Home

OS-9® for 68K Processors BLS Reference

Version 3.3

RadiSys
THE POWER OF WE

www.radisys.com
Revision A • July 2006

Copyright and publication information

Disclaimer

Reproduction notice

Table of Contents

Chapter 1: Introduction

This manual describes installing Microware OS-9® for 68K on your target system. This Board-Level Solution (BLS) provides a means for end-users and Original Equipment Manufacturers (OEMs) to quickly build standard VME-based systems with minimal effort.

This chapter includes the following sections:

- **OS-9 for 68K Targets Supplied**
- **MWOS Development Directory Structure**
- **General Installation Procedure**

OS-9 for 68K Targets Supplied

Board Level Solutions (BLS) supplied targets include:

- 68328:
  MC328ADS
- CPU32:
  MC68360 Quads (Quads) (OEM Package ONLY)
- 68040:
  MVME162 (all models)
  MVME167 (all models)
- 68060:
  MVME172 (all models)
  MVME177 (all models)
- 68030:

  MVME147

Software Packages

The CD-ROM contains Board Level Solution products for the boards listed above. BLS packages are binary based with the sources required to configure the system through device descriptors, the init module, and by adjusting the modules included in the boot.

Embedded OS-9 for 68K is the OEM Source package. This package contains all the BLS versions along with the sources to rebuild drivers, selected system modules, port specific ROM code and a variety of example drivers to aid in porting OS-9 for 68K to a new hardware platform.

Both packages allow customizing the Embedded bootfile with support for serial and parallel devices, RAM disks, SCSI disk and tape drives, SPF based TCP/IP networking support, NFS and a variety of system utilities.

MWOS Development Directory Structure

The Microware OS-9 products are developed in a directory structure that allows for multiple processors and supported boards to share common sources and definitions where possible. The directory tree is referred to as the MWOS (Microware OS) directory structure.

The MWOS directory structure is installed on the host system. This package supports Windows NT 4, Windows 2000 or Windows XP host development systems. The MWOS runtime structure can also be installed on the OS-9 target system.

For More Information

The target disk structure is discussed in **Chapter 3**.

Customizing a BLS for a target platform that requires assembly, compilation, or other build process, is performed within a PORTS directory in the MWOS development directory structure.

MWOS Subdirectories

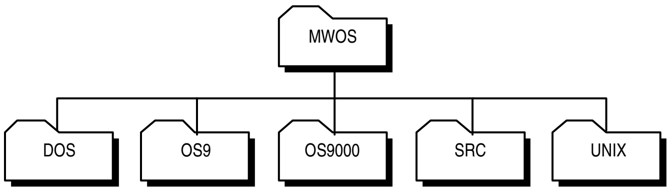


Table 1-1 MWOS Subdirectories

Directory	Contains
`OS9`	OS-9 for 68K object code is targeted under this directory. The 68K version of OS-9 has a kernel and primary system modules written in assembly code. All OS-9 specific source code, `defs` files, libraries, processor family code, and ports reside here. Most customizing of your system environment for the BLS is performed under this directory structure.
`OS9000`	OS-9 directories for processors other than the 68k family of processors. This is the C based version of OS-9.
`SRC`	All sources that are common at this level of the tree. C defs, common I/O systems, user tools, and Dual Ported I/O (DPIO) are examples of code found under the `MWOS/SRC` directory.
`DOS`	Similar to other OS directories. Contains development tools for use on a Windows cross-development host.
`MAKETMPL`	A directory for common makefile templates (include files for makefiles). Files in this directory also control which processors the template based makefiles target.

