```

Each task has its own stack, and a task other than the one currently executing contains **ticks**. Thus, to examine **ticks**, you must be executing the routine **mem1**. To do this, use the **breaki** command (**b**) to set a breakpoint within **mem1**. Set the breakpoint at an instruction after **ticks** has been initialized:

```
b #195
go
```

This causes execution to proceed until line #195 in the source code is reached. XRAY then reports a break. Now examine the variable **ticks**:

```
p ticks
```

At last, the value of **ticks** is displayed in the Command window. You can also examine arrays and structures with this command. The **mem1** routine contains an array called **addrmsg**. Examine it by entering:

p addrmsg

The value of each array element is displayed in the Command window.

The **cexpression** (**c**) command is another command that can display a variable. Examine **ticks** by using **c**:

c ticks

The Command window displays the value of **ticks** in both decimal and hexadecimal. This command is actually used to calculate the value of an expression or to assign a value to a variable. When the **c** command is used on an array or structure, it calculates the address of the specified item, which is then displayed. Try this command on an array:

c addrmsg

Notice that the address of **addrmsg** is displayed. Contrast this to what was displayed when the **p** command was used on this variable.

The **c** command can also be used as a decimal-to-hexadecimal or hexadecimal-to-decimal converter. For example, enter the following to calculate the decimal value of this number and display it in the Command window:

c 0x41

Because this number is also a printable ASCII character, the decimal equivalent of 0x41 is displayed (number 65).

You have seen symbols qualified by procedure names. In addition, XRAY lets you qualify symbols by using task selectors. When you specify a symbol in a command, you can prepend it with the name of the task using the format *taskname:* (where ROOT is the *taskname* in the example line that follows). This instructs XRAY to use a particular context when evaluating a symbol. Use the **printvalue** command to examine a symbol in the ROOT task:

p ROOT:\DEMO\root\tid[0]

This displays the variable **tid[0]** in the procedure **root** within the module DEMO. **tid[0]** exists within the context of the task ROOT.

The **monitor** command allows variables to be monitored. Monitored variables are updated each time the debugger stops executing the program. The monitored data is displayed in the Data window.

The procedure **mem1** has a variable **size**. The variable **size** contains a random number that is used to request memory segments of various sizes. Monitor this variable by entering the following:

```
monitor size
```

This instructs XRAY to monitor the variable **size** in the currently executing procedure **mem1** and open the Data window. A breakpoint is already set at line #195, which is within the task MEM1 after it has initialized **size**. To resume execution, enter the following:

```
go
```

The monitored variable is displayed in the Data window. Enter **go** three more times and observe how the monitored data changes.

The previously mentioned **cexpression** can be used to assign a value to a variable. Change the value of **size** to 256 by entering:

```
c size=256
```

The new value of **size** is displayed in the Command window and, because it is still being monitored, the Data window. In the example program, **size** is assigned a pseudo-random value, so keeping this value is acceptable.

The **nomonitor** command (**nomo**) is used to disable the monitoring of a variable and remove it from the Data window. When a variable is monitored, it is assigned a monitor line number, and the Data window displays this number. Notice that **size** is assigned line number 1. Stop monitoring the variable and remove the breakpoint by entering the following commands:

```
nomo 1
clear 1
```

Another way to display the variable **size** is by using the **expand** command. This command examines the current stack and displays all the local variables it finds in each procedure. The Trace window is used in high-level mode to display the procedure calling chain. To examine the local variables, enter the following:

```
expand
```

As the Trace window shows, only one level of calls exists at this point. **expand** also allows you to specify a particular stack level to examine.

So far, the windows have displayed information for the current running task—MEM1. To look at a non-running task, use the **scope** command. **scope** defines a new default context other than the running task. All of the window displays, trace-back information, local variables, and so on, reflect information for the new default task. Examine the ROOT task after first returning to a normal screen display with the following two commands. (Note that a colon must follow the argument when the argument to the **scope** command is a task.)

scope ROOT:

The windows now display information for the ROOT task. As expected, ROOT is waiting for an event and is therefore in the **ev_receive()** routine. The stack trace shows **psos.s**, which is the C language source file that provides the interface to the pSOS+ kernel. If you enter **expand** now, it displays all the procedure calls and local variables for the ROOT task:

expand

The **scope** command affects only the information displayed in the windows. It does not modify the current execution state, and therefore MEM1 is still the current task. To return to the display of the current task, enter **scope** without parameters:

scope

The windows now display information about MEM1.

You may need information about a symbol other than its value. The **printsymbol** command (**ps**) displays information about a specified symbol or group of symbols. This information includes the symbol name, data type, storage class, and memory location. To examine all symbols used in the procedure **mreader**, enter the following:

ps mreader

where the backslash is a *symbol qualifier convention*. For an explanation of the backslash and other symbol qualifier conventions, refer to the *XRAY Debugger for pSOSystem User's Guide*.

To display information about a specific symbol, enter the following:

ps root\tid

This displays information about the variable **tid** in the procedure **root**.

4.9.9 Profiling

One unique feature of XRAY Debugger for pSOSystem is its profiling capability. By transparently collecting statistics on the application, The XRAY debugger can provide important insights into a design's activities and performance. To enable profiling, turn on the profile flag as follows:

```
fl profile on
```

The Command window displays the following flag settings:

```
RBUG: ON
HOST: OFF
TRFR: ON
NODOTS: ON
NOMANB: OFF
NOPAGE: ON
ECHO: OFF
PROFILE: ON
```

For example, to profile the application for 30 seconds, use a timed break to halt execution after 3000 clock ticks, which is 30 seconds. (The pSOS+ configuration table specifies 100 ticks per second.) Set the breakpoint and resume execution with the following entries:

```
bc ti #&3000
go
```

On the first line, the ampersand (&) indicates the number that follows is decimal, and the pound sign (#) indicates that the next value is relative to a starting point. Otherwise, an absolute time and date would have been entered. Execution stops after the designated interval of 30 seconds. Normally, for a statistically significant profile, a much longer interval is necessary. To view the profile data, enter the following:

```
lp
```

The values displayed are decimal. Not all of the profile data can fit in the View window, so use the mouse to scroll through it.

The number of ticks listed for each task is statistical because it depends on the sampling at each clock edge. The other data, such as the number of times a task switches into a particular state and the number of system calls made, are actual counts.

To turn off profiling, enter the following:

fl profile off

4.9.10 Interactive System Calls and I/O

XRAY Debugger for pSOSystem allows you to make system calls manually to the pSOS+ kernel so that you can conveniently modify and control the application's behavior. For example, the sample application has a queue named CNSL, to which you can manually send messages.

First, check the status of queue CNSL by entering:

qq 'CNSL'

The View window in Figure 4-20 shows that one task, 'MSG•', exists and it is waiting for a message.

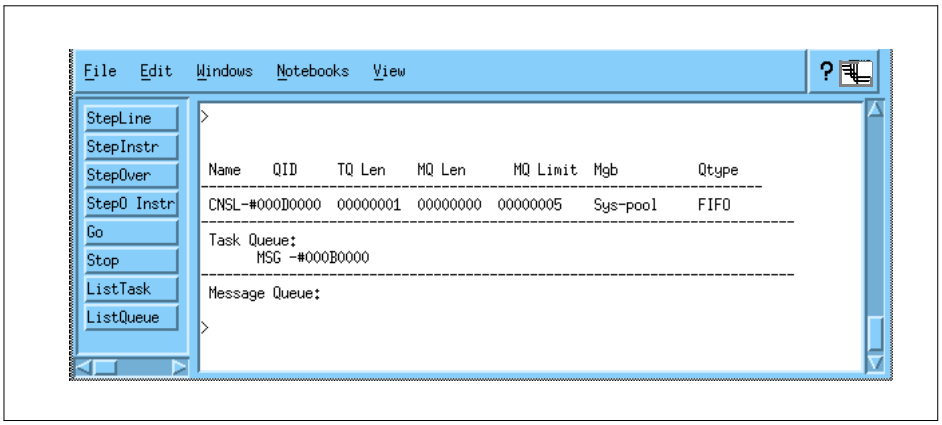


Figure 4-20 View Window Showing Queue Status

You can now use the **sy** command to send a message to CNSL. Because 'MSG•' is waiting, it receives the message and becomes ready to run:

sy q_send 'CNSL' 'ABCD' 'EFGH' 'IJKL' 'MNOP'

In this example, you enter all the command parameters on a single line. If you are not sure what parameters the call requires, you can leave out the parameters and let XRAY prompt for each parameter. When the command is done, the following pSOS+ system call return code is displayed:

```
Return code: 00000000
```

where a string of all 0s indicates that the call succeeded.

Instead of re-entering the same command, you can use the history feature that the XRAY debugger provides. It saves up to 11 previous commands. Select the stack icon at the bottom of the Command window. A menu lists the previous commands. You can select the previous command three times, as follows:

sy q_send ‘CNSL’ ‘ABCD’ ‘EFGH’ ‘IJKL’ ‘MNOP’

sy q_send ‘CNSL’ ‘ABCD’ ‘EFGH’ ‘IJKL’ ‘MNOP’

sy q_send ‘CNSL’ ‘ABCD’ ‘EFGH’ ‘IJKL’ ‘MNOP’

Now look at CNSL again:

qq 'CNSL'

The View window in Figure 4-21 shows the output of this **qq** command.

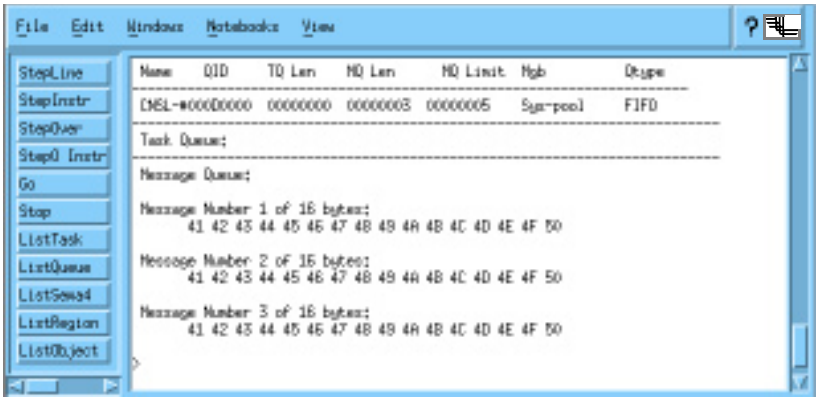


Figure 4-21 Output of the qq Command

Three messages are queued. (Remember that the first message was passed to the waiting task 'MSG•'.) Task 'MSG•' is ready to run, and when you restart execution, 'MSG•' sends all four messages to the XRAY debugger standard I/O screen.

Task 'MSG•' demonstrates another important feature of XRAY, the ability of a pSOS+ task to write to the XRAY terminal. XRAY provides an I/O screen through which application code can write to the XRAY terminal. The application accesses the Standard I/O window by calling the **db_output** routine in the file **xp_out.s**.

When execution resumes, the I/O screen appears with the messages from task 'MSG•'. To resume execution, enter the following:

```
go
```

The following should be displayed:

```
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
```

At this time, the sample application is still running because no breakpoints are set. Stop execution by clicking the **Stop** button. A message similar to the following appears:

```
Manual Break. Running: 'IO2 ' -#00060000
Stopped due to halt from user
```

Because you have manually stopped execution, the running task may not actually be 'IO2•'.

You have now completed the XRAY Debugger for pSOSystem tutorial. To terminate this debugging session, enter the following:

```
quit
```

After you enter **quit**, XRAY prompts:

```
Are you sure?
```

Enter a **y** to terminate the session.

5 XRAY Debugger for pSOSystem Tutorial: Viewport Version

This chapter provides a hands-on tutorial on the use of XRAY Debugger for pSOSystem, the multitasking debugger from Integrated Systems. The tutorial illustrates how this source-level cross debugger can be used to control and monitor a multitasking application on an HP, RS6000, or PC host. If you are using a SunOS or Solaris host, see Chapter 5, "XRAY Debugger for pSOSystem Tutorial: Viewport Version."

The XRAY Debugger for pSOSystem has the following features:

- A graphical user interface with multiple viewports
- Automatic tracking of program execution through source code files
- Traces and breaks on high-level language statements
- Breaks on task state changes and operating system calls
- Monitoring of language variables and system-level objects such as tasks, queues, and semaphores
- Profiling for performance tuning and analysis
- Full-featured C++ language support
- The ability to debug optimized code

- System Debug Mode support

In addition to supporting System Debug Mode, some versions of XRAY Debugger for pSOSystem also support Task Debug Mode. This tutorial illustrates the use of System Debug Mode only. For more information on Task Debug Mode, refer to the *XRAY Debugger for pSOSystem User's Guide*.

5.1 xraydemo Sample Application

The **apps** directory contains the subdirectory **xraydemo**, which contains a sample application also called **xraydemo**. The tutorial in this chapter uses this sample application.

The **xraydemo** sample application is a C program that contains the code for eight tasks. The ROOT task, which automatically receives control from pSOS+ after startup, creates the seven other tasks and then blocks. The seven other tasks function as follows:

Task	Function
'MEM1'	Requests memory segments of various sizes.
'MEM2'	Receives memory segments from MEM1 and frees them.
'IO1•'	Reads a block from the RAM disk device.
'IO2•'	Writes a block to the RAM disk device.
'SRCE'	Sends messages to the 'SS••' Queue.
'SINK'	Consumes messages from the 'SS••' Queue.
'MSG•'	Consumes messages from Queue CNSL and sends the contents to the XRAY Debugger for pSOSystem terminal.

NOTE: The 'IO1•' and 'IO2•' tasks consist of 4-character names where the fourth character is a required space. Similarly, the 'SS••' queue name includes two required spaces; the breve symbol (•) indicates a required space.

A task name entered with a command sometimes requires single quotes around the task name. The examples in the manual show when the quotes are required.

5.2 Creating an Executable Image

This section guides you through the steps needed to build an executable image that contains an operating system for the target board and the **xraydemo** sample application.

First, build a new working directory that contains the **xraydemo** sample and then switch to that directory. Call this directory **xd**. For example, you can enter the appropriate command sequence for your environment.

For an HP or RS6000 workstation:

```
mkdir xd
cp -r $PSS_ROOT/apps/xraydemo/* xd
cd xd
```

For an MS-DOS system:

```
xcopy \pssp\apps\xraydemo xd
cd xd
```

where, for both environments, **xd** is the name of a new working directory.

NOTE: The xraydemo application prints by using db_output(). Therefore, the constant OUTPUT_TO_DEBUGGER used in the hello application does not exist in xraydemo. Also, the value of SC_APP_CONSOLE in sys_conf.h is not relevant.

5.2.1 Customize Your Operating System

You may need to customize your operating system to work with the XRAY Debugger for pSOSystem by taking one or more of the following steps:

- Set PSS_BSP in **makefile** to an appropriate value. A PSS_BSP value has the form **$(PSS_ROOT)/bsps/*directory_name***, where *directory_name* is the name of the BSP.
- If you use XRAY Debugger for pSOSystem over a serial channel, follow the directions in Section 5.3, "Using XRAY Over a Serial Channel."
- If you use XRAY Debugger for pSOSystem over an Ethernet port, set SC_PNA in **sys_conf.h** to YES. For more information on Ethernet ports, see Section 5.4, "Using XRAY Debugger for pSOSystem Over Ethernet."

- If you use the pSOS+ kernel instead of pSOS+m kernel, set the **sys_conf.h** parameters as follows: SC_PSOS to NO and SC_PSOSM to YES.

5.2.2 Building the Executable Image

After you make the needed changes, build the executable image with the following command:

make ram.x

As noted above, you can download the **ram.x** file to the target system using a serial channel (as described in section 5.3) or an Ethernet port (as described in section 5.4). Follow the directions in the section appropriate for your environment.

5.3 Using XRAY Over a Serial Channel

This section describes how to run the **xraydemo** sample by using XRAY Debugger for pSOSystem connected to the target over a serial channel as shown in Figure 5-1. In this section both the ROM and downloaded pROBE+ operate in remote mode talking to XRAY over a serial channel. As explained previously, the operating modes of the ROM pROBE+ and downloaded pROBE+ do not need to be the same, but in most cases they are.

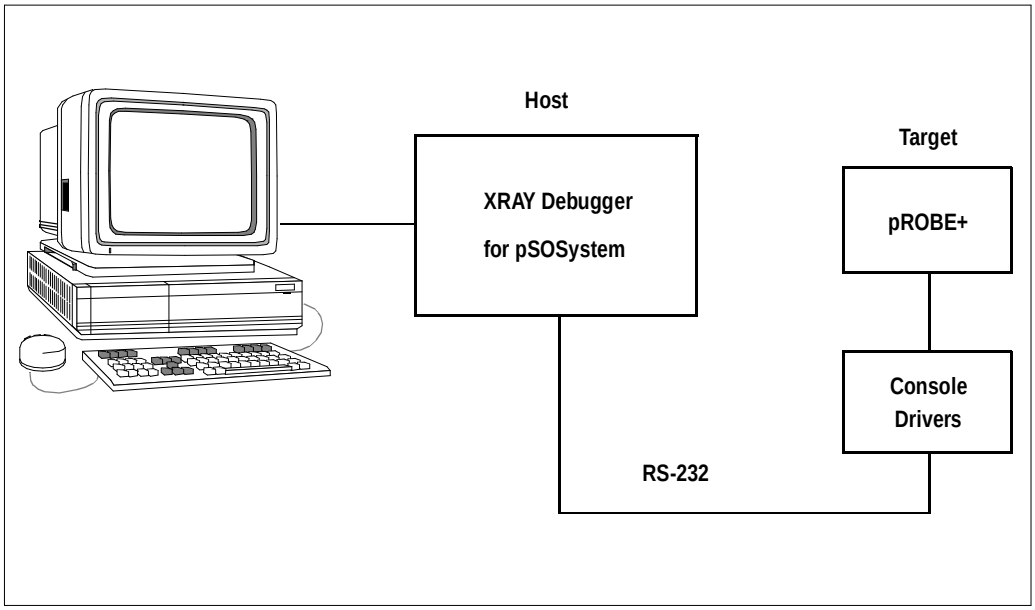


Figure 5-1 Using XRAY Debugger for pSOSystem Over a Serial Channel

5.3.1 Reconfiguring the ROMs for a Serial Channel

To reconfigure the ROMs for communication with XRAY Debugger for pSOSystem over a serial channel, reset the target CPU board and enter a character before the specified time elapses. When you power up or reset your board, a message similar to the following appears as part of the pSOSystem startup display (as shown in Figure 2-6 on page 2-11):

```
To change any of this, press any key within 5 seconds
```

Enter any key within the time limit, and a message similar to one shown in Figure 5-2 appears.

```
For each of the following questions, you can press <Return> to select the
value shown in braces, or you can enter a new value.

How should the board boot?
  1. pROBE+ stand-alone mode
  2. pROBE+ waiting for host debugger via serial connection
  3. pROBE+ waiting for host debugger via a network connection
  4. Run the TFTP Bootloader

Which one do you want? [1]
```

Figure 5-2 Reconfiguring the ROMs for a Serial Channel

To use XRAY with pROBE+ over a serial channel, enter a **2**.

You do not need to change the remaining parameters, so bypass the remaining questions by pressing [Return] until the following question appears:

```
How long (in seconds) should CPU delay before starting
up? [5]
```

Once you learn how to use the ROMs after a reset, the default of 60 seconds is unnecessarily long. Enter a value of your choice, or press [Return] to accept the current setting of 5 seconds.

When you power up or reset the board, the terminal displays a pSOSystem startup message similar to the one in Figure 5-3.

```
------------------------------------------------------------------------
START-UP MODE:
  Boot into pROBE+ and wait for host debugger via a serial connection
NETWORK INTERFACE PARAMETERS:
  LAN interface is disabled
  Shared memory interface is disabled
MULTIPROCESSING PARAMETERS:
  This board is currently configured as a single processor system
HARDWARE PARAMETERS:
  Serial channels will use a baud rate of 9600
  This board's memory will reside at 0x1000000 on the VME bus
  Processor Type :: MC68040 operating at 25 Mhz
  RAM configuration :: Parity DRAM 4 Mb
                    :: SRAM 128 Kb
  After board is reset, start-up code will wait 3 seconds
------------------------------------------------------------------------
(M)odify any of this or (C)ontinue? [M]
```

Figure 5-3 pSOSystem Startup Message for a Serial Channel

To change any of the settings shown in Figure 5-3, press any key within 5 seconds.

If a terminal emulator is running, close the emulator now because the next section requires the use of XRAY over a serial channel.

NOTE: Resetting a board does not cause the ROMs to revert to their default configuration. Furthermore, because configuration data is stored in battery-backed or nonvolatile RAM when available, powering down a board causes a return to default values only if the board lacks nonvolatile storage capability.

5.3.2 Invoking XRAY over a Serial Channel

Because XRAY Debugger for pSOSystem is a source-level debugger, it must locate the source code for the object code it is debugging. It first searches the current directory and then searches as directed by the environment variable **XRAY**. If you set **XRAY** to **@**, XRAY uses the source file pathnames embedded in the executable image. Set the **XRAY** environment variable for a particular platform as follows:

For UNIX (csh):	For MS-DOS:
setenv XRAY @	**set xray=@** (or **SET XRAY=@**)

You can now invoke XRAY Debugger for pSOSystem on the host. To invoke it for use over a serial channel, use the following syntax:

For UNIX:	For MS-DOS:
xp -e r*dev* ram.x &	**xp -e r*dev* ram.x**

where ***dev*** specifies the serial device and **ram.x** is the executable image. The following table shows some of the valid device names for the indicated hosts:

Host	Operating System	Device
HP-9000/700	HP_UX	tty01, tty02, ...
RS6000	AIX	tty0, tty1, ...
PC Compatible	MS-DOS	COM1, COM2

For example, on an HP system you can invoke XRAY Debugger for pSOSystem by entering:

xp -e rtty01 ram.x &

A sign-on screen appears while XRAY Debugger for pSOSystem initializes; downloading the application can take up to ten minutes. During this time, XRAY displays a small rotating line to indicate that loading is in progress. If an error message appears, it is likely that XRAY cannot find the file to load. When downloading is complete, XRAY displays a command prompt in the Command viewport of the XRAY screen, as shown in Figure 5-4.

Figure 5-4 XRAY Debugger for pSOSystem Command Viewport

If you plan to use the XRAY debugger over a serial channel often, see the next section for a description of how to increase the baud rate beyond the default of 9600 baud.

5.3.3 Changing the Baud Rate for a Serial Channel

You may want to increase the baud rate of the serial channel. You can increase it to either 19200 or 38400 baud if the host and target both support the higher rate. The baud rate must be changed in several places:

- The Boot ROMs should be reconfigured for the higher rate.
- The XRAY debugger uses a default baud rate of 9600. Therefore, when you invoke XRAY, give the higher rate explicitly. You do this by appending the baud rate to the name of the serial device, separated by a comma. For example, the following command changes the baud rate to 19200:

  xp -e rcom1,19200 ram.x

- The terminal or terminal emulation program must be reconfigured to operate at the higher rate.

To continue with the tutorial set up for a serial channel, see Section 5.5, "Starting the Downloaded Operating System."

5.4 Using XRAY Debugger for pSOSystem Over Ethernet

This section describes how to run the **xraydemo** sample program by using the XRAY debugger over an Ethernet connection as shown in Figure 5-5.

The exercises in this section require the pNA+ network manager in the system and your host and target must both be physically connected to an Ethernet network. The use of XRAY over Ethernet requires extra installation steps on some hosts, particularly a PC. For additional installation information, see the *XRAY Debugger for pSOSystem User's Guide*.

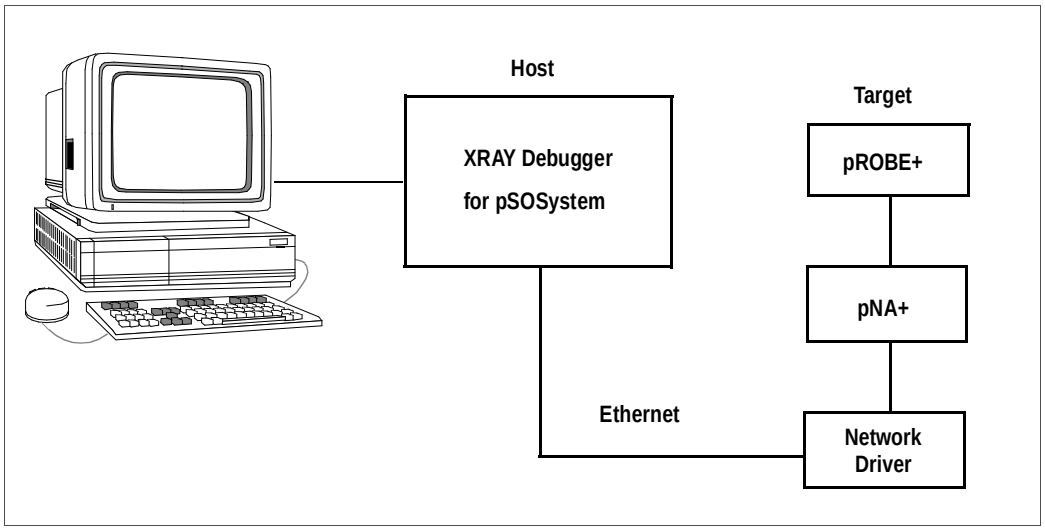


Figure 5-5 Using XRAY Debugger for pSOSystem Over Ethernet

5.4.1 Connecting to the Network

To use Ethernet, the target and host must connect over Ethernet. This is usually done by adding the target system to the office network but can also be done with a private network.

5.4.2 Reconfiguring the ROMs for Ethernet

To reconfigure the ROMs for communication with XRAY over Ethernet, reset the target CPU board and enter a character at the pSOSystem startup screen (shown in Figure 5-3 on page 5-7) before the specified time elapses. A message similar to the following appears:

```
To change any of this, press any key within 5 seconds
```

Press any key within the time limit, and a display similar to the one shown in Figure 5-6 appears.

```
For each of the following questions, you can press <Return> to select the
value shown in braces, or you can enter a new value.

How should the board boot?
  1. pROBE+ stand-alone mode
  2. pROBE+ waiting for host debugger via serial connection
  3. pROBE+ waiting for host debugger via a network connection
  4. Run the TFTP Bootloader

Which one do you want? [1]
```

Figure 5-6 Reconfiguring the ROMs for Ethernet

To use the XRAY Debugger for pSOSystem and pROBE+ debuggers over Ethernet, enter a **3**.

NOTE: The steps described in this section are always done through an ASCII terminal, regardless of the subsequent operating mode of the ROMs.

By entering a **3**, you also get the following network questions:

```
NETWORK INTERFACE PARAMETERS:
Do you want a LAN interface?  [N] y
```

Enter a **y**. (**n** would apply to multiprocessor systems where some boards communicate through shared memory rather than Ethernet.) The next prompt is as follows:

```
This board's LAN IP address? (0.0.0.0 = RARP)?
[199.99.99.99]
```

Enter the IP address of the CPU board using standard dot notation for internet addresses (four numeric fields, each separated by a period). You

should answer **n** to the next two questions because they enable features that are useful only in multiprocessor target systems:

```
Use a subnet mask for the LAN interface? [N]
Do you want a shared memory network interface? [N]
```

The next question is as follows:

```
Should there be a default gateway for packet routing? [N]
```

The usual answer is **n** unless the target and host systems are on different networks connected through a gateway. If you are not sure how to answer this, ask your system administrator.

Accept the indicated default values for the remaining questions as shown below:

```
MULTIPROCESSING PARAMETERS:

Do you want to configure a multiprocessing pSOS+m
system? [N]

HARDWARE PARAMETERS:

Baud rate for serial channels [9600]

Bus address of this board's dual-ported memory
[4000000]

How long (in seconds) should CPU delay before starting
up? [5]
```

When you power up or reset the board, the terminal displays a pROBE+ startup message similar to the one in Figure 5-7.

```
START-UP MODE:
  Boot into pROBE+ and wait for host debugger via a network connection
NETWORK INTERFACE PARAMETERS:
  IP address on LAN is 199.9.999.9
  Shared memory interface is disabled
MULTIPROCESSING PARAMETERS:
  This board is currently configured as a single processor system
HARDWARE PARAMETERS:
  Serial channels will use a baud rate of 9600
  This board's memory will reside at 0x1000000 on the VME bus
  After board is reset, start-up code will wait 3 seconds
------------------------------------------------------------------------
(M)odify any of this or (C)ontinue? [M]
```

Figure 5-7 pROBE+ Startup Message for Ethernet

Press any key to continue. No pROBE+ prompt appears because pROBE+ is now waiting for a connection from XRAY over the network.

5.4.3 Invoking XRAY on the Host for Ethernet

Invoking XRAY Debugger for use over Ethernet is similar to using XRAY over a serial channel, except that, following the **-e** option, you specify a **p** rather than an **r** and then enter the host IP address. For example, if the target system IP address is 199.9.999.9 as in Figure 5-7, the format of the command is as follows:

UNIX:	MS-DOS:
xp -e p199.9.999.9 ram.x &	**xp -e p199.9.999.9 ram.x**

A sign-on screen appears while XRAY is initializing. On some hosts you can define an alias name to represent the IP address of the target. This name can then be typed in place of the actual IP address.

NOTE: MS-DOS does not support alias names.

5.5 Starting the Downloaded Operating System

Before starting the pSOS+ kernel, verify or note the following:

- The executable image has been downloaded to the target but has not been started, as described in the preceding sections.
- The Code viewport may show the message “Not in Source Program” because the current PC is not within the executable image. Once you begin execution, this viewport normally contains the source code for the currently executing task.

You can start or boot the operating system using the **osboot** command or by using the assembly-language mode, as described in the following sections.

5.5.1 Using the osboot Command

Now you pass control to the operating system in the executable image by using the XRAY **osboot** command. The **osboot** command causes XRAY Debugger for pSOSystem to do the following:

- Pass control to the specified address.
- Terminate its connection to the pROBE+ in ROM.
- Establish a new connection with the pROBE+ debugger in the downloaded code.

If, for example, the start address of the downloaded operating system is 0x28008, enter the following:

```
osboot 28008
```

For more information about the start address, see Chapter 1, "Introduction to the pSOSystem Environment."

After a short delay, the following message appears:

```
BOOTING COMPLETE
```

This indicates that the downloaded operating system has successfully started, and the downloaded pROBE+ debugger is connected to the XRAY debugger.

5.6 Running the System Debug Mode Tutorial

At this point, the **xraydemo** executable image should be running on the target and communicating with XRAY Debugger for pSOSystem on the host. Before you begin, note that the following rules apply to all XRAY commands:

- Commands can be in either uppercase or lowercase.
- Arguments such as task names, routine names, and module names are case-sensitive and must be entered as shown.
- [Control-c] aborts the current command and returns to command mode.
- A # identifies a line number within the source code.
- Integer constants are interpreted as decimal by default but can be specified in hexadecimal by preceding them with a 0x or 0X.
- All commands are terminated by pressing [Return].
- Some commands have a corresponding function key. Refer to the *XRAY Debugger for pSOSystem User's Guide* for a list.

Your actual program output may differ slightly from that shown in this section for the following reasons:

- The board may have a different memory map, and/or the executable image may load at a different address.
- Using a serial channel for host-to-target communication instead of pNA+ can result in different output.
- Different CPUs have different register sets.

Also, note that the memory-related command examples in this section are based on an on-board RAM starting address of 0, and this is true of most boards. For a few boards, on-board RAM begins at another address, so the addresses in the commands must be adjusted. For example, if the on-board RAM begins at 0x4000000 and a tutorial command contains the address 0x1FF00, then you should enter 0x401FF00. For the start address of on-board RAM on each supported board, see Appendix A, "Board-Specific Information."

The XRAY Debugger for pSOSystem provides online help for all XRAY commands, command arguments, and keypad keys. Access the online help menu by entering:

```
>help
```

Use the cursor key to move the cursor down to the **breaki** (for break instruction) command and press [Return]. (The **breaki** command is on the second help screen.) You can also display the **breaki** command help by entering the abbreviated form of **breaki** (the letter **b**):

```
>b
```

The help utility displays the command name, its abbreviation in parentheses, the command syntax, and the command description. Press the [Esc] key to return to command mode.

5.6.1 Memory Manipulation and Viewports

This section describes XRAY commands that let you examine and modify memory. Begin by examining the first 64 vectors in the vector table:

```
>dump/l 0..0xFF
```

The **dump** command requests a hex display of memory. The **/l** displays memory in long words. **0..0xFF** specifies the first 256 bytes of target memory.

The XRAY screen is divided into several viewports. Each viewport has two sizes. The larger size is called the *zoomed* size, and the smaller is the *unzoomed* size. Notice that the Command viewport now fills the entire screen because it needed to expand to display the results of the **dump** command. The **zoom** command can be used to toggle the active viewport between its zoomed and unzoomed size. Try this command now to shrink the viewport:

>zoom

Each viewport displays information specific to its associated name (for example, the Break viewport displays breakpoint information). The *active* viewport is the one with a highlighted border. Right now this is the Command viewport. XRAY supports function keys that allow you to change the active viewport. On most keyboards these are the F1 and F2 keys. [F2] advances the display to the next viewport, and [F1] returns the display to the previous viewport. To determine which keys are used on your keyboard, refer to the *XRAY Debugger for pSOSystem User's Guide*. Cycle through the viewports by pressing [F2] (or equivalent key) five times.

In some cases, more viewports may be available than are displayed on the screen. For example, the Break viewport is not currently being displayed. For a list of the predefined viewports, refer to the *XRAY Debugger for pSOSystem User's Guide*.

Each viewport has a number, appearing in the top-right corner of the viewport. For example, the Command viewport is 1, and the Trace viewport is 4. You can specify a nonactive viewport by entering a viewport number in a command. For example, expand the Trace viewport by entering:

> zoom 4

Returning now to memory-examine and memory-modify commands, fill an area of memory with 0x12, as follows:

> fill /b 0x1FF00..0x1FFFF=0x12

The **/b** directs the **fill** command to operate on byte (8-bit) elements. Each byte in the range 0x1FF00 to 0x1FFFF is now 0x12.

This address range can vary from board to board, even in this tutorial. Verify that the range is valid by checking the appropriate board section in Appendix A, "Board-Specific Information."

Now set one byte in this range to a different value:

```
> setmem 0x1FFF0=0
```

This sets the byte at location 0x1FFF0 to 0. Search for it by entering:

```
> search 0x1FF00..0x1FFFF=0
```

XRAY Debugger for pSOSystem should report:

```
Matched: 001FFF0
```

Return to a normal screen display by entering:

```
> zoom
```

5.6.2 Starting the pSOS+ Kernel

The XRAY **restart** command sets a breakpoint at the first instruction of the pSOS+ ROOT task and passes control to the pSOS+ startup entry point. To initialize the pSOS+ kernel for the sample application, enter:

```
> restart
```

For a typical example of an XRAY Debugger for pSOSystem screen, see Figure 5-8.

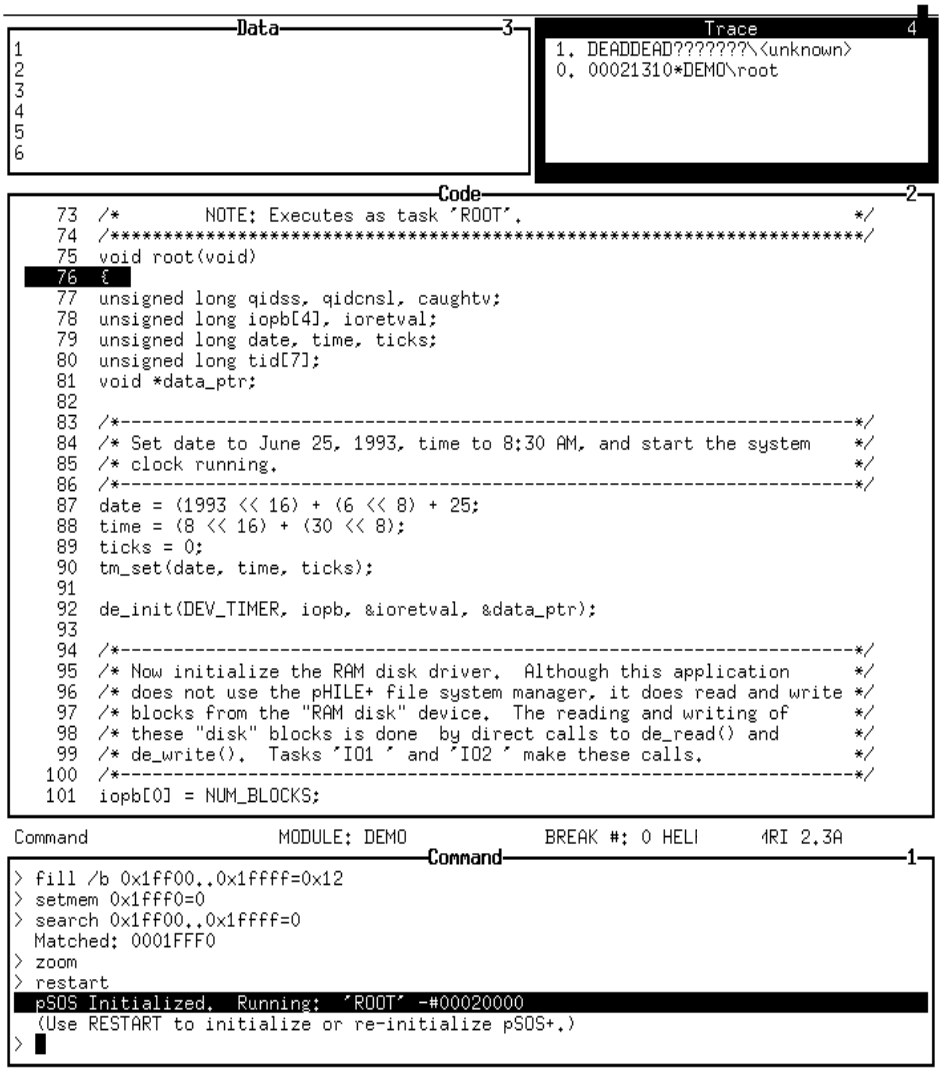


Figure 5-8 Sample XRAY Debugger for pSOSystem Screen

The Command viewport displays the following message:

```
pSOS initialized. Running 'ROOT' - #00020000
```

This identifies ROOT as the current running task and 0x00020000 as the task ID. The Code viewport is now displaying source code for the ROOT task. XRAY Debugger for pSOSystem has highlighted the opening brace of the ROOT task, which is the current point of execution. (When control is just entering a procedure, XRAY highlights the opening brace.)

The Trace viewport displays a procedure call traceback for the running task. In this case, *task* ROOT is about to execute an instruction in *procedure* **root** in module DEMO. The bottom of the traceback contains a value of 0xDEADDEAD which is a stack underflow sentinel placed there by the pSOS+ kernel when the task was created.

5.6.3 High-Level Mode and Assembly-Language Mode

XRAY supports two language modes of debugging: the high-level mode and the assembly-language mode. In high-level mode, you debug code at the C++ language level, and the Code viewport shows the C/C++ language source code. In assembly-language mode, you debug at the assembly-language level, and the Code viewport shows assembly-language code. In general, the command syntax in the two modes is similar, but the behavior of the commands is somewhat different.

Initially, XRAY is in high-level debug mode, and as you have not yet changed it, this is the current mode. When in high-level mode, a single step command allows you to execute one or more C/C++ language statements. Execute statements one at a time by entering the **step** command:

```
> step
```

Notice that the highlighted line moves down. This is because you are single-stepping lines of executable code. The complexity of the code determines whether XRAY requires more than one step to complete a single line. In some cases, XRAY may appear to execute several lines of code with a single step. This is a result of compiler optimizations.

Repeat the **step** command until the line containing the subroutine call **tm_set()** is highlighted (you may need to do this only once). In high-level mode, the **step** command ordinarily steps into a called subroutine. The **step** command does not step into an assembly-language instruction in

high-level mode, but rather it treats the assembly subroutine as an atomic operation and highlights only the high-level call. Single step again:

> step

Repeat until you are on **de_init()**. This is another assembly routine.

Before you change to assembly-language mode, make the current line the first line in the Code viewport. You can do this with a **scope** command, as follows:

> scope #92

Other uses of the **scope** command appear later in this tutorial.

The **mode** command lets you change the mode from high-level to assembly-language. Switch to assembly mode by entering:

> mode assembly

Intermixed source and assembly language now appears in the Code viewport. The lines that begin with angle brackets (>>) show the high-level code that corresponds to the assembly code. The Stack viewport replaces the Trace viewport and shows the first few long words on the top of the stack. The Registers viewport shows the contents of critical CPU registers.

The highlighted instruction is preparation for a function call. Single step some lines either by entering the **step** command repeatedly or by pressing [F9] repeatedly. (On most keyboards, the F9 key corresponds to the **step** command. Refer to the *XRAY Debugger for pSOSystem User's Guide* to see which key is used on your keyboard.) To step into the routine, press [F9] (or equivalent key) until the instruction to call a subroutine is highlighted (**jsr** for 68K processors or **callx** for 960 processors).

Now press [F9] one more time. Notice that the **step** command causes a *step into* the routine. A **stepover** command is also available. When **stepover** encounters a subroutine call, it executes the call, and halts when the called subroutine returns. In high-level mode, single-stepping occurs one source line at a time. In assembly-language mode, single-stepping occurs one microprocessor instruction at a time. Now return to high-level mode:

> mode high

The **go** instruction resumes execution of the application and optionally allows you to set a temporary breakpoint. The application continues until it encounters a break condition (if specified). For example, to resume execution with a temporary breakpoint at line #102, enter:

> go #102

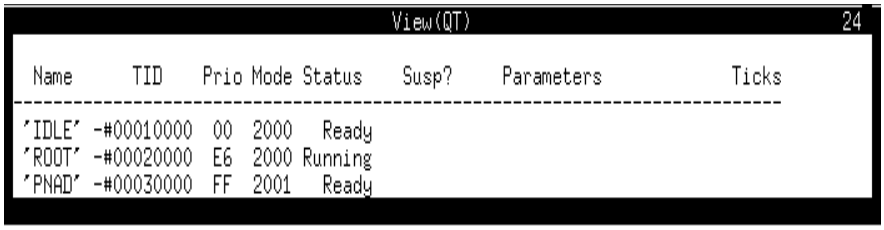
The XRAY debugger should report a break at line #102.

5.6.4 Queries and Breakpoints

The XRAY debugger offers direct access to a subset of pROBE+ commands. The majority of these commands are query commands. Refer to the *XRAY Debugger for pSOSystem User's Guide* for a complete list. Query the status of all active tasks by entering:

> qt

The View (QT) viewport displays the following:



Because you have just begun execution of the ROOT task, only the three tasks automatically created by pSOS+ and pNA+ are present: IDLE, ROOT, and PNAD (pNA+ creates the PNAD task). If pNA+ is not included in your operating system, you will not see the PNAD entry.

If you allow ROOT to execute further, it creates many more tasks. You could stop ROOT by setting a breakpoint at a statement further along in the code, but XRAY Debugger for pSOSystem supports more sophisticated types of breaks. For example, XRAY allows you to halt execution when a task makes a particular pSOS+ service call. Note that ROOT calls **ev_receive()** after creating all the other tasks and queues. Furthermore, you can use the **breakcomplex** command (**bc**) to halt execution when ROOT calls **ev_receive()**. The **bc** command accepts

options and prompts for parameters as required. Enter **bc** with the option **se** to indicate a break on a service call:

> bc se

XRAY Debugger for pSOSystem zooms the Command viewport when displaying the service calls and prompts for a pSOS+ function name. You can enter a particular pSOS+ function, or enter * to indicate any function. Enter **ev_receive** when XRAY prompts for a function, as follows:

Function: ev_receive

XRAY Debugger for pSOSystem is now prompting you to identify the call origin — either a task or an *interrupt service routine* (ISR). You can enter a particular task name or task ID, ISP, or * to indicate any task. In this case, enter ROOT when XRAY prompts for an origin (the single quotes around ROOT are required):

Origin: 'ROOT'

XRAY Debugger for pSOSystem confirms the break definition by displaying a complete list of all active breakpoints in the Break viewport. In this case, only the one service break is set (which is ROOT making the **ev_receive()** call). Return the Command viewport to normal size and resume execution:

> zoom

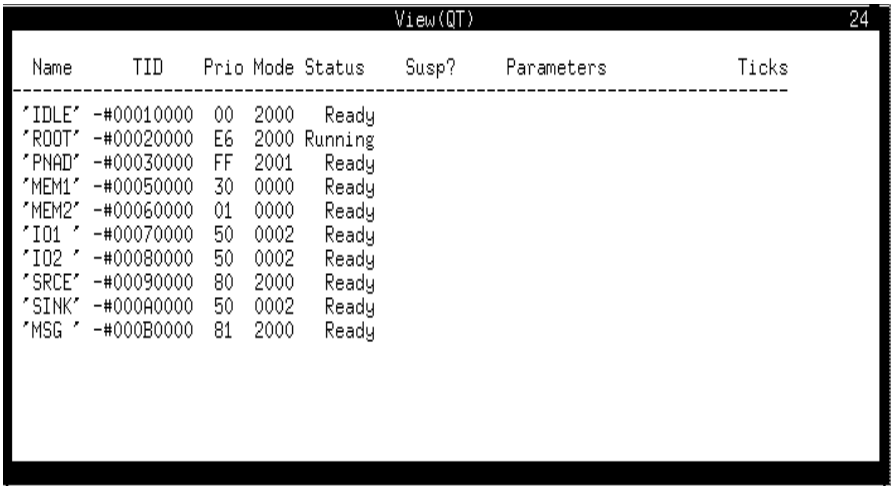
> go

Execution stops because ROOT has made an **ev_receive()** call. XRAY Debugger for pSOSystem reports that a service break was encountered, the type of call made, and the address from which the call was made. The Trace viewport shows that the call originated in the C source file **psos.s**.

At this point ROOT has completed spawning and activating the other tasks. The **query task** command (**qt**) can be used to view these tasks:

> qt

The View (QT) viewport displays the following:



ROOT is about to block, and all the other tasks are ready to run. It is useful to halt execution each time a new task executes. This can be done with a different type of complex breakpoint called the *dispatch break*. A dispatch break can be used to halt execution whenever a specified task begins executing. A wild card can also be specified, causing a break at all context switches. Define a dispatch break on any task, zoom the viewport, and resume execution by entering the following:

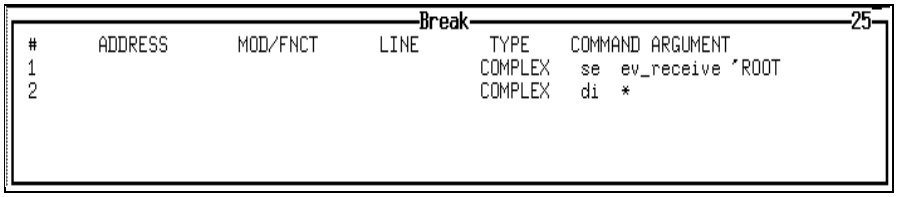
```
> bc di *
> zoom
> go
```

Execution stops because task PNAD is about to run. This is expected because, next to ROOT, PNAD has the highest priority. The display shows the last running task, the reason it stopped running, and the task about to run. Enter **go** three more times and observe which tasks execute.

The **clear** command is used to clear breakpoints. To see how many breakpoints are set, use the **breaki** command (**b**) without parameters:

```
> b
```

The Break viewport displays the following:



The service break and dispatch break are set. Breakpoints are cleared by specifying the breakpoint number or a range of numbers to clear. Thus, to clear the two breakpoints that are set, enter the following:

> clear 1..2

It is useful to examine the system after most tasks have been blocked. Because the IDLE task has the lowest priority (0), it runs only when all other tasks are blocked. Set a dispatch breakpoint to stop execution when IDLE executes by entering:

> bc di 'IDLE'

To return to a normal screen display and resume execution, enter the following commands:

> zoom
> go

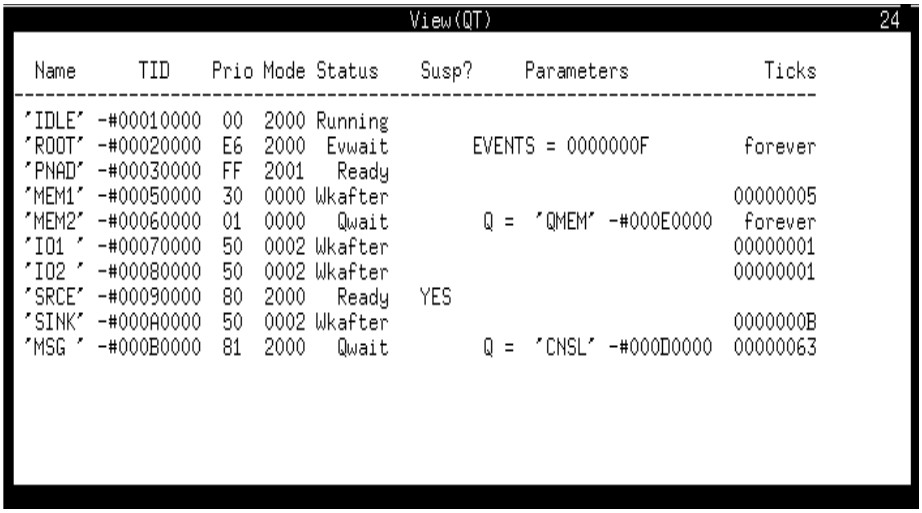
Execution stops because task IDLE is about to run. The Code viewport displays the following message because the code for task IDLE is part of the pSOS+ kernel and is therefore unknown to the XRAY debugger:

```
Not in Source module.
```

Check the state of other tasks by entering:

> qt

The View (QT) viewport displays the following:

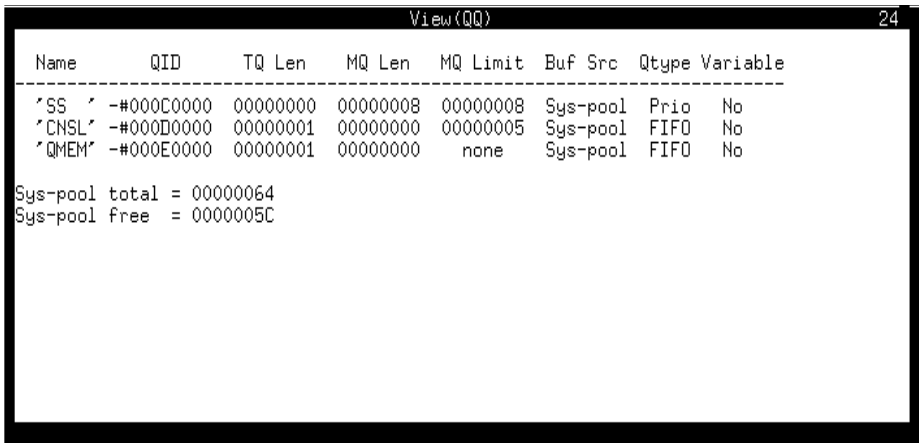


Notice that in the Status column, IDLE is running, and all other tasks besides PNAD are either paused, suspended, waiting for events, or waiting for messages. PNAD was blocked when the break occurred, but use of the network communication link since that time has changed it to the ready state.

The **qt** command is used to examine tasks. Other query commands are available to examine other pSOS+ objects. For example, examine all queues in the system by entering:

> qq

The View (QQ) viewport displays the following:



The system has several queues. The first queue is 'SS••'. It uses priority queuing and currently has eight messages pending. The second and third queues are CNSL and QMEM. They use FIFO queuing, and each has one task waiting. The system also has a queue for each open serial port for the DITI.

In addition to line numbers, breakpoints can also be set on procedure names. Resume execution and set a temporary breakpoint on the procedure **process_data** by entering the following (after you remove the dispatch break and shrink the viewport):

> clear 1
> zoom
> go process_data

Execution should halt because **process_data** has been called.

5.6.5 Symbols and Variables

XRAY commands allow you to examine and modify variables and symbols that the program uses. The **printvalue** command (**p**) is used to print the value of a variable. The **process_data** routine uses a global variable **Index**. To examine it, enter the following:

> p Index

XRAY Debugger for pSOSystem may display the error message **`At start of procedure, no local variables yet`**. If so, step into the procedure and try again:

```
> step
> p Index
```

The Command viewport displays the value of this variable. The **mem1** routine has a local variable **ticks**, and you can examine a local variable by entering:

```
> p ticks
```

When you try to examine **ticks**, the following error message appears because you are currently within the scope of the routine **process_data**:

```
Symbol not available from this scope without a qualifier
```

To examine variables in another routine, specify the routine's name:

```
> p mem1\ticks
```

Now the following error message is displayed because local variables are allocated on the stack when the routine is entered:

```
Local variable not alive
```

Each task has its own stack, and a task other than the one currently executing contains **ticks**. Thus, to examine **ticks**, you must be executing the routine **mem1**. To do this, use the **breaki** command (**b**) to set a breakpoint within **mem1**. Set the breakpoint at an instruction after the variable **ticks** has been initialized:

```
> b #195
> go
```

This causes execution to proceed until line #194 in the source code is reached. XRAY Debugger for pSOSystem then reports a break. Now examine the variable **ticks**:

```
> p ticks
```

The value of **ticks** is now displayed in the Command viewport. You can also examine arrays and structures with this command. The **mem1** routine contains an array called **addrmsg**. Examine it by entering:

```
> p addrmsg
```

The value of each array element is displayed in the Command viewport.

The **cexpression** (**c**) command is another command that can display a variable. Examine **ticks** by using **c**:

> c ticks

The Command viewport displays the value of **ticks** in both decimal and hexadecimal. This command is actually used to calculate the value of an expression or to assign a value to a variable. When the **c** command is used on an array or structure, it calculates the address of the specified item, which is then displayed. Try this command on an array:

> c addrmsg

Notice that the address of **addrmsg** is displayed. Contrast this to what was displayed when the **p** command was used on this variable.

The **c** command can also be used as a decimal-to-hexadecimal or hexadecimal-to-decimal converter. For example, enter the following to calculate the decimal value of this number and display it in the Command viewport:

> c 0x41

Because this number is also a printable ASCII character, it is displayed.

So far, you have seen symbols qualified by procedure names. In addition, XRAY allows you to qualify symbols by using task selectors. When you specify a symbol in a command, you can prepend it with the name of the task using the format *taskname:* (where ROOT is the *taskname* in the example line that follows) This instructs XRAY to use a particular context when evaluating a symbol. Use the **printvalue** command to examine a symbol in the ROOT task:

> p ROOT:\DEMO\root\tid[0]

This displays the variable **tid[0]** in the procedure **root** within the module DEMO. **tid[0]** exists within the context of the task ROOT.

The **monitor** command allows variables to be monitored. Monitored variables are updated each time the debugger stops executing the program. The monitored data is displayed in the Data viewport, but this viewport is not currently visible. You could zoom it, but another way to make this viewport visible is to use the **vactive** command (**va**). This command makes the specified viewport the active viewport:

> va 3

The procedure **mem1** has a variable **size**. The variable **size** contains a random number that is used to request memory segments of various sizes. Monitor this variable by entering the following:

```
> monitor size
```

This instructs XRAY Debugger for pSOSystem to monitor the variable **size** in the currently executing procedure **mem1**. You already have a breakpoint set at line #194, which is within the task MEM1 after it has initialized **size**. To resume execution, enter the following:

```
> go
```

The monitored variable is displayed in the Data viewport. Enter **go** three more times and observe how the monitored data changes.

The previously mentioned **cexpression** command can be used to assign a value to a variable. Change the value of **size** to 256 by entering:

```
> c size=256
```

The new value of **size** is displayed in the Command viewport and, because it is still being monitored, the Data viewport. In the example program, **size** is assigned a pseudo-random value, so keeping this value is acceptable.

The **nomonitor** command (**nomo**) disables the monitoring of a variable and removes it from the Data viewport. When a variable is monitored, it is assigned a monitor line number, and the Data viewport displays this number. These assigned line numbers refer to the expression to be removed. Notice that **size** is assigned line number 1. Stop monitoring the variable and remove the breakpoint by entering the following two commands:

```
> nomo 1
> clear 1
```

Another way to display the variable **size** is by using the **expand** command. This command examines the current stack and displays all the local variables it finds in each procedure. The Trace viewport is used in high-level mode to display the procedure calling chain. To examine the local variables, enter the following:

```
> expand
```

As the Trace viewport shows, only one level of calls exists at this point. **expand** also allows you to specify a particular stack level to examine.

So far, the viewports have displayed information for the current running task — MEM1. To look at a non-running task, use the **scope** command. **scope** is used to define a new default context other than the running task. All of the viewport displays, trace-back information, local variables, and so on, reflect information for the new default task. Examine the ROOT task after first returning to a normal screen display with the following two commands. (Note that a colon must follow the argument when the argument to the **scope** command is a task.)

```
> zoom
> scope ROOT:
```

The viewports now display information for the ROOT task. As expected, ROOT is waiting for an event and is therefore in the **ev_receive()** routine. The stack trace shows **psos.s**, which is the C language interface to pSOS+. If you enter **expand** now, it displays all the procedure calls and local variables for the ROOT task:

```
> expand
```

The **scope** command affects only the information displayed in the viewports. It does not modify the current execution state, and therefore MEM1 is still the current task. To return to the display of the current task, enter **scope** without parameters:

```
> scope
> zoom
```

The viewports now display information about MEM1.

Sometimes information other than its value is needed about a symbol. The **printsymbol** command (**ps**) provides this information. **ps** displays information about a specified symbol or group of symbols. This information includes the symbol name, data type, storage class, and memory location. To examine all symbols used in the procedure **mreader**, enter the following:

```
> ps mreader\
```

where the backslash is a *symbol qualifier convention*. For an explanation of the backslash and other symbol qualifier conventions, refer to the *XRAY Debugger for pSOSystem User's Guide*.

To display information about a specific symbol, enter the following:

```
> ps root\tid
```

This displays information about the variable **tid** in the procedure **root**.

5.6.6 Profiling

One unique feature of XRAY Debugger for pSOSystem is its profiling capability. By transparently collecting statistics on the application, XRAY can provide important insights into a design's activities and performance. To enable profiling, turn on the profile flag as follows:

```
> fl profile on
```

The Command viewport displays the following flag settings:

```
RBUG: ON
HOST: OFF
TRFR: ON
NODOTS: ON
NOMANB: OFF
NOPAGE: ON
ECHO: OFF
PROFILE: ON
```

For example, to profile the application for 30 seconds, use a timed break to halt execution after 3000 clock ticks, which is 30 seconds. By default, the pSOS+ configuration table, configured through the **sys_conf.h** file, specifies 100 ticks per second. Set the breakpoint and resume execution with the following entries:

```
> bc ti #&3000
> go
```

On the first line, the ampersand (&) indicates that the next number is decimal, and the pound sign (#) indicates that the next value is relative to a starting point. Otherwise, an absolute time and date would have been entered. Execution stops after the designated interval of 30 seconds. Normally, for a statistically significant profile, a much longer interval is necessary. To view the profile data, enter the following:

```
> lp
```

The values displayed are decimal. All of the profile data cannot fit in the View viewport, so use the up and down arrow keys to scroll through it.

The number of ticks listed for each task is statistical, because it depends on the sampling at each clock edge. The other data, such as the number

of times a task switches into a particular state and the number of system calls made, are actual counts.

To turn off profiling, enter the following:

> fl profile off

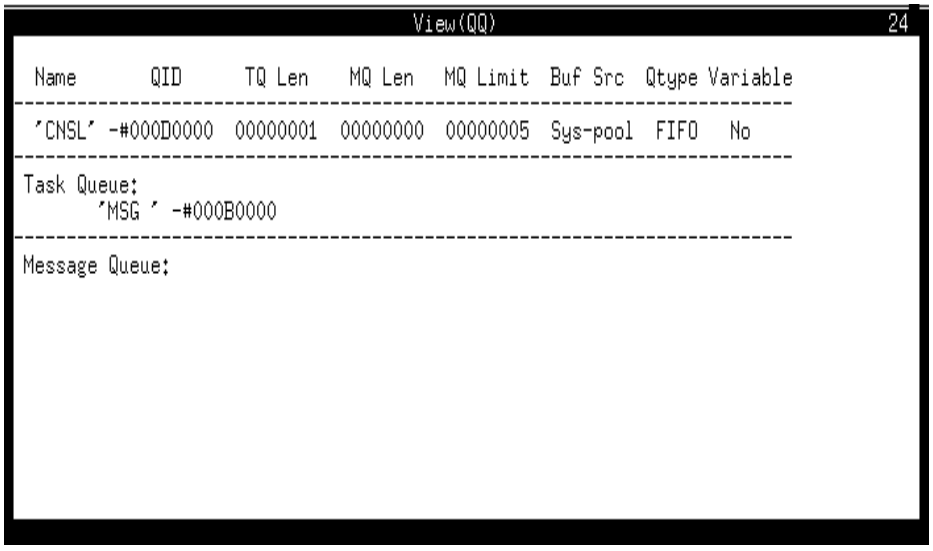
5.6.7 Interactive System Calls and I/O

XRAY Debugger for pSOSystem allows you to make system calls manually to pSOS+ so that you can conveniently modify and control the application's behavior. For example, the sample application has a queue named CNSL, to which you can manually send messages.

First, check the status of the queue CNSL by entering:

> qq 'CNSL'

The View (QQ) viewport displays the following:



The display shows that one task, 'MSG•' is running, and it is waiting for a message.

You can now use the **sy** command to send a message to CNSL. Because 'MSG•' is waiting, it receives the message and becomes ready to run:

> sy q_send 'CNSL' 'ABCD' 'EFGH' 'IJKL' 'MNOP'

In this example, you enter all the command parameters on a single line. If you are not sure what parameters the call requires, you can leave out the parameters and let XRAY Debugger for pSOSystem prompt for each parameter. When the command is done, the following pSOS+ system call return code is displayed:

```
Return code: 00000000
```

where all 0s indicates that the call succeeded.

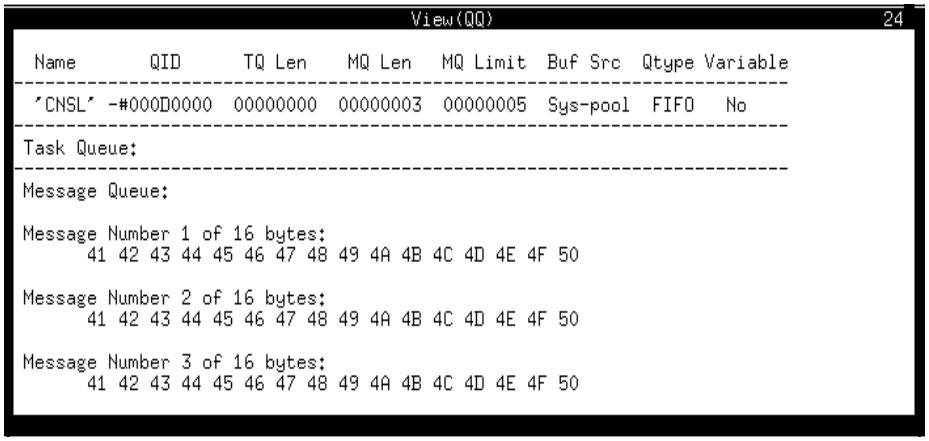
For sending more messages to a queue, XRAY supports a function key that backs up one command at a time, up to a maximum of five commands. On most keyboards, [F7] is this key (refer to the *XRAY Debugger for pSOSystem User's Guide* to determine which key is used on your keyboard). Each time you press [F7], XRAY backs up one command, but it does not execute the command until you press [Return]. For example, you can back up three commands by pressing [F7] three times, then execute the selected command by pressing [Return].

You can also repeat a preceding command three times by using the function key [F7] or equivalent and [Return] sequence three times.

Now look at CNSL again:

> qq 'CNSL'

The View (QQ) viewport displays the following:



Three messages are queued. (Remember that the first message was passed to the waiting task 'MSG•'.) Task 'MSG•' is ready to run, and

when you restart execution, 'MSG•' sends all four messages to the XRAY standard I/O screen.

Task 'MSG•' demonstrates another important feature of XRAY Debugger for pSOSystem, and that is the ability of a pSOS+ task to write to the XRAY terminal. XRAY provides an I/O screen through which application code can write to the XRAY terminal. The I/O screen is not a viewport but a separate screen. When the application does console I/O, the I/O screen temporarily replaces the high-level screen on the terminal. The application accesses the I/O screen by calling a routine in the file **db_output**.

When execution resumes, the I/O screen appears with the messages from task 'MSG•'. To resume execution, enter the following:

```
> go
```

The following should be displayed:

```
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
```

At this time, the sample application is continuing to run because no breakpoints have been set. Stop execution by entering a [Control-c]. The high-level screen now replaces the I/O screen and displays a message similar to the following:

```
Manual Break. Running: 'IO2 ' -#00080000
Stopped due to halt from user
```

Because execution has been stopped manually, the running task may not be 'IO2•'. Now that execution has stopped, the I/O screen displaying the messages from task 'MSG•' is no longer visible.

By going through this tutorial session, you have seen that XRAY has three predefined screens. Screen 1 is the high-level screen that is currently visible. Screen 2 is the assembly-language screen that was seen earlier in this tutorial session with the use of the **mode assembly** command. Screen 3 is the I/O screen where the messages from task 'MSG•' were displayed. Remember that a screen is not the same as a viewport.

The **vscreen** command is used to display a specified screen. To look at the output on the I/O screen, enter:

```
> vscreen 3
```

Now the I/O screen is displayed, and you again see the output that was displayed by task 'MSG•':

```
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
```

To get back to the high-level screen, display screen 1. While you type **vscreen 1**, the command is not echoed to the screen. This is because the command is echoed on the high-level screen, but the I/O screen is currently visible. When you press [Return] after typing the following, the high-level screen appears:

```
> vscreen 1
```

You have now completed the XRAY Debugger for pSOSystem tutorial. To terminate this debugging session, enter the following:

```
> quit
```

After you enter **quit**, XRAY prompts:

```
Are you sure?
```

Enter a **y** to terminate the session.

6 Shared Memory Multiprocessing Tutorial

This chapter takes you through the steps to build, download, and run a multiprocessing application. The pSOSystem sample multiprocessing application is called **mpdemo**, and it runs under pSOS+m.

The pSOS+m kernel allows tasks running on different processors to communicate, exchange data, and synchronize through pSOS+m as if they were running on a single processor. For example, a task on one node can use the **ev_send()** system call to signal an event to a task on another node.

The pSOS+m kernel is fully explained in the *pSOSystem System Concepts* manual.

6.1 Introduction

The **mpdemo** application can run on a target system containing any number of processors, but the tutorial presented in this chapter applies to a system with from two to eight processors.

For the **mpdemo** application, the pSOS+m kernel on a node communicates with other kernels through a software layer called the *Shared Memory Kernel Interface* (SMKI). To the application, this communication is transparent. Typically, 68K-based shared memory

systems use the VMEbus, although any hardware architecture that allows different CPUs to access a common memory area can be used. This chapter includes examples of VME-based hardware usage and helpful VME-specific information.

NOTE: This tutorial assumes you have read Chapter 2, "pSOSystem Tutorial for Workstation Hosts," and performed the tutorial. Concepts and procedures demonstrated in other tutorials are not repeated here.

6.2 mpdemo Sample Application

Directory **PSS_ROOT/apps/mpdemo** contains the multiprocessing sample application upon which this tutorial is based. The **mpdemo** application illustrates the use of kernel calls across node boundaries and the soft-fail and rejoin capabilities of pSOS+m. Directory **mpdemo** contains the following files:

makefile callout.c common.h output.c tasks.c

callouta.s drv_conf.c sys_conf.h

Figure 6-1 on page 6-3 shows the logical flow of the application once it is fully initialized. Node 1 is the master node, which must stay up and running. In the VMEbus environment, this means the master node must be the System Controller.

The same code can be downloaded and run on all nodes in the system because the application can determine the node on which it is executing by reading the node configuration table. Except for the following, the application code is the same on all nodes:

- Queue 'SRVq' and task 'SRVt' are created only on node 1.
- The ROOT task on node 1 does not execute a **k_fatal()** call because node 1 is the master node and is not allowed to fail.

The sequence executed by a client is shown in Figure 6-1.

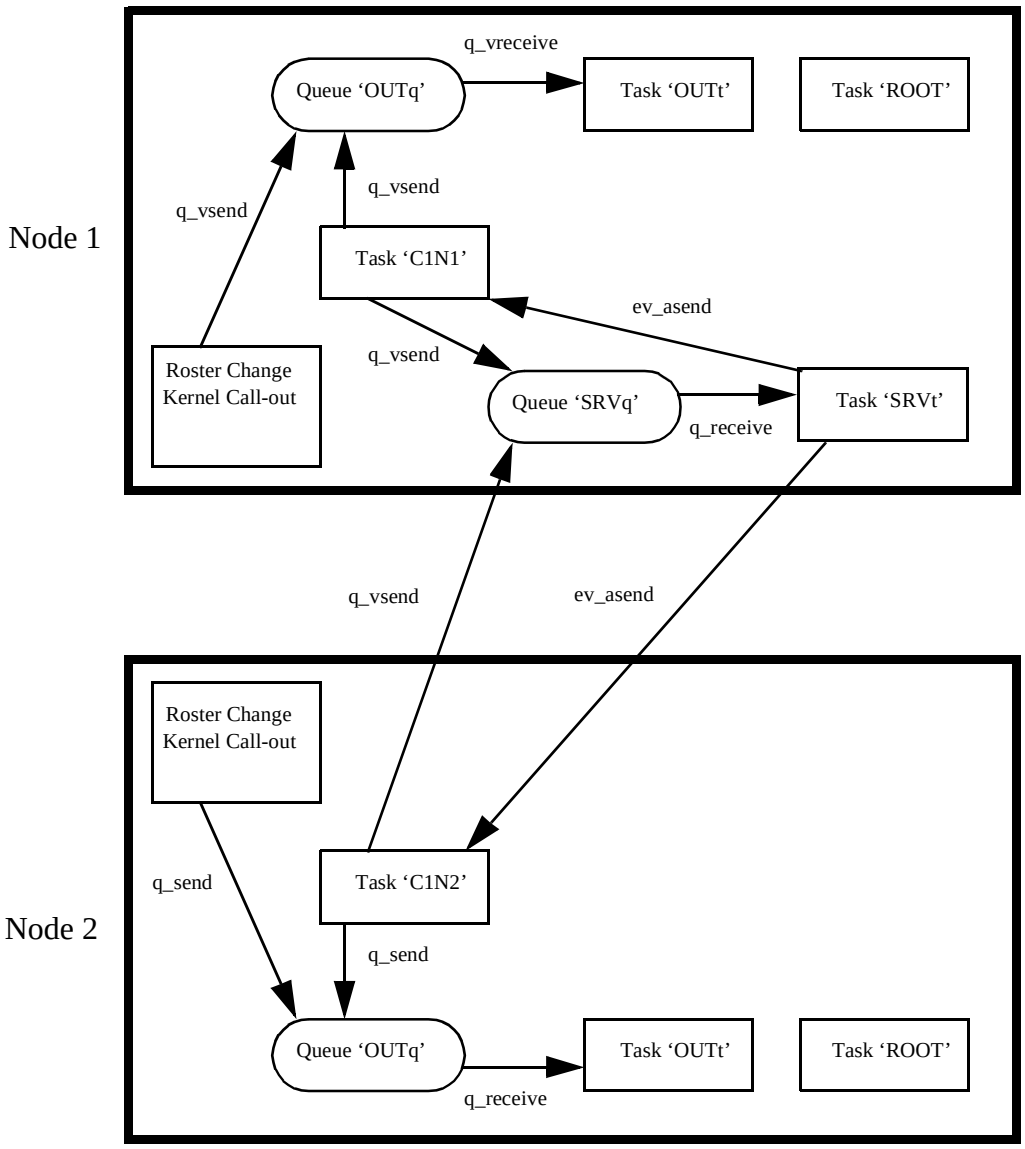


Figure 6-1 mpdemo Sample Application

6.2.1 Client Task Execution

For **mpdemo**, each node in the system contains several *client tasks*. All the client tasks execute the same code. The code executed by each client task is called **ClientTask()**. This code is an infinite loop located in **tasks.c**. The sequence executed by a client is as follows:

1. A message containing the client task's ID is sent to queue 'SRVq' on node 1.
2. The task sleeps and awaits the signalling of an event.
3. When an event is signalled to a client task by an **ev_asend()** call from task 'SRVt', the task wakes up and sends a message to queue 'OUTq'. The message contains the client task's ID and the iteration number of the loop.
4. The client task sleeps for a random amount of time.
5. The client task wakes up and repeats the loop.

The number of client tasks per node is controlled by the NUM_CLIENTS macro in **tasks.c**. Each client task described in **tasks.c** has a name in the form 'C*n*N*m*', where *n* is a number from 1 through NUM_CLIENTS and *m* is the number of the node on which the task resides.

6.2.2 Server Task Execution

Message queue 'SRVq' and server task 'SRVt' exist only on node 1 because they must always be present (and node 1 is always present). Messages posted to 'SRVq' are subsequently received by server task 'SRVt'. Each message posted to 'SRVq' is four bytes long and contains the ID of the client task that sent it.

The code for task 'SRVt' is called **ServerTask()**. This code is an infinite loop located in **tasks.c**. The sequence executed by 'SRVt' is as follows:

1. 'SRVt' blocks while it awaits a message from 'SRVq'.
2. When it receives a message, 'SRVt' wakes up.
3. 'SRVt' uses the **ev_asend()** service call to signal an event to the client task identified in the message.
4. 'SRVt' returns to waiting for a message from 'SRVq'.

6.2.3 Console Output

Each node in the system has a queue/task combination called 'OUTq' and 'OUTt' for sending output to the serial channel. This code is contained in **output.c**.

Messages posted to queue 'OUTq' are received by task 'OUTt', and the message contents are sent out on the serial channel. The messages posted to 'OUTq' come from one of two sources. Most of the messages come from the client tasks. These messages announce how many times the client tasks have completed their loops. The other messages are posted by a *roster change callout routine* (contained in **callout.c**). The pSOS+m kernel calls the roster change callout routine when a node has left or joined the system. This routine posts a message about the roster change to 'OUTq', so that notification of the change can go out over the serial channel. When nodes other than just the master node are running, **mpdemo** posts roster change messages to the terminal.

6.2.4 Soft-Fail and Rejoin

To simulate a node failure, the ROOT task on a node other than node 1 periodically executes a **k_fatal()** service call (ROOT on node 1 does not execute a **k_fatal()** call). First, ROOT sleeps for a random time period after it has completed system startup duties such as device initialization and creation and startup of other tasks. When ROOT wakes up, it executes a **k_fatal()** service call, and this causes pSOS+m to send notification of its failure to the master node. pSOS+m then calls a fatal error handler located in **callout.c**. This fatal error handler executes a software spin loop and then restarts the system by transferring control to the startup entry point of the pSOS+m kernel. In the meantime, pSOS+m on the master node removes the node from the system roster.

6.3 Planning the Target System

A well-planned configuration for the target system can greatly reduce the time required to bring up and run a multiprocessor system. This section explains how to plan the configuration.

6.3.1 Assigning Node Numbers

Each node in the system must have a unique node number. Node numbers must be in the range 1 through 8.

Node 1 must be present, and it must initialize first. This is a requirement of both the pSOS+m kernel and the SMKI. For **mpdemo**, you can assign any available node number in the range 2 through 8 to other nodes.

6.3.2 VMEbus Memory Addresses

For this tutorial, all boards in the system must have dual-ported memory accessible by both the local CPU and other processors over the VME system VMEbus. The address of the memory on the bus is usually configurable either through software or by jumpers on the board. The following rules apply to the bus addresses of the boards:

- Each processor's dual-ported memory must be assigned a unique starting address.
- No overlap of memory addresses between different boards is allowed.
- The dual-ported bus address selected for each board must be accessible by all of the other boards in the system. This usually precludes placing a board's memory at location 0 on the bus because most boards map their own memory beginning at 0 and are thus unable to access VMEbus address 0.

6.3.3 Selecting a Directory Address

The SMKI requires a *directory* structure. This directory structure must reside in a memory space where all of the nodes in the system have access to it. Each board-support package provided by Integrated Systems reserves a memory area for the SMKI directory. To find the offset of this area for each supported board, see Appendix A, "Board-Specific Information." Integrated Systems recommends that you use this area on one of the boards in the system to locate the SMKI directory.

6.4 Setting Up the Hardware

The following rules apply when multiple bus masters operate in a VMEbus system. This information should save time when you bring up a multiprocessor VMEbus system. Consult the user manuals for the VME boards you use to perform the necessary setup. The reference for the terminology and rules listed here is the VMEbus specification.

- The board designated as the System Controller should be placed in slot 1 in the chassis (farthest to the left). A system may work if the System Controller is in another slot, but not all VMEbus services will be available because slot 1 is wired differently from the other slots. Of the slots actually used, the System Controller must be the farthest on the left whether or not that position is physically slot 1 in the chassis. Otherwise, the system will not work. For example, the System Controller can occupy slot 3 as long as all other boards are to the right of it.
- The System Controller bus arbiter must be *enabled* and the other board arbiters *disabled*. This must be checked carefully because if more than one arbiter is enabled, the system might not fail immediately. Instead, it may perform erratically. Typically, the bus arbiter is enabled by a jumper.
- Bus master boards must have compatible bus arbitration modes. If you specify single-level requests, make sure all boards are set to make bus requests at level 3. A board's bus arbitration mode can be set by either jumpers, control registers, or a combination of both. The manufacturer of the VME board determines the method of setting the arbitration mode.
- Only the System Controller should drive the SYSCLK signal, and this is usually determined by a jumper setting. If more than one board drives SYSCLK, contention may result.
- Physical continuity must exist between boards in the system. To ensure this, either empty slots cannot be present between boards or the four daisy-chained bus grant lines and the daisy-chained interrupt acknowledge line must be jumpered across unused slots.

If you have not already done so, install pSOSystem Boot ROMs in each of the target boards. You should also connect serial cables from each of the target nodes to the host system or terminal.

6.5 Testing the Hardware

Once the target boards are installed, they should be tested to ensure that each board can read and write the dual-ported memory of the other boards in the system. You can do this for each board by using ROM pROBE+ to modify a memory location in a board's dual-ported memory and then reading that memory location from each of the other boards. The process requires the following steps:

1. Configure the Boot ROMs on each board to operate pROBE+ in stand-alone mode. In the dialog questions for each board, when you see the prompt "Bus Address of this board's dual-ported memory," be sure to enter the address determined for the board in Section 6.3.2, "VMEbus Memory Addresses."

2. On node 1, use the pROBE+ PM.L command to store a unique value in an unused longword of dual-ported memory. The memory where the pSOSystem executable image will be loaded ($20000 on most boards) is a good place. For example:

   ```
   pROBE+>PM.L 20000 457EE222
   ```

3. From each of the other nodes in the system, use the pROBE+ PM.L command to view the memory you modified in step 2. For example, if node 1 is at $30000000 on the VMEbus, you would use the following command to display the relevant portion of node 1's dual-ported memory:

   ```
   pROBE+>PM.L 30020000
   ```

   This would show a value of 457EE222 at $30020000.

4. Repeat steps 2 and 3 for each node in the system.

6.6 Creating a Working Directory

The first steps for using **mpdemo** require the creation of working directories. To illustrate a multiprocessor system with more than one type of target board, in this **mpdemo** tutorial you initially will create two working directories for a two-node target system. In multiprocessor systems that have only one type of board, only one working directory is actually necessary. Even if you have only one type of board, proceed as if you are using different types. Start by doing a recursive copy in the directory of your choice by entering the following:

```
cp -r $PSS_ROOT/apps/mpdemo  node1
```

where **node1** is the working directory for the master node's code. Next, make another working directory:

```
cp -r $PSS_ROOT/apps/mpdemo  node2
```

where **node2** is the working directory for the second node in the system.

6.7 Building, Downloading, and Starting the Executable Images

In each of the working directories, the **makefile** requires editing to reflect the board used for each node, and these boards preferably are those supported by Integrated Systems with a board-support package. Set PSS_BSP to indicate the board used for each node, as in the following example:

```
PSS_BSP=$(PSS_ROOT)/bsps/m167
```

Next, build the executable image, as follows:

```
make ram.hex
```

6.7.1 Configuring and Downloading to Node 1

The next phase requires configuration of the Boot ROMs. When the host system is physically connected to the target and you start up the terminal emulator (such as **tip** or **cu**), the startup dialog is displayed on the terminal. After you enter **m** for 'Modify' at the startup dialog prompt, select the following:

- Enter **1** for pROBE+ stand-alone mode.
- No LAN interface.
- No shared memory network interface.
- No default gateway.
- Enter **y** when the prompt for starting the multiprocessing configuration appears.
- Specify 8 nodes (as a maximum) even though this tutorial begins with a two-node system: this allows you to try specifying different node numbers later.
- Specify that the current CPU is (node) 1 because the system must have a master node, and this is the first CPU configuration you are performing on the target.
- The SMKI address should already have been determined through the use of information in Appendix A, "Board-Specific Information."
- The baud rate can be whatever hardware-supported rate you choose. You may need to change the terminal emulator rate to reflect your changes.
- The wait period can be whatever you want.

After you have completed the configuration, the pROBE+ prompt appears with a number appended that reflects the node number (1). Download the **ram.hex** file with the pROBE+ **dl** command. When the download is complete, transfer control to the downloaded executable image with the **go** command (as done in Chapter 2, "pSOSystem Tutorial for Workstation Hosts").

6.7.2 Configuring and Downloading to Other Nodes

The ROM configuration for node 2 is the same as node 1 except for the node number, which should be 2. When you download **ram.hex** to node 2, be sure it is the **ram.hex** file from the **node2** working directory. Transfer control to the downloaded image by entering the **go** command.

6.8 Running the Sample Application

You must start node 1 first by entering the pROBE+ **gs** and **go** commands. After the sample application starts on node 1, the terminal displays messages related to the client task running on node 1. You can do a manual break to return control to pROBE+ and examine the state of the system. For example, enter **qt** to list the tasks present on the node or **qo** to list the objects present. Be sure to resume the application with the **go** command.

Now start node 2 with the **gs** and **go** commands. After you start the application on node 2, the display for node 1 shows that node 2 has joined the system. This is a message describing the roster change and includes a *sequence number*, which is incremented every time a node joins the system.

The display for node 1 continues by showing when pSOS+m on node 2 signals a failure and when the failed node rejoins the system.

You can stop the nodes with manual breaks, by pressing a board's abort button (if present), or by turning off the system. This concludes the multiprocessing tutorial.

7 Configuration and Startup

The configuration of pSOSystem is easily changed by editing the **sys_conf.h** file. Entries in **sys_conf.h** control many system parameters, and these range from specifying the device drivers that are built into the system to the maximum number of tasks that can be active concurrently. The **sys_conf.h** file resides in the application directory. This chapter documents the **sys_conf.h** system parameters**.**

As pSOSystem is a scalable operating system, a large part of its functionality is contained in software components or building blocks. Not all of the software components are necessarily built into a pSOSystem configuration, and this depends on the capabilities you require. Each software component has its own configuration and startup requirements, and they are taken care of by the pSOSystem startup code. These requirements are documented here to help you make changes to the startup code, if necessary.

7.1 Overview

Software components, such as pSOS+, pROBE+, and pREPC+, are the basic building blocks of the pSOSystem environment. Associated with each component is a *configuration table*, which the corresponding component uses to obtain its configurable parameters. Figure 7-1 on page 7-2 shows the relationships between the various elements that the

components use to find their parameters, data areas, and other configuration information.

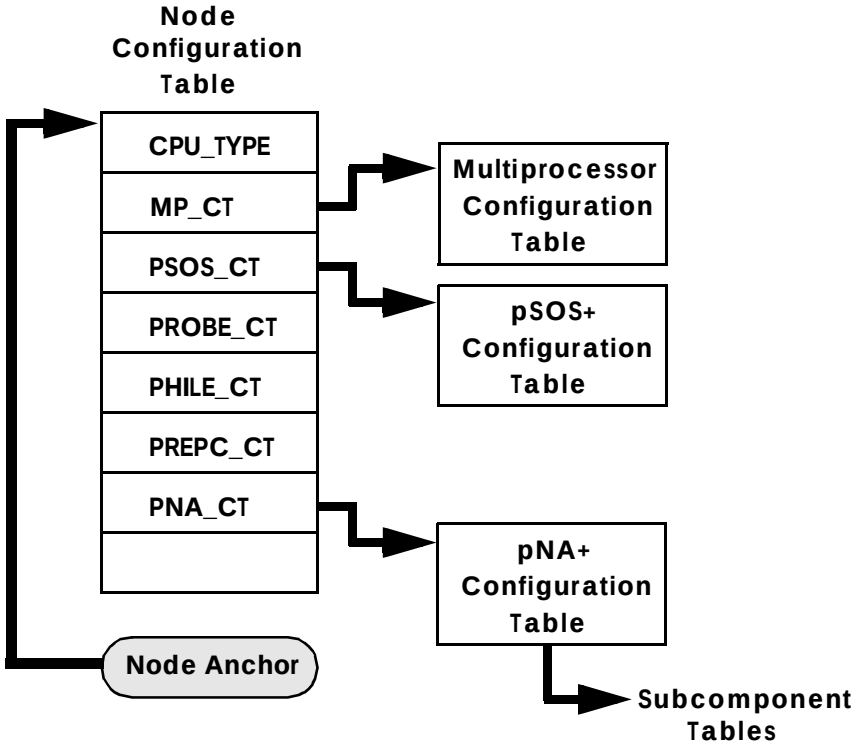


Figure 7-1 Configuration Tables

The Node Anchor is the single, fixed point of reference for all the installed software components in the system. This anchor is a critical link because each component is code and data position-independent and thus depends on the anchor to locate its configuration information.

In pSOSystem, the location of the node anchor is determined by the symbol *_anchor*. The value of *_anchor* is defined in the linker command files that are supplied with each board-support package.

The "Configuration Tables" section of the *pSOSystem Programmer's Reference* explains each of the configuration table entries. Because the

pSOSystem startup code itself sets up these tables, you do not initially need to understand all the table entries. However, as you experiment with changing the pSOSystem configuration, the information in "Component Configuration Parameters" on page 7-10 becomes more useful.

7.1.1 System Configuration File

During system startup, pSOSystem initializes all of the required component configuration tables. The code that initializes configuration tables resides in the shared, read-only file **sysinit.c** in the **PSS_ROOT/configs/std** directory. The source code in **sysinit.c** contains many lines where compilation depends on values defined in the application's pSOSystem configuration file, **sys_conf.h**. The following are determined by parameters defined in **sys_conf.h**:

- Which operating system components are built into the system.
- Characteristics of the target's serial channels.
- Whether a LAN driver is included in the system and, if so, its IP address.
- Whether a shared memory network interface (SMNI) is included in the system and, if so, its IP address.
- The optional device drivers in the system: SCSI and RAM disk drivers, the TFTP pseudo-driver, and any application-specific drivers you may have added.
- The values used for most of the component configuration table entries: for example, definitions in **sys_conf.h** determine the system's maximum number of concurrently active tasks and message queues.

Each of the sample applications supplied with pSOSystem includes a **sys_conf.h** file that contains configuration values appropriate for that application. Section 7.2, "sys_conf.h," explains the values you must define in **sys_conf.h**.

7.1.2 Parameter Storage and the Startup Dialog

Sometimes the only differences between pSOSystem configurations are the values of the parameters defined in **sys_conf.h**. For this reason,

pSOSystem allows you to place some of the **sys_conf.h** parameter values in a dedicated storage area in the target's memory. An optional *startup dialog* can be built into pSOSystem to allow review and possible modification of these parameters when pSOSystem initializes itself. The pSOSystem Boot ROMs are an example of pSOSystem and an application using the startup dialog.

7.2 sys_conf.h

This section describes the parameters that **sys_conf.h** must supply. Parameter definitions in **sys_conf.h** have the form of C macro definitions, as in the following example:

```
#define SC_PROBE YES   /* Include pROBE+ debugger */
#define KC_NTASK 20    /* Maximum of 20 active tasks */
```

You may find it helpful to refer to the example **sys_conf.h** files in the pSOSystem sample applications while reading this section.

To improve the readability of **sys_conf.h**, macros are used to define the values of some of the parameters. The code in **sysinit.c** (and other files that *include* **sys_conf.h**) assumes the use of these macros. The **types.h** and **sysvars.h** include files must be included in the **sys_conf.h** file to resolve parameter definitions.

7.2.1 Storage and Dialog Parameters

The following four parameters determine the fashion in which many of the other parameters in **sys_conf.h** are used:

Parameter	Possible Values
SC_SD_PARAMETERS	STORAGE, SYS_CONF
SC_STARTUP_DIALOG	NO, YES
SC_BOOP_ROM	NO, YES
SD_STARTUP_DELAY	Any decimal integer
SE_DEBUG_MODE	DBG_SA, DBG_XS, DBG_XN, SYS_CONF

The values of all of the parameters in **sys_conf.h** with names beginning with "SD_" can be determined either by the definitions given in **sys_conf.h** or by the data in the target's parameter storage area. If SC_SD_PARAMETERS is set to SYS_CONF, the values in **sys_conf.h** are always used for the SD_ parameter values. If SC_SD_PARAMETERS is

STORAGE, then pSOSystem attempts to use the values in the target's parameter storage area for the SD_ variables, and the values in **sys_conf.h** become default values to use if the parameter storage area either has not been initialized or has been corrupted.

If SC_SD_PARAMETERS is defined as STORAGE, you can enable the startup dialog by setting SC_STARTUP_DIALOG to YES. The startup dialog runs on the target system at startup time and allows you to view and optionally change the parameter values in the storage area. If the dialog is enabled, SD_STARTUP_DELAY specifies the number of seconds that the dialog waits for input before it boots the system.

SE_DEBUG_MODE determines how the system operates, as follows:

DBG_SA	Boot pROBE+ in stand-alone mode.
DBG_XS	Boot into pROBE+ and wait for the host debugger (system-level debugger) through a serial connection.
DBG_XN	Boot into pROBE+ and wait for the host debugger (system-level debugger) through a network connection.
STORAGE	Use the mode (DBG_SA, DBG_XS, or DBG_XN) found in the parameter storage area. If a valid mode is not found, then use DBG_SA.

7.2.2 Operating System Components

The following parameters determine which components go into the system:

Parameter	Possible Values
SC_PSOS	YES, NO
SC_PSOSM	YES, NO
SC_PROBE	YES, NO
SC_PHILE	YES, NO
SC_PREPC	YES, NO
SC_PNA	YES, NO
SC_PMONT	YES, NO
SC_PRPC	YES, NO
SC_PX	YES, NO
SC_PSE	YES, NO

SC_PTLI	YES, NO

A component parameter set to YES causes that component to be built into the system. Note that it is incorrect to set both SC_PSOS and SC_PSOSM to YES.

7.2.3 Serial Channel Configuration

The target's serial channels are referred to by their channel numbers, which start at 1. For example, a target with four serial channels would have channel numbers 1, 2, 3, and 4. The following parameters control the serial channels:

Parameter	Possible Values
SC_PROBE_CONSOLE	1, 2, 3, and so on
SC_PROBE_HOST	1, 2, 3, and so on
SC_APP_CONSOLE	1, 2, 3, and so on
SD_DEF_BAUD	4800, 9600, 19200, and so on

SC_PROBE_CONSOLE specifies the serial channel number that pROBE+ should use for its *console* channel. (The pROBE+ console displays output and receives commands.) If pROBE+ is to communicate with the XRAY host debugger over a serial channel, SC_PROBE_CONSOLE specifies the channel to use.

SC_PROBE_HOST specifies the serial channel number that should be used for the pROBE+ *host* channel. This is not normally required, so it can be disabled by specifying a 0. The host channel is explained in the *pROBE+ User's Guide*.

SC_APP_CONSOLE specifies the serial channel number used for the application's *console* channel. The console channel can be changed dynamically by making a **de_cntrl()** call to the serial driver.

When the pSOSystem terminal driver (**drivers/diti.c**) is called, the minor device number specifies the serial channel to use. For example, if the major device number specified by SC_DEV_SERIAL is 3, a **de_write()** call to device 3.3 writes the output on serial channel 3. Minor device number 0 is remapped to the application console channel, where the initial value is specified by SC_APP_CONSOLE. For a description of SC_DEV_SERIAL, see Section 7.2.7, "I/O Devices."

SC_DEF_BAUD specifies the default baud rate for the serial channels. A **de_cntrl()** call can be used to change the baud rate dynamically.

7.2.4 LAN Configuration

The following parameters control the configuration of the LAN interface:

Parameter	Explanation
SD_LAN1	YES enables the LAN interface, and NO disables it.
SD_LAN1_IP	IP address to use for LAN interface. Alternatively, SD_LAN1_IP can be set to USE_RARP, in which case pSOSystem uses RARP to obtain the IP address.
SD_LAN1_SUBNET_MASK	Subnet mask to use for LAN interface, or 0 for none.

If the target board has a LAN interface, you can enable it by setting SD_LAN1 to YES. If SD_LAN1 is NO, the values of the other SD_LAN1_* parameters are unused.

7.2.5 Shared Memory Configuration

The following parameters control the configuration of the shared memory network interface (SMNI):

Parameter	Explanation
SD_SM_NODE	Node number for this node
SD_NISM	YES to enable SMNI, otherwise NO
SD_NISM_IP	IP address of this node
SD_NISM_DIRADDR	Bus (global) address of SMNI directory structure
SC_NISM_BUFFS	Number of buffers
SC_NISM_LEVEL	2 for downloaded system, 1 for a system in ROM
SD_NISM_SUBNET_MASK	Subnet mask to use for SMNI
SD_KISM	YES to enable SMKI, otherwise NO
SD_KISM_DIRADDR	Bus (global) address of SMKI directory structure

On systems that support it, you can configure a shared memory network interface for use with pNA+ and/or a shared memory kernel interface (SMKI) for use with the pSOS+m kernel. In either case, you need to

assign a node number to each target board in the system. Node numbers are integers and start at 1. The SD_SM_NODE setting must be the same as the board's node number.

SD_NISM must be either YES or NO, depending on whether a shared memory network interface is included. If SD_NISM is YES, then SD_NISM_IP, SD_NISM_SUBNET_MASK, and SC_NISM_BUFFS specify the interface's IP address, subnet mask, and number of buffers, just as the corresponding parameters do for the LAN interface.

SD_NISM_DIRADDR is the bus address of a system-wide *directory structure* which must be accessible to all nodes in the system.

SC_NISM_LEVEL should be set to 2, except for the pSOSystem Boot ROMs. For the pSOSystem Boot ROMs, it should be 1 to allow a second, downloaded shared memory system to share the same directory structure with the one that the Boot ROMs use.

To configure a shared memory kernel interface (SMKI), set SD_KISM to YES and SD_KISM_DIRADDR to the bus address of the system-wide directory structure.

Directory structures are usually placed in a board's dual-ported RAM. Each pSOSystem board-support package reserves some space for these structures. To find the locations for these structures, see Appendix A, "Board-Specific Information."

7.2.6 Miscellaneous Parameters

The parameters in the following list do not depend on each other.

Parameter	Explanation
SC_RAM_SIZE	Amount of target memory to use (0 for all).
SD_DEF_GTWY_IP	IP address of default gateway node (0 for none).
SD_VME_BASE_ADDR	VMEbus address of a board's dual-ported RAM.

Normally, pSOSystem uses all of the unassigned memory on a board for dynamic allocation (Region 0). You can override this by setting SC_RAM_SIZE to a nonzero value. If you do, pSOSystem does not touch any memory after the first SC_RAM_SIZE bytes. This is useful when

building a Boot ROM because it allows you to make most of the board's RAM available for downloading code.

SD_DEF_GTWY_IP specifies the *default gateway* for pNA+ to use for packet routing. The default gateway is explained in the Boot ROM section of the *pSOSystem Programmer's Reference*. Note that if SC_PNA is NO, SD_DEF_GTWY_IP is not used.

SD_VME_BASE_ADDR specifies the base address of the target board's dual-ported memory on the VMEbus. This parameter is not used by non-VME target boards.

7.2.7 I/O Devices

The following parameters control the configuration of the I/O devices:

Parameter	Explanation
SC_DEV_SERIAL	Major device number of serial driver
SC_DEV_TIMER	Major device number of periodic tick timer
SC_DEV_RAMDISK	Major device number of RAM disk (0 for no RAM disk)
SC_DEV_SCSI	Major device number of SCSI driver (0 for none)
SC_DEV_SCSI_TAPE	Major device number of SCSI tape device
SC_DEV_TFTP	TFTP pseudo driver
SC_DEV_OTCP	TCP/IP for OpEN
SC_DEV_SPNA	pSOSystem backward compatibility
SC_IP	Internet Protocol
SC_ARP	Address Resolution Protocol
SC_TCP	Transmission Control Protocol
SC_UDP	User Datagram Protocol
SC_RAW	RAW sockets
SC_LOOP	Loopback
SC_DEV_SOSI	OSI for OpENbus
SC_DEVMAX	Maximum major device number in system

To include a device in the system, you must specify a major device number for it. The major device number determines the device's slot in the pSOS+ I/O switch table. To leave a device driver out of the system, use 0 or NO for the major number.

Note the following:

- Major device 0 is reserved and cannot be used to specify a device. Device number 0 means that device is not built into the system.
- No device number can exceed SC_DEVMAX. You can raise the value of SC_DEVMAX, if necessary.

The following lines should be included in **sys_conf.h** after the SC_DEV_* definitions:

```
#define DEV_SERIAL     (SC_DEV_SERIAL << 16)
#define DEV_TIMER      (SC_DEV_TIMER << 16)
#define DEV_RAMDISK    (SC_DEV_RAMDISK << 16)
#define DEV_SCSI       (SC_DEV_SCSI << 16)
#define DEV_TFTP       (SC_DEV_TFTP << 16)
#define DEV_SCSI_TAPE  (SC_DEV_SCSI_TAPE << 16)
```

7.2.8 Component Configuration Parameters

As explained in Section 7.2, "sys_conf.h," the values of many configuration table entries are controlled by *define* statements in **sys_conf.h**. The following subsections describe those parameters that can be controlled by define statements. Note that the names of the configuration table entries shown in the reference manual are in lowercase, and the corresponding **sys_conf.h** parameters are in uppercase. For example, **fc_nbuf** in the pHILE+ configuration table is controlled by FC_NBUF in **sys_conf.h**.

Although most of the component configuration table entries are determined by **sys_conf.h**, others are not because **sys_conf.h** cannot specify them. For example, **kc_code** in the pSOS+ configuration table contains the starting address of the pSOS+ kernel, but this is determined by where the linker places pSOS+; therefore, **sys_conf.h** cannot specify the address.

7.2.8.1 pSOS+ and pSOS+m Configuration Table Parameters

The following parameters in **sys_conf.h** control the values of the corresponding entries in the pSOS+ configuration table:

Parameter	Explanation
KC_RN0USIZE	Region 0 unit size
KC_NTASK	Maximum number of tasks
KC_NQUEUE	Maximum number of message queues
KC_NSEMA4	Maximum number of semaphores
KC_NMSGBUF	Maximum number of message buffers
KC_NTIMER	Maximum number of timers
KC_NLOCOBJ	Maximum number of local objects
KC_TICKS2SEC	Clock tick interrupt frequency
KC_TICKS2SLICE	Time slice quantum, in ticks
KC_SYSSTK	pSOS+ system stack size (bytes)
KC_ROOTSSTK	ROOT supervisor stack size
KC_ROOTUSTK	ROOT user stack size
KC_ROOTMODE	ROOT initial mode
KC_STARTCO	Callout at task activation
KC_DELETECO	Callout at task deletion
KC_SWITCHCO	Callout at task switch
KC_FATAL	Fatal error handler address
KC_ROOTPRI	ROOT initial priority

The following parameters in **sys_conf.h** control the values of the corresponding entries in the multiprocessor configuration table:

Parameter	Explanation
MC_NGLBOBJ	Size of global object table in each node
MC_NAGENT	Number of RPC agents in this node
MC_FLAGS	Operating mode flags
MC_ROSTER	Address of user roster change callout
MC_KIMAXBUF	Maximum length of KI packet buffer
MC_ASYNCERR	Address of error callout for asynchronous calls

7.2.8.2 pROBE+ Configuration Table Parameters

The following parameters in **sys_conf.h** control the values of the corresponding entries in the pROBE+ configuration table:

Parameter	Explanation
RC_BRKOPC	Instruction break opcode
RC_ILEVEL	pROBE+ interrupt mask
RC_SMODE	Start mode
RC_FLAGS	Initial flag settings

7.2.8.3 pHILE+ Configuration Table Parameters

The following parameters in **sys_conf.h** control the values of the corresponding entries in the pHILE+ configuration table:

Parameter	Explanation
FC_LOGBSIZE	Block size (base-2 exponent)
FC_NBUF	Number of cache buffers
FC_NMOUNT	Maximum number of mounted volumes
FC_NFCB	Maximum number of opened files per system
FC_NCFILE	Maximum number of opened files per task
FC_MSDOS	MS-DOS volume mount flag

7.2.8.4 pREPC+ Configuration Table Parameters

The following parameters in **sys_conf.h** control the values of the corresponding entries in the pREPC+ configuration table:

Parameter	Explanation
LC_BUFSIZ	I/O buffer size
LC_NUMFILES	Maximum number of open files per task
LC_WAITOPT	Wait option for memory allocation
LC_TIMEOPT	Timeout option for memory allocation
LC_SSIZE	Size of print buffer

7.2.8.5 pNA+ Configuration Table Parameters

The following parameters in **sys_conf.h** control the values of the corresponding entries in the pNA+ configuration table:

Parameter	Explanation
NC_NNI	Size of pNA+ network interface table
NC_NROUTE	Size of pNA+ routing table
NC_NARP	Size of pNA+ ARP table
NC_DEFUID	Default User ID of a task
NC_DEFGID	Default Group ID of a task
NC_HOSTNAME	Host name of the node
NC_NHENTRY	Number of host table entries
NC_NSOCKETS	Number of sockets in the system
NC_NDESCS	Number of socket descriptors/task
NC_MBLKS	Number of message blocks in the system
NC_BUFS_n	Number of n length buffers (0, 128-byte, 1K-byte, or 2K-byte)
NC_NMCSOCS	Number of sockets that can be used for multicast IP
NC_NMCMEMB	Total number of distinct multicast IP group memberships that can be added to pNA+. A maximum of 20 distinct group memberships can be added per multicast socket. Duplicate group membership addresses on a single interface do not count.
NC_NNODE_ID	Network Node ID or Router ID. This is assigned to be the source address for all unnumbered point-to-point links.
NC_BUFS_0	Number of 0-length buffers
NC_BUFS_128	Number of 128-byte buffers
NC_BUFS_1024	Number of 1K-byte buffers
NC_BUFS_2048	Number of 2K-byte buffers
SE_MAX_PNA_NC_BUFS	Maximum number of NC_BUFS types

7.2.8.6 pSE+ Configuration Table Parameters

The following parameters in **sys_conf.h** control the values of the corresponding entries in the pSE+ configuration table:

Parameter	Explanation
NBUFS_0	Number of 0-length buffers
NBUFS_32	Number of 32-byte buffers
NBUFS_64	Number of 64-byte buffers
NBUFS_128	Number of 128-byte buffers
NBUFS_256	Number of 256-byte buffers
NBUFS_512	Number of 512-byte buffers
NBUFS_1024	Number of 1K-byte buffers
NBUFS_2048	Number of 2K-byte buffers
NBUFS_4096	Number of 4K-byte buffers
SE_MAX_PSE_STRBUFS	Maximum number of stream buffer types
SE_MAX_PSE_MODULES	Maximum number of stream modules
SE_DATA_SIZE	Size of pSE+ data area (at least 3K)
SE_TASK_PRIO	Priority for pSE+ task
SE_STACK_SIZE	Stack size for pSE+ task
SE_DEF_UID	Default user ID
SE_DEF_GID	Default group ID
SE_N_FDS	Maximum number of system-wide stream descriptors
SE_N_TASKFDS	Maximum number of per-task stream descriptors
SE_N_LINKS	Maximum number of multiplexing links
SE_N_TIMEOUTS	Maximum number of timeout requests
SE_N_BUFCALLS	Maximum number of bufcall requests
SE_N_QUEUES	Number of queues
SE_N_MBLKS	Reserved for future use, must be 0

7.2.8.7 Loader Configuration Table Parameters

Parameter	Explanation
LD_MAX_LOAD	Maximum number of active loads
LD_IEEE_MODULE	Link IEEE object-load-module
LD_SREC_MODULE	Link S-record object-load-module

7.2.8.8 pMONT Configuration Table Parameters

Parameter	Explanation
PM_CMODE	Communication mode that pMONT will operate in 1 = networking, 2 = serial
PM_DEV	Major/Minor device number for serial channel if in serial mode
PM_BAUD	Baud rate for serial channel if in serial mode
PM_TRACE_BUFF	Address of trace buffer pSOSystem will allocate an address if PM_TRACE_BUFF is set to zero
PM_TRACE_SIZE	Size of trace buffer
PM_TIMER	YES confirms the existence of a second timer that pMONT will use for fine-tuned data collection

7.2.8.9 General Serial Block Configuration Parameters

Parameter	Explanation
GS_BUFS_0	Number of 0 length buffers (used in **gs_esballoc** function)
GS_BUFS_32	Number of 32-byte buffers
GS_BUFS_64	Number of 64-byte buffers
GS_BUFS_128	Number of 128-byte buffers
GS_BUFS_256	Number of 256-byte buffers
GS_BUFS_512	Number of 512-byte buffers
GS_BUFS_1024	Number of 1024-byte buffers
GS_BUFS_2048	Number of 2048-byte buffers
GS_BUFS_4096	Number of 4096-byte buffers
SE_MAX_GS_BUFS	Maximum number of buffer sizes
GS_MBLKS	Number of M-block headers

NOTE: For canonical processing, the serial driver needs buffers 64 bytes larger than the receive buffer size of the driver. For example, if the receive buffer size is 64 bytes, then the driver will need 128-byte buffers to process the incoming data. Be sure to take this into consideration when configuring the general serial block buffers.

7.3 Adding Drivers to the System

To add drivers to the pSOS+ I/O driver table, edit **drv_conf.c** in the working directory. This file contains a function called **SetUpDrivers()**. **SetUpDrivers()** calls **InstallDriver()** to install each driver in the I/O table.

To add a driver, you should add an **InstallDriver()** call to **SetUpDrivers()**. **InstallDriver()** has the following syntax:

```
void InstallDriver(
                unsigned short major_number,
                void (*dev_init) (struct ioparms *),
                void (*dev_open) (struct ioparms *),
                void (*dev_close) (struct ioparms *),
                void (*dev_read) (struct ioparms *),
                void (*dev_write) (struct ioparms *),
                void (*dev_ioctl) (struct ioparms *),
                unsigned long rsvd1,
                unsigned short rsvd2,
                unsigned short flags)
```

As you can see **InstallDriver** takes the major number of the driver given by **major_number**, uses it as an index into the I/O switch table and places the remaining arguments into their correct place in the I/O switch table.

The flags argument has special meaning to the pSOS+ kernel. Currently this flag field is used to set an **AutoInit** bit that pSOS+ will check when it is initializing. If the bit is set pSOS+ will call the initialization function, if any, for the driver when pSOS+ is initializing. This means you will not have to call the driver initialization function (through the use of **dev_init**) in your application for any driver that has this bit set. Two define statements should be used to set the **AutoInit** bit. They are found in

include/psos.h and their names are IO_AUTOINIT to set the **AutoInit** bit and IO_NOAUTOINIT to turn the setting off.

NOTE: Any driver that you plan to use with this 'AutoInit' feature must not make any system calls that need a task's context because the driver initialization function will be called before any task has been started.

For example, to add a driver as major device 6 (which has only *init* and *read* calls), you would add the following to **SetUpDrivers()** in **sysinit.c**:

```
InstallDriver(6, DriverInit, NULLF, NULLF, DriverRead,
NULLF, NULLF, 0, 0, IO_AUTOINIT);
```

where NULLF is a macro in **sysinit.c** defined as a null function pointer and IO_AUTOINIT would set the flags so the driver would be initialized at the startup of the pSOS+ kernel.

Network interfaces are added in a manner similar to that of pSOS+ drivers. **drv_conf.c** also contains a routine called **SetUpNi()**, which calls **InstallNI()** to install each network interface in the pNA+ initial interface table. The parameters to **InstallNI()** are documented in **drv_conf.c**.

7.4 Using the Boot ROMs

The pSOSystem software includes two types of Boot ROMs:

- The *pROBE+ Boot ROM* is designed for systems with only a serial communications channel. This ROM uses pROBE+ or a remote debugger to download the system image of an application and begin execution. The code that produces this Boot ROM can be found in the sample application directory (**apps/proberom**). For more information on this application, see Chapter 8, "Application Examples." For more specific information on how to build the pROBE+ ROM for the supported boards in pSOSystem, see Appendix A, "Board-Specific Information."
- The *TFTP Boot ROM* is designed for systems that have an Ethernet connection as well as a serial communications channel. The ROM can also use pROBE+ or a remote debugger to down load the system image of an application and begin execution. In addition, the ROM can download the system

image by using the TFTP protocol or a remote debugger over the network. The code that produces this Boot ROM can be found in the sample applications directory **(apps/tftp)**. See the "Applications Examples" chapter for more information on this application. For more specific information on how to build the TFTP ROM for the supported boards in pSOSystem, see Appendix A, "Board-Specific Information."

Both Boot ROMs store the configuration, if possible, in nonvolatile RAM. Once RAM is configured the way you want, it does not have to be reconfigured at each boot.

The stored configuration can also be used by the downloaded code if the downloaded code is compiled to do so. For more information on how to compile your application to use the stored information, see the "Storage and Dialog Parameters" section of this chapter.

7.4.1 pROBE+ Boot ROM

The output for the pROBE+ Boot ROM is shown in Figure 7-2.

```
pSOSystem V2.2.0
Copyright (c) 1991 - 1995, Integrated Systems, Inc.
-------------------------------------------------------------------------
START-UP MODE:
  Boot into pROBE+ stand-alone mode
NETWORK INTERFACE PARAMETERS:
  LAN interface is disabled
  Shared memory interface is disabled
MULTIPROCESSING PARAMETERS:
  This board is currently configured as a single processor system
HARDWARE PARAMETERS:
  Serial channels will use a baud rate of 9600
  This board's memory will reside at 0x1000000 on the VME bus
  After board is reset, start-up code will wait 60 seconds
-------------------------------------------------------------------------
To change any of this, press any key within 60 seconds
```

Figure 7-2 pSOSystem Startup Message

This is the pSOSystem startup screen for the Boot ROM. It displays the current Boot configuration.

Start-up mode can be one of two modes:

- pROBE+ stand-alone mode
- pROBE+ waiting for host debugger via serial connection

In stand-alone mode you will get a pROBE+ prompt when the system has completed booting. In pROBE+ waiting for host debugger via serial connection mode, the system waits for a connection via the system console port.

MULTIPROCESSING PARAMETERS: The system can be a single processor running the pSOS+ kernel or multiple processors running the multiprocessing pSOS+m kernel.

HARDWARE PARAMETERS: Shows the current baud rate the console port. The amount of time the system will wait at this screen for user input after the board has been reset is also displayed.

You can change the boot configuration by entering an **M** and a carriage return or continue by entering a **C** and a carriage return.

If you choose to modify the boot configuration, you will be prompted to enter new selections. All selections have as a default value their current setting. This means if you enter a return at any selection it maintains its current setting.

If you choose to modify the boot configuration, the first prompt you will see is shown in Figure 7-3.

```
For each of the following questions, you can press <Return> to select the
value shown in braces, or you can enter a new value.

How should the board boot?
  1. pROBE+ stand-alone mode
  2. pROBE+ waiting for host debugger via serial connection
  3. pROBE+ waiting for host debugger via a network connection
  4. Run the TFTP Bootloader

Which one do you want? [1]
```

Figure 7-3 Reconfiguring the ROMs

After you select between pROBE+ stand-alone mode and pROBE+ waiting for host debugger via serial connection, you are prompted for multiprocessing parameters.

```
Do you want to configure a multiprocessing pSOS+m system?
[Y]
```

You can answer this question with a **Y** for yes or an **N** for No. If you answer with an **N**, you will prompted for the HARDWARE PARAMETERS. If you answer with a **Y**, you will receive the following prompts:

```
How many pSOS+m nodes will it contain? [2]
```

Enter the number of nodes that are in the multiprocessing system and then a carriage return.

The next prompt is:

```
Which node should this CPU be? [1]
```

Enter the number you want the node you are using to be.

NOTE: Node number 1 will be considered the master node.

Next you are prompted for the address of the Shared Memory Kernel Interface (SMKI) directory:

```
Bus address of the SMKI directory [1000580]
```

This needs to be an address that can be accessed by all nodes in the system.

This is the end of the multiprocessing parameters section. Now you will be prompted for hardware parameters:

```
Baud rate for serial channels [9600]
```

This prompt allows you to change the baud rate of the console port of the system. The range of baud rates will depend on the system but most systems support a baud rate between 300 and 38400 bits per second.

NOTE: If you change the baud rate, you will also have to change the baud rate of the terminal or terminal emulation program you are running when the Boot ROM code resets the system configuration. This happens after you answer the last question.

The last prompt is:

```
How long (in seconds) should CPU delay before starting up?
[60]
```

Enter the amount of time in seconds you want the system to wait for input from the system console before it resumes and enters the Start-up Mode.

Once you enter a return at this final prompt, the code will set the configuration you have selected, and display the start-up screen again. At that time you should review the screen and modify the configuration parameters again as needed. If the configuration is correct, enter a **c** to run the configured Startup Mode.

7.4.2 TFTP Boot ROM

TFTP Boot ROM is a superset of the pROBE+ Boot ROM. The output for the TFTP Boot ROM is shown in Figure 7-4.

```
------------------------------------------------------------------------
START-UP MODE:
  Boot into pROBE+ and wait for host debugger via a serial connection
NETWORK INTERFACE PARAMETERS:
  LAN interface is disabled
  Shared memory interface is disabled
MULTIPROCESSING PARAMETERS:
  This board is currently configured as a single processor system
HARDWARE PARAMETERS:
  Serial channels will use a baud rate of 9600
  This board's memory will reside at 0x1000000 on the VME bus
  Processor Type :: MC68040 operating at 25 Mhz
  RAM configuration :: Parity DRAM 4 Mb
                    :: SRAM 128 Kb
  After board is reset, start-up code will wait 3 seconds
------------------------------------------------------------------------
(M)odify any of this or (C)ontinue? [M]
```

Figure 7-4 pROBE+ Startup Message

Figure 7-4 shows the PROBE+ startup screen for the Boot ROM. It displays the current Boot configuration.

At the top of the pSOSystem startup screen, as shown in Figure 7-2 on page 7-18, is the version of pSOSystem that was used to build the ROM along with the Integrated Systems copyright.

Start-up mode can be one of four modes:

- pROBE+ stand-alone mode
- pROBE+ waiting for host debugger via serial connection
- pROBE+ waiting for host debugger via a network connection
- Running the TFTP bootloader

The remaining prompts vary depending on the mode you have selected.

NETWORK INTERFACE PARAMETERS:

Displays the current state of the two possible network interfaces. If these network interfaces were enabled, additional information about how that interface was configured would be displayed.

MULTIPROCESSING PARAMETERS:

The system can be a single processor running the pSOS+ kernel or multiple processors running the multiprocessing pSOS+m kernel.

HARDWARE PARAMETERS:

Shows the baud rate of the console port. If the system you are using has a VME bus interface, the address of the board (as indicated for the rest of the boards on the bus) is displayed.

The next three lines of information, processor type and RAM configuration, are a product of a specific board level dialog for the board you are working with and will vary from board to board.

The amount of time the system will wait at this screen for user input after the board has been reset is also displayed.

NOTE: Once you configure the Boot ROM, the opening screen will change and may display additional information.

You can change the boot configuration by entering an **M** or continue by entering a **C**.

If you choose to modify the boot configuration. You are prompted to enter new selections. All selections have as a default value their current setting. Therefore, if you press [Return] at any selection, the Boot ROM maintains the current setting.

If you choose to modify the boot configuration, the first prompt you see is shown in Figure 7-5.

```
For each of the following questions, you can press <Return> to select the
value shown in braces, or you can enter a new value.

How should the board boot?
  1. pROBE+ stand-alone mode
  2. pROBE+ waiting for host debugger via serial connection
  3. pROBE+ waiting for host debugger via a network connection
  4. Run the TFTP Bootloader

Which one do you want? [1]
```

Figure 7-5 Reconfiguring the ROMs

From the choices shown in Figure 7-5, select a startup mode:

- 1 for pROBE+ stand-alone mode
- 2 for pROBE+ waiting for host debugger via serial connection
- 3 for connecting to a remote debugger via a network connection
- 4 to boot by using a TFTP bootloader

In mode 1 you will get a pROBE+ prompt when the system has completed booting. Entering a **1** at the configuration mode prompt enables you to use a ROM-resident, stand-alone pROBE+ to download a pSOSystem executable image and start that image. pROBE+ downloads the image by executing the DL command and starts the image by executing the GO command. Upon exit from configuration mode, the following pROBE+ sign-on banner appears:

```
pROBE+ V2.1.1 (68040)
COPYRIGHT 1990 - 1996, INTEGRATED SYSTEMS, INC.
ALL RIGHTS RESERVED
pROBE+>
```

In mode 2 the system will wait for a remote debugger connection via the system console port. The serial connection you are using for communication to pROBE+ now needs to be used by pROBE+ to communicate with the remote debugger, so if you are using a terminal emulator, such as the UNIX **tip** utility, exit the emulator at this time so that the remote debugger can use the connection. For a complete description of the pROBE+ debugger, see the *pROBE+ User's Guide*.

If you have a terminal connected to the target's console port, then this connection must be moved from the terminal to a port that the remote debugger can connect to on the host system. (Remember to configure the host's port the same way the terminal was set; that is, with baud rate and stop bits).

In mode 3, the system waits for a remote debugger connection via the networking port. In most cases, when the target CPU is connected to a host system by Ethernet, the target CPU is added to an existing network. As such, the target CPU must have an IP address that is compatible with that network. Consult your system administrator to obtain a usable IP address. The system administrator may need to modify one or more host system files to make the new IP address known to the host.

The Boot ROMs can also obtain their IP addresses for Ethernet by RARP (Reverse Address Resolution Protocol), in which case a RARP server on the Ethernet assigns the IP address to the ROMs when they initialize. To force the ROMs to use RARP to get their IP address, use 0.0.0.0 for the IP address.

In mode 4, the system connects to a TFTP site and download using a TFTP network connection. This provides a mechanism whereby the Boot ROMs, following a power-on/reset, can automatically download and initiate execution of a pSOSystem executable image without user intervention. The bootloader uses the TFTP protocols and can be used with any host system that supports TFTP.

The TFTP bootloader in the ROMs has been designed for ease of use. In configuration mode, the user specifies the name of the S-record file that contains the executable image and the IP address of the host system on which it resides. When the ROMs exit configuration mode, the bootloader begins operating and loads the image into memory. When the image has been loaded, the bootloader looks at the first two longwords of the loaded image to obtain the initial stack pointer and program counter. It uses this information to transfer control to the newly-loaded code. No special steps are necessary to make the pSOSystem executable image fit this model because a reset vector is always placed in the first two longwords.

The startup mode determines what questions you will see in the remaining part of the dialog. All modes present you with basic configuration questions, so you can set your configuration in nonvolatile RAM. Some modes have additional questions such as the filename of the file to download when mode 4, TFTP bootloader, has been selected. The following explanations go through all of the questions you might be

asked. Questions that are specific to a given mode are pointed out (questions are organized by parameter groups).

```
NETWORK INTERFACE PARAMETERS:
```

```
Do you want an Ethernet interface? [N]
```

If your application will be using an Ethernet interface or you have selected Startup Mode 3 or 4 and your network connection is a LAN then you should answer yes to this question. If you do answer yes to this question, then you will also see the following question:

```
This board's LAN IP address(0.0.0.0 = RARP)?
[000.000.000.000]
```

Because you have an Ethernet interface you must have an IP address. You may enter the IP address of the target system here or leave it all zeroes. If you leave it all zeros, then the boot ROM software uses Reverse Address Resolution Protocol (RARP) to obtain the target's IP address if a RARP server on the target's network has been set up to do so.

Again if you have an Ethernet interface you will be asked the following question:

```
Subnet mask for LAN (0 for none)? [000.000.000.000]
```

A subnet mask is a 32-bit quantity indicating which bits in an IP address that identify the physical network. Subnetting allows the use of some of the host-identifier bits to be used to identify local physical networks. For example, 255.255.255.000 allows 24 of the 32 bits to be used for a network identifier.

```
Do you want a shared memory network interface? [N]
```

Some systems have a common bus, a VME bus is one example, that may be used to access another systems memory or make it possible to access memory external to all boards. pSOSystem can use this shared memory as a vehicle for networking. It is called a shared memory interface. It allows two or more systems to use pNA+ to communicate between them through shared memory instead of an Ethernet. It is very useful when there is only one Ethernet connection available for a group of boards. One of the boards can be use the connection to the Ethernet and act as a router to route packets to the shared memory network. In this way boards that do not have an Ethernet connection can still be downloaded and debugged via this gateway. A **yes** answer to this question will cause

additional question about the configuration of the shared memory network starting with the following question:

```
IP address for shared memory? [000.000.000.000]
```

The shared memory network address is just like the LAN IP address however it must be different then the LAN IP address. Enter the shared memory IP address here.

```
Subnet mask for shared memory interface (0 for none)? [0]
```

This is the same idea as the subnet mask for the LAN explained previously.

```
Which node number in the shared memory system is this? [1]
```

In response to this question enter the node number of this target. No two targets can have the same node number. Node numbers for targets should be consecutive with no gaps allowed. This node number will be used to find information for the target in the SMNI (Shared Memory Interface) directory. Node one is a special node. It will be the master node used to keep track of the remaining nodes. Node one is usually the gateway node to other networks if there are any.

```
Bus address of the SMNI directory? [1000400]
```

This is a area in shared memory that is used as a table to store information about the SMNI network. It must be accessible to all nodes in the SMNI network.

```
Should there be a default gateway for packet routing?
[N] y
```

If this node is not the gateway for packet routing and the node needs to access the gateway you must enter the IP address on the shared memory network of the gateway node (the next question) Answer Y if this is true.

NOTE: If this node is to be accessed through a gateway a routing command needs to be issued on the system that wants to access it.

```
What is its IP address? [0.0.0.0] 000.00.000.000
```

Enter the IP address of the gateway of your shared memory network if you have one.

MULTIPROCESSING PARAMETERS:

These next questions deal with multiprocessing. Multiprocessing refers to pSOS+m and the use of the kernel interface pSOS+m uses. The ROMs themselves do not contain pSOS+m. However you may configure a pSOS+m system here so your downloaded system can get its configuration from nonvolatile memory.

Do you want to configure a multiprocessing pSOS+m system? [N]

A yes answer to this question will allow you to continue and configure a pSOS+m system.

How many pSOS+m nodes will it contain? [0]

pSOS+m needs to know how many nodes are in the system so it can set up its internal tables.

Which node should this CPU be? [1]

pSOS+m need to know what node number you are assigning this node. Numbers should be consecutive starting from 1 with no gaps.

Bus address of the SMKI directory [1000580]

This is a area in shared memory that is used as a table to store information about the SMKI for pSOS+m. It must be accessible to all nodes that are part of SMKI. If there is also a SMNI network, this directory must be separate and distinct from it.

HARDWARE PARAMETERS:

Baud rate for serial channels [9600]

This will set up a default baud rate for all serial channels including the console that the Boot ROMs are using to present this dialog. If a change is made to the current baud rate, it will take effect after all questions are answered. Your terminal or terminal emulator will need to be set at the new rate once all the questions are answered. The ROMs are set to 9600 baud for their first use.

Bus address of this board's dual-ported memory [1000000]

This is the address that will allow other systems on the same bus to access this boards dual-ported memory. The shared memory of any of the boards cannot start at less then the largest memory size of any

board. Care must be taken so no board overlaps the memory of another board. For example:

You had three boards on the shared memory bus.

Board #1 has 32 megabytes (0x02 00 00 00)

Board #2 has 16 megabytes (0x01 00 00 00)

Board #3 has 64 megabytes (0x04 00 00 00)

The shared memory of any of the boards cannot start at less than the largest memory size of any board. In this case, board 3 has the largest memory at 0x04 00 00 00, so no VME address can exist below 0x04 00 00 00.

To map this out, if the first board were the master connected to the Ethernet, the map should look like this:

All boards SMNI directory 0x04 00 04 00 (memory address on master node)

All boards SMKI directory 0x04 00 05 80 (memory address on master node)

Board #1 shared memory starts at 0x04 00 00 00 ending 0x05 FF FF FF

Board #2 shared memory starts at 0x06 00 00 00 ending 0x06 FF FF FF

Board #3 shared memory starts at 0x07 00 00 00 ending 0x10 FF FF FF

TFTP BOOTLOADER PARAMETERS:

Do you want to use the RARP server as the TFTP Boot server? [N]

If you have chosen TFTP mode and you entered 0 for an IP address, the target will get its IP address using RARP. By answering this question with a yes, you can also use the same server that provided you with the IP address for the TFTP of your download file.

IP address of the TFTP Boot server to boot from? [0.0.0.0]

This question will be asked if you are not using the RARP server as the TFTP server for your download file. Here you need to enter the IP address of the TFTP server to use for the download file.

What is the name of the file to be loaded and started?

```
[ram.hex]

Enter the name of the file that contains the download code
on the TFTP server.

