OS-9® Porting Guide

Version 4.7

RadiSys
THE POWER OF WE

www.radisys.com

Revision A • July 2006

Copyright and publication information

This manual reflects version 4.7 of Microware OS-9.
Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

Contents

1 Porting Overview

This chapter walks you through the process of porting OS-9® to custom hardware. The following sections are included:

Porting Summary

The OS-9 manuals use the following terms:

host is the development system used to edit and re-compile OS-9 source files

target is the reference board or system to which you intend to port OS-9

The OS-9 operating system includes the OS-9 kernel, `init` module, ticker, real time clock, I/O manager, file managers, device drivers, device descriptors, utilities, and other system modules.

Before porting to your target, complete the following steps:

Step 1. Obtain all the documentation that came with your board.

Determine the following information:

- the number of communication ports available on the target and host

  To complete installation of OS-9, you will most likely need one serial port for console communication and either one serial or one Ethernet port for debugging communications.

- the tickers available on the target

  You need one high-level, countdown ticker for time-slicing. You need a second ticker for low-level timing if you are using Hawk™ user-state debugging on the running system.

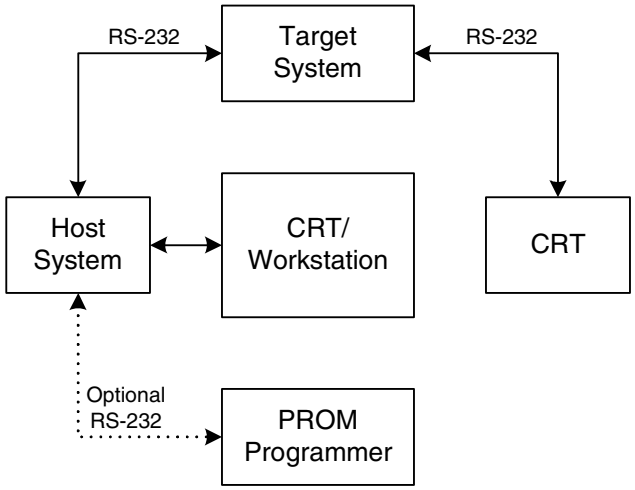
Step 2. Test and verify your hardware. You will need the following components:

- a target board
- communication cables
- a power supply cord
- hardware debugging software

Test your hardware before beginning the porting process; this avoids the need to simultaneously debug the hardware and software. While debugging your hardware, determine if there are board hardware features to help you in the debugging process.

Figure 1-1 shows a typical host and target interconnection.

Figure 1-1. Example Host and Target Interconnection



Use 9600 baud or the highest possible data rate for RS-232 links to maximize download speed. The default is 9600 baud.
The X-On/X-Off protocol is used for flow control.

Refer to the OS-9 board guide associated with your target processor for other examples about setting up hardware.

Step 3. Install the Microware software distribution onto the host system by following the installation instructions included with your Microware software distribution media. The distribution media contains all of the files that comprise the OS-9 boot code, operating system, and related utilities.

Some files are source code text files. Most of the other files are makefiles and object code files. The files are organized into subdirectories according to major subsystems (such as ROM, IO, and CMDS) in a master directory known as the MWOS structure.

During the installation process, the file system is copied into the MWOS directory structure. You need to use a hard disk based system with sufficient storage capacity to contain the entire file system.

The files in the distribution package assume this specific file and directory organization. They do not compile and link correctly if the organization is different.

The file name suffixes shown below are used to identify file types:

Table 1-1. File Name Suffixes

Suffix	Definition
`.a`	Assembly language source code.
`.c`	C language source code.
`.cc`	C++ language source code.
`.d`	Definitions (`defs`) source code (for assembly).

Table 1-1. File Name Suffixes (Continued)

Suffix	Definition
.des	EditMod description files.
.edm	Editmod generated C header file.
.h	C header file source code.
.i	RadiSys intermediate code (I-code) files.
.il	RadiSys intermediate code libraries.
.l	Library files.
.m	Macro files.
.ml	Module list files, including coreboot.ml and bootfile.ml, which create the boot images.
.o	Assembly language source from the compiler back end.
.r	Relocatable object code (for linker input), created by the assembler.
.tpl	Makefile templates.
none	Object (binary) files.

In general, OS-9 does not require file name suffixes. However, certain utilities, such as µMACS and cc or xcc, require file name suffixes to determine the mode of operation.

Porting Steps

This section discusses the various phases in the porting process. Before you begin this section, you should have completed the pre-porting steps.

If you want more details about OS-9, the modules involved in the porting process, or what occurs in OS-9 during the booting process, refer to the OS-9 Boot Code section.

Phase I: Prepare a Port Directory

To prepare the port directory, complete the following steps:

Step 1. Create a port directory for your board in the following directory:

```
<MWOS>/OS9000/<CPU Family>/PORTS
```

where <MWOS> is the tree in which you have installed your OS-9 product(s) and <CPU Family> is the name of the CPU family to which the CPU on your board belongs.

Refer to *Chapter 2, Port Directories* for a full description of the MWOS tree and the supported CPU directories.

Step 2. Create a systype.h file by copying it from one of the example ports directories into your working port directory. This example systype.h file contains comments and structure that you will use, along with the explanation in *Chapter 3, Porting the Boot Code*, to fully define the board specific definitions used throughout the porting process.

Phase II - Create the Low-Level System

Step 3. Copy the bootstrap code sources from one of the example directories into your port directory and modify for the memory layout of your board. Write customized startup code to initialize your board's memory and devices. *Chapter 3, Porting the Boot Code* walks you through this process.

Step 4. Create a low-level serial driver appropriate for your board's serial device, using the one of the example sources, along with one of the drivers included in the OS-9 for Embedded Systems source library (optional).

This low-level serial driver provides the basic I/O service to the serial hardware for displaying the OS-9 bootstrap message, and resident RomBug debugging. *Chapter 4, Creating Low-Level Serial I/O Modules* discusses the steps required to provide serial support for the boot code.

This overview assumes that you have a serial device on your target board.

Phase III - Set Up Hawk System-State Debugging (Optional)

If you want to use `sndp` or Hawk system-state debugging instead of RomBug for the remainder of your port, proceed directly to step five. If you would rather continue using RomBug for system-state debugging, proceed directly to step eight.

Step 5. Create a second serial port or an Ethernet port driver to use as the communications link for debugging.

- To create a low-level serial driver, refer to *Chapter 4, Creating Low-Level Serial I/O Modules*.
- To create a low-level Ethernet driver, refer to *Chapter 5, Creating a Low-Level Ethernet Driver*.

Step 6. Create a low-level timer module to support Hawk debugging communications. *Chapter 6, Creating a Low-Level Timer Module* discusses this issue in detail.

Step 7. Configure and test Hawk by including the following components in the boot module list and verifying the Hawk connection:

- the Hawk support modules
- the low-level serial or Ethernet driver
- the low-level timer

Refer to the *Using Hawk* manual for information on configuring Hawk.

Step 8. Create an initial Init module and boot image with `shell` as the first executable process and `term` as the system console for debugging purposes.

Step 9. (Optional) Create a PIC driver for each programmable interrupt controller on your board, if your board uses programmable interrupt controllers. Create a library of calls that access your PIC(s) to provide a transparent way for drivers to enable/disable interrupts on your board.

Refer to *Chapter 8, Creating PIC Controllers* for detailed information on creating PIC controllers.

Step 10. Write a high-level serial driver for use as your system console.

If you complete the previous steps, you have completed a port to your target board. The OS-9 shell should run on your target board as a single-tasking operating system. Complete the following step to add multi-tasking and time-slicing to the basic port.

Step 11. Create a system ticker to enable time-slicing and multi-processing.

Phase V - Adding Features to the Basic Port

Step 12. Perform any additional porting steps, including those listed below:

1. Create high-level drivers for other serial ports, clocks, and any other available devices.
2. Create high-level drivers for disk devices. Once the basic port of a board has been completed (the first two port procedures), a high-level driver for a floppy drive (or other device) can be developed.

   Once you know this driver works, you can format a floppy disk and install an OS-9 bootfile on the floppy. At this point, you can create the low-level driver (borrowing heavily from the tested code of the high-level driver) to boot the system from the floppy disk.
3. Create low-level drivers and port-specific booters to boot from the various devices available on the target.

OS-9 Boot Code

The process of booting OS-9 requires an OS-9 bootfile and boot code that initializes the system hardware, locates the OS-9 bootfile, and passes control to the OS-9 kernel.

The bootfile is a collection of the OS-9 system modules merged together into a single image, with the kernel appearing as the first module. This bootfile can exist in ROM, RAM, or flash memory. On a disk-based system, the bootfile is on the boot disk device. Tape devices can also be used as boot devices, with the bootfile on magnetic tape.

The boot code for OS-9 contains the raw machine-code bootstrap routine and a collection of separately linked but inter-dependent modules, organized as OS-9 extension modules. These modules compose the low-level system required to boot the system and provide debugging on the target.

Each low-level system module provides one or more services that may be required for a particular target. By compiling these services into separate, configurable modules, the low-level system can be rich and flexible without inflating the memory requirements for the core bootstrap code. You can build a minimal system by including only the low-level system modules required for booting.

The file `boot.c` in `<MWOS>\OS9000\SRC\ROM\` contains the following macro:

```
#define BOOTSTRAP_EDITION 62
```

In this example, the number "62" corresponds to the last entry in the edition history of the low-level boot for OS9000. The edition # will be incremented in future OS releases if/when changes are made to the bootstrap code.

Bootstrap Code (romcore)

The bootstrap code is made from a number of different files that are compiled and linked together to produce the final binary object code, `romcore`. Some of the code is not target platform-specific and is supplied in intermediate code form (files with `.i` or `.il` suffixes) or relocatable object code form (files with `.r` or `.l` suffixes).

To create the bootstrap code, you need to edit some of the source files. Next, you need to use the `make` command (`os9make` on Windows) to compile and link these files with the other intermediate and relocatable files to create the `romcore` binary image file.

The bootcode follows these steps to boot OS-9:

1. Initialize the basic CPU hardware and devices to a known, stable state.
2. Locate and initialize each boot module to make all boot services available.
3. Determine the location and extent of the target's RAM and ROM memory.
4. Call a system debugger if one is configured.
5. Call the configured system booter module to find the OS-9 bootfile.
6. Transfer control to the OS-9 kernel.

Low-Level System Modules

The `romcore` bootstrap image is merged with several low-level system modules to produce the final boot image to be burned into PROM, or loaded into RAM, NVRAM, or flash memory, prior to booting the target system.

`romcore` is the only part of the system that is not a module.

Because some of the low-level system modules provide services, they are supplied as linked memory modules in binary form. For some modules, both target-independent binary modules and source code are provided so you can make target specific changes. You should use target-independent modules for your initial port of OS-9. As more of the port is accomplished, these modules can be rebuilt to more directly target your system.

For the initial port, you need to ensure that low-level serial driver modules exist to handle the console I/O port and an auxiliary communications port. You may be able to use the example drivers for the common serial devices directly. If not, the example source code provides a guide for creating your own driver.

If you plan to use Hawk tools for downloading and remote system-state debugging, you need to ensure an appropriate low-level network driver is available. A low-level SLIP driver was provided for use with your serial port. In addition, example drivers are provided for some Ethernet devices. You use these drives directly or modify them to support your network device.

Configuration Modules

You can use the configuration modules to configure the boot system. These modules provide a way for other low-level system modules to retrieve configuration parameters describing how they should function. The low-level system modules are soft-coded to use the configured values retrieved by calling the configuration module services.

`cnfgdata` is a target-specific data module containing the configuration parameters.

The definitions of these parameters are set in the `systype.h`, `default.des` (where applicable), and `config.des` files.

While all the other low-level system modules are organized as OS-9 extension modules, `cnfgdata` is an OS-9 data module.

`cnfgfunc` is a target-independent module that retrieves configuration parameters from the `cnfgdata` boot data module.

This module could be modified to return target-specific overrides of the default information in `cnfgdata`. For example, you can override `cnfgdata` values with NVRAM or switch/jumper settings.

Boot Modules

These are the modules responsible for selecting the appropriate system boot routine and using it to locate the OS-9 bootfile from the appropriate device.

`bootsys`

is a target-independent module providing two services: a booter registration routine and a booter selection/execution routine.

The registration routine installs device specific booter modules onto a list of available booters as either an auto-booter or menu-booter.

The booter selection/execution routine is called as part of the OS-9 booting process. It either selects the appropriate auto-booter or prompts you to choose a booter from the registered menu-booters to use for booting the system. Next, it calls that booter to retrieve the OS-9 bootfile, passing parameters you enter and any defaults found for the booter in the `cnfgdata` module.

`portmenu`
: is target-independent module that retrieves a list of names of configured auto and menu booters from the configuration data module.

  `portmenu` checks each named booter against the list of available booters and, if found, registers it through the `bootsys` registration service.

`<booter>`
: includes any of the port specific booter modules capable of locating and loading the OS-9 bootfile from its target device.

  During initialization, each booter installs itself onto available booters.

`override`
: is a target-independent booter module that enables override of autobooter.

  If the space bar is pressed within three seconds after the bootstrap message displays, a boot menu is displayed. Otherwise, booting proceeds with the first autobooter.

`srecord`
: is a target-independent booter module that receives a Motorola S-record format file from the communications port and loads it into memory.

`flashb`
: is a target-independent booter support module that assists in reprogramming flash memory.

  `flashb` relocates the console, downloader, and flash programming modules from flash memory to RAM. This enables a new booter to overwrite that flash memory location. `flashb` calls the flash-specific module to program each sector, and optionally, calls a downloader module to read data for programming into flash memory.

`romboot`
: is a target-independent booter module that locates the OS-9 bootfile in the special memory list.

  Like all booters, `romboot` installs itself on the list of available booters when initialized.

`restart`
: is a target-independent booter module that restarts the boot process, if called.

`rombreak`
: is a target-independent pseudo-booter meant to drop the system into the configured system-state debugger.

`parser`
: is a target-independent booter support module providing argument-value pair parsing services.

`fdman`
: is a target-independent booter support module providing general booting services for RBF file systems.

`pcman`
: is a target-independent booter support module providing general booting services for PCF file systems (PC FAT file systems).

`scsiman`
: is a target-independent booter support module providing general SCSI command protocol services.

`<low-level SCSI module>`
: is a target-specific booter support module providing SCSI host-adaptor access services.

`IDE`
: is a target-specific standard IDE support including PCMCIA ATA PC cards.

`FDC765`
: provides PC style floppy support.

Serial Communication Modules

Two serial ports are used by the low-level system. The system console displays boot status messages, error messages, boot menus, and debugger messages from the target-resident debugger. The auxiliary communications port is a download port for communicating with a host system.

`console`
: is a target-independent module that provides high-level I/O hooks into the low-level entry points of the console serial driver.

  The available functions include `getchar()`, `getc()`, `putchar()`, `putc()`, `gets()`, and `puts()`.

`conscnfg`
: is a target-independent module that retrieves the name of the low-level driver to use for the console from the configuration data module.

  After finding the driver on a list of available drivers, `conscnfg` installs it as the console serial driver. You can modify this module to perform target-specific console configuration instead of using a `cnfgdata` module.

`commcnfg`
: is a target-independent module that retrieves the name of the low-level driver to use for the auxiliary communication port from the configuration module.

  After finding the driver on the list of available drivers, `commcnfg` initializes it as the communication serial driver. You could modify this module to perform target-specific communications port configuration instead of using a `cnfgdata` module.

`io<serial>`
: includes any of the target-specific low-level serial drivers.

  The low-level serial driver services include device initialization and de-initialization, read a byte, write a byte, and get status. Each low-level serial driver will, during module initialization, install itself on a list of available serial drivers.

`iovcons`
: is a low-level virtual console driver that is hardware independent because it transfers I/O requests to the low-level network modules (TCP/IP stack).

  `iovcons` provides a `telnetd`-like interface to the low-level system console. You can use the `telnet` command to link to the target processor board to obtain a TCP/IP connection over which the OS-9 boot messages and RomBug I/O occurs. This removes the need for a direct serial connection to the target by providing a remote console.

Low-Level Network I/O Modules

`protoman`
: is a target-independent protocol module manager.

  This module provides the initial communication entry points into the protocol module stack.

`lltcp`
: is a target-independent low-level transmission control protocol module.

`llip`
: is a target-independent low-level internet protocol module.

`llslip`
: is a target-independent low-level serial line internet protocol module.

  This module uses the auxiliary communications port driver to perform serial I/O.

`lludp`
: is a target-independent low-level user datagram protocol module.

`llbootp`
: is a target-independent low-level BOOTP protocol booter module.

`ll<ether>`
: is a target-specific low-level Ethernet driver module.

`hlproto`
: is a high-level hook into the protocol manager of the low-level system.

  This module is used when the low-level system is to be used for user-state debugging through FasTrak.

Timer Modules

The timer modules are port specific modules that use some counter/timer device of the target to provide a polling time-out mechanism for other low-level system modules. The services provided are listed below:

- Initialization: Perform any required timer initialization.
- De-initialization: Deinitialize timer.
- Set time-out value: Set a time-out value from the time of the call.
- Get time-out value: Get the time remaining until the time-out expires.

Debugger Modules

The OS-9 configuration provides for either target-resident or remote system-state debugging, depending on the debugging method and tool you select.

`dbgentry`
: is a target-independent module that provides a hook from the boot code and OS-9 kernel's `_os_sysdbg()` system call to the low-level debug server.

`dbgentry` must be present in the low-level system for debugging capability.

`dbgserv`
: is a target-independent debug server module.

The debug server contains services providing the following debugging facilities:

- monitoring exception vectors
- setting breakpoints
- setting watchpoints
- executing at full speed (until it encounters a breakpoint, watchpoint, or exception)
- tracing by single instruction
- tracing by multiple instructions

The debug server must also be present in the low-level system if any system-state debugging is required prior to the OS-9 kernel being executed.

`usedebug`
: is a target-independent module that retrieves the flag from the configuration data module indicating whether the debugger is called during system startup.

You can modify this module to perform target-specific debugger configuration instead of using a `cnfgdata` module.

`RomBug`
: is a target-independent debugger client module that provides interactive, target-resident debugging using the serial console device for the user interface.

`RomBug` uses the I/O services available through the `console` module to read commands and display output, and uses the services of `dbgserv` to perform the required debugging tasks.

The use of `RomBug` requires a low-level serial device to be available as the system console.

`sndp`
: is a target-independent system-state network debugging protocol module.

This module acts as a debugging client on the target, invoking the services of `dbgserv` to perform debug tasks. Its user interface, however, is a low-level network connection to a Hawk client on the development host. That is, `sndp` is viewed as a debug server from the standpoint of the remote, host-resident Hawk debugger.

The use of `sndp` requires the appropriate low-level network driver and protocol modules for the communication link.

Notification Module

Hawk relies on the low-level communication modules and a network driver for remote system-state debugging both before and after OS-9 is up and running. Once the OS-9 system has booted, you can use either high-level networking drivers and protocols (SPF, for example) or low-level communications to perform remote user-state debugging on the target. The high-level drivers and protocols do not use the same communications path as the low-level communications. Regardless of the communications path, if the system drops into system-state the low-level drivers/protocols must be used to communicate with the host.

Some low-level system modules require that they be informed when a transition takes place between high and low-level states in order to do special maintenance. The `notify` module provides the following services:

Table 1-2. notify Module Services

Service	Description
Registration	includes any low-level system module requiring notification of a state change can call `notify`. The calling module passes the address of a routine to be called in the event of such a state change, and the registration routine includes it on a list of such routines to be called.
De-registration	a low-level system module can call `notify` to cause its routine to be removed from the list of routines to be called in the event of a state change.
Notification	is the debugger calls `notify` when a state change takes place. `notify` passes over its list of routines requiring notification, and calls each in turn.

Miscellaneous

`flshcache`
: is a target-specific boot module that provides cache flushing routines appropriate for the target hardware.

Low-Level System Configuration

For each example target platform, the file `coreboot.ml` contains a list of the low-level system modules along with `romcore` to create the boot image. For your initial port, use the configuration given in the example ports. You will need to change the `coreboot.ml` file to use the appropriate low-level serial device drivers for your console and communications ports, and the appropriate booters and low-level communications drivers that apply to your target.

You may also want to replace the target-resident `RomBug` debugger with the modules appropriate for use with `sndp` and the remote Hawk Debugger.

OS-9 Boot Process

The booting process occurs in three phases, and are similar to the steps you take in porting OS-9. The following sections provide background information on porting and the phases of the boot process.

Apply Power to the Debugger Prompt

When power is supplied to the processor, or when a reset occurs, the processor begins executing from a fixed address. The initial value in the OS-9 boot code is a label, `cold`:. This label is defined in the bootstrap source code file `btfuncs.a`.

Once `btfuncs.a` starts executing, it performs the following tasks:

1. Branch to the label `sysinit`: in the source file `sysinit.c`. `sysinit` initializes any port specific hardware devices and then branches back to the label `sysreturn` in `btfuncs.a`.
2. Initialize the stack pointer. This relies on the memory lists defined in the bootstrap source file `rom_cnfg.h` to determine the first available RAM memory area, as well as the top-of-stack offset into it.

   Refer to *Chapter 3, Porting the Boot Code* for information on creating the `sysinit.c` file and the `rom_cnfg.h` header file for your port.

3. Call the `sysinit1()` routine in `sysinit.c`. The `sysinit1()` routine completes the initialization of target-specific hardware devices. Before returning control back to `btfuncs.a`, it calls `rompak1()` to determine if an `initext` module is present for further hardware initialization.
4. Initialize the bootstrap global data pointer and stack pointer. This relies on the memory lists defined in the bootstrap source file `rom_cnfg.h` to determine the first available RAM memory area.
5. Initialize the bootstrap global data. The `callidata()` routine in `p2privte.l` is called to initialize the global data for the bootstrap code.
6. Transfer control to `hard_reset()` in the `boot.c` source file.