Figure 1-1 OS9 Subdirectories

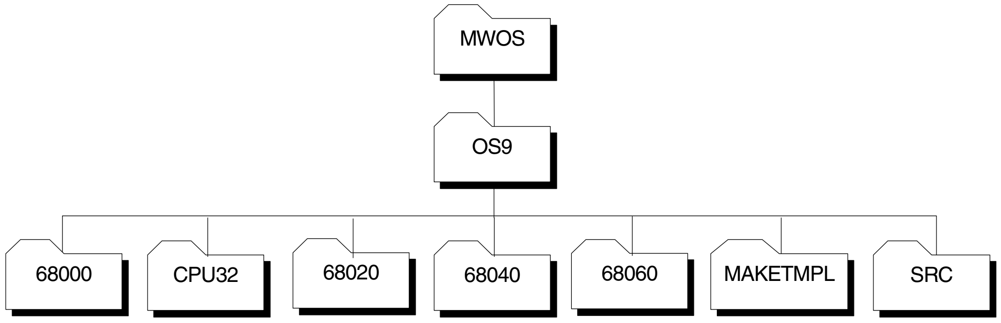


Table 1-2 OS9 Subdirectories

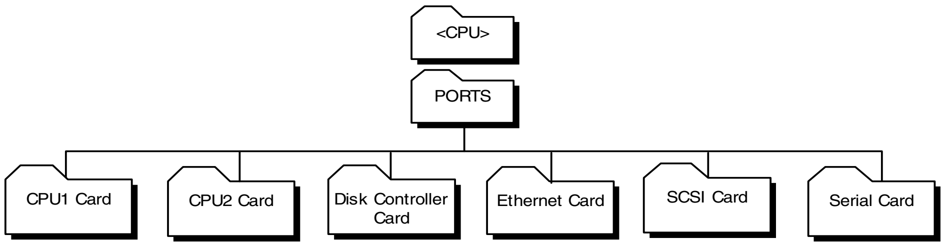
Directory	Contains
`68000`	The object code and libraries specific to the 68000 family of processors or binaries created to run on all versions of the Motorola MC68xxx family of processors. Most OS-9 for 68K utilities are compiled to run on all processors. In some cases (such as the networking utilities), speed concerns require compiling versions specifically for the 68020 and/or CPU32 families. The `68000` directory also contains code for the 68010, 68070, and 68302 processors.
`CPU32`	Files specific to the CPU32 family, such as the 68332, 68340, 68349 and 68360 processors.
`68020`	The `cmds`, libraries and ports specific to the 68020 processors.
`68040`	The `cmds`, libraries, and ports specific to the 68040 processors (MVME162 and MVME167 board ports).

Table 1-2 OS9 Subdirectories (continued)

Directory	Contains
`68060`	The `cmds`, libraries, and ports specific to the 68060 processors (MVME172 and MVME177 board ports).
`SRC`	The source files for the OS-9 for 68K drivers, descriptors, system modules, defs, and macros. `SRC` is intended to be a source directory containing hardware-specific code written to be reusable from target to target. It is not intended to be the repository for final object modules that are built from this source, although intermediate object files may be found within its subdirectories.

Each CPU directory has a `PORTS` subdirectory. The `PORTS` subdirectory provides directories for a variety of target system boards.

Figure 1-2 PORTS Subdirectories



Generally, if you are going to use peripheral cards with a variety of CPU cards, you should locate them under the `68000` directories. Drivers and card ports specific to 68020 or CPU32 family processors are located under their respective `<CPU>/PORTS` directory. The `68040/PORTS` directory contains the MVME162 and MVME167 board ports directories. The `68060/PORTS` directory contains the MVME172 and MVME177 board ports directories.

Note

In previous releases of OS-9 packages, some or all of the board ports directories were in the `68020/PORTS` directory.

Each card subdirectory has a structure that includes `CMDS` and `CMDS/BOOTOBJS` directories. CPU card directories may also contain a `BOOTLISTS` subdirectory for use in creating boots from within the `MWOS` directory structure.

Figure 1-3 SRC Subdirectories

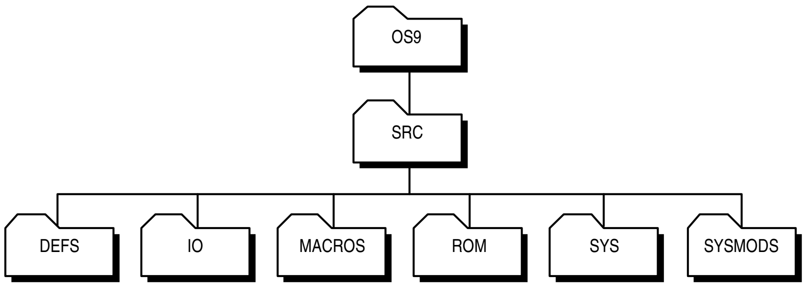


Table 1-3 SRC Subdirectories

Directory	Contains
`DEFS`	Files of definitions that apply system-wide or are processor independent. These include both assembler `.d` and C `.h include` files.
`IO`	Sources for all OS-9 for 68K-specific I/O subsystems including file managers, drivers, and descriptors. The file's subdirectories are organized by subsystem.

Table 1-3 SRC Subdirectories (continued)

Directory	Contains
`MACROS`	Files of assembly language macro definitions that apply system-wide or are target independent.
`ROM`	Sources for rebuilding boot ROM components, except for a few that share source with SCSI drivers in IO (OEM package only).
`SYS`	A repository for files and scripts that would end up residing in the OS-9 `SYS` directory on the root of the system disk.
`SYSMODS`	Sources for system extension modules.

Note

The level of source code available under the `SRC` directory depends on the type of package you purchased.

General Installation Procedure

Following is a list of basic steps to complete to install Microware OS-9 for 68K on your target system. These steps are described in detail in the following chapters.

Step 1. Install the Microware OS-9 Board Level Solution from your product CD-ROM onto your Windows-based host development system.

Step 2. Install the ROM code onto the CPU board.

Step 3. Start the target board for the first time using the OS-9 Boot that is included in the ROM Image.

For More Information

See **Chapter 2** for more information about steps 2 and 3.

Step 4. Partition and format any SCSI hard drives attached to the target system (optional).

Step 5. Start OS-9 networking for the first time using `startspf.ndbmod` (quick startup method).

Step 6. Load the hard disk with CMDS and startup scripts (optional).

Step 7. Customize and install the bootfile and utility set for the target environment.

For More Information

For all packages, see **Chapter 4**. For systems with SCSI devices, see **Chapter 3**.

Step 8. Configure the appropriate networking drivers and NFS client systems.

For More Information

See **Chapter 5** for information about network configuration.

Step 9. Set up the Hawk Development environment.

Note

OS-9 for 68K Processors MVME Board Guide provides specific information for CPU boards as well as information on configuring a number of peripheral boards supported by Microware.