How long (in seconds) should CPU delay before starting up?
[60]
```

The CPU will wait before it starts the boot process so you can alter the configuration if necessary. You can also set the amount of time it waits for you to press the return key.

7.5 System Startup Sequence

Figure 7-6 illustrates the system vectors for 68K processors.

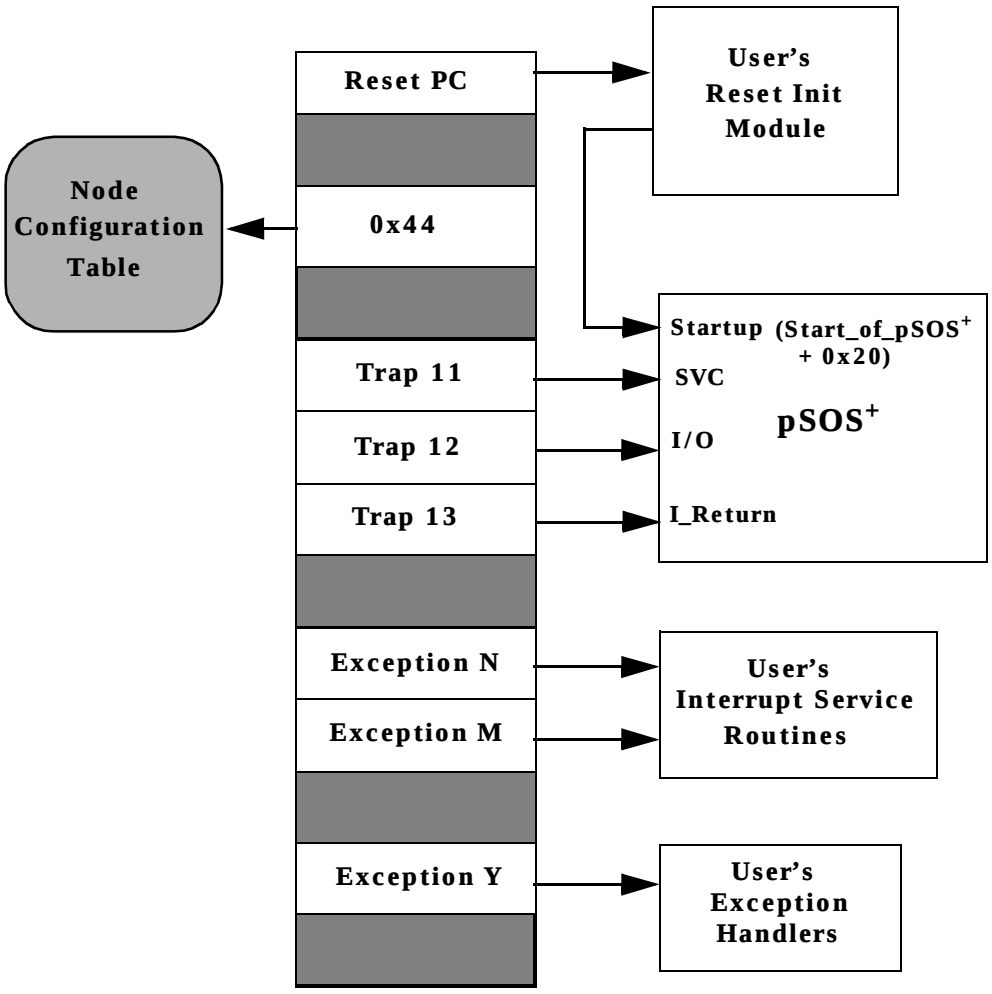


Figure 7-6 System Vectors

The component modules should be aligned on long word boundaries, but they are otherwise position-independent. Each assigned interrupt vector should be loaded with the address of the ISR that services that hardware vector. If multiple devices share a common interrupt vector, the ISR must resolve the identity of the interrupting device. Integrated Systems recommends that an error exception handler process all remaining unused and error exception vectors. If the pROBE+ monitor is present, these exception vectors can be set to point to the corresponding exception entry points to the pROBE+ debugger.

The following checklist shows the required items for a typical pSOSystem-based system:

- An optional user-supplied boot module to perform power-on or reset initialization and self-test
- pSOS+ and any other required components (for example, pROBE+ and pHILE+)
- The Node Anchor and Node configuration table
- Configuration tables for pSOS+ and other installed modules
- The application's task, ISR, and device driver code and initialized data, if any
- Exception vectors with the correct settings

In a ROM-based system, most of these items reside in ROM. For a RAM-based system or a system that is under test or integration, some of the items can be loaded into RAM either by the user's boot module or the pROBE+ System Debug/Analyzer. Furthermore, in a memory-mapped system especially, it is possible to load the application tasks or drivers dynamically at run time. Figure 7-7 on page 7-32 shows the possible system startup sequences.

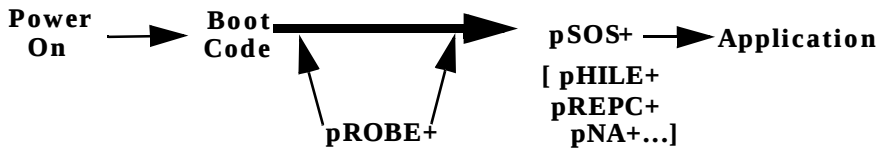


Figure 7-7 System Startup Sequences

Typically, power-on or reset passes control to the user-supplied Boot Code, which performs any necessary initialization and self-test. It should also set up the required configuration environment for the software components in the system. The configuration environment consists of the Node Anchor, the Node configuration table, and the other component configuration tables. You can completely set up this environment now or else allow the startup sequence to proceed incrementally. During debug, for example, you might set up the configuration environment just enough to let pROBE+ take over and execute. If pROBE+ is present, the Boot Code should pass control to it. pROBE+ can then be used to download other items on the checklist and start execution of the pSOS+ application. You can start pSOS+ in one of the following ways:

- By passing control to the pSOS+ Startup entry, as follows:

  ```
  BX    START_of_PSOS  + 0x20;
  ```

  Note that this assumes the CPU is in the supervisor state.

- From pROBE+, enter the **gs** command.
- Specify the **silent** startup mode in the pROBE+ configuration table, so that pROBE+ initializes itself then passes control to pSOS+ Startup without stopping.

Upon entry, pSOS+ Startup first uses the Node Anchor to locate the pSOS+ configuration table. pSOS+ then takes a segment of memory from the beginning of memory Region 0 for its data area. Within this area it uses the bottom part for its key data structures. The next memory segment goes to the system stack. Above the system stack, pSOS+ builds the required number of TCBs, QCBs, SCBs, MGBs, TMCBs, and the Object Tables.

Next, pSOS+ checks the Node configuration table. If other runtime components are present, pSOS+ locates and automatically calls the Startups of those components to allow them to set up and initialize.

The last step in pSOS+ Startup is to create and activate the IDLE system daemon task and the user ROOT task. Startup then dispatches ROOT, which takes over and executes.

NOTE: pSOS+ treats any error it encounters during startup as a fatal error. For information on handling of fatal errors, see *pSOSystem System Concepts*.

7.6 Component Customizations

If you are planning to change the pSOSystem startup code or write your own startup code, you may need to change the location of the Node Anchor.

The Node Anchor is used by each component to locate its configuration table. The location of the Node Anchor is determined in the linker command files supplied with each board-support package (**bsps/*/*.lnk**) by defining a value for symbol _anchor. You can change this if necessary, but exercise caution when doing so because the linker reserves no space for the Node Anchor.

8 Application Examples

The applications directory has several example applications in the directory **$PSS_ROOT/apps.** These applications demonstrate the use of pSOSystem and its components. Each application is located in its own directory. Each application directory contains a README file that explains the application, how to build it, what components are needed to use the application, and what output the application is expected to produce. All of the applications need pSOSystem installed. Some require additional components.

Each application contains a **makefile** that builds the application and causes it to be linked with the correct libraries. Integrated Systems recommends that you read and run the tutorials in this manual, before you use these examples.

Table 8-1 summarizes the application examples and lists these tutorials in a recommended order from simple to complex.

Table 8-1 Application Examples

Directory	Description	Page
hello	Simple one task application that displays the message “Hello, world.” This application is used in some of the tutorials. **hello** is a good starting point to get an application up and running on your target.	8-3
xraydemo	This application is used for the XRAY tutorial.	8-4

Table 8-1 Application Examples (Continued)

Directory	Description	Page
pnabench	This application is an example of a target system being used as a client to send data to a host system using TCP/IP over an Ethernet connection. This application sends data, then reports on the rate of the transfer.	8-3
philepna	This application contains several tasks that exercise the pHILE+ and pNA+ components.	8-7
proberom	This application can be used to build ROMs for non-Ethernet systems. It is also an example of how to use a startup dialog.	8-4
tftp	This application can be used to build boot ROMs for systems that contain an Ethernet connection. It is also an example of how to use a startup dialog.	8-5
nfs	This application shows how to use pSOSystem's NFS client services.	8-9
fpsp	This application is an example of the use of the floating-point library on 68040 and 68060 processors.	8-2

The following sections contain descriptions of the applications. The applications are identified by directory name.

All sample applications include the **sys_conf.h** configuration file supplied in the **applications** directory. In most cases the application will run with the default values set in the **sys_conf.h** file. In some cases you may have to adjust the configuration to suit the board-support package you are using. Check Appendix A, "Board-Specific Information," for more information on changes you may need to make for a specific board-support package.

8.1 fpsp

The **fpsp** sample demonstrates the use of the 68040/060 floating-point library. This sample application demonstrates the use of ANSI C math functions and floating-point assembly instructions. The functions that actually generate the floating-point exceptions are located in the header file **m68881.h**.

The output for this application is as follows:

```
TASK START-UP:sin(0.523599) = 0.500000
   TASK 0 ::cos(1.047198)=0.500000
   TASK 0 ::tan(0.785398)=1.00000
   TASK 1 ::area_circle(2.123457) = 14.165657
   TASK 2 ::(sin(2.651235)**2+(cos(2.651235))**2)=1.00000
```

8.2 hello

This **hello** sample is a simple program that prints a message. The application consists of a single task (ROOT) which prints out a few short messages to either the target's serial port (system console) or the XRAY Debugger for pSOSystem standard output screen and then suspends itself.

The output for this application is:

```
Hello, world
```

The **hello** program has a function named **db_output**. That function can be used in any application to redirect console output to the XRAY standard output screen. The **db_output** function directs output through pROBE+ to the XRAY standard output screen.

For example, to send the string "Hello, world" to the XRAY output screen enter the following command:

```
db_output("Hello, world\n", 0);
```

The syntax for **db_output** is:

void db_output(const char *string, ULONG port);

where **string** is the character string to be printed out and port is always 0. The **db_output** function is contained in the **bsp.lib** source code (in the **xp_out.s** file in the **devices/68k** directory).

8.3 pnabench

The pnabench sample runs a benchmark program that measures TCP throughput. To run this application, you must have pNA+ and pREPC+ installed in addition to pSOS+. This application reports the amount of

time it takes to transfer data between the target system and the host of your choice over a network.

The **README** file tells you how to build the target code.

To use this application you must first compile the server in the host directory and start it on the host. Then start the application on the target.

The output on the target looks like this (actual output will vary from system to system):

```
Sending data to the server.

Time taken to transfer 0x989680 bytes of data is 0x13 seconds.

Server's Report:
  Elapsed time = 19.519224 seconds
  Number of transfers = 3502
  Amount transferred = 10000000 bytes
  Rate = 512315.444226 bytes per second
```

The output on the server (host) is as follows:

```
Server at port 5000 and address 0
Connected to target: integrated1 on port: 1024
Receiving data from target: integrated1
.....................................
Finished receiving.
Completed 3502 requests in 19.519224 seconds.
```

8.4 xraydemo

This sample is used with the XRAY tutorial described in Chapter 4, "XRAY Debugger for pSOSystem Tutorial: Multiple Windows Version."

8.5 proberom

The proberom application is used to build pSOSystem Boot ROMs for non-networking systems. This is the only sample application that does not require pSOS+. The only component in this application is pROBE+. **proberom** is used to make Boot ROMs for pSOSystem supported boards

that do not have networking capability. When used to boot a system, this code initializes the hardware and executes the debugger, pROBE+, or allows the remote connection via the console port to a remote debugger.

The prompts that are presented by the Boot ROM code come from the **dialog.c** file located in the **configs/std** directory. The **proberom** application is a good example of how to use the functions in **dialog.c**. These functions can be used for a RAM boot also. The dialog code is included in the executable image by setting the define directive **SC_STARTUP_DIALOG** to YES in the **sys_conf.h file.** This action allows the dialog to run before any application runs or pSOS+ has been started. The functions in **dialog.c** are called during the system initialization process by the SysInit function located in the **sysinit.c** file in the **configs/std** directory.

The configuration is stored, if possible, in nonvolatile RAM so once configured the way you want, it does not have to be done over again at each boot.

The configuration can also be used by the downloaded code if the downloaded code is compiled to do so. For more details, see Chapter 7, "Configuration and Startup."

The **proberom** application uses a different linker file then a RAM based application would. The linker file is called **rom.lnk** and is located in the board- support package directory of the board you are using. This linker file is designed to load the code at a ROM address suitable for the board's hardware.

The output for this application is described in Chapter 7.

8.6 tftp

The tftp application is a superset of the **proberom** application. **tftp** is used to build the pSOSystem Boot ROMs for networking systems. It includes a TFTP bootloader program for loading and starting a pSOSystem executable image and allows for a remote Debugger connection over the network. This application is an example of how to use the tftp driver. It also shows how to put together a startup dialog.

Some of the prompts that are presented by the Boot ROM code come from the **dialog.c** file located in the **configs/std** directory as they do in the **proberom** application. The dialog code is included in the executable

image by setting the #define directive **SC_STARTUP_DIALOG** to YES in the **sys_conf.h** file.

The **tftp** application needs additional dialog and that comes from the file **apdialog.c** located in the **tftp** directory. This additional dialog is needed to support the use of a network in the boot process. The **tftp** application is a good example of how to incorporate a custom dialog that can be used to configure your application before the system boots up and the application is started. The functions in **apdialog.c** follow a specific interface. The dialog code is included in the executable image by setting the #define directive **SC_APP_PARMS** to the amount of memory in characters that is needed to store the application's parameters. This is done in the **sys_conf.h** file. This interface is detailed in the "Drivers and Interfaces" chapter of the *pSOSystem Programmer's Reference*.

Like the **proberom** application, the configuration is stored, if possible, in nonvolatile RAM once configured the way you want, it does not have to be done over again at each boot.

Also, like the **proberom** application, the configuration can be used by the downloaded code if the downloaded code is compiled to do so. See the system "Configuration and Startup" chapter for more details.

The output for the **tftp** application is detailed in the "Configuration and Startup" chapter.

The **tftp** application can be used with or without any dialog. It can be used to remotely boot a system using **tftp**. This can be done by changing the configuration in the **sys_conf.h** file. You would need to change the following define directives in **sys_conf.h:**

- **SC_SD_PARAMETERS** to SYS_CONF
- **SE_DEBUG_MODE** to DBG_AP
- **SD_LAN1** to YES
- **SD_LAN1_IP** to the target's IP address or leave it at 0 if you want to use RARP to get the IP address.
- Comment out the define directive for **SC_APP_PARMS** and **SC_APP_NAME**
- **SA_BOOT_FILE** should be set to the name of the boot file on the host system.

- **SA_HOST_IP** should be set to the IP address of the host system or 0 if you are using the RARP server and want the **RARP** server to be the boot server.

You can then build a Boot ROM with this configuration and install it into your Target System. Information on how to build a Boot ROM is contained in Appendix A, "Board-Specific Information."

8.7 philepna

The **philepna** application demonstrates the use of pSOS+, pHILE+, pNA+, and pREPC+. The application source code contains sections that are conditionally compiled depending on the presence or absence of particular components in the operating system. As shipped, it assumes the presence of pSOS+, pROBE+, pREPC+, pNA+, and pHILE+. If any of these components are not present in your system, you must edit **sys_conf.h** and change the appropriate definition lines to NO. Note also that pSOS+ (or pSOS+m) and pREPC+ are REQUIRED for this application.

Execution starts with ROOT, as usual. It initializes all of the I/O devices in the OS (console, real-time clock, RAM disk, and, if present, SCSI disks). Then it prompts you for the date and time and sets the pSOS+ clock accordingly. If pHILE+ is being used, ROOT creates and starts task FILE. If pNA+ is being used, ROOT creates and starts tasks CLNT and SRVR. Finally, ROOT prints a message stating that it has completed, and it suspends itself.

If task FILE was created, it starts running now. It initializes the ***RAM disk** volume and enters an infinite loop. In the loop, it mounts the RAM disk volume or **SCSI disk** volume if present, creates a directory and opens a file, and writes to and reads from the file. Then it deletes the file and the directory and unmounts the volume. Finally, it sleeps for a while, prints a message on the console, and repeats the loop.

Tasks CLNT and SRVR also run now in infinite loops, if they were created. They illustrate how two tasks can communicate via a TCP connection. SRVR is the higher priority of the two. It creates a socket and waits for a connection on it. Then CLNT runs because SRVR has blocked. It creates a socket and establishes a connection with the socket that CLNT created. Then CLNT repeatedly sends data through the TCP connection to SRVR, and SRVR reads it, operates on it, and sends it back

to SRVR. The CLNT task prints a message each time it runs through its loop.

If neither pNA+ nor pHILE+ is present, then FILE, CLNT, and SRVR do not get created, so after ROOT suspends itself only the IDLE task is left.

The output to the target's console will look something like the following depending on what components you are using (this run includes pHILE+ and pNA+):

```
Standard output device initialized...

Real-time clock initialized...

RAM disk initialized - size is 0xC8 blocks...

SCSI driver initialized...

* Hard disk at SCSI ID 0x5. Vendor: TEAC Model: FC-1 HF 011

* Hard disk at SCSI ID 0x6. Vendor: QUANTUM Model: LP52S
950509405

Please enter today's date (mm/dd/yyyy): 11/05/1995

Please enter the current time (hh:mm:ss): 15:00:00

Date & time successfully set

Initializing the pHILE+ sample application...

Task 'FILE' created...

pNA initialized...

Task 'SRVR' started...

Task 'CLNT' started...

'ROOT' task's initialization completed

pNA+ demo loop completed iteration 0x1

Disk volume 5.5 initialized...

pNA+ demo loop completed iteration 0x2

pNA+ demo loop completed iteration 0x3

pNA+ demo loop completed iteration 0x4

pNA+ demo loop completed iteration 0x5
```

```
pNA+ demo loop completed iteration 0x6
pNA+ demo loop completed iteration 0x7
pHILE+ demo loop completed iteration 0x1
Output will continue until system is halted.
```

8.8 nfs

The **nfs** application demonstrates the use of pSOSystem NFS client services.

This application is an example of how to use the NFS client services of pSOSystem so that it can mount the file system of a remote system, such as a workstation.

In order to run this application, you must have the following pSOSystem components in your system:

- pSOS+ (or pSOS+m) real-time kernel
- pREPC+ standard C library
- pHILE+ file system manager
- pNA+ networking manager
- pRPC+ Remote Procedure Call library

In **root.c**, NFS_SERVER_ADDR must be set to the IP address of your NFS server. This parameter is passed by **nfsmount_vol()** and is used by pNA+ to locate the server. Example:

```
#define NFS_SERVER_ADDR     0xC06736BE
                                /* NFS server IP address */
```

(Replace 0xC06736BE with the correct IP address for the host system.)

Next, NFS_FS must be set to the name of a file system that has been exported by your NFS server. Example:

```
#define NFS_FS          "/usr"      /* Remote File System */
```

Lastly, PATH_NAME must be set to the pathname of a directory on your NFS server for which the values of UID and GID defined in **sys_conf.h** have access permission. This pathname of course must be under the

NFS_FS name (NOTE: This pathname must not be a link to another directory in another file system.)

#define PATH_NAME "/xxx*/

Note the slash symbol (/) before and after the pathname.

To create a directory to use, change directories (**cd**) to the file system to be mounted (NFS_FS) and enter:

mkdir xxx

chmod a+rwx xxx

Then you should have a directory that can be accessed by any user so UID and GID should not matter.

The definitions for DIR_NAME and FILE_NAME can remain unchanged as long as they don't conflict with filenames in your chosen directory.

To ensure that the file system you want to remotely mount has been exported, check the contents of the file **/etc/xtab** on your server. If it does not contain a line with the pathname of the file system you want to mount on it, that file system has not been exported. To export it, enter the directory pathname on a line in the file **/etc/exports** on your server (Create **/etc/exports** if it does not already exist.) Make the file system available for remote mounting by using the command:

exportfs -a

Example:

Add the line

/usr

to the /**etc/exports** file. Then type command:

exportfs -a

Now, if you enter the following command:

cat /etc/xtab

you should see this output:

```
cat /etc/xtab
/usr
```

Unless you are sure that it is already running, you can start the RPC mount demon with the by entering the **rpc.mount** command. The demon that processes NFS client requests is started with the command **nfsd**.

The root task initializes serial I/O, the pSOS+ clock, pNA+, and creates the nfs_sample task. The nfs_sample task runs though various pHILE+ calls to mount a remote file system, create a file, and create a directory. Network File System support through pHILE+, pRPC+, and pNA+ is exercised by opening and closing the file, reading from and writing to the file, moving the file, changing directories and performing lseek operations.

Typical output on the target console is as follows:

```
Mounting volume “15.5”
  Creating file “15.5/xxx/tmp_file”
  Opening “tmp_file”
  Writing 0x80 bytes to “tmp_file”
  lseek to offset 0x0 from beginning of file
  Reading from “tmp_file”
  Closing “tmp_file”
  Making Directory “15.5/xxx/tmp_dir”
  Changing to “tmp_dir”
  Moving file “tmp_file” to directory “tmp_dir”
  Opening file “tmp_file”
  Lseek to offset 0x0 from end-of-file
  Old Position in File : 0x0
  Writing 0x80 bytes to “tmp_file”
  Lseek to offset 0 from beginning of file
  Reading from “tmp_file”
  Closing “tmp_file”
  Removing “tmp_file”
  Removing directory “tmp_dir”
```

```
Amounting volume “15.5”
 TEST ENDED
```

9 Understanding and Developing Board-Support Packages

This chapter explains what a pSOSystem board-support package (BSP) is and gives a detailed explanation of all functions needed and used by a BSP. This chapter also describes the process of developing a custom BSP.

The software support for a target system is called a board-support package or BSP. The BSP is a collection of functions that are hardware specific. These functions include:

- Target system hardware initialization during a system boot.
- Interface to the devices present on the Target System. These devices include timer chips, Ethernet controller chips, and serial and SCSI controller chips. The functions that control these chips are called device drivers.

pSOSystem contains several BSPs for off-the-shelf target boards. These are detailed in Appendix A, "Board-Specific Information."

- The **bsps/template** directory contains skeletons of all the functions that are needed in a BSP. You may find this useful if you are not using one of the pSOSystem supported BSPs and you have to create your own BSP. These skeleton functions are

commented to guide you in coding each function so it will interface with the rest of the elements of pSOSystem.

- To make the job of creating a custom BSP easier, pSOSystem has common functions that you can call from your custom BSP code. The use of these functions will depend on what the custom BSP needs to support. These common functions are described later in this chapter.
- pSOSystem also contains several low-level device drivers that are written in a way that makes them independent of the target system they are being used with. Through the use of macro define directives and small changes to the BSP's Makefile, these drivers can be used in a custom BSP. These drivers are also described later in this chapter.

9.1 template Directory

The **bsps/template** directory contains files that need additional coding to create a board-support package. The files in the **bsps/template** directory and the **bsps/template/src** directory contain skeletons of the functions that are used to initialize and drive the board level functions of pSOSystem. The next section, "Template File List," gives you a list of files that are in the template and a short description. The "Detailed Function Description" section provides additional details about the files in the **template** directory. All of the BSPs in pSOSystem follow this template. By knowing how the template is structured, you will also know how the supported BSPs are structured.

9.1.1 Template File List

Below is a list of files in the **bsps/template** directory.

File	Description
app.lnk	Linker file for application only (no OS).
bsp.h	Contains board-specific define statements.

File	Description
bsp.mk	Make include file used to make **bsp.libs** and applications.
os.lnk	Linker file for OS only (no application).
ram.lnk	Linker file that links OS and application together.
ramrc.lnk	Linker file for application that uses OS in ROM.
rom.lnk	Linker file used to link ROM based code.

In the **bsp/template** directory, the directory for source files is called **src**. Below is a table of files in the **src** directory.

File	Function	Description
board.c	InitBoard	Initializes the board.
	ClrAbortInt	Clears interrupt caused by front panel abort switch.
	RamSize	Gets the size of RAM on the board.
	SysInitFail	Reports a system initialization failure.
board.h		Hardware-specific defines for portable drivers.
bddialog.c	BspGetdefaults	Gets a copy of the structure containing the default hardware dialog parameters specific to the BSP.
	BspModify	Conducts a hardware-specific dialog to modify a board's parameters.
	BspPrint	Prints out the current values of the hardware-specific dialog parameters.
	BspUse	Takes the BSP parameters that have been decided on and use them to set the BSP.
bspcfg.c	BspSetup	BSP specific setup operations.
init.s	Hdwinit	Assembly code initialization of hardware.
makefile		Makefile for the board-support library.

File	Function	Description
smem.c	SetVmeAddress	Establishes the board's VMEbus address.
	SmemBus2Local	Converts a VME address to a local address.
	SmemLocal2Bus	Converts a local address to a VME address.
	SmemIntInit	Initializes this node for interprocessor interrupts and writes interrupt method data to the shared memory directory.
	SmemIntNode	Performs a software interrupt on a CPU node.
	SmemIntClear	Called as part of the shared memory ISR to clear shared memory interrupt.
timer.c	RtcInit	Starts the real-time clock for use as the pSOS+ tick clock.
	RtcIsr	Handles a clock tick interrupt.
	Delay100ms	Spins for 100 milliseconds.
	tmFreq	Returns frequency of second timer used in pMONT.
	tmReset	Resets second timer used in pMONT.
	tmRead	Reads counter value of second timer used by pMONT.

9.1.2 Detailed Function Description

This section goes into more detail about the functions and files that are in the **template** directory.

9.1.2.1 app.lnk

The **app.lnk** linker command file for linking a pSOSystem application only. **app.lnk** creates a separate downloadable file which does not contain an OS. See the **README** file in the **bsps/template** directory for more information about the options included in this file.

9.1.2.2 bsp.h

This file contains two sets of definitions:

- The characteristics and capabilities of the board and its BSP, such as processor type peripheral devices available.
- The vector numbers for the various interrupts and exceptions which can occur on the board.

The following are the define statements in the **bsp.h** file.

- **BSP_VERSION** specifies revision level of pSOSystem which this BSP was written to support. If this number does not match the pSOSystem version defined in **include/version.h**, you get a warning at compile time that says:

  BSPVERSION and pSOSystem VERSION do not match.

  Note: This check is done in the **configs/std/sysinit.c** file.

- **BSP_CPUFAMILY** is the CPU processor family for this board. For example, 68000 for the Motorola 68K processor family. This define is used in the file **include/bspfuncs.h** for various CPU type specific definitions.
- **BSP_CPU** defines the specific processor within the board's family such as 68040 or 68060. This define statement is used in **include/bspfuncs.h** and **mmulib.h** to distinguish the characteristics of processors within a CPU family.
- **BSP_FPU** defines the existence of a floating-point unit. This is used in **configs/std/sysinit.c** to set the FPU bit of the **cputype** element in the node configuration table. Components will use this to see if they should include floating-point instructions in their execution.
- **BSP_MMU** defines the existence of a memory management unit (MMU). This define statement is used in **configs/std/sysinit.c** to set the MMU bit of the 'cputype' element in the node configuration table.
- **BSP_RAM_BASE** defines the starting address of RAM as seen by CPU. This is used in **configs/std/sysinit.c**, **configs/std/pSOScfg.c**, and **configs/std/pnacfg.c** to calculate the setting of pSOS+ region-zero and the check for memory overflow

conditions at run time when allocating memory during system installation.

- **BSP_VME** indicates whether the board has a VME bus or not. This can be set to YES or NO. This define is used in **configs/std/dialog.c** to determine whether VME questions should be asked. It is also used in **configs/std/sysinit.c** to see if the VME address should be set.
- **BSP_LITTLE_ENDIAN** indicates whether the board is in little endian or big endian layout. This define is used by several device drivers to make them portable across different platforms. It can be set to YES or NO. (For 68K boards this should be set to NO.)
- **BSP_PARMS** is the size in bytes of board-specific parameters which will be stored in nonvolatile RAM and modified with the optional startup dialog. This should be set to 0 if there is no board-specific dialog. This define statement is used by **configs/std/sysinit.c** and **configs/std/dialog.c** to determine whether the call to board dialog functions and declarations should be compiled into the code and executed at startup time. The value of BSP_PARMS should be a multiple of four to ensure long-word alignment.
- **BSP_ABORTSW** determine whether a manual abort switch exists. This can be set to YES or NO. This define statement is used in the **configs/std/probecfg.c** file to compile in code that will set the interrupt vector for the manual abort interrupt.
- **BSP_SERIAL** indicates the number of serial channels supported by the hardware (0 for none). It is used in **configs/std/probecfg.c** to compile in the settings of pROBE+ hooks to the serial driver. It is also used in **configs/std/sysinit.c** for some compile-time verification.
- **BSP_SERIAL_MINBAUD** and define **BSP_SERIAL_MAXBAUD** should be set to the baud-rate range of the serial channels. These define statements are used in **configs/std/dialog.c** to check the range of answers given by the dialog user at run time.
- **BSP_LAN1** indicates whether the BSP supports a LAN driver. It can be set to YES or NO. This define statement is used in

configs/std/dialog to compile conditionally in code to support LAN driver questioning. It is also used in **configs/std/sysinit.c** for run-time verification. In an application's directory, it is used in **drv_conf.c** to compile conditionally to set up the LAN driver. If more than one LAN is attached to the target, you can add additional define statements for them. For example, the second LAN would have a define for **BSP_LAN2**. The **drv_conf.c** file would have to be expanded to set up the additional LANs.

- **BSP_LAN1_ENTRY** should be set to the name of the entry function to the LAN driver. This is used in **drv_conf.c** to set up the LAN driver. If more than one LAN is attached to the target, you can add additional define statement for them. For example, the second LAN would have a define for BSP_LAN2_ENTRY. The **drv_conf.c** file would have to be expanded to set up the additional LANs.
- **BSP_LAN1_MTU** should be set to the Maximum Transmission Unit for the LAN1 driver. This is used in the application's **drv_conf.c** file to set up the LAN1 driver. Similar to **BSP_LAN1** you can also add define statements for more than one LAN driver.
- **BSP_LAN1_HWALEN** is set to the hardware address length for your hardware, for example, Ethernet is six. It is used in an application's **drv_conf.c** file to configure the LAN driver.
- **BSP_LAN1_FLAGS** is used to tell pNA+ what kind of interface the LAN has on the system. The following can be ORed together:

Define	Hex Value	Description
IFF_POLL	8000	Interface is a polling type.
IFF_BROADCAST	0001	NI supports broadcasts.
IFF_RAWMEM	2000	Driver uses the pNA+-dependent packet interface.
IFF_MULTICAST	0800	Driver supports multicast.

- **BSP_LAN1_PKB** is set to the number of packet buffers for RAM code. This is used when creating a RAM-based system in the **bsps/*xxx*/src/bspcfg.c** file (where *xxx* is the name of the BSP directory) to configure the LAN packet buffers array. This is used when there is an i82596 LAN chip for the LAN hardware interface.
- **BSP_LAN1_PKB_ROM** is set to the number of packet buffers for ROM code. This is used when creating a ROM-based system in the **sps/*xxx*/src/bspcfg.c** file (where *xxx* is the name of the BSP directory) to configure the LAN packet buffers. As ROM requirements are usually less than the RAM requirements for LAN buffers, this number can be considerably less then BSP_LAN1_PKB. This is used with an i82596 LAN chip for the LAN hardware interface.
- **BSP_LAN1_TCB** is used for an i82596 driver to set the transmission control blocks in a RAM-based system. These are set in the **bsps/*xxx*/src/bspcfg.c** file (where *xxx* is the name of the BSP directory).
- **BSP_LAN1_TCB_ROM** is used for an i82596 driver to set the Transmission Control Blocks for a ROM-based system. These are set in the file **bsps/*xxx*/src/bspcfg.c** (where *xxx* is the name of the BSP directory).
- **BSP_LAN1_TBD** is used for an i82596 driver to set the transmission block descriptors in a RAM-based system. These are set in the **bsps/*xxx*/src/bspcfg.c** file (where *xxx* is the name of the BSP directory).
- **BSP_LAN1_TBD_ROM** is used for an i82596 driver to set the transmission block descriptors for a ROM based system. These are set in the **bsps/*xxx*/src/bspcfg.c** file (where *xxx* is the name of the BSP directory).
- **BSP_SMEM** indicates whether the hardware is supports a shared memory interface. This define statement is used in the application's **drv_conf.c** file to compile conditionally in code to install a shared memory network interface. It is also used in **configs/std/dialog.c** to compile code that asks questions about a shared memory interface. The **configs/std/sysinit.c** file also uses this define to set up the multiprocessor configuration table.

- **BSP_MP_TAS** sets the queue lock implementation for shared memory drivers. A '1' means the Test and Set instruction (TAS) is used. Because not every board supports TAS, use a '0' unless you are sure that all nodes can do TAS correctly. This is used in the files **drivers/ki_smem.c** and **drivers/ni_smem.c**. The actual function that does the TAS is located in the **drivers/ki_call.s** file.
- **BSP_INT_MODE** is used to determine the method used to signal the arrival of packets to a destination node in a shared memory system. A setting of '0' uses a polling method. A setting of '1' uses an interrupt method. If set to '1', the files **drivers/ki_smem.c** and **ni_smem.c** will compile in code to set up the interrupt procedure **SmemIsr**.
- **BSP_SCSI** is set to YES if there is a SCSI driver and hardware for a SCSI bus. If no SCSI support is available or none is needed, this should be set to NO. Once set to yes, the SCSI driver in **drivers/scsi.c** will be compiled into the system.
- **BSP_SCSIINC** is used if BSP_SCSI is set to YES. It must be set to the name of the correct include file for the actual NCR SCSI chip used. This is used in the **drivers/scsi.c** file. This can be set to **scsi/ncr53cxx.h** or **scsi/wd33c93.h**.
- **BSP_SCSI_TAPE** can be set to YES if your application needs a SCSI tape driver. If set to YES, the code for the SCSI tape driver located in **drivers/scsi.c** will be compiled into the system.
- **BSP_TIMER2** can be set to YES or NO. Setting it to YES indicates the board has a second timer and driver functions that can be used by pMONT. The **configs/std/pmontcfg.c** file will conditionally compile in code that calls the functions **tmFreq**, **tmReset**, and **tmRead**. These functions must be provided by the BSP.
- **V_BUSERR** is set to the vector number of the bus error vector. This number is used in a call to **SysSetVector** in the **configs/std/probecfg.c** file.
- **V_ADDRERR** is set to the vector number of the address error vector. This number is used in a call to **SysSetVector** in the **configs/std/probecfg.c** file.

- **V_TRAP0** is set to the vector number of the pROBE+ breakpoint vector. This number is used in a call to **SysSetVector** in the **configs/std/probecfg.c** file.
- **V_HDWBKPT** is set to the vector number of the pROBE+ hardware break point vector. This number is used in a call to **SysSetVector** in the **configs/std/probecfg.c** file. This is only needed if the hardware supports hardware breakpoints.
- **V_ABORT** is set to the vector number of the hardware abort switch. This number is used in a call to **SysSetVector** in the **configs/std/probecfg.c** file. This vector is used to enter pROBE+ when the abort switch is pressed. Note the BSP_ABORTSW must also be set to YES.
- **V_TRAP11** is set to the vector used for the pSOS+ service call. This number is used in a call to **SysSetVector** in the **configs/std/psoscfg.c** file.
- **V_TRAP12** is set to the vector used for the pSOS+ I/O call. This number is used in a call to **SysSetVector** in the **configs/std/psoscfg.c** file.
- **V_TRAP13** is set to the vector used for the pSOS+ return from interrupt vector number. This number is used in a call to **SysSetVector** in the **configs/std/pSOScfg.c** file.
- **V_TIMER** is set to the vector used for the periodic tick timer interrupt. This is available for use by the **bsps/*xxx*/src/timer.c** file (where *xxx* is the name of a BSP directory).
- **V_FLINE** is set, for boards that have processors that support floating-point, to the vector used for an F-line exception (invalid floating-point instruction used to indicate need for emulation). This is used in the **configs/std/sysinit.c** file in a **SysSetVector** call to set the vector to the pSOS+ F-line interrupt handler.
- **V_BSUN** is set, for boards that have processors that support floating-point, to the vector used for a Branch/Set On Unordered. This is used in the **configs/std/sysinit.c** file in a **SysSetVector** call to set the vector to the pSOS+ **bsun** interrupt handler.
- **V_INEX** is set for boards that have processors that support floating-point to the vector used for an Inexact. This is used in

the **configs/std/sysinit.c** file in a **SysSetVector** call to set the vector to the pSOS+ **inex** interrupt handler.

- **V_DZ** is set for boards that have processors that support floating-point to the vector used for a divide-by-zero. This is used in the **configs/std/sysinit.c** file in a **SysSetVector** call to set the vector to the pSOS+ **dz** interrupt handler.
- **V_UNFL** is set for boards that have processors that support floating-point to the vector used for an underflow. This is used in the **configs/std/sysinit.c** file in a **SysSetVector** call to set the vector to the pSOS+ **unfl** interrupt handler.
- **V_OPERR** is set for boards that have processors that support floating-point to the vector used for an operand error. This is used in the **configs/std/sysinit.c** file in a **SysSetVector** call to set the vector to the pSOS+ **operr** interrupt handler.
- **V_OVFL** is set for boards that have processors that support floating-point to the vector used for an overflow. This is used in the **configs/std/sysinit.c** file in a **SysSetVector** call to set the vector to the pSOS+ **ovfl** interrupt handler.
- **V_SNAN** is set for boards that have processors that support floating-point to the vector used for a Signaling Not-A-Number error. This is used in the **configs/std/sysinit.c** file in a SysSetVector call to set the vector to the pSOS+ **snan** interrupt handler.
- **V_UNSUPP** is set for boards that have processors that support floating-point to the vector used for an unsupported data type. This is used in the **configs/std/sysinit.c** file in a SysSetVector call to set the vector to the pSOS+ **unsupp** interrupt handler.
- **V_EFFADD** is set for boards that have processors that support floating-point to the vector used for an Unimplemented Effective Address (68060 only). This is used in the **configs/std/sysinit.c** file in a **SysSetVector** call to set the vector to the pSOS+ **effadd** interrupt handler.
- **EXCLUDED_VECTORS** can optionally set a list of vectors to be left alone by pSOSystem. Vectors in the “excluded” list will not be initialized. The list should contain the vectors, separated by commas. The most common use of this is to allow debugging of

the pSOSystem with a ROM monitor. For example, if your ROM monitor uses TRAP #1 to implement instruction breakpoints, you could let the monitor maintain control by defining EXCLUDED_VECTORS as follows:

EXCLUDED_VECTORS V_BUSERR, V_ADDRERR, V_TRACE, V_TRAP1, 17

In this case, the downloaded pSOSystem would not modify any of these vectors, so bus and address errors would be fielded by the ROM monitor. Software monitors usually make use of the trace exception as well. The 17 is the anchor slot in the vector table. For example, if the anchor is set in **rom.lnk** to 0x44, divide 0x44 by 4 = 17 decimal. This keeps the ROM anchor from getting overwritten The anchor address must be different for the RAM code used. A good address would be 0x48 (change made in **ram.lnk**).

The remaining entries in the **bsp.h** file are board specific. If you are making a custom BSP, you can add additional entries as needed.

9.1.2.3 bsp.mk

The **bsp.mk** file is included in the BSP Makefile and the application Makefile. **bsp.mk** sets the CPU variable for these files. The CPU variable is set to the processor type as defined by the compiler. For example, CPU=68040.

9.1.2.4 os.lnk

The **os.lnk** linker command file is for linking the pSOSystem software only (BSP kernel, and other components). **os.lnk** creates a separate downloadable file which does not contain an application. The application should be built with the **app.lnk** file to ensure consistency with this file. See the **bsps/template/README** file for more information about the options included in this file.

9.1.2.5 ram.lnk

The **ram.lnk** linker command file is for linking a pSOSystem application and OS (BSP kernel, and components together). This creates a downloadable file which contains the application and the OS. See the

bsps/template/README file for more information about the options included in this file.

9.1.2.6 ramrc.lnk

The **ramrc.lnk** is a linker command file that allows your downloaded system to use the OS components (pSOS+, pROBE+, etc.) which are located in ROM. This is primarily useful in the following situations:

- You are downloading your system to your target board via a serial connection. By using the components contained in your ROM, you can save considerable download time.
- Your target system does not have much RAM. By using the component code contained in ROM, you can save RAM space.

Integrated Systems recommends placing each of the components in its own linker section, which is named for the component. For example, the pSOS+ code is located in the "pSOS" section, the pROBE+ code is in the "pROBE" section, etc. The component code is linked with the rest of the system as usual, but we use the SECT command to assign hard addresses to the sections containing the components. The addresses used in the SECT command are the ROM addresses of the components. Finally, by using the ABSOLUTE command, you can leave sections containing components out of the output file.

The end result is an output file that does not contain the components, but does have references to the components resolved to the ROM addresses of the components. See the **bsps/template/README** file for more information about the options included in this file.

9.1.2.7 src/board.c

The **board.c** file is used to initialize the board specific hardware and functions that provide hardware specific information. This file contains the following functions:

- **InitBoard** initializes the board-specific hardware. This function is called from the assembly language file **bsps/*xxx*/src/init.s** (where *xxx* is the name of the BSP directory). The function that calls it is **HdwInit**. **Hdwinit** will do whatever board initialization needs to be done in assembly language. **InitBoard** should continue the board initialization that can be done in C. This function must call the **SysInitVars** function to

clear the zerovars section of RAM. The **SysInitVars** function needs to be called once the hardware for the RAM has been set up. Another function that must be called is **SysInit**. The **SysInit** function continues with the non-hardware configuration of the system which includes running any dialog at boot time, the setting up of the component's configuration tables and any driver set-up. **SysInit** must be called last in the **InitBoard** function because it does not return. The syntax is:

void InitBoard(void)

- **ClrAbortInt** clears interrupt caused by 'abort' switch or button. This is an interrupt function for the abort button if there is one. All this function needs to do is clear the hardware of the interrupt (if needed). (There is a wrapper that controls the interface with pROBE+ if it is installed in the system. The wrapper calls the pROBE+ manual break entry point.) The syntax is:

  void ClrAbortInt(void)

- **RamSize** returns the size of the onboard DRAM. This function is used by any function that needs to know how much DRAM the system contains. It is used to configure the system in the files **sysinit.c**, **pSOScfg.c**, and **pnacfg.c** in the **configs/std** directory. The syntax is:

  ULONG RamSize(void)

- **SysInitFail** reports a system initialization failure. This function is called during system start up from files in the **configs/std** directory when an unrecoverable error occurs. It should continually print an error string to the system console. The syntax is:

  void SysInitFail (const char *string)

9.1.2.8 src/board.h

The **board.h** file contains defines that are specific to the target hardware. These defines differ depending on the device drivers that are used on the board. Drivers that are supported by pSOSystem use the define statements that are located in this file. Each of these drivers contains a comment section at the top of the file that details the define statements needed by the driver. If you are writing a custom BSP, these comments

should be copied out of the driver file and placed into the **board.h** file of the custom BSP. Then code should be added for the define statement.

For example, if the driver for the i82596 driver is located in the **bsps/devices/lan/i82596.c** file, the comment section at the top of the file has the following comments for the define statement that need to go into the **board.h** file:

```
/* BD_GET_ETHER_ADDR: Macro to get Ethernet address. A mandatory
macro */
/* NOTE: This address is usually read from nonvolatile RAM. */
/* Then it will store it in the Ether_Addr array. */
```

You will need to copy the above comment (and the other macro comments) into the **board.h** file and add the appropriate define statement. For example, with the **m167** BSP, use the following define statement:

```
#define BD_GET_ETHER_ADDR \
   {\
   int i; \
   for (i = 0; i < 6; i++) \
      Ether_Addr[i] = BRDI_CNFG->ethernet_addr[i]; \
   }
```

9.1.2.9 src/bpdialog.c

The **bpdialog.c** file is used in the boot up code to execute any hardware-specific dialog during the Boot ROM dialog. It can be omitted and the define statement BSP_PARMS in **bsp.h** can be set to NO if your hardware has no specific parameters.

An example of a hardware-specific parameter is the hardware Ethernet address. Some boards need to have the hardware Ethernet address stored in hardware while others have some or none of the hardware Ethernet address stored in the hardware. In either case, the **bpdialog.c** file can provide a way during ROM boot or application startup to have a user enter or change the hardware Ethernet address.

If you are going to use this feature, you will need to provide a structure, called **ParmStruct** in the examples, to hold the data you want to save in NVRAM. BSP_PARMS in the **bsp.h** file must be set to the size of that structure. Then you need to provide the following functions:

- **BspGetdefaults** places a copy of the structure containing the default values for the hardware-specific dialog parameters in

the area pointed to by **ParmStruct**. This function is called by **configs/std/sysinit.c** at boot time. The syntax is:

void BspGetdefaults (void *ParmStruct)

- **BspPrint** prints out the current values of the hardware specific dialog parameters using a pointer to the **ParmStruct** which contains the data in NVRAM and a pointer to a print function to be used to do the printing. This function is called from **configs/std/dialog.c** at boot time. The syntax is:

  void BspPrint (void *ParmStruct, void (*PrintRoutine) (char *format,...))

- **BspModify** conducts a hardware-specific dialog to modify the parameters pointed to by ***ParmStruct**. BspModify uses a pointer to a print function to be used for printing and a pointer to a prompt function to pREPC+ for input. This function is called from **configs/std/dialog.c** at boot time. The syntax is:

  ```
  void BspModify (void *ParmStruct,
     void(*PrintRoutine) (char *format,...),
      void(*PromptRoutine)(char *prompt, PARM_TYPE ptype,
      void *paRAMptr))
  ```

- **BspUse** takes the BSP parameters that have been decided on and use them to set the BSP. In other words, this function should apply the values selected (if needed). It takes as an argument the pointer to the **ParmStruct**. It is called from **configs/std/sysinit.c** at boot time. The syntax is:

  void BspUse (void *ParmStruct)

The **bpdialog.c** file has an example of the use of these functions. The **PrintRoutine** and the **PromptRoutine** are provided by pSOSystem in the **configs/std/pollio.c** file, which is explained with the configuration files. These functions are used for printing and prompting.

9.1.2.10 src/bspcfg.c

The **bspcfg.c** file is used to provide a way to configure the BSP at application compile time. Because the **bsp.lib** is independent of the application, a means is needed to provide the application some control over the configuration of the BSP. For example, the LAN driver needs to know if multicast addressing will be required and, if so, how many multicast addresses will be needed. The LAN driver will then allocate

buffers for multicast based on the number needed. Rather than this value being hard-coded into the driver to be included in **bsp.lib**, the user can set a define statement in the application file **sys_conf.h**. The **BspSetup** function in this file uses that define statement to allocate space for the multicast buffers, and set a global pointer to allow the LAN driver access to the buffers.

The only function needed in this file is **BspSetup**. It has two arguments, the pointer to free memory **FreeMemPtr** and the pointer to the node configuration table, **NodeCfg**. The **FreeMemPtr** can be used to allocate needed memory for the BSP. The **FreeMemPtr** must be incremented by the number of bytes used and returned to the caller. The syntax is:

UCHAR *BspSetup (UCHAR *FreeMemPtr, NODE_CT *NodeCfg)

This function must be present in the BSP. However, all it needs is one line of code to return the **FreeMemPtr**. This function is called at boot time from the **configs/std/sysinit.c** file.

9.1.2.11 src/init.s

The **src/init.s** file should contain the HdwInit function. HdwInit is called by the **_START** function located in the **configs/std/begin.s** file. It is called when the system is powered on or reset. It should perform all necessary hardware initialization needed by the board. It calls a C function to complete the initialization of those things that can be done in C. The C function is called **InitBoard**.

Initialization of the hardware should set the system to a known state with the hardware configured as it should be when the system is first powered on.

Some of the functionality that might be needed in this file includes:

- RAM set up may include setting of base registers, chip select registers, RAM speed, RAM parity enable, RAM enable, and initialization of parity memory.
- CPU caches are set to a default state. You may need to write the data in the data cache out to memory and turn instruction and data cache off and invalidate both caches. (Functions are provided to do this. These functions are in the **bsps/devices/68k/cpu*xxx*.s** files. See the “Devices” section for more information on them.)

- CPU control registers are all set to a default state.

HdwInit must call the function **InitBoard** which must be the last thing it does because **InitBoard** does not return.

9.1.2.12 src/makefile

The **makefile** brings together the different parts of a BSP and creates a library called **bsp.lib** that is linked into the final system image file. To do this, the **makefile** draws files from several areas of pSOSystem. The **makefile** includes all of the files in the **bsps** directory, selected files in the **drivers** directory, and selected files in the **bsps/devices** directory. Each directory has its own make rules. The **bsps** directories define make rules in the **makefile**. The rules for the files in the **drivers** directory are defined in the **rules.mk** file. The rules for the files in the **bsps/devices** directory are defined in **bsps/devices/rules.mk**.

If you are developing a custom BSP, you will want to include some of the common code from pSOSystem in the building of your BSP. To include a file in the make from the **drivers** or **devices** directories, you need to enter two lines into the **makefile** in the **bsps** directory.

- In the **SRC_OBJ** define, a line for the object filename must be added. For example, if the file **scsi.c** was to be added to your **makefile**, you would need to add the line:

  **obj/scsi.o **

  This command executes the rule in the **drivers/rules.mk** file for **scsi.o** and compiles the **scsi.c** file.

- You need to add the **scsi.o** object file to the **bsp.lib** library. You do this by adding a line into the library section of the **makefile**. For example:

  @echo addmod obj/scsi.o >> tmp.cmd

The **bsps/template/makefile** contains examples of how this is done.

9.1.2.13 src/smem.c

The **smem.c** file contains shared memory functions for a VME bus. The following functions are needed:

- **SetVmeAddress** establishes the board's VMEbus address. This function sets the **VME_base_addr** global variable to the address supplied in the **BaseAddress** argument. The address

supplied should be the starting address of the board's VME bus address. (**VME_base_addr** is used to convert a VME address to a local RAM address on the board and to convert a local address to a VME address.) This function also initializes any hardware registers that need the VME base address. This function is called by the **SysInit** function during system startup if the define BSP_VME is set. BSP_VME is set in the **bsp.h** file. The syntax is:

void SetVmeAddress (ULONG BaseAddress)

- **SmemBus2Local** converts a VME address to a local address. This function is used in **drivers/ki_smem.c** and **drivers/ni_smem.c**. It takes as an argument a VME based address and converts it to a local bus address. The syntax is:

  void *SmemBus2Local(void *vme_addr)

- **SmemLocal2Bus** converts a local address to a VME address. This function takes a local bus address as an argument and converts it to a VME address. The syntax is:

  void *SmemLocal2Bus(void *loc_addr)

- **SmemIntInit** initializes this node for interprocessor interrupts and writes interrupt method data to the shared memory directory. This function is called by **drivers/ki_smem.c** and **drivers/ni_smem.c**. **SmemIntInit** sets up the interrupt vector and any control registers so other nodes on the VME bus can interrupt this node. Nodes may require different methods to interrupt them. To accommodate different methods there are changes via a union to the Interrupt Method Data Structure. See the INTR_METHOD in the **include/bspfuncs.h** file for more information on the Interrupt Method Data Structure. The syntax is:

  void SmemIntInit (INTR_METHOD *int_method, ULONG node, ULONG (*isr) (void))

- **SmemIntNode** performs a software interrupt on a VME node. This function is called by **drivers/ki_smem.c** and **drivers/ni_smem.c** to cause an interrupt to happen on a particular node. How your hardware handles sending an interrupt to another node is specific to your hardware and the node that is being interrupted. The syntax is:

int SmemIntNode (INTR_METHOD *int_method, ULONG target)

- **SmemIntClear** is called as part of the shared memory ISR to clear shared memory interrupt. This function is board dependent. It must clear the VME interrupt. **SmemIntClear** is called from **drivers/smem_isr.c**. This syntax is:

  void SmemIntClear(void)

9.1.2.14 src/timer.c

The **timer.c** file contains timer functions that are used by pSOS+ and pMONT to control the tick timer and profiling. This is a driver that is installed by a call to **InstallDriver** in the **drv_conf.c** file in the application directory. The following functions are needed:

- **RtcInit** starts the real-time clock for use as the pSOS+ clock. This is called through the **de_init** system call. The function needs to set up a timer to interrupt at an interval that matches the **kc_ticks2sec** entry in the pSOS+ configuration table. The syntax is:

  void RtcInit (struct ioparms *p)

  This function can access the pSOS+ configuration table through the system's anchor. The anchor is a global variable that points to the node configuration table. The **kc_ticks2sec** element can be obtained as follows:

  ```
  extern NODE_CT *anchor;
  anchor->pSOSct->kc_ticks2sec
  ```

- **RtcIsr** is an interrupt handler for clock tick interrupt. This function is the tick timer ISR. It needs to do whatever is required to acknowledge the interrupt and start the timer counting down again. Then it must call **tm_tick** to announce to pSOS+ a tick interrupt has occurred:

  ```
  tm_tick();
  ```

  The syntax is:

  static void RtcIsr(void)

- **Delay100ms** does nothing for 100 milliseconds and returns. It is used only before pSOS+ is initialized. It is used in the

SysInitFail function located in **board.c** and **RarpEth** function in **drivers/rarp.c**. The syntax is:

void Delay100ms(void)

- **tmFreq** returns the clock frequency of second timer used by pMONT. The syntax is:

  unsigned long tmFreq(void)

- **tmReset** resets second timer used by pMONT. This function initializes the second timer to zero and then starts the timer running. The syntax is:

  void tmReset(void)

- **tmRead** returns the counter value of second timer used by pMONT. The syntax is:

  unsigned long tmRead(void)

9.2 devices Directory

The **devices** directory contains low-level device drivers and low-level system functions that may be needed by a board-support package. The code in the files in this group is specific to hardware devices but not specific to the board these devices are on. The low-level code in the **devices** directory is intended to be used unaltered. This is done through the use of macros that are placed in the code where board-specific functionality is needed. To use a low-level driver in this group you will need to define the macros that are specific to the driver in a file called **board.h** located in the directory that contains your board-support package.

The low-level drivers are grouped in directories by type. For example, all low-level serial drivers are located in the **bsps/devices/serial** directory. (Each low-level driver also has a corresponding include file in the same directory.) The **rules.mk** file, located in the **bsps/devices** directory, contains the rules for making all of the low-level drivers in **bsps/devices**. If you are using one of the chips supported by a low-level driver located in the **devices** directory then you can include the **rules.mk** file in your BSP **makefile** and add the low-level driver the object list and library modules and it will be included in your BSP.

The hardware-specific code located in the **devices** directory is divided into the following directories:

- **68k** contains 68K-specific assembly language code for setup and control of Motorola 68000 processors.
- **common** contains common functions that can be used by any processor type.
- **LAN** contains chip-dependent code for several Ethernet LAN drivers along with common code for SNMP control of the LAN drivers.
- **scsi** contains chip-dependent code for SCSI drivers.
- **serial** contains chip-dependent code for serial drivers.

9.2.1 devices Directory File List

The following table contains a list of files in the **devices/68k** directory.

File	Function	Description
bhand000.s bhand030.s bhand032.s bhand040.s bhand060.s	SysBusError	System bus error handlers for the 68000, 68030, CPU32, 68040 and 68060, respectively.
cpu000.s cpu030.s cpu040.s cpu060.s cpu0x0.s		These files contain cpu specific start-up initialization code in assembly for the MC68000 through MC68060. (**cpu0x0.s** covers both MC68010 and MC68020.) They contain the following functions:
	Sys_CPU_Init	CPU initialization to default values.
	Sys_Cache_Init	Cache initialization.
	Sys_Dcache_Inhibit	Data Cache Inhibit.

File	Function	Description
	Sys_Dcache_Restore	Data Cache Restore to previously inhibited.
vect000.s vect0x0.s	SysSetVector	Install a vector in the processors vector table.
	NormalWrapper	General-purpose interrupt handler wrapper.
	NiWrapper	NETWORK INTERFACE interrupt handler wrapper.
	MbrkWrapper	This wrapper calls the appropriate handler for the interrupt and then transfers control to the “Manual Break” entry of pROBE+.
	GET_VBR	Moves the value of the current VBR into A0.
misc.s	SafeLongRead	Returns a long value at address if address is accessible, returns 0 if not.
	splx	Change the processor’s interrupt level.
buserr.c	SysHandlerInit	Initialize the bus error handler array.
	SysAddHandler	Add a bus error exception handler to the end of the handler table.
	SysRemoveHandler	Remove a bus error exception handler from the handler table.
ki_call.s		Assembly level code for the kernel interface.
nvram.c	StorageRead	Simple Read from nonvolatile memory.
	StorageWrite	Simple Write to nonvolatile memory.

The following table is a listing of the files in the **bsps/devices/lan** directory.

File	Function	Description
lan.c		LAN driver template file.
	NiLan	LAN driver entry point to perform all LAN functions.
	LanStop	Disable the LAN chip.
AM7990.c		LAN driver for AMD Local Area Network Controller for Ethernet IC-LANCE)
AM7990.h		Chip-specific #defines and structures for the AM7990.
i82596.c		LAN driver for Intel's 32-bit LAN coprocessor.
i82596.h		Chip-specific #defines and structures for the 82596.
lan360.c		LAN driver for Motorola 68360 Ethernet controller.
lan360.h		Chip-specific #defines and structures for the 68360 Ethernet.
lan_mib.c	ni_ioctl	IO-control calls that set or report LAN functionality.

The following table is a list of the files found in the **bsps/devices/serial** directory.

File	Function	Description
disi.c		Serial driver template file.
	SerialInit	Initialize information for all configured channels.
	SerialOpen	Open a serial channel.

File	Function	Description
	SerialClose	Close a serial channel.
	SerialIoct	Perform a control command.
	SerialSend	Send a message block.
ser360.c		Chip-specific serial portion of the serial driver for the Motorola 68360 Serial Communication controller.
ser360.h		Chip-specific #defines and structures for the 68360 SCC.
cd2400.c		Chip-specific serial portion of the serial driver for the Cirrus Logic CD2400 Four-Channel, Multi-Protocol Communications controller.
cd2400.h		Chip-specific #defines and structures for the CD2400.
m68681.c		Chip-specific serial portion of the serial driver for the Motorola MC68681 Dual Universal Asynchronous Receiver/Transmitter (DUART).
m68681.h		Chip-specific #defines and structures for the MC68681.
z85230.c		Chip-specific serial portion of the serial driver for the Zilog Z85230 ESCC Enhanced Serial Communication controller.
z85230.h		Chip-specific #defines and structures for the Z85230.
scc302.c		Chip-specific serial portion of the serial driver for the Motorola MC68302.
scc302.h		Chip-specific #defines and structures for the Motorola MC68302.
ser332.c		Chip-specific serial portion of the serial driver for the Motorola MC68332.

File	Function	Description
ser332.h		Chip-specific #defines and structures for the Motorola MC68332.
ser340.c		Chip-specific serial portion of the serial driver for the Motorola MC68340.
ser340.h		Chip-specific #defines and structures for the Motorola MC68340.

The following table lists the files in the **bsps/devices/scsi** directory

File	Function	Description
scsichip.c		SCSI driver template file.
	chipexec	Perform a SCSI command.
	chipinit	Initialize the SCSI host adapter.
	dma_init	Initialize the dma for the SCSI device.
	scsi_stop_commands	Stop commands from going out on the SCSI bus.
	scsi_start_commands	Restart commands going out on the SCSI bus.
WD33C93.c		Chip-specific portion of the SCSI driver for the Western Digital WD33C93 SCSI-bus interface controller.
WD33C93.h		Chip-specific #defines and structures for the WD33C93.
iscrp710.c		Script file needed for the NCR 53C710 SCSI I/O processor.
iscrp720.c		Script file needed for the NCR 53C720 SCSI I/O processor.
iscrp8xx.c		Script file needed for the NCR 53C810 and 53C820 SCSI I/O processors.
ncr53cxx.c		Chip-specific portion of the SCSI driver for the NCR 53C family of SCSI I/O controllers.

File	Function	Description
ncr53cxx.h		Chip-specific #defines and structures common to the NCR 53C family.
ncr_710.h		Chip-specific #defines and structures for the NCR 53C710.
ncr_720.h		Chip-specific #defines and structures for the NCR 53C720.
ncr_8xx.h		Chip-specific #defines and structures for the NCR 53C810 and 53C820.

9.2.2 Detailed Function Description

This section describes the functions and files that are in the **bsps/devices** directory.

NOTE: It is very important to set the BSP_LITTLE_ENDIAN define statement located in the **bsps/*xxx*/bsp.h file** (where *xxx* is the name of your BSP directory). This define statement should be set to YES if the hardware you are using is in Little-Endian byte order (some Intel Processors) or to NO if the RAM processor is a Big-Endian processor (Motorola processor). This define statement is used in many of the drivers and control files explained in the following paragraphs.

9.2.2.1 rules.mk

The **rules.mk** file contains the Make rules for low-level BSP drivers and common functions. It should be included in the **makefile** for a BSP.

9.2.2.2 68k/bhand000.s bhand030 bhand032 bhand040 bhand060

A file for each member of the 68K family handles a bus error interrupt. These files are used in conjunction with pSOS+m and the kernel interface to shared memory. The function called **SysBusError** saves the registers and information exception frame address on the stack. Then it loops through the **SysHandlers** array of function pointers and does a JSR to each one it finds. After it has gone through all the pointers in **SysHandlers**, it sets the pROBE+ BAERR entry to the return address and does an RTS.

This mechanism is used to recover from a node failure when using pSOS+m. See also the sections that describe the **bsps/devices/common/buserr.c** file, the **drivers/ki_smem.c** file, and the **drivers/ki_calls.s** file.

9.2.2.3 ki_call.s

The **ki_call.s** file contains assembly language functions that are used to support the kernel interface of pSOS+m. The following functions are used in the interface:

- **ki_call** is the entry point for the KI driver from pSOS+m. A pointer to this function is stored in the pSOS+m multiprocessor configuration table in the **mc_kicode** entry of that table. (The **mc_kicode** entry is set in the **configs/std/psoscfg.c** configuration file). The **ki_call** and the KI interface are described in the "Interfaces and Drivers" section of the *pSOSystem Programmer's Reference*. The **ki_call** function is really only a wrapper that saves the register state of the system so the 'C' function, 'ki' located in **ki_smem.c**, can be called.
- **KI_BerrorHndlr** is the kernel interface's bus error handler installed the bus handler array by the function **ki_init** in the file **drivers/ki_smem.c**. This function ensures that buffers of data are not lost on a bus error.
- **hw_tas** is used if the CPU supports the hardware test and set instruction, **TAS,** If that is the case, the functions in **ki_smem.c** use this function to lock accesses on a give memory location.

9.2.2.4 68k/cpu000.s cpu0x0.s cpu030.s cpu040.s cpu060.s

A designated file for each member of the 68K family has on-chip cache. These files have the following functions:

- **Sys_Cache_Init** initializes the CPU's caches for all of memory. With the exception of the 68060, the caches are set to operate in copy-back mode. The 68060 is set to operate in write-through mode. The Data Cache will operate in inhibited Serialized mode for all non-RAM address. This allows memory mapped I/O to function properly. The syntax is:

void Sys_Cache_Init(void);

- **Sys_Dcache_Inhibit** sets the Data Cache to, non-cached serial access mode, writes back all dirty cache lines, marks the entire Data Cache invalid, and then returns to the caller with the previous state of the Data Cache. When this call is used, the caller should save the return value (previous state) to be used in the **Sys_Dcache_Restore** call that will restore the state of the Data Cache. The syntax is:

  unsigned long Sys_Dcache_Inhibit (void);

- **Sys_Dcache_Restore** restores the Data Cache to the value in the D0 register. This value is usually the result of a previous call to **Sys_Dcache_Inhibit**. The syntax is:

  void Sys_Dcache_Restore (unsigned long);

- **GET_VBR** places the value of the Vector Base Register (VBR) in address register A0 and returns. This function is used by **bsps/devices/68k/misc.s** to make it CPU independent.

9.2.2.5 68k/vect000.s and 68k/vect0x0.s

The **vect000.s** and **vect0x0.s** files contain the vector setting and interrupt wrapper code, respectively. The **vect000.s** file should be compiled into those BSPs that have the 68000 processor member of the 68K family (such is the case for the **e302** BSP). The **vect0x0.s** file should be compiled into those BSPs that have processors other than the 68000 member of the 68K family. The following is an explanation of the functions in these files.

- **SysSetVector** sets a interrupt vector to an interrupt service function using a interrupt wrapper. The interrupt wrapper will make it possible to write the interrupt function in C. The syntax for **SysSetVector** is:

  void SysSetVector (int vector, void (*handler)(), WRAPPER_TYPE wt);

  vector is the vector number to be set.

  (***handler**)() is the address of the interrupt function to be called when the interrupt is received.

  wt is the wrapper type to be used. Wrapper type can be one

of the following:

NO_WRAPPER means no C interface will be used. This can be used to set assembly language interrupt functions.

NORMAL_WRAPPER means normal C compatible wrapper to be used for interrupt functions written in C.

NI_WRAPPER is a special wrapper that should be used for interrupt functions in C that are used with the pNA+ network interface.

MBRK_WRAPPER is used to support the manual break interrupt function. (Manual break is caused by a push of the abort button.)

- **NormalWrapper** saves scratch registers and accesses the actual interrupt through the indirect handler table. Once the function returns the scratch registers are restored and control is passed to pSOS+ through the use of a TRAP instruction to the pSOS+ I_RETURN code. This allows pSOS+ to check for dispatches because of run status that may have changed due to the interrupt.
- **NiWrapper** is an interrupt wrapper that works the same way as the **NormalWrapper** with one exception it checks a return value from the interrupt. This return value should indicate if pSOS+ is up and running. The scratch registers are restored. If the return code was nonzero, control is passed to pSOS+ through the use of a TRAP instruction to the pSOS+ I_RETURN code just like the **NormalWrapper**. If the return code was 0, an RTE instruction is done to return from the interrupt. This wrapper is needed for pNA+ because pNA+ will sometimes be running before pSOS+ has started (for example, at boot time).
- **MbrkWrapper** is an interrupt wrapper that saves the scratch registers and then calls the interrupt function for manual break. Once that function returns, the pROBE+ manual break entry point is put on the stack and the scratch registers are restored. Then when the RTS instruction is executed, control passes to the pROBE+ manual break function.

9.2.2.6 68k/misc.s

The **misc.s** file contains several functions that are useful in board initialization and device drivers.

- **MemAccessible** takes as an argument a memory location. It returns a 1 if 8 bits at that location are accessible and a 0 if not. It does this by setting up a new bus error interrupt function. Then it tries to read the memory location. If the bus error function gets called, the return value is set to zero. If not, it is set to a 1. The syntax is:

  unsigned long MemAccessible (volatile void *address);

- **SafeLongRead** is the same as **MemAccessible**; however, instead of testing the memory location for a byte access, it tests the location for a long word (4-byte) access. If the address can be read the value stored at the address is returned. If the address cannot be read, a zero is returned. The syntax is:

  unsigned long SafeLongRead (volatile void *address);

- **MemMirrorTest** tests to see if one memory location is a mirror of another. The syntax is:

  unsigned long MemMirrorTest (volatile void *Lower,
  volatile void *Upper);

  The return value is one of the following:

  1 = Addresses point to different physical RAM locations.
  0 = Upper address is just a mirror of the lower address.
  -1 = Lower address is not accessible.
  -2 = Upper address is not accessible.

- **splx** changes the current interrupt mask of the processor. This function takes one argument, the new interrupt mask level, which should be a value from 0 to 7. The function returns the old interrupt make level for use by the caller to restore the mask when necessary. The syntax is:

  unsigned long splx (unsigned long newmask);

  NOTE: Raising the interrupt mask to anything other than seven or the task's interrupt mask level is strongly discouraged, because it can cause the pSOS+ scheduler to function improperly. The task's interrupt level is set by the **t_start** and **t_mode** pSOS+ calls. These calls may set a

task's interrupt level to a value other than zero so changing the interrupt level to a level below the task's interrupt level can cause the task to be interrupted when it should not be interrupted.

9.2.2.7 68k/rules.mk

The **rules.mk** file contains all the make rules needed to create object files of the assembly files in the **68k** directory. The **rules.mk** file is included in the **bsps/devices/68k** directory.

9.2.2.8 common/buserr.c

The **buserr.c** file contains functions that manage the way bus errors are handled. These functions allow additional bus error handlers to be called when a bus error happens. These functions are needed for the soft fail feature. This allows the detection of a node failure in a multinode system that uses pSOS+m. The following functions manage the bus error interrupt:

- **SysHandlerInit** initializes the **SysHandlers** array of bus handlers. **SysHandlers** is an array of functions that will be used when an interrupt happens to call the interrupt functions using the function address stored in the array. **SysHandlerInit** causes all function pointers to be initialized to a dummy function that just does a return. This function must be called during system initialization (normally in the **InitBoard** function located in **bsps/template/board.c**). The syntax is:

  void SysHandlerInit(void);

- **SysAddHandler** adds a bus error exception handler to the first free slot it finds in the table. **SysAddHandler** takes two arguments one is the handler type which must be HNDLR_TYPE_BUS_ERROR and the second is the address of the bus error function to be installed. **SysAddHandler** returns:

  -1 if handler type was invalid
  -2 if there was no space left in the handler table

  The syntax is:

  int SysAddHandler (unsigned long HandlerType,
  void (*HandlerAddress)())

- **SysRemoveHandler** removes a bus error handler from the **SysHandlers** array and replaces it with the dummy handler. **SysRemoveHandler** takes two arguments one is the handler type which must be HNDLR_TYPE_BUS_ERROR and the second is the address of the bus error function to be removed. **SysRemoveHandler** returns:

  1 upon success
  0 if handler index was invalid

  The syntax is:

  int SysRemoveHandler (unsigned long HandlerType, unsigned long HandlerIndex)

9.2.2.9 common/nvram.c

The **nvram.c** file contains simple functions that can be used to read nonvolatile memory locations on many target systems. (All of the target systems supported by pSOSystem use this.)

The macros that can be set in the **board.h** file of the BSP to control these functions are as follows:

- **BD_NVBASE** which must be set to the starting address of the nonvolatile RAM area.

- **BD_NVSTEP_SIZE** the **StorageRead** and **StorageWrite** functions usually work on devices which have the bytes in adjacent positions in the address space. If working with a device which does not decode the lower address lines, then BD_NVSTEP_SIZE can be defined to override the default. For example, if A0 and A1 are not decoded, the BD_NVSTEP_SIZE would be 4. BD_NVSTEP_SIZE is defined in **board.h** if used.
- **BD_NV_VERIFY** is used in the **StorageWrite** function. If BD_NV_VERIFY is defined then the value just written must be verified before continuing. The verify is done by trying to read the written location until it is the same as what was written. This is done in a loop that will repeat ten times with a delay of 100 milliseconds between tries. If you use this feature, **StorageWrite** should only be done at boot time when pSOS+ is not yet up because BD_NV_VERIFY will cause the call to block and hold the CPU.
- **BD_NV_FILTER** is used in conjunction with BD_NV_VERIFY. BD_NV_FILTER is used to test only selected bits of the value read back with the one written.

The following functions are in the **nvram.c** file:

- **StorageRead** reads from a nonvolatile memory location. The syntax is:

  void StorageRead (unsigned int nbytes, void *Start, void *buff)

  nbytes is the number of bytes to read
  Start is the address to read from
  buff is the address to write to

- **StorageWrite** writes to a nonvolatile memory location. The syntax is:

  void StorageWrite (unsigned int nbytes, void *Start, void *buff)

  nbytes is the number of bytes to write
  Start is the starting address to read from
  buff is the address to write to

9.2.2.10 lan/lan.c

The **lan.c** file is a template file for a LAN driver that can interface with pNA+. This file can be used as a starting point for developing a LAN driver. It contains skeletons of the functions needed for the Network Interface. See the *pSOSystem Programmer's Reference* for more information on the network interface. The following functions are required:

- **NiLan** is the entry point to the LAN driver. See the *pSOSystem Programmer's Reference* "Interfaces and Drivers" section for more information on the network interface and the calling syntax to this function.
- **LanStop** must disable the LAN chip. This is called when pSOSystem is in the processes of switching from ROM-based to RAM-based code and needs to be off-line from the network. The syntax is:

  void LanStop (void);

- **get_LAN_indiscards** is called from **ni_ioctl** function in **devices/lan/lan_mib.c**. It returns the number of input packets that were discarded because of lack of resources. This function can also be coded as a macro instead of a C function. The syntax is:

  long get_LAN_indiscards(void)

- **get_LAN_outdiscards** is called from **ni_ioctl** function in **devices/lan/lan_mib.c**. It returns the number of output packets that were discarded because of lack of resources. This function can also be coded as a macro instead of a C function. The syntax is:

  long get_LAN_outdiscards(void)

- **get_LAN_inerrors** is called from **ni_ioctl** function in **devices/lan/lan_mib.c**. It returns the number of input packets that were discarded because of errors. This function can also be coded as a macro instead of a C function. The syntax is:

  long get_LAN_inerrors(void)

- **get_LAN_outerrors** is called from **ni_ioctl** function in **devices/lan/lan_mib.c**. It returns the number of output packets that were discarded because of errors. This function can also be coded as a macro instead of a C function. The syntax is:

  long get_LAN_outerrors(void)

9.2.2.11 lan/am7990.c

The **am7990.c** file contains the driver for the Advanced Micro Devices Local Area Network Controller of Ethernet (LANCE). This driver supports polling and interrupt driver operation and the pNA+ dependent packet interface. To use this driver you need to supply the following board dependent macros in the **board.h** file of the BSP:

- **BD_ETHER_BASE** is a physical address where the first three bytes of the Ethernet hardware address are stored. This is assumed to be set by the manufacture of the board. The bottom three bytes are set according to BD_ET_ADDR_0, BD_ET_ADDR_1 and BD_ET_ADDR_2.
- **BD_ET_ADDR_0** is the low-order byte of the Ethernet hardware address.
- **BD_ET_ADDR_1** is the middle byte of the Ethernet hardware address.
- **BD_ET_ADDR_2** is the high byte of the Ethernet hardware address.
- **BD_LANCE_CSR** is the address of the lance control and status register.
- **BD_LANCE_RAP** is the address of the lance address port.
- **BD_LANCE_MEM** is the address to the lance data area. NOTE: This must be on a quadword boundary.
- **BD_LANCE_RBUF** is the beginning address of the lance receive buffers. It must point to a valid memory address in a DMA range.
- **BD_LANCE_TBUF** is the beginning address of the lance transmit buffers. It must point to a valid memory address in a DMA range.

- **BD_CACHE_OFF** is the method to turn off the CPU cache. It can be a null macro if there is no CPU cache.
- **BD_CACHE_ON** is the method to turn on the CPU cache. It can be a null macro if there is no CPU cache.
- **BD_SET_ETHER_VEC** is used to set the Ethernet interrupt vector. It can be a null macro if the driver is not being used in interrupt mode.
- **BD_DISABLE_ETHER_INTR** is used to disable Ethernet interrupts on board. It can be a null macro if the driver is not being used in interrupt mode.
- **BD_ENABLE_ETHER_INTR** is used to enable Ethernet interrupts on board. It can be a null macro if the driver is not being used in interrupt mode.

9.2.2.12 lan/am7990.h

The **am7990.h** file contains define statements specific to the AM7990 (LANCE) chip.

9.2.2.13 lan/i82596.c

The **i82596.c** file contains the LAN driver for the Intel 82596 Product Family of high-performance, 32-bit LAN coprocessors. This driver supports poll and interrupt mode, network interface broadcasts, the pNA+ dependent packet interface, and multicast addresses. To use this driver you need to supply the following board-dependent macros in the **board.h** file of the BSP:

- **BD_ENABLE_LAN_SNOOP** turns on snooping by the LAN chip. This macro may be empty if snooping is not supported or the board hardware does not require enabling the CPU to snoop accesses by the i82596.
- **BD_GET_ETHER_ADDR** fills in the Ethernet addressing the Ether_Addr array. (This address is usually read from nonvolatile RAM or a board configuration register.)
- **BD_SET_LAN_VECTOR** sets the Ethernet ISR vector. For example, the macro might be defined as:

  BD_SET_LAN_VECTOR\SysSetVector(V_LAN,((void(*)())LAN_isr), NI_WRAPPER)

you may need to replace V_LAN with some other method of getting the vector number to set but the rest of the macro should remain the same.

- **BD_ENABLE_LAN_INT** enables the Ethernet interrupts. Most boards have some controller chip that you are required to set for the CPU to receive interrupts from the 82596. This macro must contain the code to do that setting.
- **BD_LAN_INT_PENDING** checks for a pending interrupt from the 82596 using what ever method is suitable to the boards hardware. This is usually a bit in an interrupt pending or control register on the board or interrupt controller chip.
- **BD_CLR_LAN_INT** clears an Ethernet interrupt for the 82596 using what ever method is suitable to the board's hardware. This is usually a bit in an interrupt pending or control register on the board or interrupt controller chip.
- **BD_ENABLE_LAN** enables the Ethernet on the board. Most boards do not require any special enable for the 82596 and this macro can be empty. Some boards may have chip select registers that need programming before the 82596 can be used. This macro must be coded for that initialization.
- **BD_CHANNEL_ATTN** issues a channel attention to the SCP. This is usually done by writing a 1 to a channel attention register.
- **BD_WRITE_TO_82596** writes values to the 82596 command port. It has as it only argument the value to be written. This is done by writing a value to the 82596 command register. In some cases this must be done one word (two bytes) at a time with a delay between the writes. This depends on how the 82596 was interfaced to your hardware.
- **BD_SET_SCP** sets up the 82596 System Configuration Pointer for the hardware being used.
- **BD_RESET_82596** resets the i82596 chip. This is usually done by writing to the command register. For example:

  BD_RESET_82596 BD_WRITE_TO_82596((unsigned long) 0xFFFFFFF0)

9.2.2.14 lan/i82596.h

The **i82596.h** file contains the chip-specific structures and defines for the 82596 chip.

9.2.2.15 lan/lan360.c

The **lan360.c** file is the device driver for the Motorola MC68360 Ethernet ports. It has no special board-specific definitions needed. This driver supports polled and interrupt modes, broadcast, multicast, and the use of the pNA+ dependent packet interface.

9.2.2.16 lan/lan360.h

The **lan360.h** file contains chip-specific structures and defines for the Motorola MC68360 Ethernet ports.

9.2.2.17 lan/lan_mib.c

The **lan_mib.c** file contains the common I/O control function for all Ethernet drivers. The I/O control commands allow the driver to interface with SNMP, and add or change multicast address. The **ni_ioctl** function in this file is called when a LAN driver gets a NI_IOCTL command. The calling syntax is:

unsigned long ni_ioctl (unsigned long if_num, long net_type,

unsigned long cmd, long *arg);

if_num is the interface number of the caller. The interface number is passed to the driver in the NI_INIT command.

net_type is the type of network interface being used. This can be set to E_NET for an Ethernet interface or SMEM_NET for a shared memory network interface.

cmd is the command passed in the **nientry** structure in the **NiLan** call (see the *pSOSystem Programmer's Reference* "Interfaces and Drivers" section for more information on the network interface).

arg is the command argument (if any) also passed in the **nientry** structure.

9.2.2.18 serial/disi.c

The **disi.c** file contains a skeleton for a serial driver that conforms to the Device Independent Serial Interface. This interface is fully described in the *pSOSystem Programmer's Reference* in the "Interfaces and Drivers" section.

9.2.2.19 serial/ser360.c

The **ser360.c** file contain the low-level Device Dependent Serial Driver code for the Motorola MC68360 processor. This code has been designed to support two MC68360 processors: one as a Master and one as a slave. The driver supports the SCC ports of the processor. The driver conforms to the DISIplus specification that can support HDLC (used for example with the X.25 Networking Component).

This driver can be extended to support additional MC68360 processors by expanding the **PinMap** array in the **bsps/*xxx*/src/board.h** file (where *xxx* is the BSP directory). The **PinMap** is an array in the file that configures the SCC ports on the MC68360 processor. For each port, the following pins can be configured:

- Receive Data
- Transmit Data
- Transmit Clock
- Receive Clock
- Data Terminal Ready
- Data Set Ready
- Request to Send
- Clear to Send
- Carrier Detect

The **PinMap** allows the driver to be used independent of how these pins are physically wired from the MC68360 to the serial connector. It also determines if the pin is treated as an input or an output pin.

The **PinMap** structure, defined in **ser360.h**, is as follows:

```
struct PinMap   {
        ULONG   available   /* TRUE or FALSE */
        ULONG   direc-      /* IN or OUT */
                tion
        ULONG   port;       /* PORTA, PORTB, or PORTC */
        ULONG   bit;        /* 1  to 15 bits that define the pin */
        ULONG   type;       /* GENERAL I/O or INTERNAL to chip */
        ULONG   inttype;    /* IANY or IHL */

}
```

This structure defines one of the pins mentioned earlier. For example, currently the third SCC on the Master MC68360 for the **e360** BSP is defined in the **PinMap** as follows:

/* SCC3	avail	dir	port	bit	type	interrupt */
/*RXD3*/	TRUE,	IN,	PORTA,	BIT4,	INTERNAL,	0,
/*TXD3*/	TRUE,	IN,	PORTA,	BIT5,	INTERNAL,	0,
/*TCLK3*/	TRUE,	OUT,	PORTA,	BIT12,	NTERNAL,	0,
/*RCLK3/*	TRUE,	IN,	PORTA,	BIT13,	INTERNAL,	0,
/*DTR3*/	TRUE,	OUT,	PORTB,	BIT8,	GENERAL,	0,
/*DSR3*/	TRUE,	IN,	PORTB,	BIT9,	GENERAL,	0,
/*RTS3*/	TRUE,	IN,	PORTC,	BIT2,	INTERNAL,	0,
/*CTS3*/	TRUE,	IN,	PORTC,	BIT8,	GENERAL,	IANY,
/*CD3 */	TRUE,	IN,	PORTC,	BIT9,	GENERAL,	IANY,

The first entry, the element that defines the Receive data pin, is prefaced

by the comment RXD3. The **available** field is set to TRUE meaning this SCC port is available to be used for a serial channel.

The **direction** is set to IN which determines if the bit in the Port A Data Direction Register (PADIR) of the MC68360 will get set or cleared. In this case, IN it will be cleared. **NOTE** This can be somewhat misleading because what this bit has to do with direction is anybody's guess. You really need to check the Port A Pin Assignment table in the MC68360 Motorola manual to fix the setting of this pin. For example, as you can see in the next array entry for the Transmit data pin, the direction is also IN which may not seem logical; however, it is correct according to the table provided in the MC68360 manual.

The **port** is set to Port A so this pin is connected to one of the pins in Port A. This determines the address of the control registers that need to be set.

BIT4 is the **bit** to be set in the control registers. (In this case it will be Port A control register.) This is dependent on what physical pin on the MC68360 chip is wired to the function on the serial connector. Which in this case is the Receive data pin on the serial connector wired to pin 4 of Port A.

INTERNAL defines the **type** of pin. Internal configures the pin as a dedicated on-chip peripheral. If this were set to GENERAL, the pin would be configured as a general-purpose I/O pin.

The **interrupt** element is only used for pin on Port C to enable interrupts for changes in the pin's status. If this was a Port C pin (as are the CTS and CD pins) setting, this to IANY would cause interrupts to be generated for any change in the pin. Setting the interrupt element to 0 causes an interrupt only on a high to low change. For this driver, the CTS and CD pins should have this set to IANY.

9.2.2.20 serial/ser360.h

The **ser360.h** file contains the MC68360 device-specific structures and defines needed for the ser360 driver.

9.2.2.21 serial/cd2400.c

This file contains the low-level driver code for the Cirrus Logic CL-CD2400 Communication Controller. This low-level driver is compliant with the DISI specification. (See the "Drivers and Interfaces" chapter of

the *pSOSystem Programmer's Reference* for more information on the DISI specification.) To use this driver you need to supply the following board-dependent macros in the **board.h** file of the BSP:

- **BD_MAX_SCC** must be set to the total number of serial devices in the system.
- **BD_ALLOW_INTERRUPTS** must enable interrupts to the CD2400.
- **BD_DISALLOW_INTERRUPTS** must disable interrupts to the CD2400.
- **BD_HAS_TX_INTR** produces a TRUE if a transmit interrupt is pending.
- **BD_HAS_RX_INTR** produces a TRUE if a receive interrupt is pending.
- **BD_FORCE_TX_ACK** must force a transmit interrupt acknowledge cycle to the SCC.
- **BD_FORCE_RX_ACK** must force a receive interrupt acknowledge cycle to the SCC.

9.2.2.22 serial/cd2400.h

The **cd2400.h** file contains the CDS2400/2401 device-specific structures and define statements needed for the cd2400 driver.

9.2.2.23 serial/scc302.c

The **scc302.c** file contains the low-level driver code for the Motorola MC68302 Integrated Multiprotocol Processor. This low-level driver is compliant with the DISI specification. (See the "Drivers and Interfaces" chapter of the *pSOSystem Programmer's Reference* for more information on the DISI specification.) To use the driver, you need to supply the following board-dependent macros in the **board.h** file of the BSP:

- **m68302_regs** provides the base address of the m68302 register structure.
- **BD_68302_HZ** provides the CPU clock speed in Hertz. Used to calculate the baud rate divider.
- **BD_M68302_MAX_SCC** provides the maximum number of SCCs to use for this BSP.

NOTE: For the MC68LC302 and MC68PM302, this should be two, because only two SCCs are on these MCUs. If only one SCC were required, this could be set to 1 and both ports A and B could be used as general purpose I/O ports.

9.2.2.23.1 BD_SCC302_PIN_MAP

BD_SCC302_PIN_MAP is the pin map structure used to determine which pins should be used for the modem control lines. Each pin map entry consists of 5 entries: Available, Direction, Port, Bit, and Type. Available must be TRUE if the pin for that function exists, or FALSE if that function does not exist. Direction must be OUT if the pin is an output or IN if it is an input. Port indicates the port that the pin is on and must be either PORTA or PORTB. Bit is the bit number in the port that is used for the function; it can be 1 through 15 inclusive. Type must be INTERNAL if the pin is to be used for the function or GENERAL if the pin should be used as a general purpose I/O pin instead. The values of Direction, Port, Bit, and Type can also be all 0. This is used for a pin that is dedicated to the function. For example, the SCC 1 transmitter pin would be as follows:

```
/*TXD1*/ TRUE,  0,  0,     0,     0, /*TXD1*/ \
```

As a further example, suppose SCC 3 were to be used on a BSP instead of using its pins for general purpose I/O. The SCC 3 receiver pin is shared with the port A bit 8. Its entry would be as follows:

```
/*RXD3*/ TRUE, IN, PORTA, BIT8, INTERNAL, /*RXD3*/ \
BD_DPRAM_BASE
```

This is the base address of the m68302 dual ported RAM, which is determined by the Base Address Register in the m68302.

9.2.2.24 serial/scc302.h

The **scc302.h** file contains the MC68302 device-specific structures and defines needed for the serial driver.

9.2.2.25 serial/ser332.c

The **ser332.c** file contains the low-level driver code for the Motorola MC68332 microcontroller unit (MCU). This low-level driver is compliant with the DISI specification. (See the “Drivers and Interfaces” chapter of the *pSOSystem Programmer's Reference* for more information on the DISI

specification.) To use the driver, you need to supply the following board-dependent macros in the **board.h** file of the BSP:

- **BD_REGS_BASE** is the base of the m68332 register structure to use for the driver. For example:

  ```
  #define BD_REGS_BASE   0xF00000
  ```

- **BD_M68332_HZ** is the CPU clock speed in Hertz. This value is used to calculate the baud rate of a generator constant.
- **BD_SER_IRQ_LEVEL** is the interrupt level for the QSM. This is shared by the SPI and SCI.

The following macros may rely on a chan_control structure pointer named chan that points to the channel to be affected. These macros are optional and automatically default to null if they are not defined in the **board.h** file:

- **BD_SER332_ENABLE_HWFC** enables the hardware flow control for the channel.
- **BD_SER332_DISABLE_HWFC** disables the hardware flow control for the channel.
- **BD_SER332_HWFC_RX_ON** sets the channel's hardware flow control to allow receives.
- **BD_SER332_HWFC_RX_OFF** sets the channel's hardware flow control to prevent receives.
- **BD_SER332_HWFC_TX_ON** sets the channel's hardware flow control to ask for permission to transmit.
- **BD_SER332_HWFC_TX_OFF** sets the channel's hardware flow control to indicate that no transmissions are pending.
- **BD_SER332_MQRY** queries the modem control lines supported by this driver. This macro must set the value for the control lines supported in an unsigned long called temp.
- **BD_SER332_MGET** returns the current state of the supported modem control lines for the channel. This macro must set the value in an unsigned long called temp.
- **BD_SER332_MPUT** sets the state of the supported modem control lines for the channel. This macro gets the states to set from an unsigned long called temp.

9.2.2.26 serial/ser332.h

The **ser332.h** file contains the 68332 device-specific structures and defines needed for the 68332 driver.

9.2.2.27 serial/68681.c

This file contains the low-level driver code for the Motorola MC68681 Dual Asychronous Receiver/Transmitter (DUART). This low-level driver is compliant with the DISI specification. (See the "Drivers and Interfaces" chapter of the *pSOSystem Programmer's Reference* for more information on the DISI specification.) To use the driver you need to supply the following board-dependent macros in the **board.h** file of the BSP:

- **BD_M68681_BASE** is the base of the M68681 structure to use for the driver. For example:

  ```
  #define BD_M68681_BASE ((volatile M68681 * const)0x620001)
  ```

- **BD_M68681_PAD_SIZE** is the number of bytes between the m68681 registers (it must be 1 or 3). Typically, an m68681 will be connected to the low order byte of the CPU's data bus and the bottom address line or lines are not used to access the registers of the m68681. For a CPU with a 16-bit bus, this put the registers every other byte. For a CPU with a 32-bit data bus, the registers are every fourth byte. So a 1 is used for a 16-bit bus, a 3 for a 32-bit bus.

- **BD_M68681_BRG_SET** is the baud rate set to use for the m68681; it must be either BRG_SET1 or BRG_SET2. This chooses the set of baud rates that are used by both ports of the m68681.

- **BD_ALLOW_M68681_INTERRUPTS** is used to set up any board specific registers to allow the m68681 to interrupt the CPU. For example, on an m68302 with the m68681 interrupt line connected to pin 11 of port B. For example:

  ```
  #define BD_ALLOW_M68681_INTERRUPTS m68302_regs->imr |=
  IMR_PB11
  ```

- BD_CLEAR_M68681_INTERRUPT is used to let a board-specific interrupt controller know that the interrupt has been serviced. For example, on an m68302 with the m68681 interrupt line connected to pin 11 of port B.