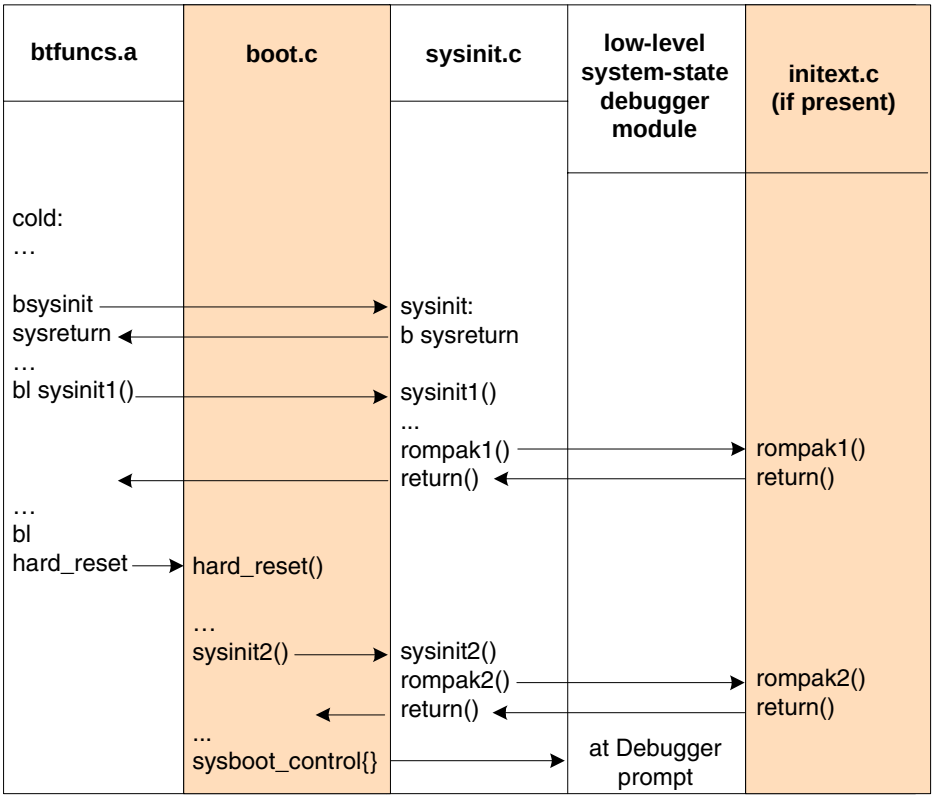
   If control is returned, which only happens if it is impossible to boot the system, control is transferred back to the `cold`: label, and the process repeats.

When `boot.c` gets control in `hard_reset()` it performs the following tasks:

1. Initialize the vector table for the processor. This is done through a call to the `initvects()` routine in the `cbtfuncs.c` file.
2. Determine the processor type and floating point unit (fpu) type. These are calls to `getfpu()` and `getcpu()` in `btfuncs.a`.
3. Search for and initializes the low-level system modules through a call to `rominfo_control()` in `romsys.l`. The `rominfo` record structure is initialized, then the memory immediately following the bootstrap code is searched for valid, contiguous low-level system modules, and each one that is found is initialized. During initialization, the low-level system modules add tables and pointers to their services onto the `rominfo` record structure.

4. Perform RAM and special memory searches, and if needed, enable memory parity checking. The memory search routines use both bus errors and pattern matching to determine the sizes of valid RAM and ROM memory segments available on the system. This relies on the memory list defined in `rom_cnfg.h` to determine the memory areas to search.
5. Insert the bootstrap global data area and stack area into the consumed memory list.
6. Call the `sysinit2()` routine in `sysinit.c`. The `sysinit2()` routine performs target-specific initializations that rely on completion of the previous steps. There may not be any, but before `sysinit2()` returns, it calls `rompak2()` to determine if an `initext` module is present for further target-specific initialization.
7. Initiate the configured low-level debugger by calling the `sysboot_control()` routine from `romsys.l`. If a low-level debugger is configured, enabled, and available, it is called at this point by the `sysboot_control()` function. The debugger displays a processor register display, and a prompt. The major steps of this phase are shown in Figure 1-2.

Figure 1-2. Files and Subroutines



Debugger Prompt to the Kernel Entry Point

On return from the debugger (once you have requested booting be continued) the bootstrap code performs the following tasks:

1. Call the boot system to find the OS-9 bootfile. `sysboot_control()` invokes the boot service provided by the `bootsys` module to oversee the location of the OS-9 bootfile by the configured booter(s). This boot service calls each registered auto-booter in turn until one is successful in locating a valid OS-9 bootfile. If there are no auto-booters, or if all fail to find a bootfile, you are presented with a menu listing of all registered menu-booters and prompted to select one. The specified booter is called and the process is repeated until a selected booter is successful in locating an OS-9 bootfile.
2. Transfer control to the OS-9 kernel. The coldstart entry point of the kernel module is calculated and control is transferred to the kernel for completion of the boot.

Kernel Entry Point to the Shell Prompt

The kernel's coldstart routine finishes the task of booting OS-9. It reads the OS-9 configuration module, `init`, and using the system configuration data stored within the kernel, performs the following tasks:

1. Initialize system global data (commonly referred to as the system globals).
2. Add the colored memory list to the memory lists found by the bootstrap code.
3. Build the kernel's RAM memory from the RAM memory list.
4. Build the module directory by searching for modules in the special memory list.
5. Execute all configured extension modules from the `PREIO` extensions list.
6. Initialize system data tables such as the path table and process table.
7. Open the system console.
8. Change directories to the system device.
9. Execute all configured extension modules from the `EXTENS` extension list.
10. Create the first process to be executed.
11. Transfer control to the system execution loop to begin process scheduling.

The OS-9 system is now booted and executing as expected.

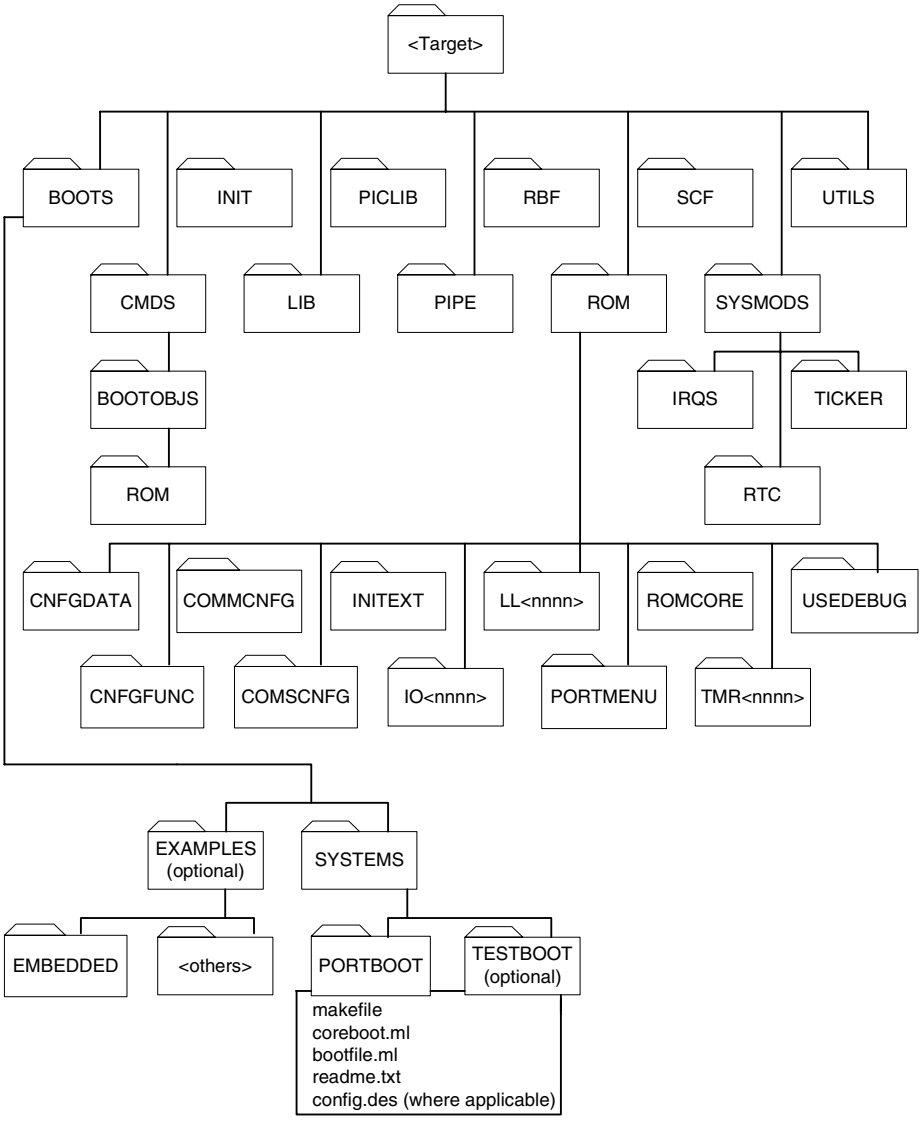
2 Port Directories

This chapter includes the following topics:

Ports Directory Structure

The following figure shows only the directories referred to in this guide. The MWOS structure includes other directories and files.

Figure 2-1. <MWOS>/OS9000 Porting Directories and Files



Creating Target Port Directories

The OS-9 boot code sources, driver sources, and system modules (such as the kernel) consist of many files when installed on your system.

Example source files for several different types of device drivers are provided, including serial, tickers, and real-time clocks. You only need support for the hardware platform your target has available.

Step 1. Answer the following questions about your hardware before beginning the porting procedure:

- What I/O devices will you use?
- How are these devices mapped into memory?
- How is the memory organized?
- What does the memory map of the entire system look like?

Step 2. Create your own working directory structure in which to design and build your port. Start by creating a subdirectory in `MWOS/OS9000/<CPU Family>/PORTS`. (`<CPU Family>` is a specific processor family directory like PPC or 80386.) This is the root of your target platform's directory structure. If your target platform is based on a processor for which there already exists a processor-specific ports directory, then your target directory can be created there instead. For example, if your target system is built on a PowerPC 603 CPU, you could choose to develop your port in `MWOS/OS9000/603/PORTS`.

Step 3. Create the necessary directories for your target and copy the following files from the corresponding directories in on the example ports as a starting point. Each target port directory structure is somewhat different depending upon the configuration of the target platform.

- `BOOTS/SYSTEMS/PORTBOOT`
- `CMDS/BOOTOBJS/ROM`
- `ROM/CONSCNFG/makefile`
- `ROM/COMMCNFG/makefile`
- `ROM/PORTMENU/makefile`
- `ROM/USEDEBUG/makefile`
- `ROM/ROMCORE/RELS`
- `ROM/ROMCORE/makefile`
- `ROM/makefile`

The `BOOTS/SYSTEMS/PORTBOOT/coreboot.ml` file contains the list of names of modules to be merged with `rom` when building the boot image.

The makefile in `ROM` invokes the makefiles in each of its appropriate subdirectories to build the bootstrap code and low-level system modules. Some of the subdirectories are disabled by default. For the initial target port, uncomment the values for the `CONSCNFG`, `COMMCNFG`, `PORTMENU`, and `USEDEBUG` macros.

Once this target port directory structure is in place, the bootstrap code can be ported.

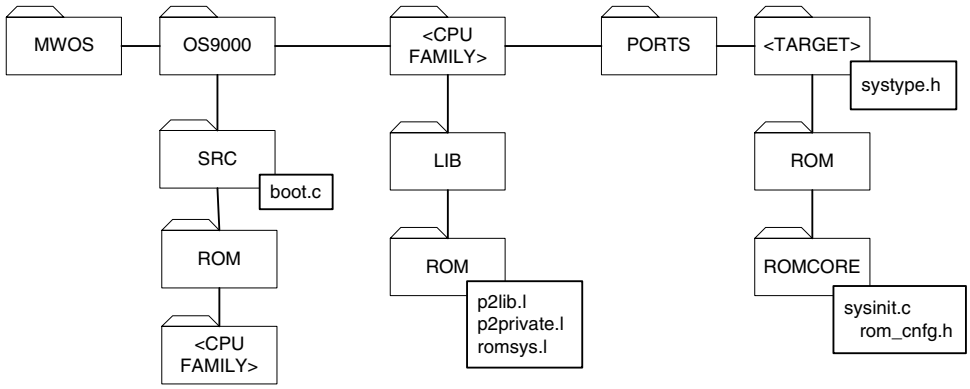
3 Porting the Boot Code

This chapter includes the following topics:

Porting the Bootstrap Code

The source files, `boot.c` and all of the files in the `<CPU Family>` subdirectory of the `ROM` directory, are used to build the bootstrap code.

Figure 3-1. <MWOS>/OS9000 Bootstrap Source Code Directories



These files, and the port-specific `sysinit.c` source file, are compiled and linked together with the distributed libraries to build the bootstrap code. The distributed libraries include the following files:

- `p2privte.l`
- `p2lib.l`
- `romsys.l`

Refer to *Appendix A, Core ROM Services*, for more information about the distribution libraries.

To port the boot code, you must create additional files to support the source files and libraries. The sample target port directories contain examples of these files that you can use as a guide.

File Name	Content Summary
`systype.h`	Target system, hardware-dependent definitions.
`rom_cnfg.h`	The bootstrap memory list and stack definitions. ROM console and boot device record definitions and the ROM memory lists.
`sysinit.c`	Target specific hardware initialization your system may require following a system reset.

Do not modify the other bootstrap source code files. If you alter these files, the port code may not function correctly.

The rom_cnfg.h File

The `rom_cnfg.h` header file contains the target system definitions only used for the bootstrap code. This includes patchable memory locations containing the following information:

- top of the bootstrap stack
- size of memory reserved for low-level system modules
- bootstrap memory lists

Some processors may require additional steps. Refer to your board guide for processor-specific porting information.

Bootstrap Stack Top and Boot Module Memory

The bootstrap code allocates memory from the first RAM memory segment of the system into three parts, as shown in Figure 3-1. The bootstrap code allocates the global data area and the stack area for its own use. It reserves the special memory pool for the low-level system modules to use.

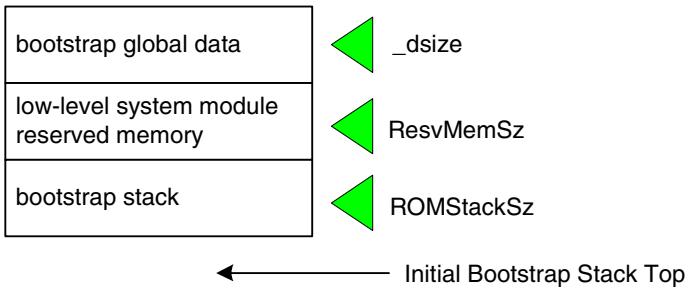
The definitions for the size of the bootstrap stack area (`ROMStackSz`) and the boot module memory pool (`ResvMemSz`) are given in `rom_cnfg.h` as shown in the following example:

```
_asm("
ROMStackSz equ $4000 KB
ResvMemSz equ $20000 128KB
romstack:
 dc.l _dsize+ResvMemSz+ROMStackSz
 dc.l ROMStackSz size of ROM stack
");
```

The linker produces a link map for the `romcore` bootstrap image when it is built. Using this map, the offset of `romstack` can be found. Once this address is known, a 32-bit value at that address can be patched to change the size of the memory area reserved for low-level system modules. Additionally, by patching in the proper 32-bit values at that address, and the following address, the size of the bootstrap stack area can be changed.

Figure 3-2 shows the memory diagram of this first RAM segment allocated by the bootstrap code.

Figure 3-2. First Memory Segment



Bootstrap Memory Lists

The ROM memory list is made of pairs of 32-bit integers specifying start and end boundaries for memory lists. The first list is used to map the system's available RAM memory. The second list is used to map special memory regions treated as ROM memory and searched in a non-destructive fashion. Special memory areas may include ROM, flash, or NVRAM memory. For example, consider the following code:

```
/*
*memory search list
*/
_asm(
memlist
    dc.l $4000,$80000  first memory segment includes
    ROM data area and stack
    dc.l $400000, $1000000  second memory segment
    dc.l 0
    dc.l $fff40000, $fff80000  ROM search area
    dc.l 0,0,0,0,0  extra fields for patching lists
);
```

In the example above, the bootstrap code performs the following tasks:

1. uses RAM from the beginning of the first memory segment for its data area and stack (The PowerPC vectors are initialized at $0-$4000.)
2. searches for RAM memory following its stack to $80000
3. searches for RAM memory in the range $400000 to $1000000

The next zero word terminates the RAM search list.

The ROM search list follows the RAM search list. In this example, the ROM search list causes the bootstrap code to search for ROM memory between $FFF40000 and $FFF80000.

These memory lists are used by the `boot.c` source file when it builds a table of available memory. Each list is searched for valid memory segments, and each valid segment is added to the memory table.

The 32-bit integer in `memlist` represents "start" and "end" boundaries for memory lists. As a general rule, avoid using all zeros or all "f's" for these boundaries.

Avoid inserting unnecessary spaces in your `rom_cnfg.h` file. Though the compilation may complete error-free with extra spaces, it may still cause build errors unnoticed until boot time.

The RAM Search

The first part of the search list defines the areas of the address space where the bootstrap code should normally search for RAM memory. This reduces the time it takes for the system to perform the search. It also prevents the search (and also OS-9) from accessing special use or reserved memory areas such as I/O controller addresses or graphics display RAM.

The first entry, or bank, in this list must point to a block of RAM large enough for storing:

- bootstrap global data
- memory required by the low-level system modules
- start-up bootstrap stack
- system global data

If the system boots from a disk or another device, the first bank needs to be large enough to also hold the size of the bootfile loaded from that device, as well as any buffers required by the boot drivers.

The RAM memory search is performed on each area in the search list by performing the following tasks:

1. reading the first four bytes of every 8K memory block of the area
2. writing a test pattern sequence. Memory is initialized to repetitions of the pattern, `Dude` (0x44756465)
3. reading the area again for comparison

   If the read matches what was written, the search assumes this was a valid RAM block and is added to the system free RAM list.

The Special Memory Search

The second part, or the special memory part of the search list, is strictly a non-destructive memory search. This is necessary so that the memory search does not overwrite modules downloaded into RAM or NVRAM.

During the porting process, you should temporarily include enough RAM (at least 256K) in the special memory list to download parts of the boot file. If this download area has parity memory, you may need to do one of the following:

- Manually initialize it.
- Disable the CPU's parity, if possible.
- Include a temporary routine in the `sysinit.c` file.

The RAM and special memory searches are performed by `boot.c` during the booting process.

The systype.h File

The `systype.h` file is an include file used in building several of the low-level system modules and OS-9 system modules. This file should be viewed as the common location for all port specific hardware definitions and configuration parameters.

The main sections of the `systype.h` file include the following definitions:

- ticker and real time clock
- low-level system module configuration
- hardware specific macros

For support of the bootstrap code, it is important to include in the `systype.h` header file any target-specific hardware definitions you want to use as you write the hardware initialization routines in the `sysinit.c` source file. Such definitions might include hardware specific bit layouts, address offsets, or initial values.

The sysinit.c File

The `sysinit.c` file should contain all special hardware initialization your system requires after a reset or system reboot. The `sysinit.c` file consists of these different sections, or entry points:

- `sysinit`
- `sysinit1`
- `sysinit2`
- `sysreset`

The sysinit Entry Point

The first entry point, `sysinit`, is called almost immediately after a reset by `btfuncs.a`. `sysinit` performs the minimum hardware actions the system may require to enable memory or initialize necessary devices during start up.

This routine does not return through the typical return machine instruction. The return to `btfuncs.a` is made directly by a branch to the `sysreturn:` label.

The `sysinit` routine is always a complete embedded assembly routine. At this point, the stack register has not been initialized to point to a stack area.The `sysinit` code must be written assuming no stack exists.

The sysinit1() Routine

The first C-routine, `sysinit1()` completes any necessary hardware initialization that was not required to be done by the `sysinit` assembly routine. In addition, it makes the call to `rompak1()` to activate any initialization routines in the `initext` module (described later in this section). While a stack is present during `sysinit1()` execution, no static storage is available.

The sysinit2() Routine

The second C-routine, `sysinit2()`, is used for any system initialization required after calling `sysinit1()`. Often, this routine consists of a routine that calls `rompak2()` and returns, as most systems can perform all their required initialization during the first call to `sysinit` and `sysinit1()`. `sysinit2()` is called after `funcs.a` and `boot.c` have completed the following tasks:

- initialized the vector table (for vectors in RAM) and the exception jump table
- performed the memory searches

The sysreset() Routine

The third C-routine, `sysreset()`, is installed as a service to enable the low-level system modules, in particular the low-level debugger, a way of initiating a software reset on the target. `sysreset()` performs any special hardware actions the system requires before attempting a software reset, for example a cache flush. It then initiates the proper instructions to reset the system, or if such a reset is not supported by the target, branches back to the `Cold`: entry point in `btfuncs.a` to initiate the reboot sequence.

The initext Module

The `initext` module is a separately linked portion of hardware initialization code providing a modular functional extension to the `sysinit1()` and `sysinit2()` routines described previously.

It is provided in source form, enabling an end-user to add hardware initialization routines specific to a target configuration that would be inappropriate to include in the base `romcore` module because of hardware modularity requirements. For example, a peripheral device implemented on a card plugged into the host bus may require specific initialization immediately following a CPU reset in the case where a bus reset could not be asserted by the processor in the `sysreset()` routine described above. This initialization code might be appropriately implemented in the `initext` module rather than a `romcore` module, since the end-user may have obtained the port from an OEM providing the base target platform.

There are two entry points to the `initext` module, `rompak1()` and `rompak2()`. When the `initext` module is present in the system immediately following the `romcore` module, `rompak1()` would be executed by `sysinit1()`, and `rompak2()` would be executed by `sysinit2()`, provided those routines attempt to call the rompak routines.

Note the following information:

- `rompak1()` is executed prior to ROM module scan.
- `rompak2()` is executed after ROM module scan and all ROM modules have been painted.
- No static storage is available for the `initext` module.

The `initext` module is built in a `ROM/INITEXT` subdirectory within the target port directory. You should defer implementation of your base `initext` module until after your initial port is completed. When you decide to start on your `initext` module, use the sources and makefile from an example port as a reference.

Configuring the Low-Level System Modules

Once the bootstrap code is ported and your low-level serial I/O drivers are ready, you need to provide some configuration data to define what your initial port looks like.

The OS-9 booting process relies on the use of a configuration data module (`cnfgdata`) to define certain default parameters used in the boot. The configuration data module provides for great flexibility in designing your system, but is not required for a simple port. We recommend you keep your initial port as simple as possible.

If you are planning to use the Hawk remote debugger during the porting process, you must use the configuration data module. Read carefully about the configuration module and the low level network configuration before attempting such a port.

For the simple port using the target resident RomBug debugger, you do not need a configuration module. Configuring the simple port involves the following steps:

1. Add to `systype.h` the definitions the low-level system modules use as default configuration values for system console and communications ports.
2. Modify the boot module makefiles to disable use of the configuration data module for the first port stage.
3. Modify the boot module list found in `coreboot.ml` to reflect the low-level system modules required for your system.

Adding Configuration Information to systype.h

`systype.h` should be modified to include definitions for the symbols `CONSNAME` and `COMMNAME`. The symbol `CONSNAME` gives the name of the console device record that the console configuration module (`conscnfg`) will, by default, select for use as the system console. Similarly, `COMMNAME` is used by `commcnfg` as the default for the communication port. For example:

```
#define CONSNAME     COMM1NAME
#define COMMNAME     "MVME1603:com2"
```

Modifying Low-Level System Module makefiles

For your initial port, disable use of the configuration data module. Later chapters discuss how to build and use this module. Modify each of the following makefiles copied earlier from an example port.

```
<Target>/ROM/COMMCNFG
<Target>/ROM/CONSCNFG
<Target>/ROM/PORTMENU
<Target>/ROM/USEDEBUG
```

These makefiles contain the definition of a macro called `SPEC_COPTS` that is defined to include the C option `-dUSECNFGDATA`. Comment this option out of the macro definition. For example, change the first line into the second line:

```
SPEC_COPTS = -d<option1> -d<option2> -dUSECNFGDATA
SPEC_COPTS = -d<option1> -d<option2> #-dUSECNFGDATA
```

Modifying coreboot.ml

The file `coreboot.ml`, copied from an example port, contains a list of low-level system modules included in the boot image when it is built.