Chapter 2: Hardware Configuration and the Initial Boot

This chapter includes the following topics:

- **Target CPU Board Configuration**
- **Sample Reconfiguration Session for MVME boards**
- **Initial OS-9 Test Boot**

Target CPU Board Configuration

Before configuring your target board, be sure to read the following reference materials:

- Follow the instructions for hardware preparation and installation described in the manuals supplied with your CPU board or system.
- Read the Motorola debugger manuals included with the PROM/flash code provided with your CPU board.
- Refer to the ***<CPU>Bug*** manual for your CPU board for details on running the initial board diagnostics.

Complete the following steps to configure your target system:

Step 1. Remove the CPU board from the system. Take precautions to protect the board from static damage.

For More Information

Refer to ***OS-9 for 68K MVME Board Guide*** for the information specific to your CPU board. The section for each CPU board contains jumper block diagrams/usage, ROM socket locations, ROM start-up sequences, and any other information specific to the board.

Step 2. Choose a method and install the OS-9/RomBug ROM code on your target CPU board.

Generally, PROMs loaded with OS-9/RomBug may coexist with the Motorola Bug PROM code pre-installed on the CPU boards or the OS-9 PROM code may be installed as the only debugger in the system. On some CPU boards, flash memory may replace some or all of the ROM memory space. For boards with additional ROM or Flash space, the OS-9 init module can be customized to search the additional areas for OS-9 modules during system boot.

- Method 1: If you are using one of the MVME boards that has FLASH memory available and you have a MOTBUG rom installed in the board, this method involves using MOTBUG to program the flash memory with the OS-9 ROM image provided.

  This is the simplest method for changing the embedded boot image on the target board. Use MOTBUG to perform the ENV, NIOT, NIOP, and PFLASH command to setup, download, and program the FLASH memory with the ROMBUG, Booters, and embedded OS-9 Boot ROM image. Once the FLASH memory is programmed, you can choose, via a jumper setting, whether to have MOTBUG start the system and then call the OS-9 ROM or to have the OS-9 ROM be the primary startup code. Choosing the OS-9 ROM as the primary startup code significantly decreases startup times.

- Method 2: Boards with no Flash memory or without MOTBUG or other monitor PROMS require that you program the PROMs with the supplied OS-9 ROM image. This is also a faster method of installing the OS-9 ROM code on your board.

  On some boards, you can choose whether to leave the MOTBUG PROMS in the primary socket(s) and install the OS-9 PROMS in a secondary socket(s) or to install the OS-9 PROMS in the primary socket(s) as the sole ROM code on the board. As with Method 1, using only the OS-9 ROM code will decrease the time required for the startup sequence.

For More Information

Refer to ***OS-9 for 68K MVME Board Guide*** for information on the specific sequences for programming flash memory or installing the OS-9 PROMS on your board.

Generally the OS-9 for 68K system uses a console port configured at 9600 baud, 8 data bits, 1 stop bit, and no parity with XON/XOFF handshaking. A three-wire RS232 cable with RX, TX, and signal ground is the standard cabling required to proceed to the next section.

For More Information

OS-9 for 68K MVME Board Guide details any exception to this configuration and the location of the console port.

Sample Reconfiguration Session for MVME boards

Once your system ROM startup method is configured, a reset or fresh powerup should bring up the OS-9 ROM code. The MVME1xx board images contain the ROMBUG debugger and a reconfiguration menu for setting and saving a number of settings in the NVRAM (Non-volatile RAM) provided on these boards.

The following example shows a configuration session for a MVME162 CPU board. The sequence of questions is identical for all the MVME1xx BLS packages; however, the data displayed and available responses may vary depending on the particular CPU board being configured.

Step 1. Power up CPU Board or press `<Reset>`.

The following lines are displayed:

```
OS-9/68K System Bootstrap

<Called>
Searching special memory list for symbol modules...

dn: 000000FF 00002000  00000000 00020000   00000000 00000001  FFFFE000 000069F8
an: FF800A6E FF800500  FFA40000 FFA20000   00007A00 00000400  00000400 000069F8
pc: FF800970  sr:2708 (--SI-7-N---)t:OFF   msp:B7B2BD3B   usp:00000000   ^isp^
0xFF800970   >43FAFBDA         lea.l 0xFF80054C(pc),a1
RomBug: g
```

Step 2. Press letter `g` to start the boot up process.

The first time the board is powered up after the OS-9 ROM code is first installed, the reconfiguration sequence is called. Additionally, if the NVRAM CRC is found to be incorrect, the following sequence is activated. The initial reconfiguration may also be activated by holding down the `<Abort>` button on the CPU card as the card comes out of the reset process. (Hold down the `<Abort>` button, press and release the `<Reset>` button, then release the `<Abort>` button.)

Note

This `Abort/Reset` sequence only works when the OS-9 ROM code is installed in the primary bank. If the Motorola debugger is installed in the primary ROM/FLASH bank, see ***OS-9 for 68K MVME Board Guide***, for information about configuring the Motorola debugger and the description of forcing a reconfiguration session.

```
*** ATTENTION ***

NVRAM was found corrupted - reconfiguration is forced.

Do you wish to reinitialize the new OS9 area
of NVRAM before entering new values:
(<yes>/<no>)? yes
```

Answering *yes* causes the NVRAM to be initialized to zero values and all booters made available. Answering *no* leaves the last known values in the NVRAM and keeps the current setting during reconfiguration.