```
#define BD_CLEAR_M68681_INTERRUPT m68302_regs->isr =
ISR_PB11
```

The following macros use a chan_control structure pointer named chan, which points to a selected channel.

- **BD_M68681_ENABLE_HWFC** enables the hardware flow control for the channel.
- **BD_M68681_DISABLE_HWFC** disables the hardware flow control for the channel.
- **BD_M68681_HWFC_RX_ON** sets the channel's hardware flow control to allow receives.
- **BD_M68681_HWFC_RX_OFF** sets the channel's hardware flow control to prevent receives.
- **BD_M68681_HWFC_TX_ON** sets the channel's hardware flow control to ask for permission to transmit.
- **BD_M68681_HWFC_TX_OFF** sets the channel's hardware flow control to indicate that no transmissions are pending.
- **BD_M68681_MQRY** queries the modem control lines supported by this driver. This macro must set the value for the control lines supported in an unsigned long called temp.
- **BD_M68681_MGET** returns the current state of the supported modem control lines for the channel. This macro must set the value in an unsigned long called temp.
- **BD_M68681_MPUT** sets the state of the supported modem control lines for the channel. This macro gets the states to set from and unsigned long called temp.

9.2.2.28 serial/68681.h

The **68681.h** file contains the 68681 device-specific structures and define statements needed for the 68681 driver. This file contains the 68681 device-specific structures and defines needed for the 68681 driver.

9.2.2.29 serial/ser340.c

The **ser340.c** file contains the low-level driver code for the Motorola MC68340 Integrated Processor Unit. This low-level driver is compliant

with the DISI specification. (See the “Drivers and Interfaces” chapter of the *pSOSystem Programmer's Reference* for more information on the DISI specification.) To use the driver you need to supply the following board dependent macros in the **board.h** file of the BSP:

- **BD_SER340_BASE** is the base of the SER340 structure to use for the driver. For example:

  ```
  #define BD_SER340_BASE \  ((volatile SER340 *
  const)(BD_M68340_REG_BASE + 0x700))
  ```

- **BD_SER340_MCR** is the value to set in the module configuration register. Must have the stop bit clear to activate the serial module.
- **BD_SER340_IL** is the level at which serial interrupts are to be generated.
- **BD_SER340_BRG_SET** is the baud rate set to use for the m68340. It must be either BRG_SET1 or BRG_SET2. This chooses the set of baud rates that are used by both ports of the m68340.
- **BD_ALLOW_SER340_INTERRUPTS** is used to set up any board specific registers to allow the serial ports to interrupt the CPU.
- **BD_CLEAR_SER340_INTERRUPT** is used to let a board specific interrupt controller know that the interrupt has been serviced.

The following macros use a chan_control structure pointer named chan, which points to selected channel:

- **BD_SER340_ENABLE_HWFC** enables the hardware flow control for the channel.
- **BD_SER340_DISABLE_HWFC** disables the hardware flow control for the channel.
- **BD_SER340_HWFC_RX_ON** sets the channel’s hardware flow control to allow receives.
- **BD_SER340_HWFC_RX_OFF** sets the channel’s hardware flow control to prevent receives.
- **BD_SER340_HWFC_TX_ON** sets the channel’s hardware flow control to ask for permission to transmit.

- **BD_SER340_HWFC_TX_OFF** sets the channel's hardware flow control to indicate that no transmissions are pending.
- **BD_SER340_MQRY** queries the modem control lines supported by this driver. This macro must set the value for the control lines supported unsigned long called temp.
- **BD_SER340_MGET** returns the current state of the supported modem control lines for the channel. This macro must set the value in an unsigned long called temp.
- **BD_SER340_MPUT** sets the state of the supported modem control lines for the channel. This macro gets the states to set from an unsigned long called temp.

9.2.2.30 serial/ser340.h

The **ser340.h** file contains the 68340 device-specific structures and define statements needed for the 68681 driver.

9.2.2.31 src/scsichip.c

The **scsichip.c** file contains a skeleton for a low-level SCSI driver. This interface is fully described in the *pSOSystem Programmer's Reference* in the "Interfaces and Drivers" section.

9.2.2.32 scsi/wd33c93.c

The **wd33c93.c** file contains the low-level SCSI driver code for the Western Digital WD33C93 SCSI Bus Interface Controller. This driver is compliant with the SCSI device driver specification in the *pSOSystem Programmer's Reference*, "Drivers and Interfaces" section. To use this driver, you need to supply the following board-dependent macros in the **board.h** file of the BSP:

- **BD_Scsi_Address** is the base address of WD33C93 SCSI chip.
- **BD_Scsi_Indirect_Reg** must be set to the address of the WD33C93 Indirect Register.
- **BD_Scsi_Aux_Status** must be set to the address of the WD33C93 Auxiliary Status Register.
- **BYTE0, BYTE1, BYTE2, and BYTE3** are byte manipulation macros that need to be defined for a particular processor. For

example, for Motorola Big-Endian processors, they are defined as follows:

```
#define BYTE0(x) (x & 0xFF)
#define BYTE1(x) ((x >> 8) & 0xFF)
#define BYTE2(x) ((x >> 16) & 0xFF)
#define BYTE3(x) ((x >> 24) & 0xFF)
```

For Little-Endian processors (for example, Intel's 960 Little-Endian processors), they are defined as follows:

```
#define BYTE0(x) ((x >> 24) & 0xFF)
#define BYTE1(x) ((x >> 16) & 0xFF)
#define BYTE2(x) ((x >> 8) & 0xFF)
#define BYTE3(x) (x & 0xFF)
```

- **BD_DISABLE_SCSI_INTR** must disable interrupts from the WD33C93.
- **BD_ENABLE_SCSI_INTR** must enable interrupts from the WD33C93.
- **BD_SET_SCSI_VEC** must set the SCSI interrupt vector.
- **BD_DISABLE_SCSI_DMA_CONTROL** must disable the DMA controller for the WD33C93.
- **BD_ENABLE_SCSI_DMA_INTR** must enable the DMA controller for the WD33C93.
- **BD_RESET_SCSI_INTR** must reset and disable interrupts for the WD33C93.
- **BD_CLEAR_SCSI_DMA_INTR** must clear any interrupt pending for SCSI dma. NOTE: this macro should only clear pending interrupts not reset or disable interrupts.
- **BD_SCSI_DMA_SETUP** must set up and enable a DMA transfer. It takes the following arguments:

  ADDR is the address of the data.
  COUNT is the number of bytes to transfer.
  DIRECTION is the direction of the transfer which can be BD_TO_SCSI or BD_FROM_SCSI where BD_TO_SCSI and BD_FROM_SCSI are defined as follows:

  ```
  #define BD_TO_SCSI 1
  #define BD_FROM_SCSI 2
  ```

- **BD_GET_SCSI_DMA_STATUS** needs to put DMA status into message. The message array must be set up as follows:

  message[0] must contain the starting address of the DMA
  message[1] must contain the numbers of bytes transferred
  message[2] must contain the DMA control status byte
  message[3] must contain the DMA status byte

  where message is defined in the **wd33c93.c** file (unsigned long message[4];)

- **BD_SCSI_DMA_STATUS** must AND into the third element of the message array to the DMA status.

  message[2] & DMA Status

- **BD_WD_CHIP** sets a chip structure to the SCSI chip address. For example:

  volatile struct wd_dev *wd_chip = (struct wd_dev *) 0x2400000

  where 0x2400000 is the base address of the WD33C93 chip.

9.2.2.33 scsi/wd33C93.h

The **wd33c93.h** file contains the WD33C93 device specific structures and define statements needed for the WD33C93 driver.

9.2.2.34 scsi/iscrp710.c, scsi/iscrp720.c, and scsi/iscrp8xx.c

The **scrp710.c, scsi/iscrp720.c,** and **scsi/iscrp8xx.c** files are compiled script files for the **NCR53C710, NCR53C720,** and the **NCR53C810/825** SCSI controller chips. This code has been compiled using the NCR NASM assembler into arrays of data that will be compiled by your C compiler and loaded into the image file. This code contains instructions used by the NCR chip.

9.2.2.35 scsi/ncr53cxx.c

The **ncr53cxx.c** file contains the low-level SCSI driver code for the NCR53C series of SCSI controllers. This file has been tested for the 53C710, 53C720, 53C810, and 53C825 SCSI controller chips. This driver is compliant with the SCSI device Driver Specification in the *pSOSystem Programmer's Reference* "Drivers and Interfaces" section. To

use this driver you need to supply the following board-dependent macros in the **board.h** file of the BSP:

- **BD_SCSI_SET_INTR** is a hook to set up the interrupt controller for use by the NCR chip.
- **BD_SCSI_POLLED** disables interrupts from the NCR chip to the CPU so the driver can be used in a polled mode.
- **BD_SCSI_ILEV_ENABLE** is a hook to set the interrupt and enable interrupts for the NRC chip in the interrupt controller.
- **BD_SCSI_SNOOP_CONTROL** is used to set the snoop control bits in the CTEST7 register of the NCR chip. A value of 1 sets SC0. A value of 2 sets SC1. A value of 3 sets both SC0 and SC1. See the NCR data manual for more information on the snoop control bits.
- **BD_SCSI_CLOCK** is set to the speed of the clock connected to the NCR chip.
- **BD_SIOP_ID** is set to the Host ID of the system. This ID depends on if the NCR chip is a 710 or an other version of the chip. In the 710 uses the HOST_ID is represented as a bit position where the other versions of the chip it as represented as a byte or a short number of the actual ID number.
- **BD_SCSI_DMA_MODE** is the value the DMODE register in the NCR chip will be set to. This setting will be dependent on how your hardware is wired to work with the NCR chip.
- **BD_SCSI_SET_NCR_BASE** is set to the base address of the NCR chip.
- **BD_SCSI_SET_VECTOR** must install the interrupt vector for the NCR chip. For example:

  ```
  #define BD_SCSI_SET_VECTOR
          SysSetVector (V_SCSI, scsi_isr,
  NORMAL_WRAPPER);
  ```

- **BD_SCSI_CPU_ACK_MODE** is used in for the NCR53C800 series only. It is the setting of the EA bits of DCNL register. This bit controls the behavior of the pin STERM related to the CPU bus activities.

- **BD_SCSI_BUS_TRANS_TYPE** determines how the TT1 pin will be set for synchronous SCSI transfers. The value of this bit is inverted on the TT1 pin. Some boards use this pin to determine what address view the NCR chip has. This is referred to a Transfer Type. BD_SCSI_BUS_TRANS_TYPE can be set to TT1_B or a zero. For example:

  #define BD_SCSI_BUS_TRANS_TYPE TT1_B

9.2.2.36 scsi/ncr53cxx.h

The **ncr53cxx.h** file contains the general structures and define statements that are common to all versions of the NCR 531 family of controller chip.

9.2.2.37 scsi/ncr_710.h, scsi/ncr_720.h and scsi/ncr_8xx.h

These files contain the NCR chip-specific structures and define statements for the driver. You would use the file that is appropriate for the chip you are using.

9.3 Configuration Files

Configuration files for BSPs are located in the **configs/std** directory. These files are used to configure the system for the different components.

9.3.1 Configuration File List

The following table is a list of the files and their functions in the **configs/std** directory.

File	Function	Description
begin.s	START	Sets initial stack pointer and status word. This is the entry point for pSOSystem.
	SysInitVars	Zeros out memory used for static variables.
begina.s		This file is used when the OS and application are built separately.

File	Function	Description
beginapi.s		Entry point for position-independent applications when linked separately.
config.mk		This file is included by an application's Makefile. It contains the rules to build the system's configuration files.
configpi.mk		This file is included by an application's Makefile. It contains the rules to build the system's configuration files. It is used to generate an application containing position-independent code (**app.x** or **app.hex**).
configre.mk		This file is included by an application's Makefile. It contains the rules to build the system's configuration files. It is used to generate an application containing relocatable code (only for **app.x**).
configxx.mk		This file is included by an application's Makefile. It contains the rules to build the system's configuration files. This fill uses the C++ compiler.
dialog.c		System start-up dialog.
	SysVarsPrint	Print out the current values of SysVars.
	SysVarsChange	Interactively lead the user through the system variables, allowing the option of changing each that is relevant.
	Dialog	Conduct the startup configuration dialog.
end.s		This file contains assembly code that must be the last code point in the image file. It is used to mark the end of the code section.
gsblkcfg.c		General serial mblk configuration.
	GSblkSetup	Set up Mblk buffer pools.
ldcfg.c		Loader configuration file.
philecfg.c		pHILE+ configuration file.

File	Function	Description
	PhileSetup	Set up the pHILE+ component.
pmontcfg.c		pMONT configuration file.
	PmontSetup	Set up the pMONT component.
	BspPmontCallout	pMONT entry and exit callout.
pnacfg.c		pNA+ configuration file.
	PnaSetup	Setup pNA+ component.
	InstallNi	Installs an NI driver.
pollio.c		Routines for serial I/O before pREPC+ can be run.
	Print	Format output and send it to the console (subset of printf).
	Prompt	Display the current value of a parameter, and allow user to change it. This function is used by the Boot-up dialog.
prepccfg.c		pREPC+ configuration file.
	MakeDeviceString	Format a device number into ASCII device name string.
	PrepcSetup	Set up the pREPC+ component.
probecfg.c		pROBE+ configuration file.
	ProbeSetup	Set up the pROBE+ component.
	ProbeEntryCallout	pROBE+ entry callout.
	ProbeExitCallout	pROBE+ exit callout.
prpccfg.c		pRPC+ configuration file.
	PrpcSetup	Set up the pRPC+ component.
psecfg.c		pSE+ configuration file.
	PseSetup	Set up the pSE+ component.

File	Function	Description
psoscfg.c		pSOS+ and pSOS+m configuration file.
	PsosSetup	Set up the pSOS+ or pSOS+m component.
	InstallDriver	Adds a device driver to PsosIO table.
sysinit.c		Main system configuration code file.
	SysInit	Initialize system variables, conduct startup dialog if configured, and set up component configuration tables.

9.3.2 Detailed Function Description

9.3.2.1 begin.s

The **begin.s** file contains the start-up code that sets the status register in the 68K CPU and initializes the Stack Pointer (SP register) in the CPU.

This file also contains a function called **SysInitVars**. This function will set to zero all memory used by static variables or zerovars. This function is usually the first function called by the **InitBoard** function in the BSP. It should be called as early as possible because if the stack is located in zero vars it will be cleared. This means that among other things you will lose the return address to whatever function you came from after you execute this function.

9.3.2.2 begina.s

The **begina.s** file contains startup code that will do a simple jump to the root function. It is used when the rule **app.x** or **app.hex** is given as a make object for the making of an application. This causes the application only to be included in the image file. This allows the application to be loaded separate from the OS.

9.3.2.3 config.mk

The **config.mk** file is included by the application's Makefile. It contains the rules to make different kinds of executable images that can be downloaded to a target system. **config.mk** also contains the rules to make the configuration files for the different components of pSOSystem.

The following targets can be made by including **config.mk**:

- **ram.x** produces a complete pSOSystem image file called **ram.x**. This file is an IEEE format file that can be downloaded via XRAY.
- **ram.hex** produces a complete pSOSystem image file called **ram.hex**. This file is an S-record format file that can be downloaded to a Target System via TFTP or the pROBE+ DL command.
- **ramrc.x** produces a pSOSystem that uses operating system components that are contained in ROM. The image file is called **ramrc.x** and is a IEEE format file that can be downloaded via XRAY.
- **ramrc.hex** produces a pSOSystem that uses operating system components that are contained in ROM. The image file is called **ramrc**.**hex** and is an S-record format file that can be downloaded to a target system via TFTP or the pROBE+ DL command.
- **rom.hex** produces a pSOSystem that will be linked at ROM address for the target system. The file produced is an S-record image file that can be downloaded to a ROM programmer and then programmed into a ROM.
- **os.x** produces an image file called **os.x**. This image file will contain the OS only and not the application code. This file can be downloaded separately from the application image. This image file is called **os.x** and is a IEEE format file that can be downloaded via XRAY.
- **os.hex** produces an image file called **os.hex**. This image file will contain the OS only and not the application code. This file can be downloaded separately from the application image. This **os.hex** and is an S-record image file that can be downloaded to a target system via TFTP or the pROBE+ DL command.
- **app.x** produces an image file called **app.x**. This image file will contain the application code only and no OS code. This file can be down-loaded separately from the OS image. The **app.x** file is an IEEE format file that can be downloaded via XRAY.
- **app.hex** produces an image file called **app.hex**. This image file will contain the application code only and no OS code. This file

can be downloaded separately from the OS image. The **app.hex** file is an S-record image file format that can be downloaded to a target system via TFTP or the pROBE+ DL command.

9.3.2.4 configxx.mk

The **configxx.mk** file contains the same features as **config.mk** except it is used with C++ instead of standard C.

9.3.2.5 dialog.c

The **dialog.c** code file contains functions that control a startup dialog. This file is intended to be used as is when making Boot ROMs. It can also be used as an example to produce your own custom start-up dialog. See the "Configuration and Startup" chapter for the complete Boot-up dialog.

The **SysInit** function in **sysinit.c** calls the function **Dialog** in the **dialog.c** file. Dialog will conduct the startup configuration dialog. This file contains conditionally compiled code. This code depends on the presence of the components you choose to configure into the System Image. The choice of components is made using the **sys_conf.h** file in your applications directory. The **sys_conf.h** file is explained in the Chapter 7, "Configuration and Startup."

The functions in the **dialog.c** file use values in the **SysVars** structure. The **SysVars** structure is a global structure of **SD_parms**. It is filled in by the **Sysinit** function in the **sysinit.c** file from a nonvolatile RAM area. The updating of this area is also handled by the **SysInit** function.

9.3.2.6 end.s

The **end.s** file contains one variable called **FreeMemStart**. This variable is used by the **SysInit** function in **sysinit.c** to allocate data areas for components and system services. **end.s** is located in its own code Section. This section is always the last section of code loaded into the image file and will be the last address in the downloaded system that contains code.

9.3.2.7 gsblkcfg.c

The **gsblk.c** file contains the function **GSblkSetup**. This function will allocate space for the General Serial message blocks that are used by the

serial drivers. The number and size of the message blocks allocated is determined by conditional defines in the **sys_conf.h** file. This is covered in the "Configuration and Startup" chapter.

9.3.2.8 philecfg.c

philecfg.c is the configuration file for pHILE+. It contains the function PhileSetup which is called from the function **BuildConfigTables** in the **sysinit.c** file. The configuration parameters are determined by conditional defines in the **sys_conf.h file**. This is covered in Chapter 7, "Configuration and Startup."

9.3.2.9 pmontcfg.c

pmontcfg.c is the configuration file for pMONT. It contains the function **PmontSetup** which is called from the function **BuildConfigTables** in the **sysinit.c** file. The configuration parameters are determined by conditional defines in the **sys_conf.h file**. This is covered in the Chapter 7, "Configuration and Startup."

9.3.2.10 pnacfg.c

pnacfg.c is the configuration file for pNA+. It contains the function **PnaSetup** which is called from the function **BuildConfigTables** in the **sysinit.c** file. The configuration parameters are determined by conditional defines in the **sys_conf.h** file. This is covered in the "Configuration and Startup" chapter.

The **pnacfg.c** file uses function **InstallNi** to install or configure a network interface driver into pSOSystem. The syntax is:

```
void InstallNi (int (*entry)(),
                int ipadd,
                int mtu,
                int hwalen,
                int flags,
                int subnetaddr,
                int dstipaddr)
```

- **int (*entry) ()** is the address of the driver's network interface function.

- **int ipadd** is the IP address of the interface.
- **int mtu** is the maximum transmission length of a packet sent or received using the interface.
- **int hwalen** is the length of the hardware address.
- **int flags** are the interface flags.
- **int subnetaddr** is the subnet address.
- **int dstipaddr** is the destination IP address used in point-to-point connections only.

The **InstallNi** function fills the values in the network interface table. A pointer to the network interface table is placed into the pNA+ configuration table in the **PnaSetup** function (also in the **pnacfg.c** file). See the "Configuration Tables" chapter in the *pSOSystem Programmer's Reference* for more information on the network interface table.

9.3.2.11 pollio.c

The **pollio.c** file contains two functions that are used by pSOSystem during the booting of the system to print output and prompt for input. These are needed because the usual functions to do printing and prompting that are contained in pREPC+ are not initialized at this point in the Boot process. These functions are as follows:

- **Prompt** displays the current value of a parameter, and allows it to be changed. This is used in the startup dialog. The syntax is:

  void Prompt (char *string, PARM_TYPE ptype, void *parm_ptr)

  string is the string to be printed before the prompt.

  ptype is the prompt type. This is used to format the prompt. The prompt type can be one of the following:

  DECIMAL formats a decimal long

  HEX formats a hex number proceeded by 0x.

  FLAG prints a [Y] or a [N] depending on the value of ***parm_ptr**. If ***parm_ptr** is 0 '[N]' will be pointed. If ***parm_ptr** is not 0 then '[Y]' will be printed.

  CHAR prints the value of ***parm_ptr** as a character.

STRING prints the value of ***parm_ptr** as a string of characters

IP prints the value of ***parm_ptr** as an IP address with the form of xxx.xxx.xxx.xxx where *xxx* indicates a decimal number corresponding to the class of IP address.

These values DECIMAL, HEX, FLAG, CHAR, STRING and IP are defined in **include/apdialog.h**.

parm_ptr is a pointer to the input data which is dependent on the **ptype**.

- **Print** is used in place of **printf** during system startup. The syntax is:

  unsigned long Print (char *format,...)

  NOTE: The **Print** function should not be used in place of printf for your application. The **Print** function blocks code execution.

9.3.2.12 prepccfg.c

The **prepccfg.c** file is the configuration file for pREPC+. It contains the function **PrepcSetup** which is called from the function **BuildConfigTables** in the file **sysinit.c**. The configuration parameters are determined by conditional defines in the file **sys_conf.h**. This is covered in the "Configuration and Startup" chapter.

The file also contains the function **MakeDeviceString**. This function formats a device number into an ASCII device name string. For example, it takes the number 0x10000 and converts it to the characters 1.0. The syntax is:

void MakeDeviceString (ULONG DeviceNr, UCHAR *DeviceStringPtr)

Where **DeviceNr** is the number to be converted and **DeviceStringPtr** is the starting address to place the converted string.

9.3.2.13 probecfg.c

The **probecfg.c** file is the configuration file for pROBE+. It contains the function **ProbeSetup** which is called from the function **BuildConfigTables** in the **sysinit.c** file. The configuration parameters are determined by conditional defines in the **sys_conf.h** file. This is

covered in the "Configuration and Startup" chapter. The **Probesetup** function uses the pointer **FreeMemPtr** (see **end.s**) to allocate memory for pROBE+'s data and stack areas.

However, you can set the define statement **RC_DATASTART** in **sys_conf.h** to the value of an address where you want pROBE+'s data and stack to be located instead of using the **FreeMemPtr**. A good reason to do this is when you are short of memory. pROBE+ uses 0x1700 bytes of memory for its data and stack area. By using the **FreeMemPtr**, it effectively takes away memory that could have been used for pSOS+'s region zero. If you are using the ROM supplied by Integrated Systems as Boot ROMs or you have made Boot ROMs using the sample applications **proberom** or **tftp**, an area of memory is set aside for the Boot ROM that once the application has been downloaded and started will no longer be used. You can reuse this area for pROBE+'s data and stack areas. The area lies between the start of RAM plus 0x800 and the start of the code section of the downloaded code. For example, if your RAM starts at location 0 and your code starts at 0x3000, then you can set RC_DATASTART to 0x800. pROBE+'s data and stack areas will then occupy from 0x800 to 0x1F00 which is safely under 0x3000 where your code starts. RC_DATASTART must always be aligned to a four-byte boundary.

In addition to the **ProbeSetup** function, two other functions are of interest in this file: the pROBE+ entry and exit call-out functions. These functions contain calls to other functions based on what things are present in the system determined by conditional defines in the **sysconf.h** file. For example, these functions call the DIPI drivers **ProbeEntry** and **ProbeExit** to set up the serial driver for pROBE+ input and output. If present, the memory management functions of the MMU library will also be called.

9.3.2.14 prpccfg.c

The **prpccfg.c** file is the configuration file for pRPC+. It contains the function **PrpcSetup** which is called from the function **BuildConfigTables** in the **sysinit.c** file. The configuration parameters are determined by conditional defines in the **sys_conf.h** file. This is covered in the "Configuration and Startup" chapter. The **PrpcSetup** function uses the pointer **FreeMemPtr** (see **end.s)** to allocate memory for pRPC+'s data and stack areas.

However, like pROBE+ you can select your own address, one that will not be used by pSOSystem for a starting point for pRPC+'s data area by setting the the **NR_DATA** define statement in the **sys_conf.h** file. See **probecfg.c** above for an explanation of a safe place to set this data area. The size of the data area is also set in **sys_conf.h** by a define statement called **NR_DATASIZE**.

9.3.2.15 psecfg.c

The **psecfg.c** file is the configuration file for pSE+. It contains the function **PseSetup** which is called from the function **BuildConfigTables** in the file **sysinit.c**. The configuration parameters are determined by conditional defines in the **sys_conf.h** file. This is covered in the "Configuration and Startup" chapter. The **PseSetup** function uses the pointer **FreeMemPtr** (see **end.s**) to allocate memory for pSE+'s data areas.

9.3.2.16 psoscfg.c

The **psoscfg.c** file is the configuration file for pSOS+. It contains the function **PsosSetup** which is called from the function **BuildConfigTables** in the **sysinit.c** file. The configuration parameters are determined by conditional defines in the **sys_conf.h** file. This is covered in the "Configuration and Startup" chapter. The **PsosSetup** function uses the pointer **FreeMemPtr** (see **end.s**) to allocate memory for pSOS+'s data areas. The function **PsosSetup MUST** be the last function called that uses the **FreeMemPtr** because **PsosSetup** will allocate all remaining memory to pSOS+'s **region 0**.

The **psoscfg.c** file also contains the function **InstallDriver**. **InstallDriver** is called from the **drv_conf.c** file in your **applications** directory. **InstallDriver** places a driver entry into the pSOS+ I/O switch table. **InstallDriver** has the following syntax:

```
void InstallDriver(
        unsigned short major_number,
        void (*dev_init) (struct ioparms *),
        void (*dev_open) (struct ioparms *),
        void (*dev_close) (struct ioparms *),
        void (*dev_read) (struct ioparms *),
        void (*dev_write) (struct ioparms *),
        void (*dev_ioctl) (struct ioparms *),
        unsigned long rsvd1,
```

unsigned short rsvd2,
unsigned short flags)

As you can see, **InstallDriver** takes the major number of the driver given by major_number, uses it as an index into the I/O switch table and places the remaining arguments into their correct place in the I/O switch table.

The flags argument has special meaning to pSOS+. Currently this flag field is used to set an **AutoInit** bit that pSOS+ checks when it is initializing. If the bit is set, pSOS+ calls the initialization function, if any, for the driver when pSOS+ is initializing. This means you will not have to call the driver initialization function (through the use of **de_init**) in your application for any driver that has this bit set.

NOTE: Any driver that you plan to use this AutoInit feature for must not make any system calls that need a task's context because the driver initialization function will be called before any task has been started.

For example, to add a driver as major device 6 (which has only *init* and *read* calls, for example), you add the following to **SetUpDrivers()** in **sysinit.c**:

InstallDriver(6, DriverInit, NULLF, NULLF, DriverRead, NULLF, NULLF, 0, 0, IO_AUTOINIT);

Where NULLF is a macro in **sysinit.c** defined as a null function pointer and IO_AUTOINIT sets the flags so the driver is be initialized at startup of pSOS+.

9.3.2.17 sysinit.c

Where **HdwInit.c** configures the target system's hardware, the **sysinit.c** file configures the target system's components, drivers, and some miscellaneous software functions.

The **sysinit.c** file is filled with conditional compile options. It draws values for these conditional code options from several sources such as **sys_conf.h** and **bsp.h**. This conditional compiled code ensures that the system image file only contains code for things you have configured into the system and nothing more. You should never need to alter this file. All configuration is done through the values you set for the define statements in the included files.

The last function that is called in the initialization process before pSOS+ is started is **SysInit**. This is the function that configures the system software. **SysInit** performs the following actions:

- A structure of default system values will be used if there are no previous values: the values for these variables initially will come from the **sys_conf.h** file. All of the values that are prefaced with SD_ in the **sys_conf.h** file are used to set this structure.
- If this is a boot of a ROM-based system (executing out of ROM) all initialized variables are copied to their correct place in RAM from where they have been stored in ROM.
- All system values stored in nonvolatile RAM are read into the SysVars structure. If the define statement **SC_SD_PARAMETERS** is set to **STORAGE** in the application's **sys_conf.h** file, these values replace the default values used in step 1. If **SC_SD_PARAMETERS** is set to **SYS_CONF**, the values set in step 1 will not be overwritten.
- If any BSP or application values are stored in nonvolatile RAM, they are read and stored.
- The **FreeMemPtr** pointer is set from the **FreeMemStart** address in **end.s**.
- The serial driver is initialized.
- If configured, the startup dialog is run and variables are set with the results of the dialog.
- Configuration tables for configured components are set up.
- If a floating-point processor is present, the floating-point exception vectors are set.
- The serial driver is initialized again because values may have changed during the running of the dialog.
- If pROBE+ was configured into the system, the **probe_init** function is called and pROBE+ takes over from here.
- If pROBE+ was not configured into the system and pSOS+ or pSOS+m was, the pSOS+ initialization function is called and pSOS+ takes over from here.

- If neither pSOS+ nor pROBE+ was configured into the system, control passes directly to the application.

9.4 drivers Directory

The **drivers** directory contains the common parts of the device drivers in pSOSystem. In some cases the entire driver is in the file in the **drivers** directory. Such is the case for the RAM disk driver. In other cases these common parts are designed to mate to device specific parts located in the **bsp/devices** directory. This is the case for the serial driver located in the file **diti.c** and the SCSI driver located in the **scsi.c** file.

9.4.1 drivers Directory File List

Below is a table of files and functions located in the **drivers** directory.

File	Function	Description
dipi.c		pROBE+'s interface to the low-level DISI serial driver.
diti.c		Device Independent Terminal Interface (DITI) that interfaces a terminal to the low-level DISI serial driver.
drv_cutl.c		Utilities used by various drivers.
	ScratchPadTest	Determines if a bit is set in a task's register.
	ScratchPadSet	Sets a bit in a task's register.
	ScratchPadUnSet	Clears a bit in a task's register.
gsblk.c		General serial message block functions.
	gsblk_initbuffers	Initializes the message block buffers configured by the user. Called by GSblkSetup in **configs/std/gsblkcfg.c**.
	gs_allocb	Allocates a message block.

File	Function	Description
	gs_esballoc	Attaches a message buffer to user buffer.
	gs_freemsg	Frees a message block and associated data blocks.
ki_smem.c		Kernel interface to shared memory used with pSOS+m.
	ki	Entry point for the driver that follows the kernel interface specification.
ni_smem.c		Network interface that enables the use of pNA+ using a shared memory interface.
	NiSmem	Entry point for the driver that follows the network interface specification.
ramdisk.c		RAM disk driver.
	RdskInit	Initialization function for de_init calls for the RAM disk driver.
	RdskWrite	Write function for de_write calls for the RAM disk driver.
	RdskRead	Read function for de_read calls for the RAM disk driver.
rarp.c		Contains routines that implement the RARP protocol.
	RarpEth	Get IP address with RARP for specified Ethernet interface.
	GetRarpServerIP	Return the IP of the last server that answered a RARP request.
rules.mk		Contains the Make rules for the files in this directory.
scsi.c		This is the upper-level common part of the SCSI driver. This is an initiator-only driver that supports SCSI disks and SCSI tape devices.

File	Function	Description
	SdrvInit	Driver initialization function for de_init calls for the SCSI driver.
	SdrvCntrl	Driver I/O control function for de_ioctl calls of a disk or tape device for the SCSI driver.
	SdskRead	Disk read function for de_read calls of a disk device for the SCSI driver.
	SdskWrite	Disk write function for de_write calls of a disk device for the SCSI driver.
	StapeRead	Tape read function for de_read calls of a tape device for the SCSI driver.
	StapeWrite	Tape write function for de_write calls of a tape device for the SCSI driver.
	StapeOpen	Tape open function for de_open calls of a tape device for the SCSI driver.
	StapeClose	Tape close function for de_close calls of a tape device for the SCSI driver.
smem_isr.c	SmemIsr	Shared-memory interrupt service routine passed to SmemIntInit function used in both the KI and NI functions.
tftp_drv.c		Driver for TFTP.
	TftpInit	TFTP driver initialization function for de_init calls.
	TftpOpen	TFTP driver open function for de_open calls.
	TftpClose	TFTP driver close function for de_close calls.
	TftpRead	TFTP driver read function for de_read calls.
	TftpCntl	TFTP I/O control function for de_ioctl calls.

9.4.2 Detailed File Description

9.4.2.1 dipi.c

The device independent pROBE+ interfaces file, **dipi.c**, is the connection between pROBE+ and the DISI serial driver. The **dipi.c** file defines seven functions that correspond to pROBE+'s configured I/O procedures:

pROBE+ Config Table Entry Name	dipi.c Function Name	Description
rc_ioinit	ProbeIOInit	Initializes the console and host channels.
rc_consts	ProbeConsts	Returns console status.
rc_conin	ProbeConin	Gets a character from the console.
rc_conout	ProbeConout	Sends a character to the console.
rc_hstst	ProbeHstst	Returns host channel status.
rc_hstin	ProbeHstin	Gets a character from the host channel.
rc_hstout	ProbeHstout	Sends a character to the host channel.

In addition to the interface functions, the interface module will also contain the functions that will change the mode of the serial driver to polled mode and save the current state of the channel. The pointers to the entry and exit functions will be entered into the pROBE+ configuration table as follows:

pROBE+ Config Table Entry Name	dipi.c Function Name	Description
rc_entry	ProbeEntry	pROBE+ entry function
rc_exit	ProbeExit	pROBE+ exit function

ProbeIOInit sets up and opens the pROBE+ console and pROBE+ host channels. The pROBE+ console and host channels defined in **sys_conf.h** as SC_PROBE_CONSOLE and SC_PROBE_HOST. During the initialization of pSOSystem, a call is made to **ProbeSetup**. In the **ProbeSetup** function, the external variable **ProbeCon** is set to SC_PROBE_CONSOLE and the external variable **ProbeHst** is set to SC_PROBE_HOST. The **ProbeIOInit** function uses the values of **ProbeCon** and **ProbeHst** to open the pROBE+ channels and set up any necessary buffers for the I/O from those channels.

ProbeConsts/**ProbeHststs** functions return an unsigned long that contains the input status of the port. **ProbeConsts**/**ProbeHststs** is in effect, a way for pROBE+ to poll for input. Internally, characters are polled for by using the **DISI** call **SerialIoctl** with the **SIOCPOLL** command. The SIOCPOLL command acts as an interrupt and checks the channel's status. If characters are received for the channel, then the **ProbeDataInd** function is called. If there has been an exception, then the **ProbeExpInd** function is called. The DIPI must be able to receive blocks of characters at a time. Once a block of characters has been received by the DIPI, the **ProbeConsts** function should check the characters in the blocks to determine the status to return instead of using the SIOCPOLL command every time. The SIOCPOLL command may still need to be used if the character being looked for is not in any current receive block. For example, if the typecode is '2', then pROBE+ is looking for a Break character. If there is no Break character in any block received, then the SIOCPOL command will be used to see if there are any more characters that have been received by the Lower Level Device Dependent Code.

ProbeConin/**ProbeHstin** returns a character that has been received from the channel. **ProbeConin**/**ProbeHstin** should be called only after **ProbeConsts**/**ProbeHststs** has indicated that a character is present in receive buffers for the channel. Internally, a character is removed from a block of characters received and returned to pROBE+.

ProbeConout/**ProbeHstout** sends the character to the port and return. Internally, a character will be put into a data block where the size is one. This data block will be part of a message block that will be used in a call to the DISI **SerialSend** function. Then the DISI call **SerialIoctl** with the command **SIOCPOLL** will be called in a loop until pROBE+'s **UDataCnf** function is called. A call to **UDataCnf** will indicate that the character has been sent. **UDataCnf** sets a flag that the function will test when the SIOCPOLL command returns. **ProbeConout**/**ProbeHstout** then returns.

ProbeEntry tells the DISI serial driver that pROBE+ is being entered by sending the I/O control command **SIOCPROBEENTRY** to the serial driver. The serial driver sets itself up for pROBE+ I/O.

ProbeExit tells the DISI serial driver that pROBE+ is being exited by sending the I/O control command **SIOCPROBEEXIT** to the serial driver. The serial driver resets itself for normal operation.

9.4.2.2 diti.c

The **diti.c** file contains the terminal device driver that follows the Device Independent Terminal Interface (DITI) specification. A full description of the DITI can be found in the "Drivers and Interfaces" section of the *pSOSystem Programmer's Reference*.

9.4.2.3 drv_cutl.c

The **drv_cutl.c** file contains common C utilities that can be used by device drivers. Currently, this file has the following utility functions:

- **ScratchPadTest** determines if a particular bit is set in a task's note-pad register. The syntax is:

  unsigned long ScratchPadTest (unsigned long tid,

  unsigned long register_number,
  unsigned long bit_number);

  tid is the Task ID of the task with the register you want to test.
  register_number in the note-pad register number to be tested.
  bit_number is the bit in the note-pad register to be tested.

- **ScratchPadSet** sets a bit in a task's note-pad register. The syntax is:

  unsigned long ScratchPadSet (unsigned long tid,

  unsigned long register_number,
  unsigned long bit_number)

  tid is the Task ID of the task with the register you want to set.
  register_number in the note-pad register number to be set.
  bit_number is the bit in the note-pad register to be set.

- **ScratchPadUnSet** clears a bit in a task's note-pad register. The syntax is:

  unsigned long **ScratchPadUnSet** (unsigned long **tid**,
  unsigned long **register_number**,
  unsigned long **bit_number**)

  tid is the Task ID of the task with the register you want to clear.
  register_number in the note-pad register number to be cleared.
  bit_number is the bit in the note-pad register to be cleared.

9.4.2.4 gsblk.c

The **gsblk.c** file contains the message block management functions that are used by the serial drivers. These message blocks are similar to streams message blocks. The structures used in these utilities are defined in **pna.h**. (for example, **mblk_t** which is the message block structure). The **gsblk.c** file contains the following functions:

- **gs_allocb** allocates a message block of type **M_DATA** and buffer of a size greater or equal to the specified size. On success, this utility returns a pointer to the allocated message block or a NULL if it fails to get a message block. The syntax is:

  mblk_t *gs_allocb (int size, int pri)

  size is the size in bytes of the data buffer required.
  pri is the priority of the allocation request which can be BPRI_LO, BPRI_MED or BPRI_HI. However, this feature is not used by **gs_allocb**.

- **gs_esballoc** attaches a message buffer to data buffer. It does this by allocating a data block of size 0, and a message block and copying the address of the supplied data buffer to the correct element of the **mblk_t structure**. The function has the following syntax:

  mblk_t *gs_esballoc (unsigned char *base, int size, int pri, frtn_t *frtn)

  base is a pointer the caller supplied address to a data buffer that should be attach to an mblock.
  size is the size of the caller's data buffer.
  pri is not used.
  frtn is a pointer to a caller supplied **frtn_t** structure. This

structure looks like this:

```
struct free_rtn
{
void   (*free_func)();
void   *free_arg;
}
typedef struct free_rtn frtn_t;
```

This structure is defined in the **pna.h** file in the **include** directory.

free_func is a caller supplied address to a function that is called as part of the **gs_freemsg** call (explained next). **gs_freemsg** calls **free_func** so the user knows the data buffer is free to use again.
free_arg is a pointer to a user-supplied argument that is passed on to the user-supplied free function.

- **gs_freemsg** frees a message and associated data blocks. The syntax is:

  void gs_freemsg (register mblk_t *mp)

  mp is the pointer to the message block to be freed.

9.4.2.5 ki_smem.c

The **ki_smem.c** file contains C source code that implements the kernel interface described in the "Interfaces and Drivers" section of the *pSOSystem Programmer's Reference*. The main entry point is the ki function. See the "Interfaces and Drivers" section of the *pSOSystem Programmer's Reference* for more information.

9.4.2.6 ni_smem.c

The **ni_smem.c** file contains the C source code that implements the Shared Memory Network Interface. pNA+ uses this interface as it would for an Ethernet to communicate with other processor boards through the use of shared memory. The main entry point is the function **NiSmem**. See the "Interfaces and Drivers" section of the *pSOSystem Programmer's Reference* for more information on the network interface.

9.4.2.7 ramdisk.c

The **ramdisk.c** file, contains a pHILE+ compatible “RAM disk” driver. The size of the RAM disk is determined by a parameter passed during the initialization call, indicating the size of the disk in blocks. (The size of each block is determined by the logical block size specified in the pHILE+ configuration table.) The memory for the RAM disk is then obtained from pSOS+ region 0. The driver contains the following functions:

- **RdskInit** is the **de_init** entry point of this driver. It initializes the RAM disk by getting memory from pSOS+ region 0.
- **RdskWrite** is the **de_write** entry point for the driver. This is not normally called directly by the user but instead used through pHILE+. pHILE+ uses this entry point to perform function such as create a pHILE+ volume and write files.
- **RdskRead** is the **de_read** entry point for the driver. This is not normally called directly by the user but instead used through pHILE+. pHILE+ will used this entry point to perform functions such as read a file.

9.4.2.8 rarp.c

The **rarp.c** file contains a utility to conduct a **RARP** over the network. This includes the following functions:

- **RarpEth** can be used to get an IP address with RARP for specified Ethernet interface. This function can be executed before pSOS+ is running but does need pNA+ to be initialized. The syntax is:

  unsigned long RarpEth(long (*NiLanPtr) (ULONG fn_code,
  union nientry *p))

  The one argument is the address of the entry point to the Ethernet driver. For **example: IPaddr = RarpEth ((long (*)())NiLan);** Where NiLan is the name of the Ethernet driver entry point.

  RarpEth function is used in the file **drv_conf.c** located in the application directory. It is used by the function **SetUpNI** to install an Ethernet driver into the Network Interface Table before pSOS+ is initialized.

- **GetRarpServerIP** will get the IP of the last server that answered a RARP request. The syntax is:

  unsigned long GetRarpServerIP(void)

  This function returns the **IP address** of the system that answered the last RARP request. This is useful if the RARP server is also the boot server. **GetRarpServerIP** can then be used to **TFTP** the system image into memory and then boot the image. An example of this is in the **tftp** sample application. The **tftp** application is used to create Boot ROMs that can boot a system via TFTP on the Ethernet. Unlike **RarpEth**, **GetRarpServerIP** must be used after the pSOS+ kernel has been initialized.

9.4.2.9 rules.mk

The **rules.mk** file contains the make rules for the files in the **drivers** directory. It should be included in the **makefile** for a BSP.

9.4.2.10 scsi.c

The **scsi.c** file contains the common code of all the **SCSI** device drivers. This driver follows the SCSI driver interface found in the "Interfaces and Drivers" section of the *pSOSystem Programmer's Reference*.

This driver is an initiator-mode-only SCSI driver.

NOTE: pSOSystem does not support a Target Mode SCSI driver.

This driver supports the following devices:

- **Hard disk drives** - Depending on the lower-level hardware and code, both 8-bit and 16-bit (wide) SCSI bus formats are supported.
- **CD-ROM drives**
- **Optical drives** - Tested with Fujitsu M2511A
- **Floppy drives** - Tested with Tech FC-1-11
- **Tape drives** - Tested with Archive 2150s and Exabyte EXB-8205

9.4.2.11 smem_isr.c

The **smem_isr.c** file contains the function **SmemIsr**. This is a common interrupt function for shared memory interrupts. It is used by both **ki_smem.c** and **ni_smem.c**. The ISR uses two call-outs to notify the KI and NI that an interrupt has happened. These call-outs are the **ki_check** and **ni_check** functions.

For the Shared Memory KI, when an interrupt comes in, the **SmemIsr** function calls the **ki_check** function (located in **ki_smem.c**). **ki_check** in turn calls the pSOS+m notification procedure. The pSOSystem notification procedure calls the KI entry point, **ki_call** (in **ki_call.s)** which in turn calls the ki function (in **ki_smem.c**) to do the work of getting the message from the shared memory.

For the Shared Memory NI when an interrupt comes in, the **SmemIsr** function calls the **ni_check** function (located in **ni_smem.c**). **ni_check** in turn calls the pNA+ announce packet entry point. The pNA+ announce packet procedure calls the NI entry point NiSmem, (located in **ni_smem.c**) to do the work of getting the packet from the shared memory interface.

9.4.2.12 tftp_drv.c

The TFTP driver **tftp_drv.c** provides the capability to handle up to eight simultaneous open channels transferring data over a network from a remote host using the TFTP protocol. (The channel numbers are coded in the minor part of the device number.) A complete example application is located in **apps/tftp**. This example serves as the Boot ROM application for pSOSystem for target systems that have networking capabilities. The TFTP driver is also used by the Network Utilities product.

This driver contains the following functions:

- **TftpInit** is the **de_init** entry point for the driver. **TftpInit** function initializes the TFTP interface. This call should be used through the **de_init** system call. For example:

  rc = de_init (DEV_TFTP, &TftpIopb, &ioretval, (void **) &data);

  DEV_TFTP is the major number of the TFTP driver shifted 16 bits to the left.

TftpIopb is a pointer to a TFTP_IOPB structure defined in **include/drv_intf.h**. This structure is not used in the **de_init** call.

ioretval stores the return value from the call (same as **rc**).

data is not used.

- **TftpOpen** is the **de_open** entry point for the driver. **TftpOpen** call allocates a free channel, starts a server task, and waits for an ACK_MSG from the server before returning. This call should be used through the **de_open** system call. For example:

  rc = de_open (**DEV_TFTP, &TftpIopb**, &**ioretval**);

  DEV_TFTP is the MAJOR/MINOR number of the channel to be opened.

  TftpIopb is a TFTP_IOPB structure (defined in **include/drv_intf.h**). It contains two elements, the IP address of the server and the name of the file to request from the server.

  ioretval stores any error code from the driver.

- **TftpRead** is the **de_read** entry point for the driver. This function reads from a remote file located on the server selected in the **de_open**. This call should be used through the **de_read** system call. For example:

  rc = de_read (DEV_TFTP, &TftpReadIopb, &ioretval);

  DEV_TFTP is the MAJOR/MINOR number of the channel to be read.

  TftpReadIopb is a **TFTP_READ_IOPB** structure (defined in **include/drv_intf.h**) that contains two elements, count which is the number of bytes to read from the file and address which is a pointer to the data area to store the data read from the file.

  ioretval stores any error code from the driver.

- **TftpClose** is the **de_close** entry point for the driver. This function closes the connection to the TFTP server. This call should be used through the **de_close** system call. For example:

  de_close (**DEV_TFTP**, **&TftpIopb, &ioretval**);

  DEV_TFTP is the MAJOR/MINOR number of the channel to be opened.

  TftpIopb is a TFTP_IOPB structure (defined in **include/drv_intf.h**). This is not currently used in the **de_close** call.

  ioretval stores any error code from the driver.

9.5 include Directory Files

The include files for BSPs are located in the **include** directory. This directory contains include files that serve as the interface to many parts of pSOSystem such as device drivers and components. The details are provided in the following table.

File	Description
apdialog.h	Prototypes of functions used in an applications dialog.
bspfuncs.h	Prototypes of driver functions and defines for board related functions.
configs.h	Definitions for the node configuration table structure.
ctype.h	Standard C character-manipulation macros. For example, ‘isalpha’.
disi.h	Structures and defines for the DISI interface.
diti.h	Structures and defines for the DITI interface.

File	Description
drv_intf.h	Driver interface structures and driver specific defines.
errno.h	Defines for pREPC+ error numbers.
gsblk.h	Defines and prototypes for the General Serial Block interface.
lan_mib.h	Definition of the mib_stat used in the NI drivers.
m68881.h	Floating-point functions.
math.h	C language math defines.
mmulib.h	Structures and defines used with the MMU library.
phile.h	Structures and defines used with pHILE+.
philecfg.h	pHILE+ configuration table structure.
pmontcfg.h	pMONT configuration table structure.
pna.h	Structures and defines used with pNA+.
pnacfg.h	pNA+ configuration table structure.
prepc.h	Structures and defines used with pREPC+.
prepccfg.h	pREPC+ configuration table structure.
probe.h	Defines used with the pROBE+ debugger.
probecfg.h	pROBE+ configuration table structure.
prpccfg.h	pRPC+ configuration table structure.
psecfg.h	pSE+ configuration table structure.
psos.h	Structures and defines used with pSOS+ and pSOS+m kernels.
psoscfg.h	pSOS+ and pSOS+m configuration table structure, structure for I/O jump table prototype for the InstallDriver function.

File	Description
scsi.h	Contains defines and command structures for the SCSI drivers.
stdarg.h	Macro definitions for variable argument lists.
stdio.h	I/O definitions for pREPC+.
sysvars.h	Definition for the SD_parms structure for the stored system variables.
types.h	Various common definitions used throughout pSOSystem.
version.h	Contains defines for version and copyright strings.

9.6 System Files

The **sys** directory contains the components and system libraries for BSPs. The components are compiled and made into a library call **sys.lib** and **syscxx.lib** for C and C++ systems. The libraries and systems services can be linked into any application.

The **sys** directory has two subdirectories: **os** and **libc**.

- The **os** directory contains the component files that you purchased, the bindings files that come with pSOSystem and the files to build a library of components.
- The **libc** directory contains products that are delivered in library form. Some of these products, such as the MMU library come with pSOSystem while others must be purchased separately, such as Network Utilities.

9.6.1 os Directory

The **os** directory contains:

- **Binding files** - These files are loaded into the system image and are the entry points to the system calls referred to the *pSOSystem System Calls* manual. The functions in these files place a specific function number in the D0 register and then

execute a trap instruction using the **SVCTRAP** (service trap) number. pSOS+ will then do what is needed to call the actual system call code using the function number.

The following is a table of bindings and corresponding filenames:

File	Description
psos.s	Run-time bindings for pSOS+
pna.s	Run-time bindings for pNA+
phile.s	Run-time bindings for pHILE+
prpc.s	Run-time bindings for pRPC+
pse.s	Run-time bindings for pSE+
ptli.s	Run-time bindings for pTLI+
pskt.s	Run-time bindings for pSKT+
pmont.s	Run-time bindings for pMONT

NOTE: pSOSystem supplies all the bindings files for the components. This allows your application to compile and link without having the component installed. However, at run time a call to a system call where the component is missing results in the following **ERR_SSFN** error message:

```
Illegal system service function number.
```

- **Component files** contain encoded data that represents a component image. These files are assembled into the system library. The system library is then linked into the system image if you choose to link in the OS. The presence of any of these files depends on the components you have installed.

  The following is a table of components and corresponding filenames:

File	Component
ks680.s	pSOS+ kernel (68000)
ks681.s	pSOS+ kernel (68010)
ks682.s	pSOS+ kernel (68020)
ks683.s	pSOS+ kernel (68360)
ks686.s	pSOS+ kernel (68060)
km680.s	pSOS+m kernel (68000)
km681.s	pSOS+m kernel (68010)
km682.s	pSOS+m kernel (68020)
km683.s	pSOS+m kernel (68360)
km686.s	pSOS+m kernel (68060)
rs68k.s	pROBE+ debugger
fs68k.s	pHILE+ file system manager
lc68k.s	pREPC+ run-time C library
ns68k.s	pNA+ networking manager
nr68k.s	pRPC+ RPC component
pn68k.s	pSE+, pTLI+ and pSKT+ OpEN components
pm68k.s	pMONT component

- **Build files** are used to build **sys.lib** and **sysxx.lib**, **makefile** and **filelist**. executing make with no arguments builds both **sys.lib** and **sysxx.lib**. you can also execute it with an argument of **sys.lib** or **sysxx.lib** to build one or the other.

  The **makefile** uses the **mklibmk** (utility name) to create a version of itself that contains lists of the correct objects for the

components you have in your system. The **mklibmk** utility takes two arguments, a directory to search and a file list of valid files; **mklibmk** creates a list of object files that will be built. A object file is named in the list if there is an entry in the file **filelist** and there is a source file in the directory for it.

After all components have been installed and when new components are added or updated you must make the **sys.lib** and **sysxx.lib** to include the new components in those libraries.

9.6.2 libc Directory

The **sys/libc** directory contains libraries that can be included in your application. The libraries in this directory will depend on what products you have purchased. The following libraries are included in the base release of pSOSystem:

Library	Description
fpu040.lib	Floating-point functions for the MC68040
fpu060.lib	Floating-point functions for the MC68060
loader.lib	Loader functions (see *pSOSystem Programmer's Reference*)
mmu68030.lib	Memory Management functions for the MC68030 processor (see *pSOSystem Programmer's Reference)*
mmu68040.lib	Memory Management functions for the MC68040 and MC68060 processors (see *pSOSystem Programmer's Reference)*

Other products such as OpEN and Network Utilities add libraries to the **libcs** directory. The manuals that accompany them describe the names and functions in their libraries.

A Board-Specific Information

This appendix contains information on individual hardware products, such as PROM locations and switch settings. The sections are organized by manufacturer and product. Table A-1 provides a summary of the specific boards described in this appendix.

Table A-1 Summary of Board-Specific Information

Board	Path	Section	Page
Motorola FADS68302	$PSS_ROOT/bsps/e302	A.1	A-2
Motorola EVS-68332	$PSS_ROOT/bsps/e332	A.2	A-7
Motorola EVS-68340	$PSS_ROOT/bsps/e340	A.3	A-11
Motorola MVME162	$PSS_ROOT/bsps/m162	A.4	A-14
Motorola MVME162LX (2xx)	$PSS_ROOT/bsps/m162_2xx	A.5	A-18
Motorola MVME162FX (5xx)	$PSS_ROOT/bsps/m162_5xx	A.6	A-22
Motorola MVME167	$PSS_ROOT/bsps/m167	A.7	A-25
Motorola MVME177	$PSS_ROOT/bsps/m177	A.8	A-29

Table A-1 Summary of Board-Specific Information

Board	Path	Section	Page
Motorola QUADS-68360	$PSS_ROOT/bsps/e360	A.9	A-33
EST SBC360	$PSS_ROOT/bsps/sbc360	A.10	A-38

A.1 Motorola FADS68302

Directory **$PSS_ROOT/bsps/e302** contains a pSOSystem Board Support-Package (BSP) for the Motorola FADS68302 evaluation board. This BSP supports the MC68302, MC68LC302, and MC68PM302 processors.

A.1.1 ROM Installation and Hardware Setup

All switches and jumpers should be set to the default values listed in the FADS68302 Manual. The **flashmem.hex** file must be loaded into the flash using the Motorola debugger. See the Section A.1.5, "Making pSOSystem Flash Code" for information on how to load the **flashmem.hex** file into the flash.

A terminal should be connected to the DB-9 connector marked P2 TERMINAL.

A.1.2 Flash Memory Programming

Rather than a pSOSystem Boot ROM set, the **e302** BSP contains an S-record file (**$PSS_ROOT/bsps/e302/flashmem.hex**), which is intended to be programmed into the flash memory of the FADS board. You can do this by using the Motorola IMPbug monitor (installed in ROM socket U48) to download **flashmem.hex** over the serial line and program it into flash memory. The following QUICCbug command should be used for downloading the file:

```
lo 40000
```

Next, use the appropriate command for your terminal emulation program to download **flashmem.hex**.

Example of downloading and programming the flash using the UNIX tip utility.

If downloading using **tip**, the **flashmem.hex** file will need to be downloaded in pieces. This can be done by entering the command:

split flashmem.hex

To program the **flashmem.hex** file into the FADS board's flash memory, take the following steps:

1. Issue the following command to the Motorola IMPbug:

   lo 40000

2. Download the **flashmem.hex** file using your terminal software.

   If you are using **tip**, you will probably need to use this modified download procedure:

   a) Download the split pieces of the file from **tip**.

   b) Type ~>

      tip prompts as follows:

      Local file name?

Enter **xaa** preceded by a pathname (for example, **~/apps/bootrom/xaa**) After the file is transferred the IMPbug will send a series of dots followed by a newline. After the newline send **xab** the same way. Repeat this until all the files created by spilt have been sent.

After the complete **flashmem.hex** has been downloaded (or split pieces have been downloaded), the Motorola IMPbug prompt should appear. Issue the following command to the IMPbug:

lof 40000 57FFF 240000

IMPbug will ask for confirmation. Retype:

lof 40000 57FFF 240000

The flash memory will be programmed with the pROBE+ ROM code.

To enter the pROBE+ now stored in the flash memory, issue the following command to the Motorola IMPbug:

```
go 240008
```

The pROBE+ boot dialog should appear.

A.1.3 Serial Channel Usage

The pSOSystem BSP for the FADS68302 includes serial drivers for all five of its serial channels. pSOSystem serial channels 1 and 2 are ports A and B of the M68681 DUART. pSOSystem serial channels 3, 4, and 5 are the M68302's on-chip serial channels. (SCC 1, 2, and 3, respectively)

A TTL-to-RS232 voltage level converter must be connected to the board for the M68302's on-chip serial channels. An example converter schematic is available through Integrated Systems Technical Support.

Note that serial channels 1 and 2 are wired differently. If both are connected to similar equipment, such as a host system, one of them requires a null modem and the other does not. Serial channel 2 may have the DCD and Tx pins reversed from their standard pinouts, if you have an early version of the board.

NOTE: This BSP will work on the old ADS68302 board by changing the define BD_M68681_BASE to 0x400001 instead of 0x620001. This define statement is in the $PSS_ROOT/bsps/e302/src/board.h file. After making the change you will need to compile the BSP.

A.1.4 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0. The ROMs for the FADS68302 use RAM addresses 0 - 0x3FFF, so do not download code to this area. The entry point for downloaded systems is 0x4008 if the operating system and application are linked together. If the operating system and application are linked separately, the entry point for the downloaded operating system is 0x30008.

ROM	0025'7FFF 0024'8000	pROBE+ code in ROM
	0024'7FFF 0024'0000	Boot ROM startup and driver code
DRAM	0007'FFFF 0000'4000	For downloaded operating system and application linked together. (When linked separately, the application occupies 0000'4000 - 0002'FFFF and operating system occupies 0003'0000 - 0007'FFFF.)
	0000'3FFF 0000'0800	Reserved for use by Boot ROMs
	0000'07FF 0000'0700	Parameter Storage Area
	0000'06FF 0000'0400	Not used by pSOSystem software
	0000'03FF 0000'0000	Vector Page and Node Anchor (Node Anchor is at $44)

A.1.5 Making pSOSystem Flash Code

The pSOSystem Boot ROMs for the FADS68302 are built using the code in sample application **apps/proberom**. You can build a custom Boot ROM set by taking the following steps and then changing the code as needed (the example commands are for a UNIX host system):

1. Copy the **proberom** file from the **PSS_ROOT/apps** directory to a working directory:

   cp -r $PSS_ROOT/apps/proberom rombuild

2. Make the working directory the current directory:

   cd rombuild

3. Edit **makefile** and set PSS_BSP to **$(PSS_ROOT)/bsps/e302** and PSS_COMLIB to **-l mcc68kab.lib** (see your MRI compiler documentation for information about setting environment variables to help the linker locate the library). If the line has a comment mark, remove it.

   NOTE: When making a Boot ROM for the ADS68302 board, you must change the code address from 240000 to 200000 in the $PSS_ROOT/bsps/e302/rom.lnk file before completing step 4.

4. Enter the following command:

   make rom.hex

5. Use the **splitrom** utility to relocate the **rom.hex** file to 0 by entering the following command:

   $PSS_ROOT/bin/$HOST/splitrom rom.hex b 240000 128 flashmem.hex

NOTE: Integrated Systems has found that running the Motorola Diagnostic Self Test clears flash memory of everything but the Motorola Debug Monitor Code. If you run this test, you will have to reload the Boot ROM code into flash memory.

For the ADS68302 boards you will need to make a PROM. In that case the **rom.hex** file is linked to be located at 0x200000, so set the PROM programmer to expect load addresses in the range 0x200000 - 0x23FFFF. Optionally, you can use a program called **splitrom** to split **rom.hex** into files that correspond to the ROMs and load at zero. To use **splitrom**, use the following command:

$PSS_ROOT/bin/$HOST/splitrom rom.hex b 200000 256 rom.u46 rom.u47

where *HOST* is the name of the host system (for example, **sunos**, **msdos**, **hpux**, and **aix**).

NOTE: When downloading from a lost system, the serial channel baud rate cannot be more than 9600 baud.

Special Note on Processor Clock Speed

The driver for the Serial Communications Controller and timer are designed to work at different system clock rates. These drivers use the #define BD_68302_HZ to determine the system clock rate. BD_68302_HZ must be set to the CPU clock rate (This CPU clock rate is half the actual crystal rate). For example, if the CPU rate was 16.67 Mhz, then BD_68302_HZ would be set as follows:

#define BD_68302_HZ = 16666667

By default, pSOSystem used a CPU clock rate of 16.67 Mhz when the **bsp.lib** was compiled for the release. If this clock needs to be changed, then the BSP must be recompiled.

A.2 Motorola EVS-68332

Directory **$PSS_ROOT/bsps/e332** contains a pSOSystem BSP for the Motorola EVS-68332 evaluation board.

A.2.1 ROM Installation and Hardware Setup

The socket and jumper numbering in this section refer to revision C of the platform board (M68332PFB). See the Motorola documentation for earlier revisions. The development interface board (M68332BCCDI) is not needed, so installing it is optional.

Install the ROMs labeled U2 and U4 in the appropriate sockets on the platform board. The pSOSystem ROMs for the M68332EVS are 27512 EPROMs. To configure the board for these ROMs, jumpers J2 and J3 must be set to CSBOOT; jumpers J4 and J7 must be set to EPROM; and jumpers J5 and J6 must be set to 27C512.

The M68332BCC comes as either revision A or revision B, and the pSOSystem ROMs work with either revision. If you use a revision A M68332BCC, jumpers J8-J13 on the platform board must be set to A. The platform board comes without jumper pins installed for J8-J13 and instead has traces that connect the revision B selections. So, to use a revision A M68332BCC, you may first have to cut these traces and install jumper pins for J8-J13.

If a revision B M68332BCC is installed but no jumper pins are installed on the platform board for J8-J13, it nevertheless works because the

default configuration of the platform board is for revision B M68332BCCs. If pins already exist at these locations, make sure the jumpers are in the B position.

On the M68332BCC, pins 2 and 3 on jumper J6 must be connected. This configuration disables the M68332BBC EPROM that contains Motorola's 332Bug and allows the pSOSystem ROMs to boot instead. Some M68332BCCs may have a trace that connects pins 1 and 2 on J6. If this connection exists, the trace must be cut. The other jumpers on the M68332BCC should have the default settings.

A terminal should be connected to the platform board DB-9 connector marked TERMINAL. The serial protocol is 9600 baud, 8-bit data, 1 stop bit, and no parity.

A.2.2 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0. The ROMs for the EVS-68332 use RAM addresses 0 - 0x3FFF, so do not download code to this area. The entry point for downloaded systems is 0x4008 if the operating system and application are linked together. Linking the operating system and application separately is not possible without adding memory to this board.

Unless additional memory has been installed on the board, you must use the **ramrc** memory configuration instead of the **ram** configuration. **ramrc** uses the pROBE+ and pSOS+ code in the Boot ROMs and thus saves RAM space.

ROM	0009'FFFF 0009'0000	pROBE+ code in ROM
	0008'FFFF 0008'C000	pSOS+ code in ROM
	0008'BFFF 0008'0000	Boot ROM startup and driver code

SRAM	0001'FFFF 0001'0000	Expansion SRAM
	0000'FFFF 0000'4000	Downloaded operating system and application
	0000'3FFF 0000'0800	Reserved for use by Boot ROMs
	0000'07FF 0000'0700	Parameter Storage Area
	0000'06FF 0000'0400	Not used by pSOSystem software
	0000'03FF 0000'0000	Vector Page and Node Anchor (Node Anchor is at $44)

A.2.3 Making a pSOSystem Boot ROM

The pSOSystem Boot ROMs for the EVS-68332 are built using the code in sample application **apps/proberom** and a modified version of the **e332** BSP. You can build a custom Boot ROM set by taking the following steps and then changing the code as needed (the example command lines are for a UNIX host system):

1. Copy **PSS_ROOT/apps/proberom** to a working directory:

   cp -r $PSS_ROOT/apps/proberom rombuild

2. Copy the contents of **PSS_ROOT/bsps/e332** to the same working directory. Use a recursive copy so that the **src** subdirectory is copied as well:

   cp -r $PSS_ROOT/bsps/e332/* rombuild

3. Make the working directory the current directory:

   cd rombuild

4. Edit **src/board.h** and set BD_BCC_VERSION to BD_BCC_TEST.

5. Switch to subdirectory **src**, enter the **make** command, and return to the working directory when the build is finished:

```
cd src
make
cd ..
```

6. Is the **makefile,** set PSS_BSP to "." (which means *the current directory*).

7. In the **dummy.c** file, and add the following lines to the end of the file:

```
extern void psos1(void);
static void a(void){psos1();}
```

8. Enter the following command:

```
make rom.hex
```

The resulting file is **rom.hex**. It is an S-record hex file you can load into a PROM programmer. This file is linked to be located at 0x80000, so set the PROM programmer to expect load addresses in the range 0x80000 - 0x81FFFF. Optionally, you can use a program called **splitrom** to split **rom.hex** into files that correspond to the ROMs and load at 0. To use **splitrom**, use the following command:

```
$PSS_ROOT/bin/$HOST/splitrom rom.hex b 80000 128 rom.u4 rom.u2
```

where *HOST* is the name of the host system (for example, **sunos**, **msdos**, **hpux**, and **aix**.

A.2.4 Additional Information

The M68332EVS is very limited in memory. The system comes with only the 64 Kbytes of SRAM on the M68332BCC. An additional 64 Kbytes can be added by inserting a 32-Kbyte SRAM in sockets U1 and U3 on the platform board. The pSOSystem software maps the 64 Kbytes of the M68332BCC RAM to 0x0000-0xFFFF. If RAM is installed in U1/U3, it is mapped to 0x10000-0x1FFFF. The chip-select logic for U1/U3 is set for 0 wait states, so it requires SRAMs of 100 ns or faster.

If you use a revision A M68332BCC, you must edit the **board.h** file in the **$PSS_ROOT/bsps/e332/src** directory and change the BD_BCC_VERSION definition to REVA. By default, the pSOSystem software is set to build applications for revision B M68332BCCs. The pSOSystem ROMs work on either revision.

If Your Code is Not Working:

Two revisions of the 68332 BBCC exist: revision A and revision B (this BSP supports both). You can determine which revision is present two ways. You can determine the revision at compile-time by setting the BD_BCC_VERSION definition in **PSS_ROOT/bsps/e332/src/board.h** to either BD_BCC_REVA or BD_BCC_REVB. You can determine the revision at run time by setting the definition of BD_BCC_VERSION to BD_BCC_TEST. However, the test to determine the board revision may corrupt the RAM and therefore should be used only in code that runs from (EP)ROM. If you set BD_BCC_VERSION to BD_BCC_TEST and run the resulting code from RAM, it may not work. If you *incorrectly* define BD_BCC_VERSION to BD_BCC_REVA or BD_BCC_REVB, the resulting code does not work. pSOSystem ROMs should be made with BD_BCC_VERSION set to BD_BCC_TEST so they can work with either board revision.

The Delay100ms call for the **e332** BSP calculates a constant based on the CPU clock rate. It divides the CPU Hz by a constant. This constant was determined at 16.78 MHz running the code out of ROMs with six wait states programmed. If the number of wait states for the ROM changes or the code is run in the faster SRAM, you must adjust this constant.

A.3 Motorola EVS-68340

Directory **$PSS_ROOT/bsps/e340** contains a pSOSystem BSP for the Motorola EVS-68340 evaluation board.

A.3.1 ROM Installation and Hardware Setup

The socket and jumper numbering that follows applies to revision C of the platform board (M68340PFB). Consult the Motorola documentation for earlier revisions. The development interface board (M68340BCCDI) is not needed, so installing it is optional.

Install the ROMs labeled U2 and U4 in the appropriate sockets on the platform board. The pSOSystem ROMs for the M68340EVS are 27512 EPROMs. To configure the board for these ROMs, jumpers J2 and J3 must be set to CSBOOT; jumpers J4 and J7 must be set to EPROM; and jumpers J5 and J6 must be set to 27C512.

On the M68340BCC, pin 2 must connect to pin 3 on jumper J2. This disables the M68340BBC EPROM containing Motorola's 340Bug and allows the pSOSystem ROMs to boot instead. The other jumpers on the M68340BCC should have the default settings.

A terminal should be connected to the platform board DB-9 connector marked TERMINAL. The serial protocol is 9600 baud, 8-bit data, 1 stop bit, and no parity.

A.3.2 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0. The ROMs for the EVS-68340 use RAM addresses 0 through 0x3FFF, so do not download code to this area. The entry point for downloaded systems is 0x4008 if the operating system and application are linked together. Linking the operating system and application separately is not possible without adding memory to this board.

Unless additional memory has been installed on the board, you must use the **ramrc** memory configuration instead of the **ram** configuration. The **ramrc** configuration uses the pROBE+ and pSOS+ code in the Boot ROMs and thus saves RAM space.

ROM	0007'8FFF 0006'9000	pROBE+ code in ROM
	0006'8FFF 0006'5000	pSOS+ code in ROM
	0006'4FFF 0006'0000	Boot ROM startup and driver code

SRAM	0001'FFFF 0001'0000	Expansion SRAM
	0000'FFFF 0000'4000	Downloaded operating system and application
	0000'3FFF 0000'0800	Reserved for use by Boot ROMs
	0000'07FF 0000'0700	Parameter Storage Area
	0000'06FF 0000'0400	Not used by pSOSystem software
	0000'03FF 0000'0000	Vector Page and Node Anchor (Node Anchor is at $44)

A.3.3 Making a pSOSystem Boot ROM

The pSOSystem Boot ROMs for the EVS-68340 are built using the code in sample application **apps/proberom**. You can build a custom Boot ROM set by taking the following steps and then changing the code as needed (the example command lines are for a UNIX host system):

1. Copy **PSS_ROOT/apps/proberom** to a working directory:

   cp -r $PSS_ROOT/apps/proberom rombuild

2. Make this working directory your current directory:

   cd rombuild

3. Edit **makefile** and set PSS_BSP to **$(PSS_ROOT)/bsps/e340**. If the line has a comment mark, remove it.

4. Edit **dummy.c** and add the following lines to the end of the file:

   ```
   extern void psos1(void);
   static void a(void){psos1();}
   ```

5. Enter the following:

   make rom.hex

The resulting file is **rom.hex**. It is an S-record hex file you can load into a PROM programmer. This file is linked to be located at 0x60000, so you must set the PROM programmer to accept load addresses in the range 0x60000 - 0x7FFFFF. Optionally, you can use a program called **splitrom** to split **rom.hex** into files that correspond to the ROMs and load at 0. Execute **splitrom** by entering the following command:

$PSS_ROOT/bin/$HOST/splitrom rom.hex b 60000 128 rom.u4 rom.u2

where *HOST* is the name of the host system (for example, **sunos**, **msdos**, **hpux,** and **aix**).

A.3.4 Additional Information

The M68340EVS is very limited in memory. The system comes with only 64 Kbytes of SRAM on the M68340BCC. You can add an additional 64 Kbytes by inserting 32 Kbytes SRAMs in sockets U1 and U3 on the platform board. The pSOSystem environment maps the 64 Kbytes of the M68340BCC RAM to 0x0000-0xFFFF. If RAM is installed in U1/U3, it is mapped to 0x10000-0x1FFFF. The chip-select logic for U1/U3 is set for zero wait states, so SRAMs 100 ns or faster are required.

A.4 Motorola MVME162

Directory **$PSS_ROOT/bsps/m162** contains a pSOSystem BSP for the Motorola MVME162 board.

A.4.1 ROM Installation and Hardware Setup

Install the ROM in socket U38. The pSOSystem ROMs for the MVME162 are 27C040 OTPROMs. To configure the board for these ROMs, connect pins 2-3 on jumper J21. The MVME162 has the ability to power up either from the ROM in socket U38 or from the MBUG monitor that comes programmed into flash memory. To enable the pSOSystem ROM, the jumper connecting pins 9 and 10 of jumper block J22 must be removed prior to powerup.

A terminal can be connected either to the front panel connector marked SERIAL PORT 1/CONSOLE or, if a transition module is used (MVME712M, MVME712A, MVME712AM, or MVME712B), to the transition module connector marked SERIAL PORT 2/TTY01. Using a transition module may require a null modem adapter, and this depends on the module's jumper settings. The serial protocol is 9600 baud, 8-bit data, 1 stop bit, and no parity. For an Ethernet interface, a cable from the connector marked ETHERNET on the transition module must be connected to the Ethernet transceiver.

Be sure the system controller jumper is set correctly, with pins 1 and 2 of jumper J1 connected if the board is the system controller and disconnected if the board is not.

A.4.2 Serial Channel Usage

The MVME162 has two serial channels. They are pSOSystem serial channels 1 and 2.

A.4.3 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0. The Boot ROMs use RAM from 0x800 through 0x1FFFF, so do not attempt to use them to download code to this area. The entry point for downloaded systems is 0x28008 if the operating system and application are linked together. If the operating system and application are linked separately, the entry point for the downloaded operating system is 0x60008.

These boards are available with varying DRAM sizes. The sizes include 4 Mb, 8 Mb, 16 Mb, and 32 Mb. In the memory map below, *xx* should be replaced by 03, 07, 0F, or 1F as appropriate.

NVRAM	FFFC'00FF FFFC'0000	Parameter Storage Area

ROM	FF83'9FFF FF82'2000	pNA+ code in ROM
	FF82'1FFF FF81'2000	pROBE+ code in ROM
	FF81'1FFF FF80'E000	pSOS+ code in ROM
	FF80'DFFF FF80'0000	Boot ROM startup and driver code
RAM	0*xx*F'FFFF 0002'8000	Downloaded operating system and application, when linked together. When linked separately, application occupies 0002'8000 - 0005'FFFF and operating system occupies 0006'0000 - 0*xx*F'FFFF
	0002'7FFF 0000'0800	Reserved for use by Boot ROMs
	0000'07FF 0000'0700	Not used by pSOSystem software
	0000'06FF 0000'0580	Reserved for Shared Memory Kernel Interface (SMKI) Directory
	0000'057F 0000'0400	Reserved for Shared Memory Network Interface (SMNI) Directory
	0000'03FF 0000'0000	Vector Page and Node Anchor (Node Anchor is at $44)

A.4.4 Making a pSOSystem Boot ROM

The pSOSystem Boot ROM for the MVME162 are built using the code in sample application **apps/tftp** and a slightly modified version of the **m162** BSP. You can build a custom Boot ROM by taking the following steps and then changing the code as needed (the example commands are for a UNIX host system):

1. Copy **PSS_ROOT/apps/tftp** to a working directory:

```
cp -r $PSS_ROOT/apps/tftp rombuild
```

2. Make the working directory the current directory:

   ```
   cd rombuild
   ```

3. If you are using the pSOS+m kernel instead of the pSOS+ kernel, edit **sys_conf.h** and set SC_PSOS to NO and SC_PSOSM to YES.

4. Enter the following command:

   ```
   make rom.hex
   ```

The resulting file is **rom.hex**. It is an S-record hex file you can load into a PROM programmer. This file is linked to be located at 0xFF800000, so set the PROM programmer to expect load addresses in the range 0xFF800000 - 0xFF83FFFF. Optionally, you can use a program called **splitrom** to relocate **rom.hex** to 0.

To use **splitrom**, enter the following command:

```
$PSS_ROOT/bin/$HOST/splitrom rom.hex b FF800000 256 rom.u38
```

where *HOST* is the name of the host system (for example, **sunos**, **msdos**, **hpux**, and **rs6000**).

A.4.5 VMEbus Configuration Information

The following list contains configuration information for VMEbus. The item gives a pSOSystem default, when applicable.

- The System Controller is jumper-enabled.
- The programmable bus arbitration mode is either round-robin or priority. The pSOSystem default is round-robin.
- The default for the programmable bus request level is 3.
- The programmable bus release mode can be either release on request or release when done. The pSOSystem default is release on request.
- The pSOSystem default for the register-enabled FAIR bus request mode is enabled.
- The programmable master data bus width is either 32 or 16 bits. The pSOSystem default is 32 bits.

- Register-enabled re-arbitration is possible after a 256-microsecond arbiter timeout. The pSOSystem default is for re-arbitration to be enabled.
- The programmable VMEbus global timeout is 8, 64, or 256 microseconds, or else disabled. The pSOSystem default is the 256-microsecond VMEbus global timeout.
- The programmable bus request timeout can be 1, 32, or 64 microseconds, or else disabled. The pSOSystem default is the 32-millisecond bus request timeout.

A.5 Motorola MVME162LX

Directory **$PSS_ROOT/bsps/m162_2xx** contains a pSOSystem BSP for the Motorola MVME162LX board. NOTE the LX boards all labeled as MVME 162-2xx. Where xx is the specific board. MVME162LX boards are in the 200 series.

A.5.1 ROM Installation and Hardware Setup

Install the ROM in socket U24. The pSOSystem ROMs for the MVME162 are 27C040 OTPROMs. To configure the board for these ROMs, connect pins 2-3 on jumper J21. The MVME162 has the ability to power up either from the ROM in socket U26 or from the MBUG monitor that comes programmed into flash memory. To enable the pSOSystem ROM, the jumper connecting pins 9 and 10 of jumper block J22 must be removed prior to powerup.

A terminal is connected to the front panel connector marked CONSOLE 1. This requires a null modem adapter. The serial protocol is 9600 baud, 8-bit data, 1 stop bit, and no parity. For an Ethernet interface, a cable from the connector marked ETHERNET PORT on the front panel must be connected to the Ethernet transceiver.

Be sure the system controller jumper is set correctly, with pins 1 and 2 of jumper J1 connected if the board is the system controller and disconnected if the board is not.

A.5.2 Serial Channel Usage

The MVME162 has four serial channels. They are pSOSystem serial channels 1, 2, 3, and 4.

A.5.3 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0. The Boot ROMs use RAM from 0x800 through 0x1FFFF, so do not attempt to use them to download code to this area. The entry point for downloaded systems is 0x28008 if the operating system and application are linked together. If the operating system and application are linked separately, the entry point for the downloaded operating system is 0x60008.

These boards are available with varying DRAM sizes. The sizes include 4 MB, 8 MB, 16 MB, and 32 MB. In the memory map below, *xx* should be replaced by 03, 07, 0F, or 1F as appropriate.

NVRAM	FFFC'00FF FFFC'0000	Parameter Storage Area
ROM	FF83'4FFF FF82'2000	pNA+ code in ROM
	FF82'1FFF FF81'2000	pROBE+ code in ROM
	FF81'1FFF FF80'E000	pSOS+ code in ROM
	FF80'DFFF FF80'0000	Boot ROM startup and driver code

RAM	0*xx*F'FFFF 0002'8000	Downloaded operating system and application, when linked together. When linked separately, application occupies 0002'8000 - 0005'FFFF and operating system occupies 0006'0000 - 0*xx*F'FFFF
	0002'7FFF 0000'0800	Reserved for use by Boot ROMs
	0000'07FF 0000'0700	Not used by pSOSystem software
	0000'06FF 0000'0580	Reserved for Shared Memory Kernel Interface (SMKI) Directory
	0000'057F 0000'0400	Reserved for Shared Memory Network Interface (SMNI) Directory
	0000'03FF 0000'0000	Vector Page and Node Anchor (Node Anchor is at $44)

A.5.4 Making a pSOSystem Boot ROM

The pSOSystem Boot ROM for the MVME162 are built using the code in sample application **apps/tftp** and a slightly modified version of the **m162** BSP. You can build a custom Boot ROM by taking the following steps and then changing the code as needed (the example commands are for a UNIX host system):

1. Copy **PSS_ROOT/apps/tftp** to a working directory:

   cp -r $PSS_ROOT/apps/tftp rombuild

2. Make the working directory the current directory:

   cd rombuild

3. If you are using the pSOS+m kernel instead of the pSOS+ kernel, edit **sys_conf.h** and set SC_PSOS to NO and SC_PSOSM to YES.

4. Enter the following **make** command:

   make rom.hex

The resulting file is **rom.hex**. It is an S-record hex file you can load into a PROM programmer. This file is linked to be located at 0xFF800000, so set the PROM programmer to expect load addresses in the range 0xFF800000 - 0xFF83FFFF. Optionally, you can use a program called **splitrom** to relocate **rom.hex** to 0.

To use **splitrom**, enter the following command:

$PSS_ROOT/bin/$HOST/splitrom rom.hex b ff800000 256 rom.u38

where *HOST* is the name of the host system (for example, **sunos**, **msdos**, **hpux**, and **aix**).

A.5.5 VMEbus Configuration Information

The following list contains configuration information for VMEbus. The item gives a pSOSystem default, when applicable.

- The System Controller is jumper-enabled.
- The programmable bus arbitration mode is either round-robin or priority. The pSOSystem default is round-robin.
- The default for the programmable bus request level is 3.
- The programmable bus release mode can be either release on request or release when done. The pSOSystem default is release on request.
- The pSOSystem default for the register-enabled FAIR bus request mode is enabled.
- The programmable master data bus width is either 32 or 16 bits. The pSOSystem default is 32 bits.
- Register-enabled re-arbitration is possible after a 256-microsecond arbiter timeout. The pSOSystem default is for re-arbitration to be enabled.
- The programmable VMEbus global timeout is 8, 64, or 256 microseconds, or else disabled. The pSOSystem default is the 256-microsecond VMEbus global timeout.
- The programmable bus request timeout can be 1, 32, or 64 microseconds, or else disabled. The pSOSystem default is the 32-millisecond bus request timeout.

A.6 Motorola MVME162FX

Directory **$PSS_ROOT/bsps/m162_5xx** contains a pSOSystem BSP for the Motorola MVME162FX board. NOTE the FX boards all labeled as MVME 162-5xx. Where xx is the specific board. MVME162LX boards are in the 500 series.

A.6.1 ROM Installation and Hardware Setup

Install the ROM in socket U47. The pSOSystem ROMs for the MVME162 are 27C040 EPROMs. To configure the board for these ROMs, connect pins 2-3 on jumper J21. The MVME162 has the ability to power up either from the ROM in socket U47 or from the MBUG monitor that comes programmed into flash memory. To enable the pSOSystem ROM, the jumper connecting pins 9 and 10 of jumper block J22 must be removed prior to powerup.

A terminal can be connected either to the front panel connector marked SERIAL PORT 1/CONSOLE or, if a transition module is used (MVME712M, MVME712A, MVME712AM, or MVME712B), to the transition module connector marked SERIAL PORT 2/TTY01. Using a transition module may require a null modem adapter, and this depends on the module's jumper settings. The serial protocol is 9600 baud, 8-bit data, 1 stop bit, and no parity. For an Ethernet interface, a cable from the connector marked ETHERNET on the transition module must be connected to the Ethernet transceiver.

Be sure the system controller jumper is set correctly, with pins 1 and 2 of jumper J1 connected if the board is the system controller and disconnected if the board is not.

A.6.2 Serial Channel Usage

The MVME162 has two serial channels. They are pSOSystem serial channels 1 and 2.

A.6.3 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0. The Boot ROMs use RAM from 0x800 through 0x1FFFF, so do not attempt to use them to download code to this area. The entry point for downloaded systems is 0x28008 if the operating

system and application are linked together. If the operating system and application are linked separately, the entry point for the downloaded operating system is 0x60008.

These boards are available with varying DRAM sizes. The sizes include 4 MB, 8 MB, 16 MB, and 32 MB. In the memory map below, *xx* should be replaced by 03, 07, 0F, or 1F as appropriate.

NVRAM	FFFC'00FF FFFC'0000	Parameter Storage Area
ROM	FF83'4FFF FF82'2000	pNA+ code in ROM
	FF82'7FFF FF81'2000	pROBE+ code in ROM
	FF81'7FFF FF80'E000	pSOS+ code in ROM
	FF80'DFFF FF80'0000	Boot ROM startup and driver code
RAM	0*xx*F'FFFF 0002'8000	Downloaded operating system and application, when linked together. When linked separately, application occupies 0002'8000 - 0005'FFFF and operating system occupies 0006'0000 - 0*xx*F'FFFF
	0002'7FFF 0000'0800	Reserved for use by Boot ROMs
	0000'07FF 0000'0700	Not used by pSOSystem software
	0000'06FF 0000'0580	Reserved for Shared Memory Kernel Interface (SMKI) Directory
	0000'057F 0000'0400	Reserved for Shared Memory Network Interface (SMNI) Directory
	0000'03FF 0000'0000	Vector Page and Node Anchor (Node Anchor is at $44)

A.6.4 Making a pSOSystem Boot ROM

The pSOSystem Boot ROM for the MVME162 are built using the code in sample application **tftp** and a slightly modified version of the **m162** BSP. You can build a custom Boot ROM by taking the following steps and then changing the code as needed (the example commands are for a UNIX host system):

1. Copy **PSS_ROOT/apps/tftp** to a working directory:

   cp -r $PSS_ROOT/apps/tftp rombuild

2. Make the working directory the current directory:

   cd rombuild

3. If you are using the pSOS+m kernel instead of the pSOS+ kernel, edit the **sys_conf.h** file and set SC_PSOS to NO and SC_PSOSM to YES.

4. Enter the following **make** command:

   make rom.hex

The resulting file is **rom.hex**. It is an S-record hex file you can load into a PROM programmer. This file is linked to be located at 0xFF800000, so set the PROM programmer to expect load addresses in the range 0xFF800000 - 0xFF83FFFF. Optionally, you can use a program called **splitrom** to relocate **rom.hex** to 0.

To use **splitrom**, enter the following command:

$PSS_ROOT/bin/$HOST/splitrom rom.hex b FF800000 256 rom.u38

where *HOST* is the name of the host system (for example, **sunos**, **msdos**, **hpux**, and **aix**).

A.6.5 VMEbus Configuration Information

The following list contains configuration information for VMEbus. The item gives a pSOSystem default, when applicable.

- The System Controller is jumper-enabled.
- The programmable bus arbitration mode is either round-robin or priority. The pSOSystem default is round-robin.
- The default for the programmable bus request level is 3.

- The programmable bus release mode can be either release on request or release when done. The pSOSystem default is release on request.
- The pSOSystem default for the register-enabled FAIR bus request mode is enabled.
- The programmable master data bus width is either 32 or 16 bits. The pSOSystem default is 32 bits.
- Register-enabled re-arbitration is possible after a 256-microsecond arbiter timeout. The pSOSystem default is for re-arbitration to be enabled.
- The programmable VMEbus global timeout is 8, 64, or 256 microseconds, or else disabled. The pSOSystem default is the 256-microsecond VMEbus global timeout.
- The programmable bus request timeout can be 1, 32, or 64 microseconds, or else disabled. The pSOSystem default is the 32-millisecond bus request timeout.

A.7 Motorola MVME167

Directory **$PSS_ROOT/bsps/m167** contains a pSOSystem BSP for the Motorola MVME167 board.

A.7.1 ROM Installation and Hardware Setup

Install the ROMs labeled XU1 and XU2 in the appropriate sockets. The MVME167 ROM size configuration is done by software, so no jumper size selection is necessary.

A transition module (MVME712M) must be connected to the MVME167 and a terminal should be connected to the transition module connector marked SERIAL PORT 1/CONSOLE. The serial protocol is 9600 baud, 8-bit data, 2 stop bits, and no parity. Using a transition module may require a null modem adapter, and this depends on the MVME712M's jumper settings. For an Ethernet interface, a cable from the connector marked ETHERNET on the transition module must be connected to the Ethernet transceiver.

Be sure the system controller jumper is set correctly, with pins 1 and 2 of jumper J1 connected if the board is the system controller and disconnected if the board is not.

A.7.2 Serial Channel Usage

The **m167** BSP supports all four serial channels on the target board. The pSOSystem serial channel numbers correspond to the labels on the MVME712M module. For example, output to pSOSystem serial channel 3 is sent to serial port 3 on the transition module.

A.7.3 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0. The Boot ROMs use RAM from 0x800 through 0x27FFF, so do not attempt to use them to download code to this area. The entry point for downloaded systems is 0x28008 if the operating system and application are linked together. If the operating system and application are linked separately, the entry point for the downloaded operating system is 0x60008.

The MVME167 is available with varying DRAM sizes. The sizes include 4 MB, 8 MB, 16 MB, 32 MB, and 64 MB. In the memory map diagram, *xx* should be replaced by 03, 07, 0F, 1F, or 3F as appropriate.

NVRAM	FFFC'00FF FFFC'0000	
ROM	FF83'DFFF FF82'B000	pNA+ code in ROM
	FF82'AFFF FF81'B000	pROBE+ code in ROM
	FF81'AFFF FF81'7000	pSOS+ code in ROM
	FF81'6FFF FF80'0000	Boot ROM startup and driver code

RAM	0*xx*F'FFFF 0002'8000	Downloaded operating system and application, when linked together. When linked separately, application occupies 0002'8000 - 0005'FFFF and operating system occupies 0006'0000 - 0*xx*F'FFFF
	0002'7FFF 0000'0800	Reserved for use by Boot ROMs
	0000'07FF 0000'0700	Not used by pSOSystem software
	0000'06FF 0000'0580	Reserved for Shared Memory Kernel Interface (SMKI) Directory
	0000'057F 0000'0400	Reserved for Shared Memory Network Interface (SMNI) Directory
	0000'03FF 0000'0000	Vector Page and Node Anchor (Node Anchor is at $44)

A.7.4 Making a pSOSystem Boot ROM

The pSOSystem Boot ROMs for the MVME167 are built using the code in sample application **tftp** and a slightly modified version of the **m167** BSP. You can build a custom Boot ROM set by taking the following steps and then changing the code as needed (the example commands are for a UNIX host system):

1. Copy **PSS_ROOT/apps/tftp** to a working directory:

   cp -r $PSS_ROOT/apps/tftp rombuild

2. Make the working directory the current directory:

   cd rombuild

3. If you are using the pSOS+m kernel instead of the pSOS+ kernel, edit the **sys_conf.h** file, and set SC_PSOS to NO and SC_PSOSM to YES.

4. Enter the following command:

   make rom.hex

The resulting file is **rom.hex**. It is an S-record hex file you can load into a PROM programmer. This file is linked to be located at 0xFF800000, so set the PROM programmer to expect load addresses in the range 0xFF800000 - 0xFF87FFFF. Optionally, you can use a program called **splitrom** to divide **rom.hex** into files corresponding to the two ROMs.

To use **splitrom**, enter the following command:

$PSS_ROOT/bin/$HOST/splitrom rom.hex ws ff800000 128 rom.u1 rom.u2

where *HOST* is the name of the host system (for example, **sunos**, **msdos**, **hpux**, and **aix**).

A.7.5 VMEbus Configuration Information

The following list contains configuration information for VMEbus. The item gives a pSOSystem default, when applicable.

- The System Controller is jumper-enabled.
- The programmable bus arbitration mode is either round-robin or priority. The pSOSystem default is round-robin.
- The pSOSystem default for the programmable bus request level is 3.
- The programmable bus release mode is either release on request or release when done. The pSOSystem default is release when done.
- FAIR bus request mode is enabled.
- The programmable master data bus width is either 32 or 16 bits. The pSOSystem default is restricted to 32 bits.
- Register enabled re-arbitration can occur after a 256-microsecond arbiter timeout. The pSOSystem default is to have re-arbitration enabled.
- The programmable VMEbus global timeout can be 8, 64, or 256 microseconds, or else it is disabled. The pSOSystem default is the 256-microsecond VMEbus global timeout.

- The programmable bus request timeout can be 1, 32, or 64 microseconds, or else disabled. The pSOSystem default is the 32-millisecond timeout.

A.8 Motorola MVME177

Directory **$PSS_ROOT/bsps/m177** contains a pSOSystem BSP for the Motorola MVME177 board.

A.8.1 ROM Installation and Hardware Setup

Install the ROMs labeled XU1 and XU2 in the appropriate sockets. The MVME177 ROM size configuration is done by software, so no jumper size selection is necessary.

A transition module (MVME712M) must be connected to the MVME177 and a terminal should be connected to the transition module connector marked SERIAL PORT 1/CONSOLE. The serial protocol is 9600 baud, 8-bit data, 2 stop bits, and no parity. Using a transition module may require a null modem adapter, and this depends on the MVME712M's jumper settings. For an Ethernet interface, a cable from the connector marked ETHERNET on the transition module must be connected to the Ethernet transceiver.

If pins 1 and 2 are jumpered, the board is the system controller. If no pins are jumpered, the board is not the system controller. If pins 2 and 3 are jumpered, the auto-system controller is enabled and the board automatically becomes the system controller if it is in slot 1.

A.8.2 Serial Channel Usage

The **m177** BSP supports all four serial channels on the target board. The pSOSystem serial channel numbers correspond to the labels on the MVME712M module. For example, output to pSOSystem serial channel 3 goes to Serial Port 3 on the transition module.

A.8.3 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0. The Boot ROMs use RAM from 0x800 through

0x27FFF, so do not attempt to use them to download code to this area. The entry point for downloaded systems is 0x28008 if the operating system and application are linked together. If the operating system and application are linked separately, the entry point for the downloaded operating system is 0x60008.

The MVME177 is available with varying DRAM sizes. The sizes include 4 MB, 8 MB, 16 MB, 32 MB, and 64 MB. In the memory map diagram, *xx* should be replaced by 03, 07, 0F, 1F, or 3F as appropriate.

NVRAM	FFFC'00FF FFFC'0000	
ROM	FF84'1FFF FF82'F000	pNA+ code in ROM
	FF82'EFFF FF81'F000	pROBE+ code in ROM
	FF81'EFFF FF81'B000	pSOS+ code in ROM
	FF81'AFFF FF80'0000	Boot ROM startup and driver code

RAM	0*xx*F'FFFF 0002'8000	Downloaded operating system and application, when linked together. When linked separately, application occupies 0002'8000 - 0005'FFFF and operating system occupies 0006'0000 - 0*xx*F'FFFF
	0002'7FFF 0000'0800	Reserved for use by Boot ROMs
	0000'07FF 0000'0700	Not used by pSOSystem software
	0000'06FF 0000'0580	Reserved for Shared Memory Kernel Interface (SMKI) Directory
	0000'057F 0000'0400	Reserved for Shared Memory Network Interface (SMNI) Directory
	0000'03FF 0000'0000	Vector Page and Node Anchor (Node Anchor is at $44)

A.8.4 Making a pSOSystem Boot ROM

The pSOSystem Boot ROMs for the MVME177 are built using the code in sample application **apps/tftp** and a slightly modified version of the **m177** BSP. You can build a custom Boot ROM set by taking the following steps and then changing the code as needed (the example commands are for a UNIX host system):

1. Copy **PSS_ROOT/apps/tftp** to a working directory:

   cp -r $PSS_ROOT/apps/tftp rombuild

2. Make the working directory the current directory:

   cd rombuild

3. If you are using the pSOS+m kernel instead of the pSOS+ kernel, edit **sys_conf.h** and set SC_PSOS to NO and SC_PSOSM to YES.

4. Enter the following command:

   make rom.hex

The resulting file is **rom.hex**. It is an S-record hex file you can load into a PROM programmer. This file is linked to be located at 0xFF800000, so set the PROM programmer to expect load addresses in the range 0xFF800000 - 0xFF87FFFF. Optionally, you can use a program called **splitrom** to divide **rom.hex** into files corresponding to the two ROMs.

To use **splitrom**, enter the following command:

$PSS_ROOT/bin/$HOST/splitrom rom.hex ws ff800000 256 rom.u1 rom.u2

where *HOST* is the name of the host system (for example, **sunos**, **msdos**, **hpux**, and **aix**).

A.8.5 VMEbus Configuration Information

The following list contains configuration information for VMEbus. The item gives a pSOSystem default, when applicable.

- The System Controller is jumper-enabled.
- The programmable bus arbitration mode is either round-robin or priority. The pSOSystem default is round-robin.
- The pSOSystem default for the programmable bus request level is 3.
- The programmable bus release mode is either release on request or release when done. The pSOSystem default is release when done.
- FAIR bus request mode is enabled.
- The programmable master data bus width is either 32 or 16 bits. The pSOSystem default is restricted to 32 bits.
- Register enabled re-arbitration can occur after a 256 microsecond arbiter timeout. The pSOSystem default is to have re-arbitration enabled.
- The programmable VMEbus global timeout can be 8, 64, or 256 microseconds, or else it is disabled. The pSOSystem default is the 256-microsecond VMEbus global timeout.

- The programmable bus request timeout can be 1, 32, or 64 microseconds, or else disabled. The pSOSystem default is the 32-millisecond timeout.

A.9 Motorola QUADS-68360

Directory **$PSS_ROOT/bsps/e360** contains a pSOSystem BSP for the Motorola QUADS-68360 evaluation board.

A.9.1 Hardware Setup

Connect a terminal to the QUADS DB-9 connector marked “RS-232 PORT P5.” The protocol is 9600 baud, 8 data bits, 1 stop bit, and no parity.

If you will be using Ethernet, a transceiver should be connected to the connector marked “AUI P7.”

A.9.2 Flash Memory Programming

Rather than a pSOSystem Boot ROM set, the **e360** BSP contains an S-record file (**$PSS_ROOT/bsps/e360/flashmem.hex**), which is intended to be programmed into the flash memory of the QUADS board. You can do this by using the Motorola QUICCbug monitor (installed in ROM socket U48) to download **flashmem.hex** over the serial line and program it into flash memory. The following QUICCbug command should be used for downloading the file:

```
lo 400000
```

Next, use the appropriate command for your terminal emulation program to download the **flashmem.hex** file as described in the following section.

A.9.2.1 Downloading and Programming Flash Memory

To download and program flash memory using the UNIX **tip** utility, you will need to download the **flashmem.hex** file in segments. You can do this by entering the following command:

```
split flashmem.hex
```

To program the **flashmem.hex** file into the QUADS board's flash memory, take the following steps:

1. Issue the following command at the Motorola QUICCbug prompt:

   lo 400000

2. Download the **flashmem.hex** file using your terminal software.

   If you are using **tip**, you will probably need to use the following modified procedure:

   a) Download the split pieces of the file from **tip**.

   b) Enter the following characters:

      ~>

      tip prompts as follows:

      ```
      Local file name?
      ```

   c) Enter **xaa** preceded by the pathname. For example:

      ~/apps/bootrom/xaa

      After the file is transferred, the IMPbug sends a series of dots followed by a newline character.

   d) After the newline, send the **xab** file the same way.

   e) Repeat this process until you have sent all files created by the **split** command.

3. After you have downloaded the complete **flashmem.hex** file (or you have downloaded split segments), enter the following command at the QUICCbug prompt:

   lof 480000 4FFFFF 80000

The above procedure programs flash memory with the pROBE+ ROM code.

A.9.2.2 Starting the Boot Monitor Code

Once the above procedure has been completed, you can start the pSOSystem Boot monitor code with the QUICCbug **go** command, as follows:

```
go 80008
```

In the future, Motorola will give QUICCbug the ability to recognize a program in flash memory and automatically start it, thus eliminating the need to enter the **go 80008** command every time the board is reset.

A.9.3 Serial Channel Usage

The pSOSystem BSP for the QUADS-68360 supports only one serial channel, SCC channel 3, which is wired to the connector labelled "RS-232 PORT P5." The RTC requires that the TICKS2SEC value in the **sys_conf.h** file be set to 95, 19, 5, or 1. SC_APP_CONSOLE and SC_PROBE_CONSOLE must be set to 7 for all applications in the **sys_conf.h** file.

A.9.4 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0x400000. The Flash boot code uses RAM from 0x400000 through 0x427FFF, so do not download code to this area. The entry point for downloaded systems is 0x428008, if the operating system and application are linked together. If the operating system and application are linked separately, the entry point for the downloaded operating system is 0x460008.

The following table shows pSOSystem memory usage. The QUADS board can be equipped with 1, 2, 4, or 8 megabytes of DRAM.

DRAM	00BC'FFFF 007E'7FFF 005F'3FFF 004F'A000 0042'8000	8 MB of DRAM 4 MB of DRAM 2 MB of DRAM 1 MB of DRAM For downloaded operating system and application linked together. (When linked separately, application occupies 0042'8000 - 0045'FFFF and operating system occupies 0046'0000 - 00xF'FFFF.)
	0042'7FFF 0040'0600	Reserved for use by Flash Boot Code
	0040'05FF 0040'0400	Unused by pSOSystem software
	0040'03FF 0040'0000	Vector Page and Node Anchor (Node Anchor is at $40'0044)
Flash Memory	000F'FFFF 0008'6000	Not used by pSOSystem software
	0008'5FFF 000A'3000	pNA+ code in flash memory
	000A'2FFF 0009'3000	pROBE+ code in flash memory
	0009'2FFF 0008'F000	pSOS+ code in flash memory
	0008'8FFF 0008'0000	pSOSystem boot code

A.9.5 Making the pSOSystem Flash Boot Code

The file **flashmem.hex** was built using the code in sample application **apps/tftp**. You can customize it by taking the following steps and then changing the code as needed (the example commands are for a UNIX host system):

1. Copy **PSS_ROOT/apps/tftp** to a working directory:

   cp -r $PSS_ROOT/apps/tftp bootcode

2. Make the working directory the current directory:

   cd bootcode

3. Edit **Makefile** in order to set **PSS_BSP** to **$(PSS_ROOT)/bsps/e360**. If this line is commented out, remove the comment mark.

4. Change the following define statements in your **sys_conf.h** file:

   Change SC_APP_CONSOLE from 1 to 7.
   Change SC_PROBE_CONSOLE from 1 to 7.
   Change KC_TICKS2SEC from 100 to 95.

5. Enter the following command:

   make rom.hex

6. Rename the output file to **flashmem.hex** as follows:

   mv rom.hex flashmem.hex

A.9.6 Special Notes

A.9.6.1 Tick Timers

When building an application to run on the QUADS board, you must set KC_TICKS2SEC in **sys_conf.h** to one of the following values:

1, 5, 19, or 95

You also need to make the following changes in your **sys_conf.h** file:

Change SC_APP_CONSOLE from 1 to 7.
Change SC_PROBE_CONSOLE from 1 to 7.
Change KC_TICKS2SEC from 100 to 95.

The BSP for the QUADS board includes a special tick timer initialization routine called **Rtc360Init**, which is called automatically by the pSOS$^+$ kernel during startup. Thus, it is not necessary to initialize the tick timer via a **de_init()** call, as it is on most other boards. However, because all of the pSOSystem sample applications make a **de_init()** call to initialize the

tick timer, the **e360** BSP includes a routine to handle this call as well. The **de_init()** routine for the tick timer does not initialize the tick timer if **Rtc360Init** has already done so.

A.9.6.2

The QUADS board has specific power supply requirements. If these requirements are not satisfied, the Ethernet connection will not work. Here are some hints that may help if your Ethernet does not work:

- The QUADS board has 2 positive voltage requirements: +5 and +12. If you are using two supplies for these voltages, make sure the grounds of both supplies are connected together. Relying on the board to provide a "nonfloating ground" is not sufficient.
- Test the +5 voltage output of your power supply; it must be at least 5 volts with a load. A lower voltage causes the Ethernet connection to fail or work sporadically.

A.10 EST SBC360 Evaluation Board

Directory **$PSS_ROOT/bsps/sbc360** contains a pSOSystem BSP for the Embedded Support Tools Corporation SBC360 evaluation and prototyping board.

A.10.1 Hardware Setup

Install the ROMs labeled U5, U4, U7, and U2 in the appropriate sockets. Set the rotary switch to E. Install the NV RAM Module from the kit into socket U6.

Connect a terminal to the SBC360 connector marked "Phone Jack RS-232" on the Daughter Card. The protocol is 9600 baud, 8 data bits, 1 stop bit, and no parity.

If you will be using Ethernet, a transceiver should be connected to the connector marked "P5" on the Daughter Card.

A.10.2 SCC Channel Usage

The four SCC channels are used as follows for the SBC360:

- Channel 1 is used for the Ethernet controller at the p5 connection on the Daughter Card.
- Channel 2 can be use for a serial connection if the connection is bought out through the expansion connectors.
- Channel 3 is used for the system console at the RS-232 connector on the Daughter Card.
- Channel 4 can be used for a serial connection if the connection is bought out through the expansion connectors.

A.10.3 SMC Channel Usage

The SMC channels are not supported.

The RTC requires that the TICKS2SEC value in the **sys_conf.h** file be set to 95. SC_APP_CONSOLE and SC_PROBE_CONSOLE must be set to 3 for all applications in the **sys_conf.h** file.

A.10.4 Memory Layout

For the purpose of using the tutorials in this manual, on-board RAM begins at address 0x400000. The Flash boot code uses RAM from 0x400000 through 0x427FFF, so do not download code to this area. The entry point for downloaded systems is 0x428008, if the operating system and application are linked together. If the operating system and application are linked separately, the entry point for the downloaded operating system is 0x460008.

The following table shows pSOSystem memory usage. The SBC360 board can be equipped with 2, 4, 8 or 16 Mbytes of DRAM.

DRAM	0197'BFFF 00BC'FFFF 007E'7FFF 005F'3FFF 0042'8000	16 Mbytes of DRAM 8 Mbytes of DRAM 4 Mbytes of DRAM 2 Mbytes of DRAM For downloaded operating system and application linked together. (When linked separately, application occupies 0042'8000 - 0045'FFFF and operating system occupies 0046'0000 - 00xF'FFFF.)
	0042'7FFF 0040'0600	Reserved for use by Boot ROMs
	0040'05FF 0040'0400	Not used by pSOSystem software
	0040'03FF 0040'0000	Vector Page and Node Anchor (Node Anchor is at $40'0044)
NVRAM	100000	
ROM Memory	000F'FFFF 000A'E000	Not used by pSOSystem software
	0003'FFFF 0002'3000	pNA+ code in ROM memory
	0002'2FFF 0001'3000	pROBE+ code in ROM memory
	0001'2FFF 0000'F000	pSOS+ code in ROM memory
	0000'EFFF 0000'0000	pSOSystem boot code

A.10.5 Making the pSOSystem ROM Boot Code

The file **rom.hex** is built using the code in sample application **tftp**. You can customize it by taking the following steps and then changing the code as needed (the example commands are for a UNIX host system):

1. Copy **PSS_ROOT/apps/tftp** to a working directory:

   cp -r $PSS_ROOT/apps/tftp bootcode

2. Make the working directory the current directory:

   cd bootcode

3. Edit **Makefile** in order to set **PSS_BSP** to **$(PSS_ROOT)/bsps/sbc360**. If this line is commented out, remove the comment mark.

4. Enter the following command:

   make rom.hex

5. Use the utility **splitrom** to divide the **rom.hex** file into four parts for the four ROMs by entering the following command:

   $PSS_ROOT/bin/$HOST/splitrom rom.hex b 0 128 rom.u23 rom.u22 rom.u25 rom.u20

A.10.6 Pinout Diagram

The pinout architecture for the Motorola QUADS-68360 evaluation board is shown in Figure A-1 on page A-42 and in Figure A-2 on page A-43.

NOTE: The pinout diagram is a functional example. It is not a final definition of a correct design.

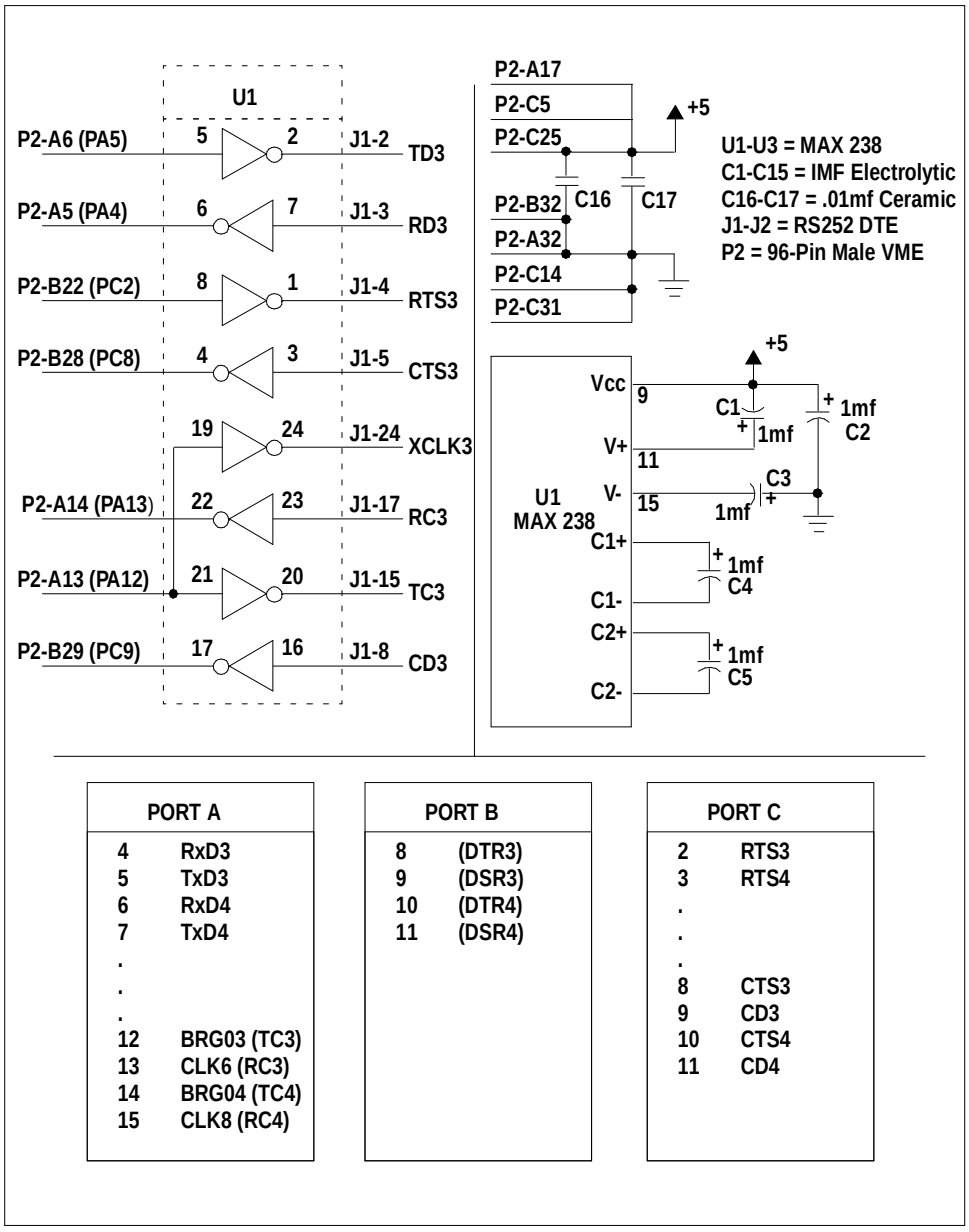


Figure A-1 Pinout for Motorola QUADS-68360 Evaluation Board (Sheet 1)

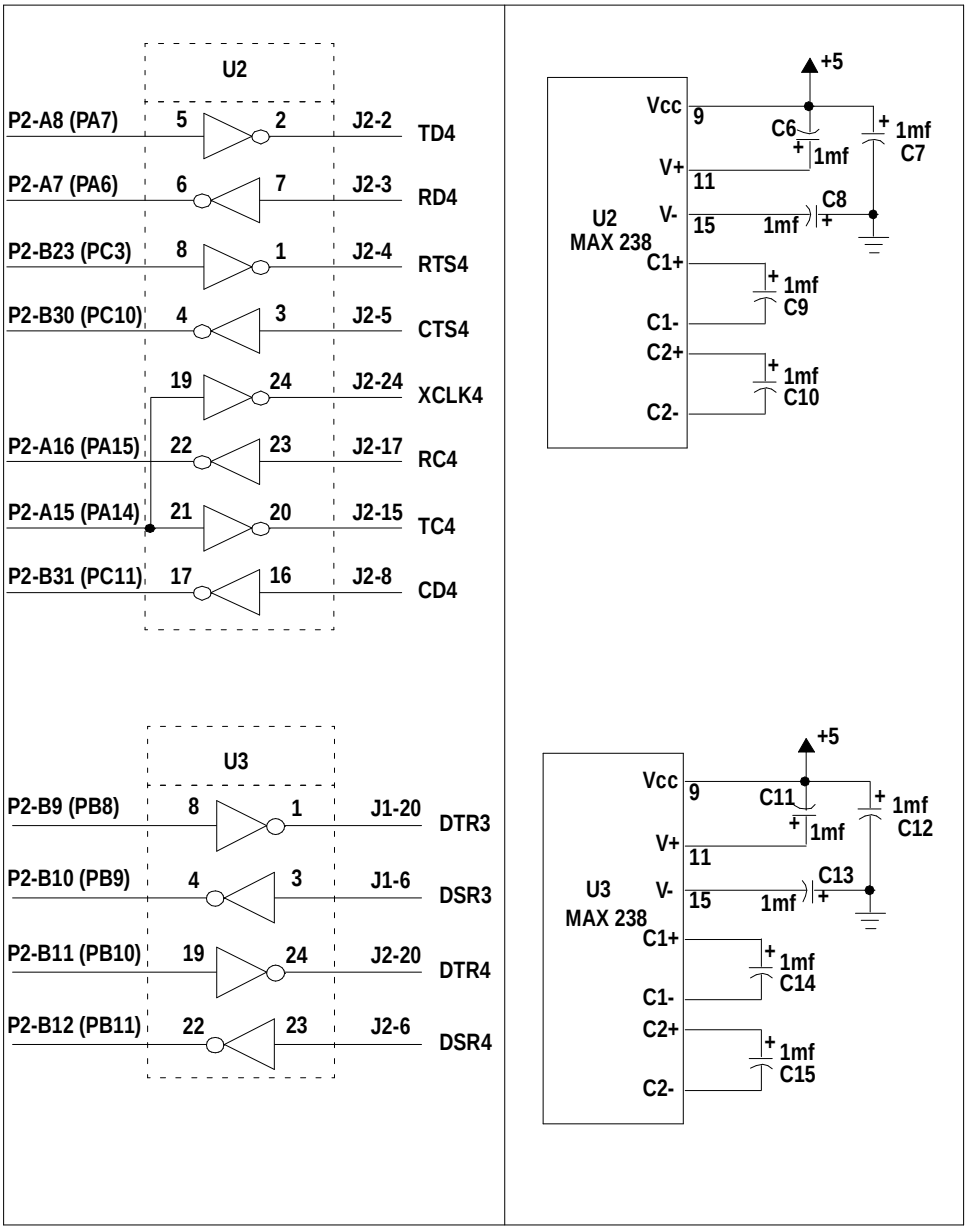


Figure A-2 Pinout for Motorola QUADS-68360 Evaluation Board (Sheet 2)

Glossary

apps	**apps** is the directory that contains a number of sample applications, such as the **hello** program.
anchor	*See* Node Anchor.
BOOTP Client Code	With the BOOTP client code you can send a BOOTP request packet and get necessary information for booting your target.
bootloader	A program that runs automatically at startup. In pSOSystem, this program runs some hardware tests and then passes control to the pROBE+ or the pSOSystem startup program. The bootloader resides in read-only memory (ROM).
Boot ROM	A hardware device that contains the startup code for the target system. This hardware is usually accessed automatically when the system is powered up or reset. It usually contains at least one method to download an executable image to the system.
BSP	Board-Support Package. *See* pSOSystem Board-Support Package.

build	The process that produces an executable image. This usually involves a **makefile** that contains rules to compile, assemble, link, and load files from source code. *See also* makefile.
client	A computer that accesses shared network resources provided by another computer called a server.
Configuration files	Source files that control the configuration of the pSOSystem environment are called configuration files. Configuration files exist for all systems built with the pSOSystem software, and these files are compiled and linked into the executable image. A set of the common configuration files resides in the **PSS_ROOT/configs/std** directory.
Configuration table	A collection of configuration settings for a pSOSystem component stored in parameters in the **sys_conf.h** file.
drv_conf.c	**drv_conf.c** is the driver configuration file. It contains two routines that are called during system startup to install pSOSystem drivers in the appropriate tables.
Ethernet	A local area network (LAN) developed by Xerox in 1976. Ethernet uses a bus topology and relies on the form of access known as CSMA/CD to regulate traffic on the main communication line. Network nodes are connected by coaxial cable or by twisted-pair wiring.
Executable image	The pSOSystem software and the application code are linked together on the host system. This combination is then downloaded to the target. The downloaded software is called the executable image.
hello	**hello** is the sample application program that prints a short message and exits. This program resembles the Hello World program familiar to most programmers.

Kernel Interface (KI)	A software layer that provides a set of standard services for the pSOS+m kernel. The kernel uses these services to transmit and receive packets when communicating with other kernels on other nodes to form a multiprocessor independent system. This layer makes pSOS+m independent of the actual transfer medium.
sys.lib	pSOSystem system library of operating system components and run-time bindings the application uses to make system calls to the components. It can be found in the **PSS_ROOT/sys/os/** directory.
makefile	A data file used by the **make** utility within pSOSystem to produce a library or executable image that can be downloaded to a target. The **makefile** contains rules that control the execution of programming tools such as compilers, assemblers, linkers, and loaders.
mpdemo	**mpdemo** is the pSOS+m sample multiprocessing application.
msgarray	The **msgarray** structure is an array of messages. The program code sets up the array as a circular linked list.
multithreading	The running of several processes in rapid sequence (multitasking) within a single program.
Network Interface (NI)	A software layer that provides a set of standard services to pNA+. The NI services are independent of the transmission medium.
Node Anchor	The Node Anchor is the single fixed point of reference for all the installed software components in the pSOSystem software. The anchor is a critical link because each component in the pSOSystem environment is code and data position-independent and depends on the anchor to locate its configuration information.

pHILE+ File System Manager	pHILE+ is the file management component of the pSOSystem software. It offers a superset of the UNIX file system capabilities. The pHILE+ internal organization and operation have been carefully crafted to provide an unsurpassed level of performance, integrity, and flexibility.
pNA+	pNA+ is the networking component of the pSOSystem software that provides TCP/IP capabilities.
pREPC+	pREPC+ is a component of the pSOSystem. It contains more than 85 run-time functions that comply with the ANSI C standard library, including character and string handling functions, general utilities, and I/O functions such as `printf` and `scanf`.
pROBE+	The pROBE+ debugger is a comprehensive system debugger and analyzer for the pSOSystem environment. Co-resident with the kernel on the target system, the pROBE+ debugger uses detailed structural information about the pSOS+ kernel to let you observe and control system execution.
pRPC+	pRPC+ is the remote procedure call subcomponent of the pNA+ component of the pSOSystem software. It is a block of code that extends the pNA+ feature set, and relies on the pNA+ and other components for resources and services.
pSOS+m	pSOS+m is the pSOSystem multitasking kernel which allows tasks running on different processors to communicate, exchange data, and synchronize as if they were running on a single processor.
pSOSystem	pSOSystem is an operating system used on embedded controllers. Its code consists of read-only *object* libraries, *include* files, and *source* files.
pSOSystem board-support package (BSP)	BSP is the hardware specific code in the pSOSystem software and is contained in the **PSS_ROOT/bsps/** directory.

pSOSystem Boot ROMs	Boot ROMs are the ROM monitors that come with the pSOSystem software. The Boot ROM set is actually a pSOSystem executable image downloaded to ROM.
pSOSystem directory tree	The pSOSystem directory tree is the central location on the host system. It contains the shared pSOSystem code so that multiple users can have access to it.
pSOSystem environment	The pSOSystem environment provides a standard set of services for the application code. It usually contains the pSOS+ kernel, along with the following companion software elements: pROBE+, pNA+, pHILE+, device drivers, interrupt handlers, and configuration tables to customize the pSOSystem environment for a particular target system.
PSS_BSP	An environment variable used by pSOSystem during the build process to point to the directory containing the Board-Support Package needed by the application.
PSS_ROOT	An environment variable used by pSOSystem during the build process to point to the directory containing the pSOSystem root directory.
ram.hex	An executable image in S-record format for Motorola processors or Intel Extended Hexadecimal format for Intel processors, suitable to download to the target board's RAM.
ram.map	**ram.map** is the map file.
ram.x	An executable image in IEEE-695 format, suitable to load to the target board's RAM with XRAY for pSOSystem.
readme file	This file contains the instructions for using the sample applications in the **apps** directory.
SDM	*See* System Debug Mode.
Service breakpoints	Service breakpoints stop execution whena pSOS+ service call is made.

Shared Memory Kernel Interface (SMKI)	A kernel interface that uses shared memory as a communication medium. *See also* KI.
Shared Memory Network Interface (SMNI)	A software layer used by the pSOS+m kernel when communicating with other kernels. The SMKI directory structure must reside in a memory space where all of the nodes in the system have access to it.
single threading	Within a program running a single process at a time.
SMKI	*See* Shared Memory Kernel Interface.
start address	Start address is the address used by the host to pass control to the target for starting the executable image. By default, the start address is the image load address plus 4.
System Debug Mode	A mode of debugging where the entire system executes and halts together. Both tasks and interrupt service routines (ISRs) can be debugged in System Debug Mode.
sys_conf.h	**sys_conf.h** is the pSOSystem configuration file. It is a C include file that must reside in the working directory.
target system	The embedded computer is called the target system. Two major software elements run on the target hardware: the pSOSystem software and the application code. The application code is what makes one target system different from another.
Working Directory	Working directory is a directory where you build a pSOSystem executable image. You can locate your working directory under **PSS_ROOT**.
XRAY Debugger for pSOSystem	XRAY is a multitasking debugger that can be used to control and monitor a multitasking application in the high-level mode and the assembly-language mode. In addition to supporting System Debug Mode, some versions of XRAY also support Task Debug Mode.

xraydemo	The **xraydemo** is a sample application. It is a C program that contains the code for the ROOT task which in turn creates seven other tasks: MEM1, MEM2, 'IO1•', 'IO2•', SRCE, SINK, and 'MSG•'.
XRAYLIB	**XRAYLIB** is an environment variable used to specify the directories the XRAY debugger should search when looking for the **startup.xry** file.

Index

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Document Title: pSOSystem Getting Started:
68K Processors MRI Release

Part Number: 000-5001-004

Revision Date: March 1996