To finish the configuration of your initial port, use the asterisk (*) to comment out the use of the configuration modules `cnfgdata` and `cnfgfunc`, and replace the low level I/O modules names in this list with the ones appropriate for your target. The I/O modules used in the example ports are usually named `io<device>`.

Do not remove the `console`, `conscnfg`, or `commcnfg` module names, and be sure to add the appropriate low-level serial I/O module names after `console`, but before the `conscnfg` or `commcnfg` module names. Once the new port is proven, the console and communication ports can be removed if desired.

Do not change the order of the low-level system module names or the system may not boot.

The ROM Image

The OS-9 ROM image is a set of files and modules that collectively make up the operating system. The specific ROM image contents can vary depending on hardware capabilities and user requirements of the system in use. To simplify the process of loading and testing OS-9, the ROM image is divided into two parts: the low-level image, called `coreboot`, and the high-level image, called `bootfile`.

Coreboot

The coreboot image is generally responsible for initializing hardware devices and locating the high-level (or bootfile) image as specified by its configuration. It is also responsible for building basic structures based on the image it finds and passing control to the kernel to bring up the OS-9 system.

Bootfile

The bootfile image contains the kernel and other high-level modules (initialization module, file managers, drivers, descriptors, applications). The image is loaded into memory based on the device you select from the boot menu. The bootfile image normally brings up an OS-9 shell prompt, but can be configured to automatically start an application.

Building the ROM Image

Once you have ported the bootstrap code, written (or copied) the sources and makefile for your low level serial I/O modules, and configured your system, you are ready to build the ROM image:

Step 1. Use the makefile `<Target>/ROM/makefile` to build your low-level system modules. This makefile forces a make within each of the subdirectories included in its `TRGTS` macro to build the low-level system modules.

Step 2. Use the makefile `<Target>/BOOTS/SYSTEMS/PORTBOOT/ makefile` to build your boot image. This makefile not only creates the `rom` file, but also oversees the creation of the coreboot and bootfile.

There must be at least four bytes of padding between the coreboot and bootfile images in the merged `rom` file.

You may get errors when running make. If these problems are not related to low-level system modules, you can ignore the errors. This is because you only need the `coreboot` file for testing and it is created before the make exits with errors while trying to build the `rom` file.

4 Creating Low-Level Serial I/O Modules

This chapter includes the following topics:

Creating the Low-Level Serial I/O Modules

While it is not absolutely necessary to have a serial I/O console device on your system, it is strongly recommended that your initial port include both a console device and an auxiliary serial I/O communications device.

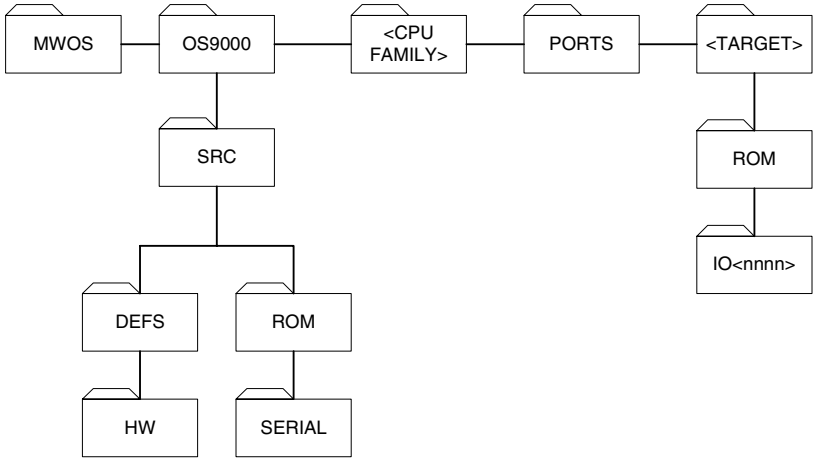
The console I/O routines are used by the bootstrap code and low-level system modules for error messages, and by the debugger and menu-booters for interactive I/O. The communications port is used by the debuggers as a download and talk-through port. The communications port can also be used as the SLIP device for low level network communications with the Hawk remote debugger.

Source code is provided for several low-level serial modules that you can configure and use in your system without modification. If your target has a serial device for which no I/O module already exists, use the example sources as a guide to write your own. If both the console port and communications port use the same type of hardware interface, you only need to build one low-level I/O module.

If you are writing your own low-level serial driver, be advised that in order to use Hawk's module download feature you will need to implement the polled interrupt service routine, `cons_irq()`. The distributed low-level serial I/O module sources are in the following directory: `MWOS/SRC/ROM/SERIAL`.

Create a subdirectory for your own source code if you are building your own I/O module.

Figure 4-1. Low-Level Serial I/O Source Code Directories



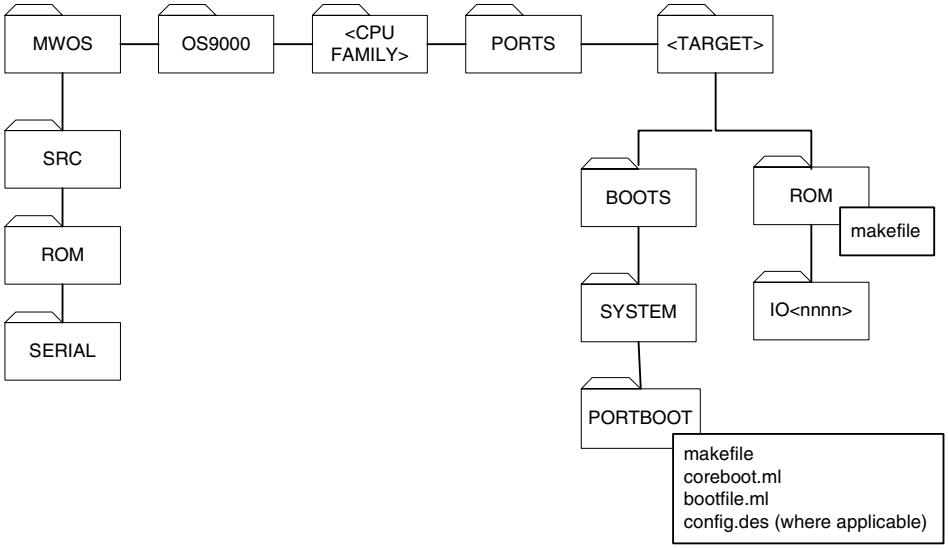
In addition to the directories listed earlier, each example port directory contains `<Target>/ROM/IO<nnnn>` directories containing makefiles used to build the low-level I/O module used in the port. You need to create such a directory and makefile for your serial devices in your ports directory. Use the example makefiles as a guide.

Device specific include files (`<xxxx>.h`) are normally kept in the `MWOS/SRC/DEFS/HW` directory. These are typically chip-specific definitions and are to be shared by both low-level (ROM) and high-level (OS) drivers.

Building the Low-Level Serial I/O Modules

The makefile for your I/O module should be created in a properly named subdirectory of your ports `ROM` directory. (Consider, for example, `<Target>/ROM/IO<nnnn>`.) Use the makefiles from the example ports as a guide.

Figure 4-2. Low-Level Serial I/O Source Code Directories



To add your low-level serial I/O module to the system, complete the following steps:

Step 1. Edit the makefile, `<Target>/ROM/makefile`.

Step 2. Add your device directory name to the list of targets used to define the `TRGTS` macro.

Step 3. Add the low-level serial I/O module name into the `corboot.ml` file in the `PORTBOOT` directory.

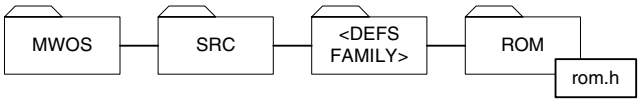
By doing this, your low level I/O module is rebuilt along with the bootstrap code and the rest of the low-level system modules when:

- `<Target>/ROM/makefile` is invoked and included in the `rom` file,
- and, `<Target>/BOOTS/SYSTEMS/PORTBOOT/makefile` is invoked creating the boot image `coreboot`.

The Console Device Record

A console device (`consdev`) record is maintained for each low level serial I/O device included with the low-level system modules. This record is used to access the services of the I/O module, and to maintain lists of such devices. The definition of `consdev` appears in the header file, `rom.h`, and appears below for illustration.

Figure 4-3. Console Device Record Directory




```
struct consdev {
    idver    infoid;                          /* structure version tag */
    void      *cons_addr;                   /* port address of I/O device*/
    u_int32    (*cons_probe)(Rominfo, Consdev),    /* h/w probe service */
        (*cons_init)(Rominfo, Consdev), /* initialization service */
        (*cons_term)(Rominfo, Consdev);  /* de-initialization service*/
    u_char    (*cons_read)(Rominfo, Consdev);            /* read service */
    u_int32   (*cons_write)(char, Rominfo, Consdev),     /* write service */
        (*cons_check)(Rominfo, Consdev);  /* character check service */
    u_int32   (*cons_stat)(Rominfo, Consdev, u_int32),
              (*cons_irq)(Rominfo, Consdev),
              (*proto_upcall)(Rominfo, void*, char*);
    u_int32   cons_flags;                         /* device flags */
    u_char    cons_csave,                        /* read ahead stash */
           cons_baudrate,                 /* communication baud rate */
           cons_parsize,                   /* parity, data bits, stop bits */
           cons_flow;                          /* flow control */
    u_int32   cons_vector,                       /* interrupt vector */
              cons_priority,                     /* interrupt priority */
             poll_timeout;
    u_char    *cons_abname,                      /* abreviated name */
       *cons_name;                    /* full name and description */
    void      *cons_data;                       /* device specific data */
    void      *upcall_data;
    Consdev   cons_next;                    /* next serial device in list*/
    u_int32   cons_level;                       /* interrupt level */
    int       reserved;
};
```

Low-Level Serial I/O Module Services

The following entry points describe the services required of each low-level serial I/O module to support the booting process.

Table 4-1.

Function	Description
cons_check()	Check I/O Port
cons_init()	Initialize Port
cons_irq()	Polled Interrupt Service Routine for I/O Device
cons_probe()	Probe for Port
cons_read()	Read Character from I/O Port
cons_stat()	Set Status on Console I/O Device
cons_term()	De-initialize Port
cons_write()	Write Character to Output Port
notification_handler()	Handle Callback from Notification Services

cons_check()
Check I/O Port

Syntax

```
u_int32 cons_check(
    Rominfo       rinf,
    Consdev       cdev);
```

Description

`cons_check()` interrogates the port to determine if an input character is present and returns the appropriate status.

Parameters

`rinf`
: points to the `rominfo` record structure.

`cdev`
: points to the console device record for the device.

cons_init()
Initialize Port

Syntax

```
u_int32 cons_init(
    Rominfo       rinf,
    Consdev       cdev);
```

Description

`cons_init()` initializes the port. It resets the device port, sets up for transmit and receive, and sets up baud rate, parity, bits per type, and number of stop bits. `cons_init()` also registers a notification handler described below, with the notification services.

Parameters

`rinf`
: points to the `rominfo` record structure.

`cdev`
: points to the console device record for the device.

cons_irq()
Polled Interrupt Service Routine for I/O Device

Syntax

```
u_int32 cons_irq(
    Rominfo       rinf,
    Consdev       cdev);
```

Description

`cons_irq()` is an interrupt service routine. It is installed for the device that performs the following polling interrupt service on receipt of a device interrupt:

1. Disable further interrupts on the device.
2. Clear the interrupt from the device.
3. Initialize the low-level polling timer.
4. Set the polling time-out value and loops through the process of checking the device and timer until either a character is received or the time-out occurs.
5. Send a character that is received up the protocol stack by calling the uplink routine installed in the console device structure.
6. Repeat steps two through five until a time-out occurs.
7. Re-enable device interrupts and returns.

Parameters

`rinf`
: points to the `rominfo` record structure.

`cdev`
: points to the console device record for the device.

cons_probe()
Probe For Port

Syntax

```
u_int32 cons_probe(
    Rominfo       rinf,
    Consdev       cdev);
```

Description

`cons_probe()` tests to see if the hardware described by the console device record `cdev` is present. This could be a read of an I/O register based on the value of `cons_addr` in the console device record.

Parameters

`rinf`
: points to the `rominfo` record structure.

`cdev`
: points to the console device record for the device.

cons_read()
Read Character From I/O Port

Syntax

```
u_char cons_read(
    Rominfo       rinf,
    Consdev       cdev);
```

Description

`cons_read()` returns a character from the device's input port. `cons_read()` repeatedly calls `cons_check()` until a character is present. `cons_read()` should not echo the character. The only special character handling it might perform is XON-XOFF processing if the `CONS_SWSHAKE` flag is set in the `cons_flow` field of the console device structure.

Parameters

`rinf`
: points to the `rominfo` record structure.

`cdev`
: points to the console device record for the device.

cons_stat()
Set Status on Console I/O Device

Syntax

```
u_int32 cons_stat(
    Rominfo       rinf,
    Consdev       cdev,
    u_int32       code);
```

Description

`cons_stat()` changes the operational mode of the I/O module.

Parameters

`rinf`
: points to the `rominfo` record structure.

`cdev`
: points to the console device record for the device.

`code`
: is the low-level setstat code indicating operational mode change.

The supported setstat codes are defined in `MWOS/SRC/DEFS/ROM/rom.h` and described as follows:

CONS_SETSTAT_POLINT_OFF

`Indication`
: Issue when `hlproto` no longer requires the services of the communications port.

`Operation`
: Disable receive interrupts on port.

CONS_SETSTAT_POLINT_ON

`Indication`
: Issue when `hlproto` requires the services of the communications port for user-state connections.

`Operation`
: Verify configuration of low-level timer, enable receive interrupts on port.

CONS_SETSTAT_ROMBUG_OFF

`Indication`

Issue indirectly through notification services when the RomBug debug client returns control from any breakpoint, exception, or `d_sysdebug` entry.

`Operation`

Restore any applicable port- or chip-specific configuration (including interrupts).

CONS_SETSTAT_ROMBUG_ON

`Indication`

Issue indirectly through notification services when the RomBug debug client gets control on any breakpoint, exception, or `d_sysdebug` entry.

`Operation`

Save any applicable port- or chip-specific configuration (including interrupts). Disable any receive interrupts on port.

CONS_SETSTAT_RXFLOW_OFF

`Indication`

Issue when a driver user (such as `llslip`) needs to restrict the flow of received data.

`Operation`

If hardware handshaking is configured, assert hardware flow control (on), otherwise if software handshaking is configured, send an X-OFF.

CONS_SETSTAT_RXFLOW_ON

`Indication`

Issue when a driver user (such as `llslip`) needs to restore the flow of received data.

`Operation`

If hardware handshaking is configured, turn off hardware flow control, otherwise if software handshaking is configured, send an X-ON.

cons_term()
De-initialize Port

Syntax

```
u_int32 cons_term(
    Rominfo       rinf,
    Consdev       cdev);
```

Description

`cons_term()` shuts the port down by disabling transmit and receive.

Parameters

`rinf`
: points to the `rominfo` record structure.

`cdev`
: points to the console device record for the device.

cons_write()
Write Character To Output Port

Syntax

```
u_int32 cons_write (
    char          c,
    Rominfo       rinf,
    Consdev       cdev);
```

Description

`cons_write()` writes a character to the output port with no special character processing (for a low-level serial driver that does not use software handshaking).

The sample sources also contain the following serial I/O module entry points to support the user state Hawk remote debugger. For the initial port, it is not necessary to include these entry points because the previous functions are sufficient for support of system-state operation at the low-level. The following entry points support the use of low-level serial I/O module while in user state after the system is booted. This functionality is required for use of the I/O module by the user-state Hawk debugger.

Parameters

`c`
: is the character written to the output port.

`rinf`
: points to the `rominfo` record structure.

`cdev`
: points to the console device record for the device.

notification_handler()
Handle Callback from Notification Services

Syntax

```
u_void notification_handler(
    u_int32       direction,
    void          *cdev);
```

Description

`notification_handler` issues calls to `cons_stat()` with the `CONS_SETSTAT_ROMBUG_ON` and `CONS_SETSTAT_ROMBUG_OFF` codes.

Parameters

`direction`
: is the direction value provided from the notification services: the `NTFY_DIR_TOROM` value indicates a transition into the ROM from the operating system; the `NTFY_DIR_TOSYS` values indicates a transition to the operating system from the ROM.

`cdev`
: points to the console device structure for the device.

Starting-up the Low-Level Serial I/O Module

During the early stages of system bootup, the bootstrap code searches for and initializes all low-level system modules included in the boot image. The initialization entry point for the low-level system modules is supplied in a relocatable (.r) file. This entry point branches to the C function `p2start()` which you need to provide for each of your low-level I/O modules. The initialization routine performs these tasks:

- allocates/initializes the console device record for the device.
- makes the entry points for its services available through the `consdev` record.
- initializes configuration data for the I/O device.
- installs its `consdev` record on the list of I/O devices in the console services record.

An example `p2start()` routine for a low level I/O module follows. (The console device record is allocated in the module's static data area.)

```
consdev    cons_r;    /* allocate console device record */
error_code p2start(
Rominfo rinf,         /* bootstrap services record structure pointer */
u_char *glbls)        /* bootstrap global data pointer */
{
    Cons_svcs    console = rinf->cons;
                       /* get the console services record pointer*/
    Consdev      cdev;
                       /* local console device structure pointer */
       /* verify that a console services module has been initialized */

    if (console == NULL)
       return (EOS_NOCONS);
          /*cannot install w/o the console services record*/

    /* initialize device record for our device */
    cdev = &cons_r;   /* point to our console device record */
    cdev->struct_id = CONSDEVID;     /* id and version tags */
    cdev->struct_ver = CDV_VER_MAX;
    /* export our service routine entry points */
    cdev->cons_probe = &io16450_probe;
    cdev->cons_init = &io16450_init;
    cdev->cons_term = &io16450_term;
```

```
    cdev->cons_read = &io16450_read;
    cdev->cons_write = &io16450_write;
    cdev->cons_check = &io16450_check;

    /*  The following services are not required for the initial port */
    /*
    cdev->cons_stat = &io16450_stat;
    cdev->cons_irq = &io16450_irq;*/

    /* initialize the device configuration data */
    cdev->cons_addr = (void *)COMM2ADDR;
                /* base address of I/O port */
    cdev->cons_baudrate = CONS_BAUDRATE_9600;
                /* communication baud rate  */
    cdev->cons_vector = COMMVECTOR;        /* interrupt vector */
    cdev->cons_priority = COMMPRIORITY;    /* interrupt priority */
    cdev->poll_timeout = 2000;
                /* polling routine timout value*/
    cdev->cons_abname = (u_char *)COMM2ABNAME;
                /* abreviated device name */
    cdev->cons_name = (u_char *)COMM2NAME;     /* device name */

    /* install the record structure on the list of available I/O modules */
    cdev->cons_next = console->rom_conslist;
    console->rom_conslist = cdev;

    return (SUCCESS);
}
```

The default value definitions used to initialize the device configuration data should be placed in the target-specific `systype.h` header file, leaving the I/O module source code portable across platforms.

If the same I/O module is used with the console and communications ports, an additional console device record, (for example, `comm_r`) should be allocated and initialized with the proper data for the communications port. Both console device records should be added to the list of available devices.

The console and communications port configuration modules (`conscnfg` and `commcnfg`), using the configuration data module (`cnfgdata`), determine which console device record is selected as console and communications port.

5 Creating a Low-Level Ethernet Driver

Low-level Ethernet drivers enable communications between the target and the host. Ethernet drivers support boot device and debugger operations, and can provide other functionality, such as console services.

Low-level Ethernet drivers communicate to low-level IP (`llip`), receiving and sending data as required. `llip` also communicates with the low-level TCP (`lltcp`) and low-level UDP (`lludp`) protocols, forwarding datagrams up to the appropriate protocol and receiving datagrams to be delivered to the low-level driver. `lltcp` and `lludp` communicate with the `protoman` module handling the protocol services to communicate with network booters, virtual consoles, and debugger modules.

This chapter includes the following topics:

Creating a Low-Level Ethernet Driver

Use the following steps to create a low-level Ethernet driver.

Step 1. Obtain information about the Ethernet chip on the target board.

- Get data book from the manufacturer.
- Obtain packet drivers for the chip to test out on a PC. Several packet driver collections are available on the World Wide Web, on FTP sites, and by mail.
- Obtain a reference board with the supported chip.
- Determine the chip's memory management map for mbufs.
- Determine how the information is transmitted, for example in a circular buffer or FIFO buffer.

Step 2. Review the supplied example Ethernet drivers to find the one that most closely fits the capabilities and requirements of the Ethernet chip on your target board. For an example Ethernet driver, see the `ll21040` file in the `<MWOS>/SRC/ROM/PROTOCOLS` directory.

Step 3. Edit the example you selected to include the information specific to the target Ethernet chip.

Step 4. Add the driver to your boot code.

Step 5. Remake the boot code.

Step 6. Test communications using the Ethernet driver you created.

Required Ethernet Driver Functions

The following sections define the required functions for implementing a low-level Ethernet driver.