Hardware Reconfiguration

For each question, you can select one of the following:

Table 2-1 Hardware Reconfiguration Responses

Key	Result
Backspace	Backspace over the value and enter a new value.
Tab	Toggle between the current and default values.
Esc	Exit the reconfiguration sequence.
Return	Accept the value displayed.

Board Memory Size

```
Warning: you are forced to enter the default value.

Board Memory Size: [current=0x0] [default=0x400000]
(<new_value><cr>/<tab>/<esc>)? 0x0
(<new_value><cr>/<tab>/<esc>)? 0x400000
```

Table 2-2 Board Memory Sizes

Board	Memory Size
MVME162 MVME167 MVME172 MVME177	the memory size is filled in automatically and cannot be changed

Board ID

```
Board ID: [current=0] [default=15]
(<new_value><cr>/<tab>/<esc>)? 0
(<new_value><cr>/<tab>/<esc>)? 15
```

The board ID is used in conjunction with the Microware ISP Backplane drivers. The default ID of 15 is used if no backplane driver is to be used or if this CPU board is the only one in the system. For systems using SPF based networking (all new installs) select the default value.

For systems using the discontinued ISP backplane driver refer to the 3.x manuals for appropriate reconfiguration settings.

WARNING
You are forced to enter the default value.

Group ID

```
Group ID: [current=0x0] [default=0xd2]
(<new_value><cr>/<tab>/<esc>)? 0x0
(<new_value><cr>/<tab>/<esc>)? 0xd2
```

VME Interrupt Levels

```
VME interrupt levels enabled: [current=<none>] [default=1234567]
(<newlist><cr>/<tab>/<esc>)?
(<newlist><cr>/<tab>/<esc>)? 1234567
```

If this CPU board is to service interrupts on the VME bus, select the levels of interrupts the board is suppose to acknowledge. Generally, the left-most board/system controller CPU is set to service all interrupts, while the other CPUs are set to **none**. More complicated interrupt schemes can be implemented if more than one CPU is on the same VME backplane. Only one board should be set to service any particular interrupt level.

VME Slave Memory

```
VME Slave Memory: [current=disabled] [default=enabled]
(enabled<cr>/disabled<cr>/<tab>/<esc>)? disabled
(enabled<cr>/disabled<cr>/<tab>/<esc>)? enabled
```

The VME slave memory is enabled to allow VME bus boards access to the CPU cards' on-board RAM.

VME Slave Memory Address

```
   VME Slave Memory Address: [current=0x0] [default=0x0]
   (<new_value><cr>/<tab>/<esc>)? 0x0
```

The VME slave address is the VME bus address at which the local board's memory appears on the VME bus. That is, when other boards want to access this CPU board's on-board memory, this is the starting address of the CPU board's memory when addressed from other cards on the VMEbus. The board should be set so each board's memory appears in non-overlapping address spaces on the VME bus.

SCSI Reset on ROMBUG startup

```
SCSI Reset on ROMBUG startup: [current=disabled] [default=enabled]
(enabled<cr>/disabled<cr>/<tab>/<esc>)? disabled
(enabled<cr>/disabled<cr>/<tab>/<esc>)? enabled
```

Generally the SCSI bus is issued a reset when the CPU card is reset. This feature may be disabled if the reset is not desired.

Boot System Reconfiguration

The following questions are used to configure the boot-up sequence for the CPU. Disabling the debugger causes the Boot Menu to display first. Using the OS-9 break command causes a system reset if the debugger is disabled.

Disabling the Boot Menu causes the system to follow the sequence entered under the Booter Priority Order sequence in attempting to boot the system. You may enter a given device more than once. Once the menu is disabled, you need to use the Abort/Reset method previously described to force the ROM code into the reconfiguration sequence.

```
Debugger: [current=enabled] [default=enabled]
(enabled<cr>/disabled<cr>/<tab>/<esc>)? enabled

Boot Menu: [current=enabled] [default=enabled]
(enabled<cr>/disabled<cr>/<tab>/<esc>)? enabled

Boot Drivers Available:

1 - Boot from SCSI(SCCS) hard drive
2 - Boot from Viper tape drive
3 - Boot from Teac SCSI floppy drive
4 - Boot from BOOTP i82596 LANC
5 - Boot from a non-volatile (Static) RAM disk
6 - Load Bootfile from ROM
7 - Boot from ROM
8 - Boot Manually Loaded Bootfile Image
The priority of these boot drivers can be set below.
This priority determines the order that the boot drivers
will be selected when your system is configured to boot
automatically. It also determines the order that they
appear in the "boot driver menu" as well.
Booter Priority Order: [current=12345678] [default=12345678]
(<newlist><cr>/<tab>/<esc>)? 12345678

Is this the configuration you want:
(y)es will reconfigure and restart the system.
(n)o will restart this reconfiguration dialogue.
(<yes>/<no>)? yes
```

Repeat the process or enter **yes** to restart the system with the new values for hardware and boot system configuration.

Initial OS-9 Test Boot

Once you have configured the NVRAM data structure, you are ready to boot OS-9 for the first time. The following example uses the configuration as set up in the previous section on an MVME162 CPU-based system. The OS-9/RomBug image contains a ROM-based OS-9 for 68K Boot for your CPU board with drivers for the serial devices, a SCSI hard disk, SPF based TCP/IP networking, and the user state HAWK target daemons. This boot also contains a small set of utilities that simplify downloading additional modules.

The following sequence shows the OS-9 for 68K system first being booted from ROM, then being loaded from ROM, and finally example output from some of the utilities contained in OS-9 for 68K in ROM.

After the reconfiguration sequence, you see the following display:

```
OS-9/68K System Bootstrap

<Called>
Searching special memory list for symbol modules...


dn: 000000FF 00002000  00000000 00020000   00000000 00000001  FFFFE000 000069F8
an: FF800A6E FF800500  FFA40000 FFA20000   00007A00 00000400  00000400 000069F8
pc: FF800970  sr:2708 (--SI-7-N---)t:OFF   msp:B7B2BD3B   usp:00000000   ^isp^
0xFF800970   >43FAFBDA         lea.l 0xFF80054C(pc),a1
RomBug: g

After the RomBug initial register dump is displayed, press g <ret> to bring up the
boot menu.  In this first example we will boot from ROM.

BOOTING PROCEDURES AVAILABLE -------- <INPUT>

Boot from Viper tape drive ---------- <vs>
Boot from Teac SCSI floppy drive ---- <fs>
Boot from SCSI(SCCS) hard drive ----- <hs>
Boot from BOOTP backplane ----------- <bp>
Boot from BOOTP am7990 LANCE -------- <le>
Load Bootfile from ROM -------------- <lr>
Boot from ROM ----------------------- <ro>
Boot Manually Loaded Bootfile Image - <ml>
Reconfigure the boot system --------- <rc>
Restart the system ------------------ <q>

Select a boot method from the above menu.
```

Step 1. Enter **ro** <**Return**> to boot the system from ROM. The ROM area is searched and if an OS-9 kernel is found, it is executed from its position in the ROM. Next, the kernel searches for the rest of the OS-9 system modules and executes them from ROM and the following displays:

```
Now searching memory ($FF820000 - $FF827FFF) for an OS-9 Kernel...

An OS-9 kernel module was found at $FF820000
A valid OS-9 bootfile was found.
Sysgo can't chx to 'CMDS'
Sysgo can't open 'SYS/startup' file
$
```

OS-9 is now running as shown by the $ prompt from the shell.

The system in the ROM is set up to use the device /dd as the default system device. In this case /dd is a standard RAM disk, which is empty when first initialized. Therefore when sysgo looks for a CMDS directory and the SYS/startup files, neither are found. When sysgo doesn't find the CMDS directory or the Startup file, it prints the warning messages and continues to process toward forking mshell on the console port.

If /dd were a hard disk or Non-volatile RAM disk, sysgo would have performed a chx to CMDS and would have attempted to run the SYS/startup file as a shell script before forking mshell on the console.

The system is now using modules running out of ROM. Since access time for code in ROM is generally slower than the same code running from RAM, you should reset the system and reboot using the `Load Bootfile from ROM` boot option. This option causes the OS-9 for 68K system in ROM to be loaded into RAM much the same as if it had been booted from a disk, tape, or network booter. To reboot the system, press the reset button or execute the break `(break)` command, then reset the system using the reset `(rst)` command from RomBug.

```
$ break
WARNING: Timesharing HALTED.
 (type ‘G’ to resume.)

<Called>
dn: B0BD06FF 1BADD00D  00000000 003E0003   00000000 003E3038  000012D0 00000000
an: 003E42A4 00000000  FF86BE74 003E42BC   003E42C8 003E4288  003EB000 003FFAE0
pc: 0000BA16  sr:2708 (--SI-7-N---)t:OFF   msp:003E6FA0   usp:003E4284   ^isp^
0x0000BA16   >4CDF7001         movem.l (a7)+,d0/a4-a6
RomBug: rst
OS-9/68K System Bootstrap
```

```
<Called>
Searching special memory list for symbol modules...


dn: 000000FF 00002000  00000000 00020000   00000000 00000001  FFFFE000 000069F8
an: FF800A6E FF800500  FFA40000 FFA20000   00007A00 00000400  003EB000 000069F8
pc: FF800970  sr:2708 (--SI-7-N---)t:OFF   msp:003E6FA0   usp:00000000   ^isp^
0xFF800970    >43FAFBDA         lea.l 0xFF80054C(pc),a1
RomBug: g
```

Step 2. Again, enter the go command to bring up the boot menu, by pressing letter **g** and the <**Return**> key. Select the `Load Bootfile` option from ROM boot options by entering `lr` and pressing the <**Return**> key.

```
BOOTING PROCEDURES AVAILABLE -------- <INPUT>

Boot from Viper tape drive ---------- <vs>
Boot from Teac SCSI floppy drive ---- <fs>
Boot from SCSI(SCCS) hard drive ----- <hs>
Boot from BOOTP backplane ----------- <bp>
Boot from BOOTP am7990 LANCE -------- <le>
Load Bootfile from ROM -------------- <lr>
Boot from ROM ----------------------- <ro>
Boot Manually Loaded Bootfile Image - <ml>
Reconfigure the boot system --------- <rc>
Restart the system ------------------ <q>
Select a boot method from the above menu: lr
Now searching memory ($FF820000 - $FF83FFFF) for an OS-9 Kernel...

An OS-9 kernel module was found at $FF820000
A valid OS-9 bootfile was found.
Sysgo can't chx to 'CMDS'
Sysgo can't open 'SYS/startup' file
$
```

The OS-9 for 68K system is now running. This time the system is executing from RAM.