Proto_srvr Structure

This structure, defined in `rom.h`, is common to all protocols and drivers and identifies the modules in the low-level `protoman` protocol list.

```
struct proto_srvr {
   idver        infoid;              /* id/version for proto_srvr */
#if  defined(NEWINFO)
   Proto_srvr next;                  /* next protocol stack in list */
   u_int32      proto_type_id;         /* protocol id */
   error_code    (*proto_install)(Rominfo, u_char *),
                                     /* Installation */
                 (*proto_iconn)(Llpm_conn, u_int32),
                                     /* initiate conn */
                (*proto_read)(Llpm_conn, u_int32, LlMbuf *),
                                     /* read conn */
                (*proto_write)(Llpm_conn, u_int32),
                                     /* write conn */
                (*proto_status)(Llpm_conn, u_int32, void *),
                                     /* get status */
                (*proto_tconn)(Llpm_conn, u_int32),
                                     /* terminate conn */
                (*proto_deinstall)(Rominfo),
                                     /* De-installation */
                (*proto_timeout)(Rominfo, Proto_srvr),
                                     /*timeout processing*/
                (*proto_upcall)(Rominfo, Proto_srvr, void*);
                                     /* LL ISR upcall */
   void         *proto_data;         /* server local data */
                                     /* structure ptr */
   u_int32      proto_data_size,     /* protocol's data area size */
                proto_conn_cnt;    /* number of active connections */
   Consdev      proto_cons_drvr;     /* llvl serial comm console */
                                     /* (slip) */
   u_int16      proto_mtu,      /* Max Xmission Unit for protocol*/
                proto_hdr_len,  /* Space requirements for header*/
               proto_tlr_len; /* Space requirements for 8 trailer*/
   u_char       proto_flags;  /* Protocol status & type flags*/
   u_char       proto_rsv1;          /* align on longword boundary */
```

```
    u_int32   proto_addr;                /* V1 only - IP address, null */
                                        /* except drivers */
    u_char    *proto_globs;              /* Pointer to protocol srvr */
                                        /* globals */
    u_int32   proto_vector,              /* vector for lldrivers */
              proto_priority;            /* llisr priority */
    void      *proto_port_addr;          /* lldriver port address */
    /* fields added at V2 */
    u_char    proto_ipaddr[16];          /* Extended IP address */
    u_char    proto_hwaddr[16];          /* Physical (MAC) address */
    u_int32   proto_irqlevel;            /* IRQ level for low-level */
                                        /* (drivers */
    char      *proto_drv_name;           /* name identifier of protocol */
                                        /* /driver */
    u_int32   proto_rsv2[6];             /* reserved for emergency */
                                        /* expansion */
#else
     int        reserved;
#endif
};

/* values for proto_flags */
#define PVR_RELIABLE           0x01    /* reliable protocol */
#define PVR_LLISR_REG_REQ      0x02    /* request LLISR registration */
#define PVR_LLISR_REG_ERR      0x04    /* the LLISR reg req failed */

/* The following flag is to be used to indicate which driver to use, if
* the interface IP address does not match that specified during the bind.
*/

#define PVR_DRV_USEME          0x08
#define     PVR_BOOTDEV        0x10    /* To indicate interface */
                                       /* booted from */
#define     PVR_MWRSV0         0x20
/* We might use these for distinguishing protocols/drivers at some point
*/
#define PVR_DRIVER             0x40
#define PVR_PROTOCOL           0x80
```

```
/* Reserved Flags for Microware for proto_rsv1 field of proto_srvr */
#define PVR_MWRSV1              0x01
#define PVR_MWRSV2              0x02
#define PVR_MWRSV3              0x04
#define PVR_MWRSV4              0x08

/* For OEM User use */
#define PVR_OEM1                0x10
#define PVR_OEM2                0x20
#define PVR_OEM3                0x40
#define PVR_OEM4                0x80

/* subcodes for implementation by proto_status() */
#define SS_IntEnable            0x01
#define SS_IntDisable           0x02
#define SS_RombugOn             0x03
#define SS_RombugOff            0x04
```

The Low-Level Ethernet Driver Entry Point Services

In each of the entry points of the driver module, complete the following steps:

Step 1. Save the current global variables pointer.

Step 2. Set the global variables pointer to the driver's variables when it is called (before accessing them).

Step 3. Restore the original global variables pointer at all exit points.

This can be done using the `swap_globals` function provided in the `p2lib` library.

Table 5-1. Entry Points

Function	Description
`proto_deinstall()`	Low-level driver de-installation entry point
`proto_iconn()`	Low-level driver initiate connection entry point
`proto_install()`	Installs the low-level ethernet driver
`proto_read()`	Low-level driver polled read entry point
`proto_status()`	Low-level driver status entry point
`proto_tconnl()`	Low-level driver terminate connection entry point
`proto_timeout()`	Low-level driver timeout entry point
`proto_upcall()`	Low-level driver upcall for interrupt processing
`proto_write()`	Low-level driver write entry point

proto_deinstall()
Low-Level Driver De-installation Entry Point

Syntax

```
error_code (*proto_deinstall)(Rominfo rinf);
```

Description

`proto_deinstall()` is the low-level driver de-installation entry point. It takes the driver `proto_srvr` off the protoman protocols/driver list. The service de-initializes the chip and frees the memory allocated for the buffers. It also removes its name from the notification services list.

Parameters

`rinf`
: points to the `rominfo` structure.

proto_iconn()
Low-Level Driver Initiate Connection Entry Point

Syntax

```
error_code (*proto_iconn)(
   Llpm_conn     conn_entry,
   u_int32       index);
```

Description

`proto_iconn()` is the low-level driver initiate connection entry point. This service performs the driver related connection specific initialization.

Parameters

`conn_entry`
: is not used in the drivers but is present because the protocols also use the same prototypes. This entry point is called by `hlproto`, to turn on/off the interrupts. It is also called by the notification handler routine.

`index`
: points to the appropriate `proto_srvr` (tcp, ip, udp, slip).

proto_install()
Installs the Low-Level Ethernet Driver

Syntax

```
error_code (*proto_install)(
    Rominfo       rinf,
    u_char        *globs);
```

Description

proto_install() installs the low-level Ethernet driver module. The service initializes the chip and masks the interrupts. It initializes the proto_srvr structure, sets all the entry points, and installs itself on the protocol list in the low-level protocol manager structure. Each driver must allocate the memory for the receive buffers and save the pointer.

Each driver has to allocate its own pool of mbufs. Set the PVR_LLISR_REG_REQ and PRM_LLISR_REG_REQ bits so hlprotoman can register the LLISRs to run in interrupt driven mode. The PVR_DRIVER flag in proto_flags indicates the module is a driver module. If the service knows the IP address, it sends a gratuitous ARP.

Parameters

rinf
: points to the rominfo structure.

globs
: points to the module global variables. You should save this pointer in the proto_globs field of the proto_srvr structure so you can access the module global variables.

proto_read()
Low-Level Driver Polled Read Entry Point

Syntax

```
error_code (*proto_read)(
    Llpm_conn     conn_entry,
    u_int32       index,
    LlMbuf        *rmb);
```

Description

`proto_read()` is the low-level driver polled read entry point. It polls the chip and returns if it has a good packet or if it was called with the low-level connection entry flag set to indicate `nonblocking` read, and the timer expires.

The suggested algorithm, while waiting for a packet, is to periodically check the timer routine if a `nonblocking` read is specified and returns if the timer reaches a value of zero, or if it receives a valid packet. In the latter case, the service processes the Ethernet packet before passing it up the stack.

Parameters

`conn_entry`
: is not used in the drivers but is present because the protocols also use the same prototypes. This entry point is called by `hlproto`, to turn on/off the interrupts. It is also called by the notification handler routine.

`index`
: points to the appropriate `proto_srvr` (tcp, ip, udp, slip).

`rmb`
: points to the global mbuf pool.

proto_status()
Low-Level Driver Status Entry Point

Syntax

```
error_code (*proto_status)(
   Llpm_conn     conn_entry,
   u_int32       code,
   void          *ps);
```

Description

`proto_status()` is the low-level driver status entry point.

Parameters

`conn_entry`
: is not used in the drivers but is present because the protocols also use the same prototypes. This entry point is called by `hlproto`, to turn on/off the interrupts. It is also called by the notification handler routine.

`code`
: specifies what the caller expects to be done. It can have the following values:
  - `SS_IntEnable` to enable interrupts (called by `hlproto`).
  - `SS_IntDisable` to disable interrupts (called by `hlproto`).
  - `SS_RombugOn` to indicate a change from user to system state (called by the notification handler).
  - `SS_RombugOff` to indicate a change from system to user state (called by the notification handler).

`ps`
: points to the `proto_srvr` structure.

proto_tconnl()
Low-Level Driver Terminate Connection Entry Point

Syntax

```
error_code (*proto_tconn)(
    Llpm_conn     conn_entry,
    u_int32       index);
```

Description

`proto_tconn()` is the low-level driver terminate connection entry point. This service does the driver related connection specific termination (converse of `proto_iconn()`).

Parameters

`conn_entryis not used in the drivers but is present because the protocols also use t`
he same prototypes. This entry point is called by `hlproto`, to turn on/off the interrupts. It is also called by the notification handler routine.

`index`
points to the appropriate `proto_srvr` (tcp, ip, udp, slip).

proto_timeout()
Low-Level Driver Timeout Entry Point

Syntax

```
error_code (*proto_timeout)(
    Rominfo       rinf,
    Proto_srvr    ps);
```

Description

`proto_timeout()` is the low-level driver time-out entry point. This entry point is called by the `hlproto` thread to provide for any kind of time-out needed. The sample drivers do not use this and, therefore, it is nulled out in `proto_install()`.

Parameters

`rinf`
: points to the `rominfo` structure.

`ps`
: points to the `proto_srvr` structure.

proto_upcall()
Low-Level Driver Upcall For Interrupt Processing

Syntax

```
error_code (*proto_upcall)(
    Rominfo      rinf,
    Proto_srvr   pd,
    void*        c);
```

Description

`proto_upcall()` is the low-level driver upcall routine for interrupt processing. It is called on the interrupt context from the `commonIRqEntry` point in `hlproto`.

This service is used primarily with receive interrupts. If the service receives a valid IP packet, it updates the ARP table to eliminate sending out an ARP packet. If it is an ARP packet, the service processes it, replies to it if `proto_upcall()` is the destination address, and also saves the sender's hardware address. The `arp_tblupdate()` function updates the tables, if needed. If the service receives an IP packet, it calls the `proto_upcall()` entry point of the next protocol on the stack (IP for now).

Before doing any interrupt processing, this service restores the interrupt status register and the mask register so it does not miss other packets while processing one.

Parameters

`rinf`
: points to the `rominfo` structure.

`pd`
: points to the `proto_srvr` structure.

`cdata`
: (packet, character) being passed. This is typecast void because each level typecasts it.

proto_write()
Low-Level Driver Write Entry Point

Syntax

```
error_code (*proto_write)(
   Llpm_conn     conn_entry,
   u_int32       index);
```

Description

`proto_write()` is the low-level driver write entry point.

When called from the next upper layer protocol module on the stack, (IP for now), the service puts the Ethernet headers in place and hands them to the chip to send out on the wire. The service masks the interrupts during the entire processing time and does not return until the packet has been sent out on the wire.

Parameters

`conn_entry`
: is not used in the drivers but is present because the protocols also use the same prototypes. This entry point is called by `hlproto`, to turn on/off the interrupts. It is also called by the notification handler routine.

`index`
: points to the appropriate `proto_srvr` (tcp, ip, udp, slip).

Additional Utility Functions

The following utility functions are used with mbufs:

Table 5-2. Utility Functions

Function	Description
`find_n_init_mbuf()`	Find and initialize an mbuf
`init_eth_mbuf()`	Initialize an mbuf

find_n_init_mbuf()
Find and Initialize an mbuf

Syntax

```
error_code find_n_init_mbuf(
    u_char        *rmb,
    LlMbuf        *mb);
```

Description

`find_n_init_mbuf()` finds and initializes an `mbuf`.

This function returns an `ENOBUF` error if it cannot find an `mbuf`.

Parameters

`rmb`
: points to the global mbuf pool.

`mb`
: points to the returned `mbuf` so it can be used.

init_eth_mbuf()
Initialize an mbuf

Syntax

```
void init_eth_mbuf();
```

Description

The `init_eth_mbuf()` function is called from `proto_install()` to initialize an `mbuf` after allocating memory for the mbuf pool.

Low-Level ARP

The ARP included with the low-level Ethernet driver has minimal functionality.

Low-level Ethernet drivers do not avoid sent ARP requests. Whenever a driver receives an ARP/IP packet, it saves the sender's hardware address (if the packet is directed to this driver), assuming the driver has sent a request, since it wants to communicate with the driver. The low-level Ethernet driver processes the ARP request and replies to the sender. The driver also updates an ARP table without removing any entries.

Table 5-3. Low-Level ARP Functions

Function	Description
`arpinit()`	Low-level ARP init function
`arpinput()`	ARP input processing routine
`arpresolve()`	Resolves hardware addresses
`arptbl_update()`	Update ARP table
`arpwhohas()`	ARP packet request for hardware address
`in_arpinput()`	ARP input processing and replying function

arpinit()
Low-Level ARP init Function

Syntax

```
error_code arpinit(Rominfo rinf);
```

Description

`arpinit()` function allocates memory for the ARP table and ARP buffer and initializes the buffer.

Parameters

`rinf`
: points to the `rominfo` structure.

arpinput()
ARP Input Processing Routine

Syntax

```
void arpinput(
    Proto_srvr    psrvr,
    LlMbuf        mb,
    Rominfo       rinf);
```

Description

`arpinput()` is the ARP input processing routine. The routine checks for common length and type. Only IP protocol packets are processes when `in_arpinput()` is called.

Parameters

`psrvr`
: points to the proto_srvr structure.

`mb`
: points to the received packet.

`rinf`
: points to the `rominfo` structure.

arpresolve()
Resolves Hardware Addresses

Syntax

```
error_code arpresolve(
    Llpm_conn     conn_entry,
    u_char        *desten);
```

Description

`arpresolve()` looks into the ARP table and, if successful in finding the entry for the destination address in the `Llpm_conn conn_entry`, copies the hardware address to that destination address, `desten`. If `arpresolve()` cannot find the entry, it returns a non-zero value. This service is called from the `proto_write()` routine, and assumes it is always able to resolve the address without ever having to send an ARP request. If however, it does have to send requests, it calls `arpwhohas()` to broadcast the request. In this case the `proto_write()` function would have to be suspended until `arpresolve()` gets a response and is able to resolve the hardware address.

Parameters

`conn_entry`
: is not used in the drivers but it is present because the protocols also use the same prototypes. This entry point is called by `hlproto`, to turn on/off the interrupts. It is also called by the notification handler routine.

`desten`
: is the destination address.

arptbl_update()
Update ARP Table

Syntax

```
error_code arptbl_update(
    Proto_srvr    psrvr,
    LlMbuf        mb,
    Eth_header    eth);
```

Description

The ARP table update function is called from the driver's `proto_read()` and `proto_upcall()` routines. It performs ARP table updates if the sender included this service's Ethernet address (through this service's gratuitous ARP or other means) and did not make an ARP request. This prevents this service from having to make ARP requests. This service compares this address to its own address to determine if the packet was directed to it.

In addition, packets that are not directed to this service are filtered by returning ERROR, preventing the service from searching the stack in the interrupt context.

Parameters

`psrvr`
: points to the `proto_srvr` structure.

`mb`
: points to the packet received.

`eth`
: points to the Ethernet address of the packet received.

arpwhohas()
ARP Packet Request For Hardware Address

Syntax

```
error_code arpwhohas(
    Proto_srvr    psrvr,
    struct in_addr* addr,
    Rominfo       rinf);
```

Description

`arpwhohas()` broadcasts an ARP packet and asks for the hardware address of the machine with the supplied IP address, `addr`.

This is used only in `proto_install()` when the driver does a gratuitous ARP informing the world of its hardware address. This function can be used in the future for sending ARP requests.

Parameters

`psrvr`
: points to the `proto_srvr` structure.

`addr`
: is the IP address.

`rinf`
: points to the `rominfo` structure.

in_arpinput()
ARP Input Processing and Replying Function

Syntax

```
void in_arpinput(
    Proto_srvr    psrvr,
    LlMbuf        mb,
    Rominfo       rinf);
```

Description

`in_arpinput()` is called by `arpinput()`. If the ARP request is directed to this function, it caches the sender's hardware address and replies to the request with its hardware address. If the ARP table is full, the request is discarded.

Currently, there is no mechanism to reuse the stale entries. This means requests may be discarded if the table is full.

Parameters

`psrvr`
: points to the `proto_srvr` structure.

`mb`
: points to the packet received.

`rinf`
: points to the `rominfo` structure.

Miscellaneous Functions

Table 5-4. Additional Functions

Function	Description
`in_broadcast()`	Determines if Address is a Broadcast Address

in_broadcast()
Determines If Address Is a Broadcast Address

Syntax

```
int in_broadcast(LLpm_conn conn_entry);
```

Description

`in_broadcast()` determines if the destination address in the `Llpm_conn` pointed to by `conn_entry` is a broadcast address. This does not handle subnetting, however. The function returns a non-zero value if the address is a broadcast address, and a zero value (`SUCCESS`) if not.

Parameters

`conn_entry`
: is not used in the drivers but is present because the protocols also use the same prototypes. This entry point is called by `hlproto`, to turn on/off the interrupts. It is also called by the notification handler routine.

6 Creating a Low-Level Timer Module

This chapter includes the following topics:

Creating the Timer Module

A timer module is required whenever timing services are required. The following list includes examples of when you should use a timer module:

- Low-level network protocols are being used for booting.
- An autobooter has been configured with a specified delay.
- User-state Hawk debugging must be done using low-level communications.
- The high-level driver is `scllio` and it is operating in interrupt driven mode.

Low-level timers are polled instead of interrupt driven. A simple programmable counter is usually adequate. The timer services values are in terms of microseconds, though the counter resolution for a timer does not need to be that small. If the counter resolution is greater than a microsecond, the timer services would have to guarantee at least the specified time had elapsed, perhaps rounding up to the next value given the counter resolution.

It is not generally advisable to use the same device for the system ticker as for the low-level timer. However, under some circumstances, it may be done.

Refer to *Chapter 10, Creating a Ticker* for more information about tickers.

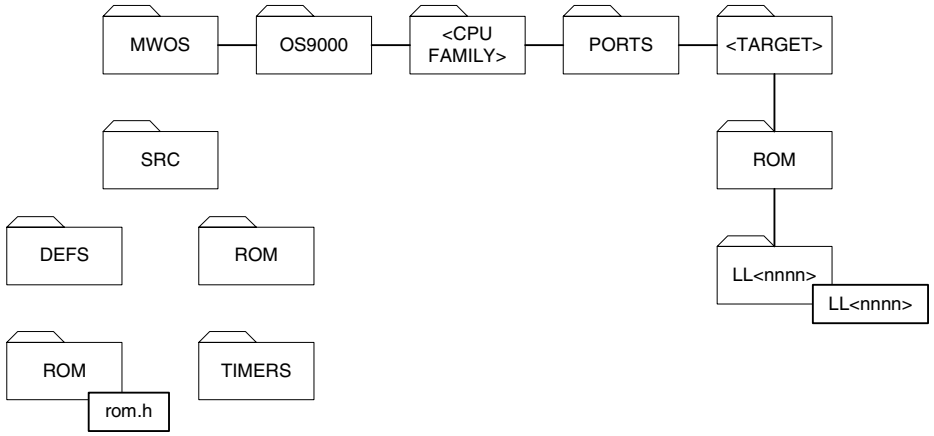
An example of a software timer can be found in the following directory:

`MWOS/SRC/ROM/ TIMERS/SWTIMER`

This example needs to be calibrated to the target platform, given a fixed CPU speed and caching configuration. The software timers have no upper bound on elapsed time, but the specified time must have elapsed. You may be able to configure and use the source code for one of the included example low-level timer modules without modification. If your target has a counter/timer for which no timer module already exists, use the example sources as a guide to write your own timer module.

The low-level timer module sources are in the `MWOS/SRC/ROM/TIMERS` directory. Create a subdirectory for your own source code if you are writing your own timer module. Try to keep your source specific to the particular counter device and not introduce target-specific constants.

Figure 6-1. Creating a New Low-Level Timer Module Directory



In addition to the source directories, each example port directory contains `<Target>/ROM/LL<nnnn>` directories containing makefiles used to build the low-level timer module used in the port. You need to create such a directory and makefile for your timer module in your ports directory. Use the example makefiles as a guide.

The Timer Services Record

A timer module establishes a single timer services record for the system. This record is used to access the services of the timer module and to maintain any necessary state information. The definition of the `tim_svcs` record is in the header file, `MWOS/SRC/DEFS/ROM/rom.h` as follows:

Timer Services Record Example

```
typedef struct tim_svcs {
    idver      infoid;                 /* id/version for tim_svcs */
    error_code (*timer_init)(Rominfo); /* initialize the timer */
    void       (*timer_set)(Rominfo, u_int32);
                                        /* set timeout value & start */
    u_int32    (*timer_get)(Rominfo);
                           /* get time left, zero = expired*/
    void       (*timer_deinit)(Rominfo);
                                        /* de-initialize timer */
    void       *timer_data;            /* local data structure */
    u_int32    rom_delay;   /* delay loop counter, 1us delay */
    int        reserved;  /* reserved for emergency expansion*/
} tim_svcs, *Tim_svcs;
```

Low-Level Timer Module Services

The following entry points describe the services required of each low-level timer module.

Table 6-1. Timer Module Entry Points

Function	Description
`timer_deinit()`	De-initialize timer
`timer_get()`	Get the Time remaining
`timer_init()`	Initialize the timer
`timer_set()`	Arm the timer

timer_deinit()
De-initialize Timer

Syntax

```
void timer_deinit(Rominfo rinf);
```

Description

Deactivate the timer.

Parameters

`rinf`
: points to the `rominfo` structure.

timer_get()
Get the Time Remaining

Syntax

```
u_int32 timer_get(Rominfo rinf);
```

Description

Determine the amount of time remaining. If the time-out has elapsed, stop the counter and return a zero value.

Parameters

`rinf`
: points to the `rominfo` structure.

timer_init()
Initialize Timer

Syntax

```
error_code timer_init(Rominfo rinf);
```

Description

Initialize the hardware for operation. Ensure the timer is not already initialized.

Parameters

`rinf`
: points to the `rominfo` structure.

timer_set()
Arm the Timer

Syntax

```
void timer_set(
    Rominfo       rinf,
    u_int32       timeout);
```

Description

Begin timing with the `timeout` value specified. Set the counter to the corresponding value.

Parameters

`rinf`
: points to the `rominfo` structure.

`timeout`
: is the value at which to begin the countdown.

Starting the Low-Level Timer Module

During the early stages of system bootup, the bootstrap code searches for and starts low-level system modules included in the boot image. The start-up entry point for the low-level system modules is supplied in a relocatable (`.r`) file in the distribution. This entry point branches to the C function `p2start()` you need to provide for your timer module. The start-up routine should perform these tasks:

1. Ensure no other timer module has been installed.
2. Allocate and initialize the timer services record. Allocation may be done passively by defining the timer services record as a module global variable.
3. Make the entry points for its services available through the timer services record.
4. Allocate and initialize any device-specific data structure.
5. Install the timer services structure into the `rominfo` record.

Building the Low-Level Timer Module

Create the makefile for your timer module in a properly named subdirectory of your port's ROM directory. Use the makefiles from the example ports as a guide.

Complete the following steps to add your low-level timer module to the system:

Step 1. Edit the `makefile` file in `<Target>/ROM`.

Step 2. Add your timer directory name to the list of directory names used to define the `TRGTS` macro.

Step 3. Add the timer module name into the `coreboot.ml` file in `<Target>/BOOTS/SYSTEMS/PORTBOOT`.

By doing this, you ensure your timer module is rebuilt along with the bootstrap code and the rest of the low-level system modules when:

- `<Target>/ROM/makefile` is invoked and included in the `rom` file
- `<Target>/BOOTS/SYSTEMS/PORTBOOT/makefile` is invoked creating the boot image `coreboot`.

7 Creating an Init Module

This chapter includes the following topics:

Creating an init Module

Init modules are non-executable modules of type `MT_SYSTEM`. An init module contains a table of system start-up parameters. During start-up, init specifies the initial table sizes and system device names, but init is always available to determine system limits. It must be in memory when the system is booting and usually resides in the `sysboot` file or in ROM.

An init module begins with a standard module header. The module header's `m_exec` offset is a pointer to the system's constant table. The fields of this table are shown here and defined in the `init.h` header file. Within the `INIT/default.des` file is a section for the init module variables that need to be modified for a particular system.

Refer to the *OS-9 Device Descriptor and Configuration Module Reference* for a list of the init fields and the procedures for configuring the init module. See your target's board guide for the init modules specific to your board.

Init Macros

The macros defined here override the default macros contained in the file `/MWOS/OS9000/SRC/DESC/init.des`.

The following macros must be set in the `INIT/default.des` file and do not have defaults in the `init.des` file.

Table 7-1. Init Module Override Macros

Name	Description and Example
`INSTALNAME`	A processor-specific character string used by programs such as `login` to identify the system type. `#define INSTALNAME` "Motorola MVME1603" `#define INSTALNAME` "PC-AT Compatible 80386"
`TICK_NAME`	A processor-specific character string identifying the tick module name. The tick module handles the periodic interrupts for OS-9`s time slicing and internal timings. `#define TICK_NAME` "tk1603" `#define TICK_NAME` "tk8253"
`RTC_NAME`	A character string identifying the real time clock module name. `#define RTC_NAME` "tk8253"
`SYS_START`	A character string identifying the name of the first process to start after the system boots. `#define SYS_START` "CMDS/shell"
`SYS_PARAMS`	A character string containing the parameters to be passed to the first process. `#define SYS_PARAMS` "\n"

Table 7-1. Init Module Override Macros (Continued)

Name	Description and Example
CONS_NAME	A character string identifying the console terminal descriptor module name. `#define CONS_NAME "/term"`
SYS_DEVICE	A character string identifying the initial mass storage device descriptor module name. This must be defined, but can be a null string if none exists. `#define SYS_DEVICE`

Optional Macros

The following describes macros that can be modified. These macros do not have to be included in the `INIT/default.des` file because they have default values defined in `init.des`. However, if your first port does not include a ticker (explained in *Chapter 10, Creating a Ticker* and *Chapter 11, Selecting Real-Time Clock Module Support*) then you should define the `COMPAT` macro with a value made up of at least the `B_NOCLOCK` flag.

Table 7-2. Init Module Optional Macros with Default Values

Name	Description and Example

MPUCHIP	A processor-specific number identifying the MPU chip; for example, 403, 603 or 80386.
	`#define MPUCHIP 603`
	`#define MPUCHIP 80386`

OS_VERSION	A number defining the version of the operating system. Default value is the currently shipped version. `#define OS_VERSION 2 /* version 2.x */`
OS_REVISION	A number defining the revision of the operating system. Default value is the currently shipped revision. `#define OS_REVISION 0 /*rev. x.0*/`

OS9K_REVSTR	A processor-specific character string identifying the operating system.
	`#define OS9K_REVSTR "OS-9000/PowerPC(tm)"`
	`#define OS9K_REVSTR "OS-9000 V2.1 for Intel x86"`

SITE	A customer defined number. An example of the use of this number would be to denote the location where the operating system was installed. Default value is `0`. `#define SITE 0`
PROCS	A number specifying the initial number of entries in the process table. Must be divisible by 64. Default value is `64`. `#define PROCS 64`
PATHS	A number specifying the initial path table size. Must be divisible by 64. Default value is `64`. `#define PATHS 64`

Table 7-2. Init Module Optional Macros with Default Values (Continued)

Name	Description and Example
SLICE	Is the number of clock ticks for each process' time slice. The actual duration for a time slice is this number times the tick rate. Default value is 2. `#define SLICE 2`
SYS_PRIOR	A number defining the priority of the initial process. Default value is 128. `#define SYS_PRIOR 128`
MINPTY	A number defining the system minimum executable priority. Default value is 0. `#define MINPTY 0` Refer to the *OS-9 Technical Manual* for a explanation of priority.
MAXAGE	A number defining the system maximum age. Default value is 0. `#define MAXAGE 0`
EVENTS	A number specifying the initial event table size. Must be divisible by 8. Default value is 32. `#define EVENTS 32` Refer to the *OS-9 Technical Manual* for an explanation of priority.

COMPAT	The compat word contains bit flags that are configuration parameters for the operating system. Default value is 0.
	The init.h file defines the flags that can be used:
	`B_GHOST`
	Do not retain ghost (sticky) modules if set
	`B_WIPEMEM`
	Patternize allocated and returned memory if set
	`B_NOCLOCK`
	`Do not automatically set system clock`
	`B_EXPTBL`
	Do not automatically expand system tables
	`B_UDATMOD`
	Align user-state data modules to MMU page boundary, if SSM is currently in use
	`B_NOCRC`
	Disable the validation of the CRC for new modules.
	`#define COMPAT B_WIPEMEM | B_GHOST`

Table 7-2. Init Module Optional Macros with Default Values (Continued)

Name	Description and Example
EXTENSIONS	A character string containing the names of OS-9 extension modules executed as the system is booting and after the OS-9 I/O system has been initialized. These modules do not need to be present in the boot file but are executed if present. OS-9 system modules provided are: `cache` provides cache enabling and flushing. `fpu` provides software floating point math, if necessary. `ssm` provides memory protection. `#define EXTENSIONS "OS9P2 ssm"`
PREIOS	A character string containing the names of the OS-9 extension modules to be executed prior to the initialization of the OS-9 I/O system. `#define PREIO` “picirq”
IOMAN_NAME	A character string identifying the name of the module handling I/O system calls. `#define IOMAN_NAME “ioman”`
SYS_TIMEZONE	A number specifying the local time zone in minutes from Greenwich Mean Time. Default value is `0`. `#define SYS_TIMEZONE 0`
MAX_SIGS	A number specifying the maximum number of signals that can be queued for a process at any given time. Default value is `32`. `#define MAX_SIGS 32`
MEMLIST	The offset to “colored” memory list. `#define MEMLIST memlist`
MEMTBL	The colored memory list. `#define MEMTBL`

8 Creating PIC Controllers

This chapter includes the following topics:

Reviewing the PowerPC Vector Code

The vector code information discussed in this section relates to PowerPC processors only.

Architecture

The PowerPC vector code consists of a table of 256- byte entries, one for each vector. Each entry contains the exception handling code for that vector. When an exception occurs, the processor saves the current program counter (PC) and the current machine state register (MSR), then transfers control to the appropriate vector. The PC is loaded with the address of the vector and the MSR has the same value as before except that the applicable exception and address translation enable bits are cleared. Consult the user manual for your hardware platform for specific exception processing information.

OS-9 Vector Code Service

The standard OS-9 PowerPC vector code is located in the following directory:

`MWOS/OS9000/SRC/SYSMODS/VECTORS`

This directory is divided into two categories of service, the external interrupt code and the general exception code. The main difference in the two sets of vector code is the software stack used when the high-level C code exception handler is called.

The IRQ vector code saves the current state on the current process system stack and then switches to a dedicated IRQ service stack. If the system was already in an IRQ context, the dedicated IRQ stack and the current system stack are the same and the vector code does not change the stack. The interrupt service code continues to use the IRQ stack. The general exception code uses the current process system stack throughout the context of the exception.

The standard exception handlers save registers `r0-r14`, `lr`, `ccr`, `ctr`, `xer`, `srr0`, and `srr1`. Both exception handlers use the same registers to save the context of the system and dispatch to the appropriate high-level handler. The vector code associated with the system call vector functions similar to the general exception vector code, except the system call vector code does not change the value of `r3` (as stated in the `r3` definition in this section) prior to calling the high-level exception handler.

The following list describes the important register usage in the handlers:

`sprg0` Prior to the exception, the `sprg0` register contains a pointer to the kernel's global static storage area. The software IRQ stack is located just below the kernel’s globals.

`sprg1` Prior to the exception, the `sprg1` register contains a pointer to the top of the current processes system state stack.

`sprg2` This register is used by both categories of handlers as a temporary register for preserving the state of the current process condition codes register.

`sprg3` This register is also used by both categories of handlers as a temporary register for saving the current stack pointer.

`r1` Upon calling the high-level C code exception handler, `r1` contains the stack pointer for use for the duration of the exception handling. It is also pre-decremented by eight bytes to account for the stack space required by the C code handler to save the content of the link register and the current value of the stack pointer. The Ultra C/C++ compiler normally allocates these eight bytes for subroutine calls.

The OS-9 operating system assumes the r1 register points to software stack. If the exception is coming from user-state, r1 is assumed to point to the current process user-state stack. If the exception is coming from system state, then r1 is assumed to point into either the IRQ stack or the current process system-state stack.

`r2` Upon calling the high-level C code exception handler, `r2` points to the static storage area associated with the handler. This is the same static storage pointer specified in the `F_IRQ` service request used to install the exception handler.

`r3` This register, like `r2`, also contains the pointer to the exception handler's static storage area specified in the `F_IRQ` service request. This is true for all of the exception handlers except the system call vector code. This handler leaves `r3` unchanged because it is assumed to hold a pointer to the service requestor's parameter block.

`r4` This register, for all of the exception handlers, contains a pointer to the short stack generated by the vector table code. It contains the partially saved state of the processor at the time of the exception. The complete content of this stack is described in the `regppc.h` header file (located in the `MWOS/OS9000/PPC/DEFS` directory).

This stack image is passed as a parameter to the target C code exception handler to allow handlers to gain access to the conditions of the exception if necessary. If additional registers other than the ones saved in the short stack are to be modified by the exception handler, then the handler must save the content of those registers prior to modification. However, the format of the short stack cannot be modified.

`r5` This register contains the vector number of the exception that just occurred. It is passed as a parameter to the exception handler, which may be useful to the handler.

`lr` The link register is used by the exception handlers to dispatch to the target C code exception handler. The C code handler is called using the `blrl` instruction so the link register is updated with the return address to the vector exception handler. The C code handler then saves the current link register value on the stack in the eight-byte location allocated by the vector code. The return code of the handler restores the link register and returns to the vector code.

The C compiler, by default, generates code to save registers `r14-r31`, if the code generated uses any of the registers.

Initialization

The vectors are initialized twice during the full booting sequence. The first initialization occurs during the low-level or bootstrap booting process.

The OS-9 low-level boot code initializes the vectors so it can catch any exceptions that may occur during this portion of the booting sequence. The second and final initialization occurs during the high-level or kernel's boot stage. Here the kernel links to the `vectors` module and calls its execution offset entry point (where the vectors initialization code resides). The vectors are initialized by copying the exception code from the `vectors` module into each 256-byte vector table entry. Each block of the vectors code has a unique label associated with the first and last instruction of the code. These labels are used by the initialization code to copy the vectors code into the vector table entries requiring that block of code.

In addition to copying the vector code into the tables, there are usually three other operations the initialization code must perform for most of the vectors. There may be other initialization requirements dictated by the complexity of the hardware platform. This description assumes the simplest case and describes what is required of the vector code.

1. The first additional operation is to patch the instruction loading the vector number of the associated vector with an immediate effective address mode. The target instruction is identified with a specific label the installation code can use to calculate the offset to use for the patch operation. The immediate value of the target instruction is modified to contain the vector number passed to the C code handler.
2. Another immediate form load instruction must be patched to contain the value of the offset of the exception table entry structure within the kernel's exception service routine table (also known as the interrupt polling table) for the target vector. Each entry in the exception service routine table is a four-byte pointer to the first of a list of exception table structures associated with the vector as defined by the `excpt.h` header file (in the `MWOS/OS9000/SRC/DEFS` directory).

   The patch value is calculated by adding the offset of the beginning of the kernel's exception service routine table to the vector number multiplied by four. This offset to the exception service routine list for the vector is patched into the vector code in order to save execution time and vector code space in dispatching to the target service routines. Again, each of these patchable instructions can be located within the vector table entry by using the instruction's label to calculate the offset of the instruction within the vector code.
3. The last four-byte word of each vector table is patched with the offset into the associated vector, the location where the OS-9 low-level debugger is allowed to take over the vector. In most cases, this is the location of the actual `blrl` instruction dispatching to the C code handler. The low-level debugger uses this offset value to dynamically patch the vector code to allow itself to monitor exception and breakpoints as instructed.

Interrupt Vector

The vector code for the interrupt vector is typically unique from the code for the other vectors. Since many boards have different external interrupt control mechanisms, the code handling the specifics of the interrupt control is located in an OS-9 extension module specific to the port. In this case the interrupt vector entry contains the usual portion of vector code that switches to the current process' system stack, and saves the current state of all of the volatile CPU registers on the stack. The interrupt code then switches to the system's interrupt stack prior to dispatching to the specialized interrupt controller support code.

Modifying the Interrupt Vector

The standard exception code for each of the vectors performs the same series of state-saving operations. A defined set of registers is preserved on the exception stack prior to calling the C code handler for the given exception. The set of general registers saved is defined by the subroutine calling conventions used by the Ultra C/C++ compiler. The compiler always treats registers `r0`, `r3-r13`, `xer`, `ctr`, `lr`, and `ccr` as volatile registers and register `r1` and `r2` as dedicated registers. The compiler also uses a caller register-save algorithm for subroutine calls.

The calling block must save the current value of any of the volatile registers it wants to preserve because the called subroutine is always allowed to destroy the content of the set of volatile registers. This is why the vector exception handlers preserve these registers. The exception code must be written to assume that the content of all of the volatile registers will be destroyed by the C code handlers.

Because of these conventions, the vector exception handlers are not coded to maximize efficiency but rather to maintain the integrity of the process context state. If for some reason a dedicated application requires a decrease in exception latency from what the standard exception handlers provide, it is possible to modify the vector exception handlers and the C code handlers to achieve these requirements.

The Ultra C/C++ compiler is capable of generating code using a callee register-save subroutine calling convention. In this case, the code is generated to preserve the contents of any of the registers it expects to destroy. This allows the C code exception handlers to be compiled to preserve the content of the registers it uses, making it possible to reduce the burden of context saving required by the vector exception handler.

The vector exception handler can be written to use a bare minimum of registers to dispatch to the C code handler, thus reducing its context save operation to only the set of registers it modifies in getting to the C code handler. While this makes it possible to reduce the latency in servicing an exception, this callee-save convention is not more efficient in the case where many C code handlers reside on the same vector and have to be polled to locate the target handler.

In this case, each of the C code handlers being compiled for the callee-save mode saves and restores all of the registers used in the body of the handler. As each C code handler is called by the dispatcher, it saves and restores multiple registers in determining the need to service the exception.

By the time the target C code handler is located, many more context preservation and restoration operations may have been performed than in the more general caller register-save compiler convention. This reduced context saving scheme for the vectors code should only be used under controlled circumstances where latency can be kept to a minimum and only one or two C code handlers are associated with a given exception or interrupt.

If this technique is used, there are certain restrictions on kernel and debugger usage, because a short stack frame is expected to be present under these circumstances. These are:

1. The debugger cannot be used to monitor the interrupt exception. However, breakpoints can still be processed within the interrupt service routine because they cause their own exceptions that create the short stack for debugger operations.
2. The interrupt vector code, after receiving control back from the interrupt service routine, would directly return control back to the interrupted thread of execution itself, instead of calling the kernel exit routine.

An exception to this would be if the interrupt service routine changed a task state that would require a task switch (for example, sent a signal). Then the interrupt vector code would have to build a short stack after the interrupt service routine completed, and call the kernel exit routine to cause the task switch to occur.

Interrupt Controller Support

Since the PowerPC architecture defines only one vector entry for interrupt processing, it is typical for a target platform to implement one or more external interrupt controller(s) to control and prioritize multiple interrupts external to the processor. OS-9 allows controlling code for this target-dependent interrupt controller structure to be modularized independently of the standard vector dispatching code and the device drivers.

The controlling code can be divided into two classes, interrupt enable/disable, and interrupt acknowledge/ dispatch. Example interrupt enable/disable functions are implemented in library functions accessible to device drivers.

The interrupt acknowledge and dispatch functions are implemented as system extension modules that install themselves as an interrupt handler on the interrupt vector. An example `picirq` module implements the acknowledge and dispatching code for “8259-like” interrupt controllers.

The dispatching code in an interrupt controller module maps the interrupt line to a “logical interrupt vector” and then searches the interrupt polling table associated with the logical vector for handlers to execute until one of them returns a value other than `EOS_NOTME` in register `r3`. Device drivers would then register the interrupt service routine on the logical interrupt vector (during device initialization) instead of the physical vector.

Another example `vmeirq` module implements the acknowledge and dispatching code for a VMEchip2/vmepci bridge chip set used on the MVME1603 reference target. The interrupts from the VMEchip2 are run through the bridge chip and cascaded into the main interrupt controller. As a result, the `vmeirq` module installs its acknowledge and dispatching handler on one of `picirq`'s logical interrupt vectors. Device drivers servicing the VME interrupts then install their handlers on the logical vectors serviced by `vmeirq`. This demonstrates how cascaded interrupt controllers of differing types can be supported. The interrupt controller module required for your port should be added to the `PREIO` extension list of the `init` module.

9 Using Hardware-Independent Drivers

This chapter includes the following topics:

Simplifying the Porting Process

An OS-9 driver module is required for your console device. If the Microware-supplied serial drivers include a driver based on the same device your target platform uses, you only need to set up the proper configuration labels for the device within the `systype.h` file. The Microware-supplied driver and driver-specific descriptor sources are located in the following direcotry:

`MWOS/OS9000/SRC/IO/SCF/DRVR`

SCF Driver (scllio)

`scllio` is defined as having all of the following characteristics:

- a high-level OS-9 SCF driver satisfying I/O requests by calling into a low-level serial driver
- not associated with any particular hardware device
- a port-independent module

In addition, all hardware specific operations are performed by the low-level driver called by `scllio`.

`scllio` can be in polled input or interrupt driven input modes. The `v_pollin` flag in the device descriptor controls the input mode. Output is always polled. As a result, it is not intended that `scllio` replace a high-level serial driver targeting the specific serial port. Instead, `scllio` is designed to be a useful tool in the porting process.

Once the low-level driver has been written, and a RomBug prompt achieved, `scllio` can be configured as the high-level console to bring the system up to a shell prompt before a proper high-level driver is completed. `scllio` is also useful for initial testing of the polled-interrupt mode of a low-level driver. Polled interrupt support is necessary if the low-level driver is to be used to support Hawk communication. This mode is also used by `scllio` when in interrupt driven mode. This is a less complex use of the polled interrupt mode of the low-level driver. `scllio` can be used to test this mode without involving the various network layers of the TCP/IP communications stack.

The low-level device driver called by `scllio` is specified in the device descriptor used with `scllio`. The device descriptor field `v_llconsname` points to a string containing the abbreviated name (the `cons_abname` field of the console device record) of the low-level console you want `scllio` to communicate through. Two special name strings, `consdev` and `commdev`, can be used in this field to specify, respectively, the configured system console device and communication port. These allow generic specification of a high-level console and communication port based strictly on low-level configuration. The reference platform in OS-9 for Embedded Systems contains an example device descriptor for use with `scllio`.

Virtual Console (iovcons)

The low-level virtual console driver, `iovcons`, appears to the caller to be a standard low-level serial driver. Unlike standard serial-drivers, however, `iovcons` does not communicate with a serial hardware device. Instead `iovcons` transfers I/O requests to the low-level communication modules (TCP/IP stack) in the same way daemons supporting the Hawk debugger do. The configuration of the low-level communication system determines whether the output device used is an Ethernet port or SLIP operating over a serial device.

`iovcons` provides a `telnetd`-like interface to the low-level system console. You can `telnet` to the target processor board to obtain a TCP/IP connection over which the OS-9 boot messages and RomBug input/output occurs. This removes the need for a direct serial connection to the target by providing a remote console.

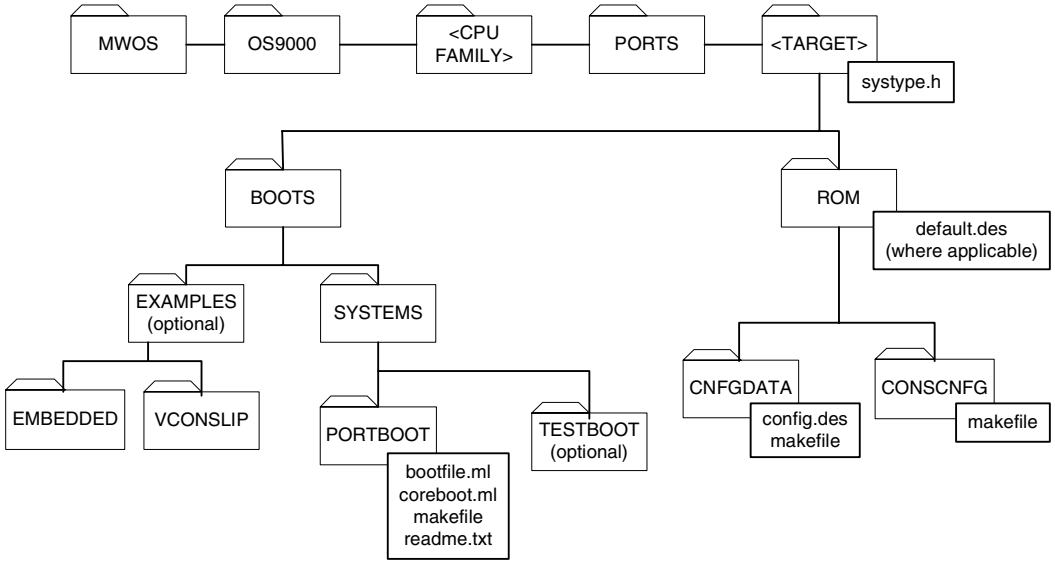
Configuration

Since `iovcons` relies on the low-level networking modules, it must be initialized after these modules in the boot sequence. As a result, the low-level module list used to build the system must be ordered so references to `iovcons` and `conscnfg` appear after the references to the networking modules. The following excerpt from an example `bootfile.ml` file illustrates the required ordering.

```
* Console modules
../../../PPC/CMDS/BOOTOBJS/ROM/console
CMDS/BOOTOBJS/ROM/iosmc
CMDS/BOOTOBJS/ROM/commcnfg
*
* Communications protocol modules
../../../PPC/CMDS/BOOTOBJS/ROM/protoman
../../../PPC/CMDS/BOOTOBJS/ROM/lltcp
../../../PPC/CMDS/BOOTOBJS/ROM/llip
../../../PPC/CMDS/BOOTOBJS/ROM/llslip
*
* Virtual Console
../../../PPC/CMDS/BOOTOBJS/ROM/iovcons
CMDS/BOOTOBJS/ROM/conscnfg
```

The `conscnfg` module looks to the console record in the `cnfgdata` module to determine which low-level driver should be installed as the system console driver. To configure `iovcons` as your system console, declare `VirtualConsole` as the name of your console device in the console port declarations section of `config.des`.