For More Information

See **Chapter 4** for more information on `init` module search lists.

You may now execute a number of standard OS-9 utilities contained in this ROMed version of OS-9 for 68K:

`mfree`	Shows the amount of free memory in the system.
`procs`	Shows the running processes.
`mdir`	Displays all the modules available in the system.
`mdir -e`	Displays additional information including the address where the module is located.

The following example shows the results for running mfree:

```
$ mfree -e
Minimum allocation size:       4.00 K-bytes
Number of memory segments:   1
Total RAM at startup:       4096.00 K-bytes
Current total free RAM:     3860.00 K-bytes

Free memory map:

    Segment Address          Size of Segment
   -----------------    --------------------------
       $1A000          $3C5000     3860.00 K-bytes

$ procs
 Id PId Grp.Usr   Prior   MemSiz Sig S     CPU Time    Age Module & I/O
  2   0    0.0      128     4.00k   0 w        0.00 8760:05 sysgo <>>>term
  3   2    0.0      128    12.00k   0 w        0.21 8760:05 shell <>>>term
 14   3    0.0     128    32.00k0        *       0.09   0:00 procs <>>>term
```

You can also try some of the other utilities such as `setime`, `date`, `tmode`, and `devs`.

Congratulations, you have successfully booted your OS-9 for 68K system.

Chapter 3: Optional Hard Disk and Initial Networking Startup

At this point you should have booted your OS-9 target system.

If your board supports connection to a SCSI hard drive and/or has networking capabilities, read the following sections:

- **Connecting A SCSI Hard Disk**
- **Network Configuration**
- **Establishing the Hard Disk Root Directory**

Connecting A SCSI Hard Disk

Manual Installation Information

If you haven't already done so, shut down the target machine and connect a SCSI hard disk to the CPU board. There are various hardware locations for the SCSI connector, including the following:

- A connector located on the faceplate of the VME card.
- The MVME712 P2 Bus connector or the 712 breakout module.
- A connector on the surface of the card.

Be sure to observe proper SCSI termination requirements. The end devices on a SCSI bus should be set to provide termination for the bus. No other devices on the bus should have termination enabled.

In some cases termination is controlled by jumper selection while other devices/boards may require the addition or removal of resistor packs. New devices may have automatic termination. Refer to the device hardware manuals for more information.

If the hard drive is 4 Gigabytes or less in size, you can follow the instructions in the following section. If the drive is larger than 4 Gigabytes, proceed to the appropriate section.

Formatting 4 Gigabyte or Smaller Drives

If your hard drive is 4 Gigabytes or less in size, use the descriptors in the ROM boot to format the drive without using the partition utility. The following sequence uses the format utility to format a 224 meg drive called `/h0fmt`. If the drive is larger than 4 Gigabytes, format will automatically size it to 4 Gigabytes. If you then use the partition utility, you should select 4 Gigabytes for the size of the first partition.

```
$ tmode nopause
$ format /h0fmt
                Disk Formatter
OS-9/68K V3.1   Motorola VME162 - 68040
```

```
------------  Format Data  ------------
Fixed values:
               Disk type: hard
             Sector size: 512
  Physical Disk capacity: 485601 sectors
                         (248627712 bytes)
   Logical Disk capacity: 0 sectors
                         (0 bytes)
           Sector offset: 0
            Track offset: 0
              LSN offset: $000000
 Minimum sect allocation: 32

Variables:
Sector interleave offset: 1

Formatting device:  /h0fmt
proceed?             y
this is a HARD disk - are you sure? y
```

Note

SCSI hard disks generally do not require physical formatting unless you wish to change the physical sector size. If you do choose to do a physical format, you should know that on large drives, the formatting operation may take several hours or more. During these hours, the drive-in-use light may or may not be on. Physical formatting is still appropriate if the drive has unreadable sectors.

On most SCSI drives, performing a verify is not necessary because the drive should complete the verify with zero sectors bad. Selecting yes causes every sector of the disk to be read and is appropriate if you have any concerns about the physical format of the drive.

```
physical verify desired?  n
volume name:  Machine Name: /H0
```

At this point if you chose to verify the drive, you will see a display of track numbers as they are verified. If no verify was selected, you should see the driver light blink a number of times as format writes the bitmat on the disk.

The drive is now ready for use. After starting the networking as described later in the chapter, proceed to **Chapter 4** for instructions on loading the disk with basic commands, startup scripts, and other useful files.

Formatting and Partitioning Drives Larger than 4 Gigabytes

If you are connecting a SCSI hard drive that is larger than 4 Gigabytes, you must partition the drive into virtual drives of less than 4 Gigabytes.