Figure 9-1. scllio and iovcons Directories and Files



The console device record name is specified in the `config.des` (or `default.des`, where applicable) file in the `CNFGDATA` directory. All other fields of the console record are ignored by `iovcons`. The following excerpt from `config.des` provides an example of how to declare a macro:

```
/* Console port */

#define CONS_NAME VirtualConsole
```

10 Creating a Ticker

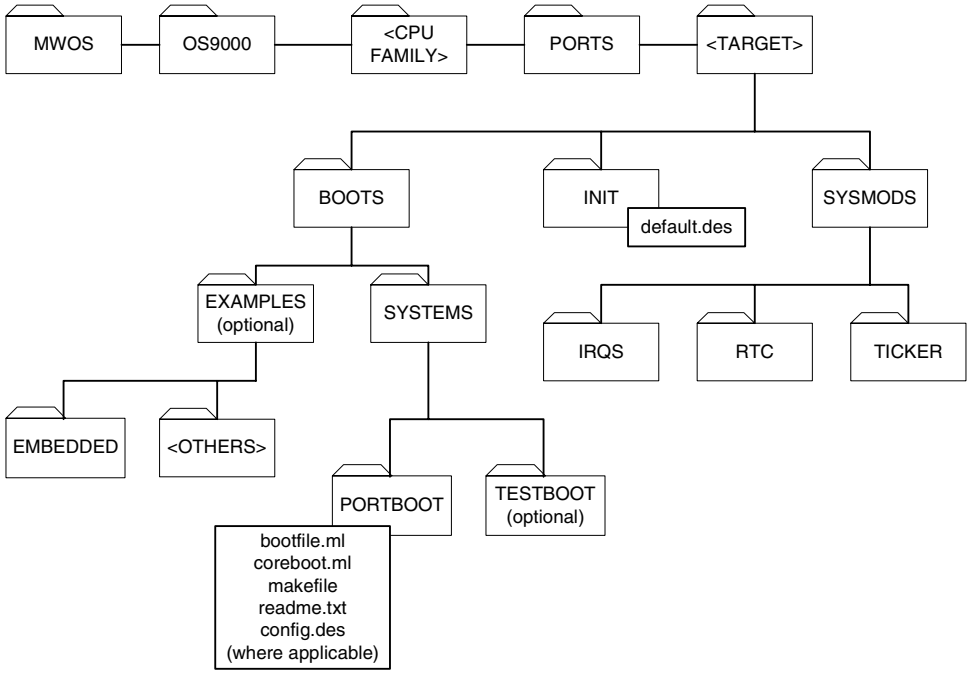
This chapter includes the following topics:

Guidelines for Selecting a Tick Interrupt Device

The interrupt level associated with the timer should be as high as possible. A high interrupt level prevents ticks from being delayed and/or lost due to interrupt activity from other peripherals. Lost ticks cause the kernel's time-keeping functions to lose track of real-time. This can cause a variety of problems in processes requiring precise time scheduling.

The interrupt service routine associated with the timer should be able to determine the source of the interrupt and service the request as quickly as possible.

Figure 10-1. Ticker Setup Directories and Files



Ticker Support

The tick functions for various hardware timers are in the `TICKER` directory.

There are two ticker routines:

- tick initialization entry routine

  This routine is called by the kernel and enables the timer to produce interrupts at the desired rate.

- tick interrupt service routine

  This routine services the tick timer interrupt and calls the kernel's clock service routine.

The ticker module name is user-defined and should be included in the `init` module.

OS-9 Tick Time Setup

You can set the tick timer rate to suit the requirements of the target system. You should define the following variables:

- Ticks per Second
  This value is derived from the count value placed in the tick timer's hardware counter. It reflects the number of tick timer interrupts occurring each second. Most systems set the tick timer to generate 100 ticks per second, but you can vary it. A slower tick rate makes processes receive longer time slices, making multitasking appear sluggish. A faster rate may burden the kernel with extra task-switching overhead due to increased rate for swapping of active tasks.
- Ticks per Time Slice
  This parameter is stored in the `init` module's `m_slice` field. It specifies the number of ticks occurring before the kernel suspends an active process. The kernel checks the active process queue and activates the highest priority active task. The `init.des` module sets this parameter to a default value of 2, but this can be modified by defining the `SLICE` macro in the `default.des` file to the desired value.

  ```
  #define SLICE     2   /* ticks per time slice */
  ```

- Tick Timer Module Name
  The name of the tick timer module is specified in the `init` module. Use the `TICK_NAME` macro in the `default.des` file in the `INIT` directory to define this name.

  For example: `#define TICK_NAME "tk8253"`

Tick Timer Activation

You must start the tick timer to allow the kernel to begin multitasking. This is usually performed by the `setime` utility or by an `_os_setime()` system call during the system startup procedures.

When `_os_setime()` is called, it attempts to link to the tick-timer module name specified in the `init` module. If the tick-timer module is found, the module's entry point is called to initialize the tick timer hardware.

An alternative is to clear the `B_NOCLOCK` bit of the compatibility flag in the `init` module. If this bit is cleared, the kernel automatically starts the tick timer during the kernel's cold start routine. This is equivalent to a `setime -s`.

Refer to the *Utilities Reference* manual for information about using `setime` or the *OS-9 Technical Manual* for information about `_os_setime()`.

Debugging the Ticker

The kernel can automatically start the system clock during its coldstart initialization. The kernel checks the `init` module's `m_compat` word at coldstart. If the `B_NOCLOCK` bit is clear, the kernel performs an `_os_setime()` system call to start the tick timer and set the real time.

This automatic starting of the clock can pose a problem during clock driver development, depending on the state of the real-time clock hardware and the modules associated with the tick timer and real-time clock. If the system software is fully debugged, you should not encounter any problems.

If your system has a working tick module, but no real-time clock support, and the `B_NOCLOCK` bit in the init module's `m_compat` byte is clear, the kernel will perform the `_os_setime()` call. The tick timer code will be executed to start the tick timer, but the tick module will return an error because it lacks real-time clock hardware.

The system time will be invalid and time slicing will occur. You can correctly set the real-time once the system is up. For example, you can run `setime` from the startup file or a shell command line.

For more information about debugging the ticker in a system with a real-time clock, refer to *Chapter 11, Selecting Real-Time Clock Module Support*.

11 Selecting Real-Time Clock Module Support

This chapter includes the following topics:

Real-Time Clock Device Support

Real-time clock devices (especially those equipped with battery backup) enable the real time to be set without operator input. OS-9 does not directly support the real-time functions of these devices, although the system tick generator can be a real-time clock device.

The real-time functions of these devices are used with the tick timer initialization. If the system supports a real-time clock, write the tick timer code so the real-time clock is accessed to read the current time or set the time after the ticker is initialized.

Refer to Figure 10-1 for an illustration of the ticker setup directory structure.

Real-Time Clock Support

The real-time clock functions for various real-time clock devices are in the `MWOS/OS9000/SRC/SYSMODS/RTC` directory. The two real-time clock routines are listed below:

- get time

  This routine reads the current time from the real-time clock device.

- set time

  This routine sets the current time in the real-time clock device.

Automatic System Clock Startup

The kernel can automatically start the system clock during its coldstart initialization. The kernel checks the `init` module's `m_compat` word at coldstart. If the `B_NOCLOCK` bit is clear, the kernel performs an `_os_setime()` system call to start the tick timer and set the real time.

This automatic starting of the clock can pose a problem during clock driver development, depending on the state of the real-time clock hardware and the modules associated with the tick timer and real-time clock. If the system software is fully debugged, you should not encounter any problems.

Below are common scenarios and their implications:

1. The system has a working tick module and real-time clock support.
   If the `B_NOCLOCK` bit in the `init` module's `m_compat` byte is clear, the kernel performs the `_os_setime()` call. The tick timer code is executed to start the tick timer running and the real time clock code is executed to read the current time from the device.

   If the time read from the real-time clock is valid, no errors occur and system time slicing and time keeping functions correctly. You do not need to set the system time.

   If the time read from the real-time clock is not valid, the real-time clock code returns an error. This can occur if the battery back-up malfunctions. The system time is invalid, but time slicing occurs. You can correctly set the real time once the system is up.

2. The system does not have a fully functional/debugged tick timer module and/or real-time clock module.

   In this situation, executing the tick and/or real-time clock code has unknown and potentially fatal effects on the system. To debug the modules, prevent the kernel from performing an `_os_setime()` call during coldstart by setting the `B_NOCLOCK` flag in the `init` module's `m_compat` word. This enables the system to come up without the clock running. Once the system is up, you can debug the clock modules as required.

Debugging Disk-Based Clock Modules

Do not include clock modules in the bootfile until they are completely debugged. Use the following steps to debug the clock modules:

Step 1. Make the `init` module with the `B_NOCLOCK` flag in the `m_compat` byte set.

Step 2. Exclude the modules to be tested from the bootfile.

Step 3. Apply power to the system.

Step 4. Load the tick/real-time clock modules explicitly.

Step 5. Use the system-state debugger or a ROM debugger to set breakpoints at appropriate places in the clock modules.

Step 6. Run the `setime` utility to access the clock modules.

Step 7. Repeat steps three through six until the clock modules are operational.

Use the following steps to include the clock modules when they are operational:

1. Remake the `init` module so the `B_NOCLOCK` flag is clear.
2. Remake the bootfile to include the new `init` module and the desired clock modules.
3. Reboot the system.

Debugging ROM-Based Clock Modules

There are two possible scenarios for ROM-based systems:

- If the system boots from ROM and has disk support, exclude clock modules from the ROMs until they are fully debugged. They can be debugged in the same manner as for disk-based systems.
- If the system boots from ROM and does not have disk support, exclude the clock modules from the ROMs and download them into special RAM until they are fully debugged. Downloading into RAM is required so you can set breakpoints in the modules.

To debug the clock modules, complete the following steps:

Step 1. Make the `init` module with the `B_NOCLOCK` flag in the `m_compat` byte set.

Step 2. Program the ROMs with enough modules to bring the system up, but do not include the clock modules under test.

Step 3. Apply power to the system so that it enters the ROM debugger.

Step 4. Download the modules to test into the special RAM area.

Step 5. Bring up the system.

Step 6. Use the system-state debugger or ROM debugger to set breakpoints at appropriate places in the clock modules.

Step 7. Run the `setime` utility to access the clock modules.

Step 8. Repeat steps 3 through 7 until the clock modules are operational.

When the clock modules are operational, complete the following steps:

Step 1. Remake the `init` module so the `B_NOCLOCK` flag is clear.

Step 2. Remake the bootfile to include the new `init` module and the desired clock modules.

Step 3. Reboot the system.

12 Creating Booters

This chapter includes the following topics:

Creating Disk Booters

After creating and debugging the basic disk driver, you can create a disk booter for the same device. You can use the example disk booters as prototypes.

The basic function of the disk boot routine is to provide the device-specific routines needed to load a bootfile containing the OS-9 system modules.

1. The boot file is established on the disk as a special file by the `bootgen` utility.
2. A target-independent module, `fdman`, is aware of the standard RBF file system layout and is called by the disk boot routine to find the boot file and initiate the transfer.
3. `fdman` then calls back the disk boot routines to accomplish the transfer of specific blocks of data from the disk.

If the device is a SCSI device:

1. The disk boot data transfer routines call the services of a target-independent `scsiman` module to manage the SCSI command protocol.
2. `scsiman` uses the services of a target-specific low-level host-adapter module to manage the transfer across the SCSI bus.

If your target requires a SCSI boot implementation, you need to create a host-adapter module specific to your target, using the example modules as prototypes.

Since the boot system can pass both configured and user parameters to booters, a `parser` module is provided to process the argument lists and place the values in parameter structures accessible from the C language.

The `parser`, `fdman`, `scsiman`, and the host-adapter modules are implemented as *pseudo-booters*. During module startup, they build up a standard boot device record (`bootdev`) with null service pointers and install it onto the list of available booters. Instead of using the `bt_data` field to point to module globals, it points to a pseudo-booter-specific record structure holding pointers to the pseudo-booter's services and any applicable data. The services of `fdman` and `scsiman`, and those required of any SCSI host-adapter are listed in the following sections.

The Boot Device (bootdev) Record and Services

Each booter module establishes one or more boot device records on the list of available boot devices in the Boot Services (`boot_svcs`) record. The definition of the `bootdev` record appears in the header file, `MWOS/SRC/DEFS/ROM/rom.h`, and appears below for illustration.

```
typedef struct bootdev bootdev, *Bootdev;
struct bootdev {
   idver     infoid;
   void     *bt_addr;                 /* the port address */

      /* check for device existence */
      u_int32 (*bt_probe) (Bootdev bdev, Rominfo rinf),

      /* initialize boot device */
      (*bt_init) (Bootdev bdev, Rominfo rinf),

      /* read data from boot device */
      (*bt_read) (u_int32 blks, u_int32 blkaddr, u_char *buff,
         Bootdev bdev, Rominfo rinf),

      /* write data from boot device */
      (*bt_write) (u_int32 blks, u_int32 blkaddr, u_char *buff,
         Bootdev bdev, Rominfo rinf),

      /* terminate the boot device */
      (*bt_term) (Bootdev bdev, Rominfo rinf),

      /* bring boot in from device */
      (*bt_boot) (Bootdev bdev, Rominfo rinf);

      u_int32   bt_flags;               /* misc. flags */
      u_char    *bt_abname,             /* abreviated name */
                *bt_name;               /* full name and description */

      void      *bt_data;              /* special data for boot device */
      Bootdev   bt_next;                /* next device in the list */
      Bootdev   bt_subdev;              /* sub-device record */
      u_char    **user_params;          /* user parameter array */
      u_char    **config_params;     /* configuration parameter array */
      u_char    *config_string;      /* configuration parameter string */
      u_int32   autoboot_delay;       /* autoboot delay time */
      u_int32     bt_reserved[4];   /* reserved for emergency expansion */
};
```

The following entry points describe the services required of each boot device. Pseudo-booters provide none of these services.

Table 12-1. Boot Device Entry Points

Function	Description
bt_boot()	Boot from device
bt_init()	Initialize device
bt_probe()	Probe/verify device
bt_read()	Read data from device
bt_term()	De-initialize device
bt_write()	Write data to device

bt_boot()
Boot From Device

Syntax

```
u_int32 bt_boot(
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

This is the main entry point called by the boot system when this boot device is selected. At this time any parameters can be parsed, and the `bt_init()` service is called. SCSI device booters are likely to call scsiman's `ll_install()` routine to install the host adapter module. Disk booters are likely to follow with a call to fdman's `read_bootfile()` routine described later. Finally, `bt_term()` is be called before returning control back to the boot system.

Parameters

`bdev`
: points to the disk booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

bt_init()
Initialize Device

Syntax

```
u_int32 bt_init(
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

This routine initializes the device as necessary.

Parameters

`bdev`
: points to the disk booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

bt_probe()
Probe/Verify Device

Syntax

```
u_int32 bt_probe(
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

The boot system calls `bt_probe()` to determine if the device is available. Usually, this routine at least confirms a boot area can be returned back to the boot system. Devices with fixed configuration can also be probed to determine if they exist. Devices that can be reconfigured by the user probably cannot determine this at this time, since the return value is used when presenting the boot menu to determine if the device should be marked as available. SCSI device booters are likely to determine if the `scsiman` module is available as part of the probe.

Parameters

`bdev`
: points to the disk booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

bt_read()
Read Data From Device

Syntax

```
u_int32 bt_read(
    u_int32       blks,
    u_int32       blkaddr,
    u_char        *buff,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

This routine causes block reads to occur. For disk booters, it is likely to be called from `fdman` or `scsiman` routines. Otherwise, it would be called from the booter's own `bt_boot()` routine.

Parameters

`blks`
: is the number of blocks to read.

`blkaddr`
: is the address of the block on the media.

`buff`
: points to the buffer in which to store the data.

`bdev`
: points to the disk booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

bt_term()
De-initialize Device

Syntax

```
u_int32 bt_term(
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

This routine deinitializes the device as necessary.

Parameters

`bdev`
: points to the disk booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

bt_write()
Write Data To Device

Syntax

```
u_int32 bt_write(
    u_int32       blks,
    u_int32       blkaddr,
    u_char        *buff,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

This optional routine causes block writes to occur. Currently, it is never called, but the service was defined in case some custom low-level utility required the function of a custom booter.

As with the low-level serial and timer modules, the booter modules are started at a `p2start()` entry point. This entry is responsible for building the necessary `bootdev` records and installing them on the list of available booters. Remember the `portmenu` module services discussed earlier are still required to configure the appropriate booters for autobooting or menu presentation.

Parameters

`blks`
: is the number of blocks to read.

`blkaddr`
: is the address of the block on the media.

`buff`
: points to the buffer in which to store the data.

`bdev`
: points to the disk booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

The parser Module Services

Access to the `parser` module services are through the `paman_svcs` structure defined in `MWOS/SRC/DEFS/ROM/parse.h`.

Table 12-2. paman_svcs Functions

Function	Description
`getnum()`	Convert numeric string to value
`parse_field()`	Parse keyword equals value string from key table entry

getnum()
Convert Numeric String To Value

Syntax

```
u_int32 getnum(char *p);
```

Description

`getnum()` converts the numeric string pointed to by the `p` parameter into a value and returns it.

Parameters

`p`
: points to the numeric string.

parse_field()

Parse Keyword Equals Value String From Key Table Entry

Syntax

```
u_int32 parse_field(
    char          *argv,
    u_int32       *s,
    char          *kf,
    int           ktflag,
    int           j,
    Rominfo       rinf);
```

Description

`parse_field()` compares the string pointed to by the `kf` parameter and the keyword portion of the string (before the equal sign) pointed to by `argv`. If the two are not equal, the service returns `FALSE`. If they are equal and the `ktflag` value is 1, the service places the pointer of the value portion of the string (after the equal sign) into `s[j]`. If they are equal and the `ktflag` is zero, `parse_field` places the converted numeric value of the value portion of the string (after the equal sign) into `s[j]`. Generally, this service is called within a booter's loop, incrementing through each potential parameter that a booter can recognize.

Parameters

`argv`
: points to the keyword.

`s`
: points to the value portion of the string.

`kf`
: points to the compare string.

`ktflag`
: is a flag.

`j`
: is an incrementer.

`rinf`
: points to the `rominfo` structure.

The fdman Module Services

Access to the `fdman` module services are through the `fdman_svcs` structure defined in `MWOS/SRC/DEFS/ROM/fdman.h`.

Table 12-3. fdman_svcs Functions

Function	Description
`fdboot()`	Validate bootfile
`get_partition()`	Locate bootable partition
`read_bootfile()`	Read bootfile from device

fdboot()
Validate Bootfile

Syntax

```
error_code fdboot(
    u_char        *addr,
    u_int32       size,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`fdboot()` scans a loaded image to determine the validity of a bootfile.

Parameters

`addr`
: is the address of the loaded image.

`size`
: is the address of the loaded image.

`bdev`
: points to the disk booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

get_partition()
Locate Bootable Partition

Syntax

```
error_code get_partition(
    u_int32       lsnoffs,
    u_int8        pari_start,
    u_int8        pari_end,
    u_char        *sect0,
    u_int32       *offs,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`get_partition()` finds the first bootable partition on the disk within the specified partition range.

Parameters

`lsnoffs`
: is the original logical sector offset of the drive.

`pari_start`
: is the starting partition number to scan.

`pari_end`
: is the ending partition number to scan.

`sect0`
: points to the sector zero buffer.

`offs`
: points to the partition offset pointer.

`bdev`
: points to the disk booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

read_bootfile()
Read Bootfile From Device

Syntax

```
error_code read_bootfile(
    u_int32       ssize,
    u_int32       lsnoffs,
    u_int8        pari_start,
    u_int8        pari_end,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`read_bootfile()` attempts to read in the first bootfile found on the disk within the specified partition range.

Parameters

`ssize`
: is the sector size of the disk.

`lsnoffs`
: is the original logical sector offset of the drive.

`pari_start`
: is the starting partition number to scan.

`pari_end`
: is the ending partition number to scan.

`bdev`
: points to the disk booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

The scsiman Module Services

Access to the scsiman module services are through the scsi_svcs structure defined in MWOS/SRC/DEFS/ROM/scsiman.h.

Table 12-4. scsi_svcs Functions

Function	Description
da_execnoxfer()	Execute a SCSI command without data transfer
da_execute()	Execute a SCSI command with data transfer
init_tape()	Initializes a sequential device
initsccs()	Initializes a direct access device
ll_install()	Install a low-level SCSI host adapter module
readsccs()	Reads a direct access device
rewind_tape()	Rewinds a sequential device
sq_execnoxfer()	Execute a SCSI command without data transfer
sq_execute()	Execute a SCSI command with data transfer

da_execnoxfer()
Execute a SCSI Command Without Data Transfer

Syntax

```
error_code da_execnoxfer (
    u_int32       opcode,
    u_int32       blkaddr,
    u_int32       bytcnt,
    u_int32       cmdopts,
    u_int32       cmdtype,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`da_execnoxfer()` issues a command to direct access devices.

Parameters

`opcode`
: is the SCSI command code.

`blkaddr`
: is the direct access device block address.

`bytcnt`
: is the size of the data transfer in bytes.

`cmdopts`
: are option flags (booters should use 0).

`cmdtype`
: indicates the type of command, standard or extended (`CDB_STD` or `CDB_EXT`).

`bdev`
: points to the booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

da_execute()
Execute a SCSI Command With Data Transfer

Syntax

```
error_code da_execute (
   u_int32      opcode,
   u_int32      blkaddr,
   u_int32      bytcnt,
   u_int32      cmdopts,
   u_char       *buff,
   u_int32      xferflags,
   u_int32      cmdtype,
   Bootdev      bdev,
   Rominfo      rinf);
```

Description

`da_execute()` issues a command to direct access devices and manages the subsequent data transfer.

Parameters

`opcode`
: is the SCSI command code.

`blkaddr`
: is the direct access device block address.

`bytcnt`
: is the size of the data transfer in bytes.

`cmdopts`
: are option flags (booters should use 0).

`buff`
: points to the data buffer.

`xferflags`
: specifies the data transfer direction (`INPUT` or `OUTPUT`).

`cmdtype`
: indicates the type of command, standard or extended (`CDB_STD` or `CDB_EXT`).

`bdev`
: points to the booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

init_tape()
Initializes a Sequential Device

Syntax

```
error_code init_tape(
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`init_tape()` initializes a sequential device for subsequent access.

Parameters

`bdev`
: points to the booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

initsccs()
Initializes a Direct Access Device

Syntax

```
u_int32 initsccs(
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`initsccs()` initializes a direct access device for subsequent access.

Parameters

`bdev`
: points to the booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

ll_install()
Install a Low-Level SCSI Host Adapter Module

Syntax

```
error_code ll_install(
    char          *name,
    u_int8        *portaddr,
    u_int8        selfid,
    u_int8        reset,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`ll_install()` installs the low-level SCSI host-adapter module. The port address, `selfid` and `reset` values are placed into the appropriate `llscsi_svcs` record and the host-adapter's `ll_init()` routine is called.

Parameters

`name`
: points to the name of the host-adapter module.

`portaddr`
: is the address of the SCSI port.

`selfid`
: is the host adapter's SCSI identification

`reset`
: is a flag to indicate if the host adapter should reset the SCSI bus or not.

`bdev`
: points to the booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

readsccs()
Reads a Direct Access Device

Syntax

```
u_int32 readsccs(
    u_int32       numsects,
    u_int32       blkaddr,
    u_char        *buff,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`readsccs()` reads data from a direct access device.

Parameters

`numsects`
: is the number of blocks to transfer.

`blkaddr`
: is the direct access device block address.

`buff`
: points to the data buffer.

`bdev`
: points to the booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

rewind_tape()
Rewinds a Sequential Device

Syntax

```
error_code rewind_tape(
   Bootdev      bdev,
   Rominfo      rinf);
```

Description

`rewind_tape()` positions a sequential device to the beginning of information.

Parameters

`bdev`
points to the booter's `bootdev` record.

`rinf`
points to the `rominfo` structure.

sq_execnoxfer()
Execute a SCSI Command Without Data Transfer

Syntax

```
error_code sq_execnoxfer(
   u_int32      opcode,
   u_int32      blkcount,
   u_int32      opts,
   u_int32      action,
   Bootdev      bdev,
   Rominfo      rinf);
```

Description

`sq_execnoxfer()` issues a command to sequential devices.

Parameters

`opcode`
: is the SCSI command code.

`count`
: is the size of the data transfer in blocks or bytes.

`opts`
: are option flags (booters should use 0).

`action`
: is the immediate state flag.

`bdev`
: points to the booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

sq_execute()
Execute a SCSI Command With Data Transfer

Syntax

```
error_code sq_execute(
   u_int32      opcode,
   u_int32      count,
   u_int32      opts,
   u_int32      action,
   u_char       *buff,
   u_int32      xferflags,
   u_int32      bytemode,
   Bootdev      bdev,
   Rominfo      rinf);
```

Description

`sq_execute()` issues a command to sequential devices and manages the subsequent data transfer.

Parameters

`opcode`
: is the SCSI command code.

`count`
: is the size of the data transfer in blocks or bytes.

`opts`
: are option flags (booters should use 0).

`action`
: is the immediate state flag.

`buff`
: points to the data buffer.

`xferflags`
: specifies the data transfer direction (`INPUT` or `OUTPUT`).

`bytemode`
: indicates if the count is a block or byte count.

`bdev`
: points to the booter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

The SCSI Host-Adapter Module Services

Access to the host-adapter services are through the `llscsi_svcs` structure defined in `MWOS/SRC/DEFS/ROM/scsiman.h`. If a host adapter module requires global variables, a pointer can be kept in the `reserved2` field of the `llscsi_svcs` structure. Each of the following services would need to make `swap_globals()` calls to set the module globals for the duration of the service.

Table 12-5. llscsi_svcs Functions

Function	Description
`llcmd()`	Execute a raw SCSI command
`llexec()`	Execute specified SCSI command
`llinit()`	Initializes host adapter interface
`llterm()`	Terminate host adapter interface

llcmd()
Execute a Raw SCSI Command

Syntax

```
error_code llcmd(
    u_int8        *cmd,
    u_int8        *dat,
    u_int32       drive_id,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`llcmd()` executes the specified SCSI command.

Parameters

`cmd`
: points to a raw SCSI command block.

`dat`
: points to the data buffer.

`drive_id`
: is the target SCSI identification.

`bdev`
: points to the host adapter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

llexec()
Execute Specified SCSI Command

Syntax

```
error_code llexec(
    Scsicmdblk    cmd,
    u_int32       atn,
    u_int32       llmode,
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`llexec()` executes the SCSI command contained in `cmd`. The `adn` and `llmode` fields are passed down to the host adapter module from scsiman. However, the host adapter does not need to honor these fields since using SCSI attention or synchronized transfers during boot is not required.

Parameters

`cmd`
: points to the SCSI command block.

`atn`
: is the attention flag.

`llmode`
: is the mode flag.

`bdev`
: points to the host adapter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

llinit()
Initializes Host Adapter Interface

Syntax

```
error_code llinit(
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`llinit()` initializes the low level SCSI controller for usage. Normally `llinit()` is called by the `scsiman` module through the `ll_install()` service.

Parameters

`bdev`
: points to the host adapter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

llterm()
Terminate Host Adapter Interface

Syntax

```
error_code llterm(
    Bootdev       bdev,
    Rominfo       rinf);
```

Description

`llterm()` terminates usage of the host adapter module. Any memory explicitly allocated for driver usage can be returned at this time.

Parameters

`bdev`
: points to the host adapter's `bootdev` record.

`rinf`
: points to the `rominfo` structure.

Configuration Parameters

Some of the standard configuration parameters recognized by the example booter modules follow. Not all booters support all parameters.

Table 12-6. Standard Configuration Parameters

Keyword Description	Port Address of Interface
port	Address of interface (coded as 0xFF00<bus#> <device/unit#> for autoconfigured PCI devices)
si	Starting partition index number
ei	Ending partition index number
device	Name of low-level SCSI host adaptor module
reset	SCSI reset flag
aux_device	Name of secondary interface
aux_port	Address of secondary interface
debug	Debugging flags for SCSI booters - bit values are: SCSI_DEBUG_CMD 1 SCSI_DEBUG_DATIN 2 SCSI_DEBUG_DATOUT 4 SCSI_DEBUG_MSGIN 8 SCSI_DEBUG_MSGOUT 0x10 SCSI_DEBUG_STATUS 0x20 SCSI_DEBUG_INFO 0x40 When SCSI debug options are employed it is recommended that the SCSI_DEBUG_INFO option be used. This displays useful information as debug information is processed. Debug phases are displayed in the following form: Data Out Phase: {} Data IN Phase: () Command Phase: [] When the SCSI_DEBUG_INFO flag is used, the following message is display at the start of each SCSI command: SCSI Debug Enabled. DO={} DI=() CMD=[] Drive ID: 0

A Core ROM Services

The modularity of the boot code is accomplished by grouping the services into subsets and providing access to these subsets through record structures. This appendix describes the core structures and their services for all target systems, along with the library services available to all modules. The following sections are included:

The rominfo Structure

The rominfo structure is the focal point of all modularized boot code services. It consists of pointers to all the sub-structures, organized by the type of service provided. The definition of the rominfo structure resides in the include file, MWOS/SRC/DEFS/ROM/rom.h, and appears below (simplified) for illustration.

```
typedef struct rominfo {
   idver           infoid;          /* id_version for rominfo */
   Hw_config       hardware;        /* hardware config struct ptr */
   Mem_svcs        memory;          /* memory services struct ptr */
   Rom_svcs        rom;             /* rom services struct ptr */
   Mod_svcs        modules;         /* module services struct ptr */
   Tim_svcs        timer;           /* timer services struct ptr */
   Cons_svcs       cons;            /* console services struct ptr */
   Proto_man       protoman;        /* protocol manager struct ptr */
   Dbg_svcs        dbg;             /* debugger services struct ptr */
   Boot_svcs       boots;           /* boot services struct ptr */
   Os_svcs         os;              /* OS services struct ptr */
   Cnfg_svcs       config;        /* configuration services struct ptr */
   Notify_svcs     notify;         /* notification services struct ptr */
   u_int32         reserved;
} rominfo, *Rominfo;
```

The rominfo structure and all its substructures have an infoid field defined as the type idver:

```
typedef struct idver {
     u_int16     struct_id,       /* structure identifier */
          struct_ver;             /* structure version */
     u_int32     struct_size;     /* allocated structure size */
} idver, *Idver;
```

The infoid field provides identification and version information about the structure. Modules explicitly allocating structures through a rom_malloc call can also use the struct_size subfield to save the actual size of the memory segment allocated. This is useful when actual size differs from the size requested, and for later explicit freeing, where the actual size needs to be known. The version information can be used to determine the existence of added fields as the structures mature from release to release.

Hardware Configuration Structure

The definition of the `hw_config` structure resides in the include file, `MWOS/SRC/DEFS/ROM/rom.h`, and appears below for illustration.

```
typedef struct {
   union hw_config {

      struct cpu68k_config {
         idver  infoid;              /* id/version for hw_config */
         u_int32cc_cputype,          /* specific cpu type */
            cc_fputype,          /* specific fpu type */
            cc_mmutype,          /* specific mmu type */
            cc_intctrltype;      /* interrupt controller type */
         } cpu68k;

      struct cpu386_config {
         idver  infoid;              /* id/version for hw_config */
         u_int32cc_cputype,          /* specific cpu type */
                cc_fputype,          /* specific fpu type */
                cc_intctrltype;     /* interrupt controller type */
         } cpu386;

      struct cpuppc_config {
         idver  infoid;              /* id/version for hw_config */
         u_int32cc_cputype,          /* specific cpu type */
                cc_fputype,          /* specific fpu type */
                cc_intctrltype;     /* interrupt controller type */
         } cpuppc;

      } cpu;

      /* cache flushing routine */
      void   (*flush_cache)(u_int32 *addr, u_int32 size, u_int8 type,
             Rominfo rinf);
      int    reserved;          /* reserved for emergency expansion */

} hw_config, *Hw_config;
```

Of the CPU-specific configuration fields, only `cputype` and `fputype` are currently used. The other fields are provided for future use.

The `flush_cache()` service is provided by a separate module (`flshcach`) that only needs to be installed if caching is available and expected to be active. The debugger and other modules that build code segments at runtime require this service.

flush_cache()
Flush the Caches

Syntax

```
u_int32 flush_cache(
    u_int32       *addr,
    u_int32       size,
    u_int8        type,
    Rominfo       rinf);
```

Description

Flush the specified cache region.

Parameters

`addr`
: points to the region of memory to flush.

`size`
: is the size of the region of memory to flush. If zero, all cache tables are to be flushed.

`type`
: is the type of cache to be flushed (if applicable). The available values are:

- `HW_CACHETYPE_INST` - instruction cache
- `HW_CACHETYPE_DATA` - data cache

`rinf`
: points to the `rominfo` structure.

Memory Services

The definition of the mem_svcs structure is in the include file, MWOS/SRC/DEFS/ROM/rom.h.

```
typedef struct mem_svcs {
    idver        infoid;                 /* id/version for mem_svcs */
    Dumb_mem     rom_memlist;            /* the limited memory list */
    Rom_list    rom_romlist;            /* rom memory list */
    Rom_list    rom_bootlist;           /* boot memory list */
    Rom_list     rom_consumed;         /* memory consumed by roms */
    Rom_list     consumed_next;   /* next free consumed list entry */
    Rom_list     consumed_end;          /* last entry in consumed list */
    u_char       *rom_ramlimit;         /* RAM limit (highest address */
    u_int32      rom_totram;            /* total ram found */

                /* get memory */
       u_int32     (*rom_malloc)(u_int32 *size, char **addr, Rominfo),

                /* free memory */
                     (*rom_free)(u_int32 size, char *addr, Rominfo);

                /* clear memory */
       void        (*mem_clear)(u_int32 size, char *addr);

       u_char      *rom_dmabuff;     /* 64k DMA buffer for >16MB systems */
       int         reserved;         /* reserved for emergency expansion */
} mem_svcs, *Mem_svcs;
```

Most of the fields in the mem_svcs structure are for internal bookeeping within the raw romcore module and the rom_malloc() and rom_free() services. The rom_bootlist entry pointer is used by booters to communicate the location and size of a boot image.

Table A-1.mem_svcs Functions

Function	Description
mem_clear()	Clear memory
rom_free()	Free allocated memory
rom_malloc()	Allocate memory

mem_clear()
Clear Memory

Syntax

```
void mem_clear(
    u_int32       size,
    char          *addr);
```

Description

`mem_clear()` clears memory. Memory allocated from boot memory pools is always cleared.

Parameters

`size`
is the size of the memory region in bytes to clear.

`addr`
is the address of the memory region to clear.

rom_free()
Free Allocated Memory

Syntax

```
u_int32 rom_free(
    u_int32       size,
    char          *addr,
    char          Rominfo);
```

Description

`rom_free()` returns memory to memory pool. If the request is being made after the operating system is up, an `_os_srtmem()` call is made on behalf of the caller.

Parameters

`size`
is the size of the memory region in bytes being returned.

`addr`
is the address of the memory region being returned.

`Rominfo`
is the `rominfo` pointer.

rom_malloc()
Allocate Memory

Syntax

```
u_int32 rom_malloc(
    u_int32       *size,
    char          **addr,
    Rominfo       rinf);
```

Description

`rom_malloc()` allocates memory from the memory pool. If the request is being made after the operating system is up, an `_os_srqmem()` call is made on behalf of the caller.

Parameters

`size`
: points to the size of the memory being requested in bytes. If the size is rounded up to some block size multiple during allocation, the value is adjusted to reflect the actual block size allocated.

`addr`
: points to where the address of memory allocated is returned.

`rinf`
: is the `Rominfo` pointer.

ROM Services

The definition of the `rom_svcs` structure resides in the include file, `MWOS/SRC/DEFS/ROM/rom.h`, and appears below for illustration.

```
typedef struct rom_svcs {

    idver    infoid;                  /* id/version for rom_svcs */
    void     *rom_glbldata,           /* global data pointer */
              *rom_excptjt,            /* exception jump table */
              *rom_initsp;             /* initial stack pointer */
    u_int32  *rom_vectors;            /* the vector table */

    void     (*rom_start)();          /* reset pc */

    u_char   *kernel_extnd;           /* the kernel extension */
    u_char   *debug_extnd;            /* the debugger extension */
    u_char   *rom_extnd;              /* the ROM extension */

    u_int32   (*use_debug)(Rominfo rinf);     /* debugger enable routine */

    char      *rom_hellomsg;          /* hello message pointer */
    int       reserved;              /* reserved for emergency expansion */

       } rom_svcs, *Rom_svcs;
```

Most of the `rom_svcs` fields are informational. The `rom_hellomsg` field enables runtime customization of the first bootstrap message printed to the console. The `use_debug()` services is provided by the `usedebug` module to indicate if the debugger should be activated just prior to the boot system starting the boot process.

Module Services

The definition of the `mod_svcs` structure resides in the include file, `MWOS/SRC/DEFS/ROM/rom.h`, and appears here for illustration.

```
typedef struct mod_svcs {
   idver     infoid;                 /* id/version for mod_svcs */

                  /* init module as P2 */
         u_int32    (*rom_modinit)(u_char *modptr, Rominfo rinf),

            /* deinit module */
            (*rom_moddeinit)(),

            /* insert into list */
           (*rom_modins)(u_char *modptr, Mod_list *mleptr, Rominfo rinf),

            /* delete module from list */
            (*rom_moddel)(u_char *modptr, Rominfo rinf);

            /* find module start ptr */
         void       (*rom_findmod)(u_char *codeptr, u_char **modptr);

            /* find module list entry */
         u_int32     (*rom_findmle)(u_char *modptr, Mod_list *mleptr,
Rominfo rinf);

            /* scan for modules */
         void      (*rom_modscan)(u_char *modptr, u_int32 hdrchk, Rominfo
rinf);

         Mod_list  rom_modlist;          /* low-level module list */
         char      *kernel_name;       /* pointer to kernel name string */

                  /* validate module */
         u_int32    (*goodmodule)(u_char *modptr, u_int32 bootsize,
                  u_int32 *modsize, u_int32 kerchk, Rominfo rinf);

         int       reserved[4];       /* reserved for emergen expansion */
     } mod_svcs, *Mod_svcs;
```

The most commonly used services are `goodmodule()` and `rom_modscan()`.

The `goodmodule()` service is used by most booters to validate the loadfile image. The `rom_modscan()` service is used to extend the runtime configurability of the low-level system modules.

Table A-2. Module Service Functions

Function	Description
`goodmodule()`	Validate bootfile modules
`rom_findmle()`	Find module list entry
`rom_findmod()`	Find beginning of module
`rom_moddeinit()`	De-initialize low-level system modules
`rom_moddel()`	Delete module from module list
`rom_modinit()`	Initialize low-level system modules
`rom_modins()`	Insert module into module list
`rom_modscan()`	Scan for modules

goodmodule()
Validate Bootfile Modules

Syntax

```
u_int32 goodmodule(
    u_char        *modptr,
    u_int32       bootsize,
    u_int32       *modsize,
    u_int32       kerchk,
    Rominfo       rinf);
```

Description

This service validates a bootfile module, optionally checking if the module is the kernel.

Parameters

`modptr`
: is the address of the module.

`bootsize`
: is the size of all modules within the boot image.

`modsize`
: is a pointer to the returned size of the module in bytes (if it is good).

`kerchk`
: is a flag specifying if the module should be checked as the kernel. A non-zero value indicates the module's name must match the kernel name for the service to succeed.

`rinf`
: is a pointer to the `rominfo` structure.

rom_findmle()
Find Module List Entry

Syntax

```
u_int32 rom_findmle(
    u_char        *modptr,
    Mod_list      *mleptr,
    Rominfo       rinf);
```

Description

This service returns the module list entry for the specified module.

Parameters

`modptr`
: points to the low-level system module.

`mleptr`
: points to the returned module list entry pointer.

`rinf`
: points to the `rominfo` structure.

rom_findmod()
Find Beginning Of Module

Syntax

```
void rom_findmod(
    u_char        *codeptr,
    u_char        **modptr);
```

Description

This service scans back from the specified code pointer until it finds a module header.

Parameters

`codeptr`
: points to code within the module.

`*modptr`
: points to the returned address of the module header.

rom_moddeinit()
De-initialize Low-Level System Modules

Syntax

```
u_int32 rom_moddeinit();
```

Description

This service is currently not implemented.

rom_moddel()
Delete Module From Module List

Syntax

```
u_int32 rom_moddel(
    u_char        *modptr,
    Rominfo       rinf);
```

Description

This service deletes a module list entry from the module list and frees it.

Parameters

`modptr`
: points to the low-level system module.

`rinf`
: points to the `rominfo` structure.

rom_modinit()
Initialize Low-Level System Modules

Syntax

```
u_int32 rom_modinit(
    u_char        *modptr,
    Rominfo       rinf);
```

Description

This routine starts the low-level system module.

Parameters

`modptr`
points to a low-level system module.

`rinf`
points to the `rominfo` structure.

rom_modins()
Insert Module Into Module List

Syntax

```
u_int32 rom_modins(
    u_char        *modptr,
    Mod_list      *mleptr,
    Rominfo       rinf);
```

Description

This service allocates a module list entry and inserts it onto the module list.

Parameters

`modptr`
: points to the low-level system module.

`mleptr`
: points to the returned module list entry pointer.

`rinf`
: points to the `rominfo` structure.

rom_modscan()
Scan For Modules

Syntax

```
void rom_modscan(
    u_char        *modptr,
    u_int32       hdrchk,
    Rominfo       rinf);
```

Description

This service scans for contiguous modules starting at the specified address and starts them in order of occurrence. When a module is not found, the scan terminates.

`rom_modscan()` enables low-level system modules to be found in memory regions other than the base ROM area (for example, external ROM or flash, on PCMCIA, Industry Pak, or other bus carriers), and enables them to be configured depending on the presence or absence of that memory region.

Parameters

`modptr`
: is the base address to scan for modules.

`hdrchk`
: is a flag to specify if the module header parity should be checked. If the value is non-zero, the header parity is validated.

`rinf`
: points to the `rominfo` structure.

p2lib Utility Functions

Three libraries are shipped as part of this distribution:

- `p2privat.l`
- `romsys.l`
- `p2lib.l`

The `p2privte.l` and `romsys.l` libraries are only used by the bootstrap code (`romcore`). The `p2lib.l` library contains functions you can use to customize your own low-level system modules. The `p2lib.l` functions are explained in this appendix.

Table A-3.`p2lib.l` Functions

Function	Description
`getrinf()`	Get the rominfo structure pointer
`findrinf()`	Returns or converts a rominfo structure pointer
`hwprobe()`	Check a system hardware address
`inttoascii()`	Convert an integer to ASCII
`os_getrinf()`	Get pointer to rominfo structure after boot
`outhex()`	Display one hexidecimal digit
`out1hex()`	Display a hexidecimal byte
`out2hex()`	Display a hexidecimal word
`out4hex()`	Display a hexidecimal longword
`rom_udiv()`	Unsigned integer division
`setexcpt()`	Install exception handler
`swap_globals()`	Exchange current globals pointer

getrinf()
Get the Rominfo Structure Pointer

Syntax

```
error_code getrinf(Rominfo *rinf_p);
```

Description

`getrinf()` finds and returns the pointer to the `rominfo` structure from the system globals.

Parameters

`rinf_p`
is the address where `getrinf()` stores the pointer to the `rominfo` structure. (Output)

The current globals register needs to be set to point at the system globals when the service is invoked.

findrinf()

Returns or Converts a Rominfo Structure Pointer

Syntax

```
error_code findrinf(Rominfo *rinf_p);
```

Description

`findrinf()` returns a pointer to the `rominfo` structure. If an old (original) style rominfo structure is passed in, findrinf() will convert it to a new style rominfo structure.

Parameters

`rinf_p`
: is the address where `findrinf()` updates the pointer to the `rominfo` structure. (Input/Output)

Possible Errors

EOS_NOROMINFO	missing or invalid rominfo pointer specified

If using the `d_sysrom` system global, `findrinf()` can be used to convert it to a new style rominfo structure pointer. Alternatively, os_getrinf() could be used.

hwprobe()
Check a System Hardware Address

Syntax

```
error_code hwprobe(
    void          *addr,
    u_int32       ptype,
    Rominfo       rinf);
```

Description

`hwprobe()` sets up the appropriate handlers to catch machine check exceptions, and probes the system memory at the specified address, attempting to read either a byte, word, or long. In the event of a machine check, an error is returned. `SUCCESS` is returned if the read is successful.

Parameters

`addr`
: is the specific memory address you want probed. (Input)

`ptype`
: is the probe type, either byte, word, or long.

`rinf`
: points to the `rominfo` structure.

inttoascii()
Convert an Integer To ASCII

Syntax

```
char *inttoascii(
    u_int32       value,
    char          *bufptr);
```

Description

`inttoascii()` converts its input value to its base 10 ASCII representation stored in `bufptr`. The caller must ensure `bufptr` points to a sufficient storage space for the ASCII representation. `inttoascii()` returns `bufptr`.

Parameters

`value`
is the integer value to be converted.

`bufptr`
points to the location where the ASCII value is stored. (Output)

For ARM users:
If you are using the ARMv4 platform, you must link `inttoascii()` with `os_lib.l` in order for the function to work properly.

os_getrinf()

Get Pointer to Rominfo Structure After Boot

Syntax

```
error_code os_getrinf(Rominfo *rinf_p);
```

Description

`os_getrinf()` returns a pointer to the `rominfo` structure. It will only work correctly if the OS-9 kernel system globals have been initialized, otherwise an error is returned.

Parameters

`rinf_p`
: is the address where `os_getrinf()` stores the pointer to the `rominfo` structure. (Output)

Possible Errors

EOS_NOROMINFO	the `d_sysrom` system global is not initialized

outhex()
Display One Hexidecimal Digit

Syntax

```
void outhex(
    u_char        n,
    Rominfo       rinf);
```

Description

`outhex()` displays one hexidecimal digit on the system console. The lower 4 bits of the character `n` are displayed using the `putchar()` service of the system console device.

Parameters

`n`
: is the character for which the hex value is to be displayed.

`rinf`
: points to the `rominfo` structure. (Input)

out1hex()
Display a Hexidecimal Byte

Syntax

```
void out1hex(
    u_char        byte,
    Rominfo       rinf);
```

Description

`out1hex()` displays the hexidecimal representation of a byte on the system console device.

Parameters

`byte`
is the byte for which the hex value is to be displayed.

`rinf`
points to the `rominfo` structure. (Input)

out2hex()
Display a Hexidecimal Word

Syntax

```
void out2hex(
   u_init16      word,
   Rominfo       rinf);
```

Description

`out2hex()` displays the hexidecimal representation of a word on the system console device.

Parameters

`word`
: is the word for which the hex value is to be displayed.

`rinf`
: points to the `rominfo` structure. (Input)

out4hex()
Display a Hexidecimal Longword

Syntax

```
void out4hex(
    u_int32       longword,
    Rominfo       rinf);
```

Description

`out4hex()` displays the hexidecimal representation of a longword on the system console device.

Parameters

`longword`
: is the longword for which the hex value is to be displayed.

`rinf`
: points to the `rominfo` structure. (Input)

out8hex()
Display Hexidecimal Quadword

Syntax

```
void out8hex(unsigned long long quadword,
             Rominfo            rinf);
```

Description

`out8hex()` displays the hexidecimal representation of a quadword (64 bits) on the system console device

Parameters

`quadword`
: is the quadword for which the hex value is to be displayed.

`rinf`
: points to the rominfo structure. (Input)

rom_udiv()
Unsigned Integer Division

Syntax

```
unsigned rom_udiv(
    unsigned      dividend,
    unsigned      divisor);
```

Description

`rom_udiv()` provides an integer division routine that does not rely on the presence of a built-in hardware division instruction.

Parameters

`dividend`
is the number to be divided.

`divisor`
is the number by which the dividend is to be divided.

setexcpt()
Install Exception Handler

Syntax

```
u_int32 setexcpt(
    u_int32       vector,
    u_int32       irqsvc,
    Rominfo       rinf);
```

Description

`setexcpt()` installs an exception handler on the system exception vector table for the specified exception. This is usually used with the `setjmp()` and `longjmp()` C functions to provide a bus fault recovery mechanism prior to polling hardware.

Parameters

`vector`
: is the number of the exception for which the handler should be installed.

`irqsvc`
: points to the exception handling code you want installed.

`rinf`
: points to the `rominfo` structure. (Input)

swap_globals()
Exchange Current Globals Pointer

Syntax

```
u_char *swap_globals(u_char *new_globals);
```

Description

`swap_globals()` replaces the caller's global data pointer with a new value and returns the old value.

Parameters

`new_globals`
: is the value to be assigned to the global data pointer. (Input)

B Optional ROM Services

There are several optional categories of service for a final production boot ROM, which can be implemented according to your desired configuration. Since these services are modularized, they may be left out to conserve required ROM and RAM space, or be included to meet a functional requirement.

This appendix includes the following topics:

Configuration Module Services

The configuration services module, `cnfgfunc`, provides access to data built into the configuration data module. The definition of the `cnfg_svcs` structure resides in the include file, `MWOS/SRC/DEFS/ROM/rom.h`, and appears below for illustration.