The OS-9 ROM boot contains the partition utility, written specifically for partitioning and formatting your hard drive. The OS-9 for 68K Random Block File manager (RBF) can format drives up to 4 Gigabytes in size. For drives larger than 4 Gigabytes, RBF uses a system of breaking the disk up into virtual partitions. The term virtual partition is used because no partition table exists on the drive. Instead, the partition utility creates a number of new RBF device descriptors that access the physical drive as a number of virtual drives based on starting Logical Sector Numbers (LSN) and size of the virtual drive/partition. In terms of RBF they appear as separate drives while in terms of the physical drive the are virtual partitions.

The /h0 descriptor in the ROM boot can be used to run the partition command as described in the following section. The descriptors are then saved on the first partition of the hard disk and help maintain the connection between the partitioning information and the physical drive that was partitioned. The startup file residing in the SYS directory then loads the rest of the descriptors from the drive’s first partition, so the rest of the virtual partitions can be accessed.

Note

Because of a possibility of mixing descriptors and the drive they partition, it is strongly recommended that a consistent method is used for partitioning large drives.

In the following example the drive is broken up into 4 Gigabyte partitions with the final partition being the only one less than 4 Gigabytes.

WARNING
Using a descriptor that is improperly matched to the virtual drive partition can lead to file structure corruption.

Step 1. View the options on partition:

```
$ partition -?
Syntax:   partition [<opts>] <device name> {[<descs>] [<opts>]}
Function: partition a large (>4GB) hard disk
Options:
     -r         overwrite existing files
     -w=<dir>   output generated device descriptors to <dir>
     -z=<path>  read partition device descriptors from <path>
```

Step 2. Start the partition utility using the /h0 descriptor that is already part of the ROM boot. The h0 drive will also be used to write the descriptors after they are created. Once the first partition is formatted and the descriptors are saved in the Root directory of the disk, they can be loaded into memory so that the additional partitions can be accessed. You can also use the /dd ramdisk drive with the -w option. However, you must copy the descriptors to the hard drive so they won't be lost at system reset.

The default values are used in most cases. The example drive is approximately 21.61 Gigabytes. The convention of h01 refers to the drive with a SCSI ID of 0 (zero), partition 1.

Note
The /h01 and /h0 will refer to the same partition once the /h01 partition has been formatted.

```
$ partition -w=/h0 /h0<CR>
enter partition name (h01) =><CR>
enter a partition size (4.00GB) =><CR>

enter partition name (h02) =><CR>
enter a partition size (4.00GB) =><CR>

enter partition name (h03) =><CR>
enter a partition size (4.00GB) =><CR>

enter partition name (h04) =><CR>
enter a partition size (4.00GB) =><CR>

enter partition name (h05) =><CR>
enter a partition size (4.00GB) =><CR>

enter partition name (h06) =><CR>
enter a partition size (1.61GB) =><CR>

1. create new partition
2. edit existing partition
3. delete existing partition
4. display partition information
5. write device descriptors for partitions
6. format a partition
7. format all partitions
8. exit
please enter command number =>4
partition names and sizes:
--------------------------
partition 1 (h01): 4.00GB (8388607 blocks)
partition 2 (h02): 4.00GB (8388607 blocks)
partition 3 (h03): 4.00GB (8388607 blocks)
partition 4 (h04): 4.00GB (8388607 blocks)
partition 5 (h05): 4.00GB (8388607 blocks)
partition 6 (h06): 1.61GB (3379609 blocks)

1. create new partition
2. edit existing partition
3. delete existing partition
4. display partition information
5. write device descriptors for partitions
6. format a partition
7. format all partitions
8. exit
```

You can edit or delete partitions to make any changes before actually formatting the virtual partitions in the next step. Using the `format all partitions` produces the following queries. You can also choose to format single partitions using option 6.

At a minimum you must format the first partition, /h01 so that the descriptors can be saved to the drive.

```
please enter command number =>7
format partition h01? y
formatting partition h01...
format partition h02? y
formatting partition h02...
format partition h03? y
formatting partition h03...
format partition h04? y
formatting partition h04...
format partition h05? y
formatting partition h05...
format partition h06? y
1. create new partition
2. edit existing partition
3. delete existing partition
4. display partition information
5. write device descriptors for partitions
6. format a partition
7. format all partitions
8. exit
```

Step 3. Finally, the descriptors are saved to the Root of the first partition. After the descriptors are saved, they can be reloaded for accessing the additional partitions.

```
please enter command number =>5
1. create new partition
2. edit existing partition
3. delete existing partition
4. display partition information
5. write device descriptors for partitions
6. format a partition
7. format all partitions
8. exit

please enter command number =>8
paritions may need formatting - format now? n
$ dir /h0
```

```
                            Directory of /h0 22:34:43
h01              h01fmt           h02              h02fmt           h03
h03fmt           h04              h04fmt           h05              h05fmt
h06              h06fmt
$ load -d /h0/*
$ free /h02
"h02" created on: Aug 16, 1999
Capacity: 8388607 sectors (512-byte sectors, 32-sector clusters)
8388448 free sectors, largest block 8388448 sectors
4294885376 of 4294966784 bytes (4095.92 of 4095.99 Mb) free on media (99%)
4294885376 bytes (4095.92 Mb) in largest free block
```

The drive is now ready to use. For ease of loading the partitions in the future, store them in a sub-directory of the root directory called `PARTITIONS` on a partitioned drive, for example `/h0/PARTITIONS`.