```
typedef struct cnfg_svcs {

    idver  infoid;      /* id/version for cnfg_svcs */

    /* configuration service */

    error_code  (*get_config_data)(enum config_element_id id,
                                   u_int32 index, Rominfo rinf, void *buf);

    /* pointer to configuration data module */

    void   *config_data;

    int    reserved;      /* reserved for emergency expansion */

} cnfg_svcs, *Cnfg_svcs;
```

If no low-level system modules require the configuration services, the `cnfgfunc` and `cnfgdata` modules can be omitted.

get_config_data()
Obtain Configuration Data Element

Syntax

```
error_code(
    enum config_element_id id,
    u_int32       index,
    Rominfo       rinf,
    void          *buf);
```

Description

`get_config_data()` returns the value of the configuration element identified by `id` in the caller supplied location specified by `buf`. The following tables list the available identifiers, their definition, and field type/size.

Table B-1.Console Configuration Elements

Configuration Elements	Description	Type/Size
CONS_REVS	structure version	u_int16
CONS_NAME	console name	char *
CONS_VECTOR	interrupt vector number	u_int32
CONS_PRIORITY	interrupt priority	u_int32
CONS_LEVEL	interrupt level	u_int32
CONS_TIMEOUT	polling timeout	u_int32
CONS_PARITY	parity size	u_int8
CONS_BAUDRATE	baud rate	u_int8
CONS_WORDSIZE	Character size	u_int8
CONS_STOPBITS	Stop bit	u_int8
CONS_FLOW	Flow control	u_int8

Table B-2.Debugger Configuration Elements

Configuration Elements	Description	Type/Size
DEBUG_REVS	Structure version	u_int16
DEBUG_NAME	Default debugger client name	char *
DEBUG_COLD_FLAG	Flag the client should be called at cold start, or not	u_int32

Table B-3.Protoman Configuration Elements

Configuration Elements	Description	Type/Size
LLPM_REVS	Structure version	u_int16
LLPM_MAXLLPMPROTOS	Max. # of protocols on protocol stack	u_int16

Table B-3.Protoman Configuration Elements (Continued)

Configuration Elements	Description	Type/Size
LLPM_MAXRCVMBUFS	Number of maximum receive mbuffs	u_int16
LLPM_MAXLLPMCONNS	Max. # of low level protoman connections	u_int16
LLPM_IFCOUNT	Number of hardware config entries	u_int32

Table B-4.Low-Level Network Interface Config Elements

Configuration Elements	Description	Type/Size
LLPM_IF_IP_ADDRESS	IP address	u_int8[16]
LLPM_IF_SUBNET_MASK	Subnet mask	u_int8[16]
LLPM_IF_BRDCST_ADDRESS	Broadcast address	u_int8[16]
LLPM_IF_GW_ADDRESS	Gateway address	u_int8[16]
LLPM_IF_MAC_ADDRESS	MAC (Ethernet) address	u_int8[16]
LLPM_IF_TYPE	Type of hardware interface	u_int8
LLPM_IF_ALT_PARITY	Alternate serial port parity	u_int8
LLPM_IF_ALT_BAUDRATE	Alternate serial port baud rate	u_int8
LLPM_IF_ALT_WORDSIZE	Alternate serial port word size	u_int8
LLPM_IF_ALT_STOPBITS	Alternate serial port stop bits	u_int8
LLPM_IF_ALT_FLOW	Alternate serial port flow control	u_int8
LLPM_IF_FLAGS	Interface flags	u_int16
LLPM_IF_NAME	Name of hardware interface	char *
LLPM_IF_PORT_ADDRESS	Replacement HW interface address	u_int32
LLPM_IF_VECTOR	Interrupt vector number	u_int32
LLPM_IF_PRIORITY	Interrupt priority	u_int32
LLPM_IF_LEVEL	Interrupt level	u_int32
LLPM_IF_ALT_TIMEOUT	Alternate serial port timeout	u_int32
LLPM_IF_USE_ALT	Alternate usage flags	u_int32

Table B-5.Boot System Configuration Elements

Configuration Elements	Description	Type/Size
BOOT_REVS	Structure version	u_int16
BOOT_COUNT	Number of boot system configuration entries	u_int32
BOOT_CMDSIZE	Maximum size of user input string	u_int32

Table B-6.Booter Configuration Elements

Configuration Elements	Description	Type/Size
BOOTER_ABNAME	Abbreviated booter name	char *
BOOTER_NEWAB	Replacement abbreviated name	char *
BOOTER_NEWNAME	Replacement full name	char *

Table B-6.Booter Configuration Elements (Continued)

Configuration Elements	Description	Type/Size
BOOTER_AUTOMENU	Auto/Menu registration flag	u_int8
BOOTER_PARAMS	Parameter string	char *
BOOTER_AUTODELAY	Autoboot delay time (in microseconds)	u_int32

Table B-7.Notification Services Configuration Elements

Configuration Elements	Description	Type/Size
NTFY_REVS	Structure version	u_int16
NTFY_MAX_NOTIFIERS	Maximum number of registered notifiers	u_int32

Console I/O Module Services

The console module provides a high level I/O interface to the entry points of the low-level serial device driver configured as the system console. These services are made available through the console services field of the `rominfo` structure. Assuming the variable `rinf` points to the `rominfo` structure, `rinf->cons` can be used to reference the console services record.

The header file `MWOS/SRC/DEFS/ROM/rom.h` contains the structure definitions for the `rominfo` structure and the console services record, `cons_svcs`.

The console services are required when any of the following conditions are met:

1. Console dialog is required to boot the system (for example, using a boot menu or menus).
2. Local system-state debugging with RomBug is required.
3. The communications port is required to support downloading or remote debugging.

If none of these are required in the final system, the console module, the corresponding low-level serial modules, and the console and communications port configuration modules can be omitted.

The following services are available through the console services record.

Table B-8.Console Services

Function	Description
rom_fprintf()	Write a printf-style string to the system console
rom_getc()	Read the first character
rom_getchar()	Read the first character from the system console
rom_gets()	Read a null-terminated string from the system console
rom_putc()	Output one character
rom_putchar()	Output a character to the system console
rom_puterr()	Write error code to the system console
rom_puts()	Write a null-terminated string to the system console

rom_fprintf()
Write a Printf-style String to the System Console

Syntax

```
void rom_fprintf(Rominfo rinf, char *fmt, ...);
```

Description

`rom_fprintf()` calls the low-level write routine of the console device record configured for use as the system console. `rom_fprintf()` writes the specified printf-style format string to the console device after replacing the printf escapes with the specified variable arguments. The following escapes are recognized:

%%	a single '%' character

%b	bit-field value. This escape consumes two variable arguments: a 32-bit unsigned integer value and a character pointer. The character pointer controls how the integer value is printed. The first character of the string contains the base for printing the integer value. Bases 2 through 16 are supported. The remainder of the string is a series of "bit records" where the first character of the record has a value from 1 to 32 indicating a bit number and the remainder of the record is the mnemonic for the bit. Bits are numbered from least significant (1) to most significant (32). Mnemonics may contain any printable ASCII character except space (' '). If the specified bit number in the integer value is set the mnemonic is printed. For example,
	`cons->rom_fprintf(rinf, "reg = %b\n", reg, "\2\1bit1\2bit2");`
	might print:
	`reg = 1011<bit1,bit2>`

%c	8-bit ASCII character
%d, %i	32-bit signed decimal integer*
%o	32-bit unsigned octal integer*
%p	pointer value. Pointer values are printed in base 16 with a leading "`0x`".*
%s	NUL terminated character string. If the value of the character pointer is NULL, "`(null)`" is printed in its place.
%u	32-bit unsigned decimal integer*
%x, %X	32-bit unsigned hexadecimal integer (always prints lower-case)*

* This escape may also include a field width, using space or zero padding. The field width is specified by the decimal number that appears between the % and format letter. If the first digit of this number is 0, zero padding is enabled. Otherwise, space padding is used. Use zero padding with %p since the 0x is printed and does not count as part of the field.

Illegal escapes are simply printed as part of the output and do not consume any of the variable arguments.

rom_fprintf() can also be used like vprintf(). If fmt is NULL, then the first variable argument is assumed to be the format string and the second variable argument is assumed to be an already started va_list. This allows calling modules to provide a printf()-like wrapper around rom_fprintf(). For example, if a module had a global variable for the ROM info structure (rinf) then the following wrapper could be used to conveniently wrap the call to rom_fprintf(), allowing callers to use the same syntax as printf().

```
#include <ROM/rom.h>
#include <stdarg.h>

extern Rominfo rinf;

void lprintf(char *fmt, ...)
{
    if (rinf->cons && rinf->cons->rom_fprintf) {
        va_list vp;

        va_start(vp, fmt);
        rinf->console->rom_fprintf(rinf, NULL, fmt, vp);
        va_end(vp);
    }
}
```

Parameters

rinf
: points to the rominfo structure.

fmt
: points to the first character of the printf-style string to output.

Example

```
rinf->cons->rom_fprintf(rinf, "value = %x\n", value);
```

rom_getc()
Read the First Character

Syntax

```
char rom_getc(
    Rominfo       rinf,
    Consdev       cdev);
```

Description

`rom_getc`() calls the low-level read routine of the specified console device record to read a single input character from the associated console device.

`rom_getc`() returns the character read.

Parameters

`rinf`
points to the `rominfo` structure.

`cdev`
points to the console device record for the console device to be used.

Example

```
char ch;
ch = rinf->cons->rom_getc(rinf, cdev);
```

rom_getchar()
Read First Character From the System Console

Syntax

```
char rom_getchar(Rominfo rinf);
```

Description

`rom_getchar`() calls the low-level read routine of the console device record configured for use as the system console. `rom_getchar`() reads a character from the console. XON or XOFF characters not processed by the low-level read are ignored.

If echoing is enabled for the console, `rom_getchar`() calls `putchar`() to echo this character. The character is then returned by `rom_getchar`().

Parameters

`rinf`
: points to the `rominfo` structure.

Example

```
ch = rinf->cons->rom_getchar(rinf);
```

rom_gets()
Read a Null-Terminated String From the System Console

Syntax

```
char *rom_gets(
    char          *buff,
    u_int32       count,
    Rominfo       rinf);
```

Description

rom_gets() calls the low-level read routine of the console device record configured for use as the system console. rom_gets() reads a null-terminated string from the console into the buffer designated by the pointer buff. The rudimentary line editing feature of <backspace> is supported by rom_gets().

rom_gets() returns to the caller when it receives a carriage return character (0x0d), or when the number of characters designated by count has been read.

Parameters

buff
: points to the input buffer into which the string is read.

count
: is the integer used as the size of the input buffer including the null termination.

rinf
: points to the rominfo structure.

Example

```
str = rinf->cons->rom_gets(buffer, count, rinf);
```

rom_putc()
Output One Character

Syntax

```
void rom_putc(
    char          c,
    Rominfo       rinf,
    Consdev       cdev);
```

Description

`rom_putc()` calls the low-level write routine of the specified console device record to output a single character to the associated console device.

Parameters

`c`
: is the character to output.

`rinf`
: points to the `rominfo` structure.

`cdev`
: points to the console device record for the console device to be used.

Example

```
rinf->cons->rom_putc(ch, rinf, cdev);
```

rom_putchar()
Output a Character To the System Console

Syntax

```
void rom_putchar(
    char          c,
    Rominfo       rinf);
```

Description

`rom_putchar`() calls the low-level write routine of the console device record configured for use as the system console. `rom_putchar`() writes the specified character to the console. If the character is a carriage return character `(0x0d)`, `rom_putchar`() also writes a line feed character `(0x0a)` to the console.

Parameters

`c`
: is the character to output.

`rinf`
: points to the `rominfo` structure.

Example

```
rinf->cons->rom_putchar(ch, rinf);
```

rom_puterr()
Write Error Code To the System Console

Syntax

```
void rom_puterr(
    error_code    stat,
    Rominfo       rinf);
```

Description

`rom_puterr`() converts the specified error code to a null terminated ASCII string representation of the form `AAA:BBB:CCC:DDD` and outputs this string to the system console using the `rom_putc`() service.

Parameters

`stat`
: is the value of the error code to be displayed

`rinf`
: points to the `rominfo` structure.

Example

```
rinf->cons->rom_getchar(status, rinf);
```

rom_puts()
Write a Null-Terminated String To the System Console

Syntax

```
void rom_puts(
    char          *buff,
    Rominfo       rinf);
```

Description

`rom_puts`() calls the low-level write routine of the console device record configured for use as the system console. `rom_puts`() writes a null terminated string to the console device.

Parameters

`buff`
: points to the first character of the string to output.

`rinf`
: points to the `rominfo` structure.

Example

```
rinf->cons->rom_puts(buffer, rinf);
```

Notification Module Services

The definition of the `notify_svcs` structure resides in the include file `MWOS/SRC/DEFS/ROM/rom.h`.

```
typedef struct notify_svcs {

    idver     infoid;                 /* id/version for notify_svcs */

    /* handler registration service */
    error_code      (*reg_hndlr)(Rominfo rinf, u_int32 priority,
                    void (*handler)(u_int32 direction, void *parameter),
                    void *parameter, u_int32 *hndlr_id);

    /* handler deregistration service */
    error_code      (*dereg_hndlr)(Rominfo rinf, u_int32 hndlr_id);

    /* notification service */
    error_code      (*rom_notify)(Rominfo rinf, u_int32 direction);

    Notify_hndlr      torom_list,     /* ordered lists of handlers */
                    tosys_list,
                      empty_list;     /* empty list of available records */
    u_int32   last_direction; /* direction of last notification call */
    int       reserved;             /* reserved for emergency expansion */

} notify_svcs, *Notify_svcs;
```

The notification services, `reg_hndlr`() and `dereg_hndlr`(), are commonly used from a low-level driver requiring notification to preserve and restore the state of a hardware interface shared between high-level drivers under the control of the operating system and low-level drivers required for remote debugging communications or local console support.

If no low-level drivers require the notification services, then the `notify` module may be omitted.

Table B-9.Notification Services

Function	Description
`dereg_hndlr()`	Remove registration for notification handler
`reg_hndlr()`	Register notification handler

dereg_hndlr()
Remove Registration For Notification Handler

Syntax

```
error_code dereg_hndlr(
    Rominfo       rinf,
    u_int32       hndlr_id);
```

Description

This service deregisters a notification handler.

Parameters

`rinf`
points to the `rominfo` structure.

`hndlr_id`
is the handler ID returned when the handler was registered.

reg_hndlr()
Register Notification Handler

Syntax

```
error_code reg_hndlr(
   Rominfo      rinf,
   u_int32      priority,
   void (*handler)(
      u_int32   direction,
      void      *parameter),
   void         *parameter,
   u_int32      *hndlr_id);
```

Description

This service registers a notification handler.

Parameters

`rinf`
: points to the `rominfo` structure.

`priority`
: specifies the priority of execution relative to the other registered handlers. Lower numbers are executed prior to higher numbers when transitioning from the operating system to the ROM. When transitioning back, the handlers are executed in the opposite order.

`handler`
: points to the actual handler being registered. Its parameters are the transition direction and a local parameter pointer.

`parameter`
: specifies the parameter value to be passed to the handler on its activation. This typically points to a data structure defined by the handler.

`hndlr_id`
: specifies the address where the handler identification is to be returned.

Compressed Booter Services

The definition of the compressed booter resides in the include file `MWOS/SRC/DEFS/ROM/rom.h`.

```
#define COMPRSVCID 0xA8E0


#define COMPR_VER_MIN 1
#define COMPR_VER_MAX 1


  /* defined service types are */
#define COMPR_INFLATE 1 /* zip/gzip/zlib inflate method */
#define COMPR_DEFLATE 2 /* zip/gzip/zlib deflate method */


typedef struct compr_svcs {
   idver  infoid;/* id/version for excpt_svcs */
   /* compress/uncompress call */
   error_code(*work)(u_int32 type, u_int8 *dest, u_int32 *dest_len,
                CONST_ROM_H u_int8 *source, u_int32 source_len, Rominfo
rinf);
   u_int32*compr_ll;  /* compression modules linked list*/
   u_int32reserved1;/* reserved for emergency expansion */
   u_int32reserved2;/* reserved for emergency expansion */
} compr_svcs, *Compr_svcs;
```

Compression services are routinely used by booters to handle the uncompression of a bootfile. The only currently supported service is a `zlib` inflate provided in the uncompress module. Other compression and decompression services can be chained to the compression services structure. The `zlib` deflate service is defined; however, a module that supports it is not provided.

For more information on `zlib`, refer to http://www.gzip.org/zlib/, which is the canonical web site for zlib. The `zlib` sources provided with OS-9 are slightly modified from the standard distribution.

The uncompress module may be omitted if compressed bootfiles or other compressed data are not used.

Compressing the Bootfile

When creating a bootfile, you can specify to compress the module you are creating. If you are using the Configuration Wizard to build your bootfile, simply check the Compress Bootfile check box in the Wizard's Master Builder window before building the image; this will automatically compress the bootfile.

If you are not using the Wizard to build your OS-9 image, you can compress the bootfile by completing the following steps from your chosen application:

In order to add compression, you must add the compress module to the coreboot when performing or updating a board port.

Step 1. Open the makefile in which you want to include a compressed bootfile.

Step 2. Edit the makefile by typing the following command in an appropriate location:

```
$(ODIR)/os9kboot.z: $(ODIR)/os9kboot
-$(DEL) $(ODIR)/os9kboot.z
mbc $(ODIR)/os9kboot -o=$(ODIR)/os9kboot.z
```

The above command implements the `mbc` utility, which replaces the uncompressed bootfile with one that is compressed.

For more information on the `mbc` utility, refer to the *Utilities Reference* manual, included with this product CD.

Step 3. Edit the ROM target to use `os9kboot.z` instead of `os9kboot`.

Step 4. Run the makefile.

Step 5. Transfer the ROM image to your board, performing all necessary steps.

Step 6. Boot the target board. The following boot menu appears:

```
Press the spacebar for a booter menu

BOOTING PROCEDURES AVAILABLE ------------ <INPUT>

Boot embedded OS-9 in-place ------------- <bo>
Copy embedded OS-9 to RAM and boot ------ <lr>
Boot over Ethernet (Intel Enet Pro 100) - <eb>
Load bootfile via kermit Download ------- <ker>
Enter system debugger ------------------- <break>
Restart the System ---------------------- <q>

Select a boot method from the above menu: lr

Compressed bootfile found at $00300000
A valid OS-9 bootfile was found.
$
```

C piclib.l Functions

This appendix discusses the `piclib.l` functions.

Overview

The functions to enable and disable interrupts on programmable interrupt controllers (PICs) have been externalized into libraries to minimize the platform dependency on driver sources and binaries. There are three types of libraries available for different driver requirements. Examples of them for "8259-like" (PC) PICs are supplied in your OS-9 Embedded release.

Library Types

The first two types of libraries are for drivers that are to be specialized for a particular target platform. The example libraries are `piclib.il` (for I-code linking and inlined code optimization) and `piclib.l` (for linking with driver relocatable assembler output files during debugging). These libraries are built to be target platform-specific, since the target determines the I/O location of the PIC and mapping of interrupt numbers to vectors is established by the port to the target.

The third type of library is a subroutine module that can be accessed through a helper library linked with the driver. This enables drivers to be distributed in object form with plug-in cards for bus-based systems and remain portable across target platforms with possibly differing interrupt controllers or mappings. The example `picsub` module is built from a special root psect and the `piclib.l` library mentioned above.

For CPU boards that do not employ an interrupt controller, the distribution provides the `nopiclib.il` and `nopiclib.l` libraries, and the `nopicsub` subroutine modules. These libraries do nothing but return a `SUCCESS` status.

The services available to drivers are `_pic_enable()` and `_pic_disable()`.

Table C-1.PIC Services

Function	Description
`_pic_disable()`	Disable interrupt on PIC hardware
`_pic_enable()`	Enable interrupt on PIC hardware

_pic_disable()
Disable Interrupt On PIC Hardware

Syntax

```
error_code _pic_disable(u_int32 irqno);
```

Description

`_pic_disable()` disables the appropriate vector on the interrupt controller hardware. Generally, this function is called just before a system module uninstalls the interrupt handler on the specified vector.

Parameters

`irqno`
is the OS-9 vector number to disable on the PIC.

_pic_enable()
Enable Interrupt On PIC Hardware

Syntax

```
error_code _pic_enable(u_int32 irqno);
```

Description

`_pic_enable()` enables the appropriate vector on the interrupt controller hardware. Generally, this function is called after a system module has installed an interrupt handler on the specified vector.

Parameters

`irqno`
is the OS-9 vector number to enable on the PIC.