The startup file can then load the descriptors using commands such as:

```
Load -d /h0/partitions/*  OR Load -d /h01/partitions/*
Load -d /h11/partitions/*
```

Once the `mv` or `copy` and `del` commands are available on the target machine, it is easy to move the descriptor files out of the root directory into the `PARTITION` subdirectory.

Formatting SCSI Drives (other than those set to 512 Bytes per Sector)

OS9/68K supports SCSI drives formatted with a variety of physical sector sizes. Sizes of 256, 512,1024, and 2048 have all been verified.   Currently, most drives come from the factory set to 512 byte sectors.

Creating a Device Descriptor

To physically format a drive (other than by using the default setting as read from the SCSI drive), a device descriptor for the drive you want to format must be created with the sector size set appropriately. A size of “zero” tells the rbvccs driver to query the drive and use the sector size for which the drive is formatted. Values for the sector size generally range from 256 bytes to 2K bytes. Currently, there are a number of drives that only support a single physical sector size; these report errors when attempts are made to change them from the supported size.

In addition, some SCSI drives have commands for changing the characteristics of the drive. When a descriptor is opened or attached with a sector size defined, the driver makes calls to set the SCSI drive characteristics described by the descriptor.

If the call is successful and sets the drive characteristics differently from the characteristics at the last formatting of the drive, the data on the drive will no longer be accessible. In most cases, the changes are only set in RAM and may be returned to their original settings by opening the device with a descriptor configured with the original settings for the drive. Drives will usually revert to their last formatted configuration on power-up. Therefore, cycling the power to the SCSI drive may allow the drive to again be accessed using a descriptor set for auto configuration.

As an example of this behavior, assume the following descriptors are available for a SCSI drive set to ID 1. All descriptors exist for this drive, but have different sector size settings as described. The drive is currently formatted at 512 Bytes/sector.

h1	Standard descriptor with sector size set to 0 (zero) for SCSI id 1.
h1_256	Same as h1 but with the sector size field set to 256 bytes/sector
h1_512	Same as h1 but with the sector size field set to 512 bytes/sector

The drive contains data which is accessible as /h1. The user runs the format utility with descriptor h1_256.

$ format /h1_256

Opening the drive with this descriptor causes the driver to set the drive characteristics to now be 256 bytes per sector. Even if the user decides to exit the format program at the continue/quite prompt, the drive has had it's characteristics altered. Since the drive was not reformatted, the data still resides on the drive, but it is not accessible via the /h1 device since the drive has now been configured for 256 bytes sectors

Accesses to the h1_512 device causes the drive to be configured back to the original 512 byte sector sizes. This makes the drive readable via the h1 and h1_512 device descriptors/names.

```
$ dir /h1_512
```

Cycling the power on the drive may also restore the 512 byte/sector configuration; this makes the drive readable via the h1 device descriptor. In some cases, however, newer drives that do not support the sector size changes may require a power cycle to clear a latched error condition.

Note

Once the physical format has been started using a new sector size descriptor, the data is on the drive is permanently destroyed. Due to drive size, a physical format may take up to several hours to complete. If the device descriptor has the SCSI option disconnect set (bit 0 in the SCSI option flags) the SCSI bus is released for access to other devices during the format operation.

The disconnect bit is generally not set in descriptors in standard shipping descriptors. If disconnect is not set for the drive to be formatted, the SCSI bus will remain busy and inaccessible during the entire format operation.

Network Configuration

The example file, `MWOS/SRC/SYS/startspf.ndbmod` contains a sequence of commands used to start networking on your target system for the first time.

You must edit the file to set the information for your networking environment. Once the file is edited, you can use your editor to cut lines from the file and paste them into the OS-9 command line running on the target. Alternatively, you can type them on the command line and edit them as you proceed.

If your system has a hard drive that was configured according to the previous section, you can save the `inetdb3` module to the disk once it is created. Alternatively, once FTP has been started, you can transfer your edited `startspf.ndbmod` file to the hard drive and simply run it as a shell script to start the networking after each reboot.

For More Information

Starting the networking for ROM based systems is described in **Chapter 5**.

Following is the example `startspf.ndbmod` file:

```
-t
****************************************************************************
* Initial Startup sequence for networking using the Rommed Boot           *
*                                                                          *
* Lines that start with ** should be used as examples to set values       *
*   appropriate for your network enviroment.  Lines currently not commented *
*   out should not need to be customized for the initial startup of the    *
*   target board.                                                          *
*                                                                          *
* Edit this file and if possible cut/paste each appropriate line onto the  *
*   console command line to initially start the SPF/Lancom networking on   *
*   the target board.                                                      *
*                                                                          *
* NOTE: Once a storage device is available, the inetdb3 can be saved to    *
*       disk and simply reloaded via a startup file.                       *
*                                                                          *
****************************************************************************
```

```
*
* Inititialize System MBuf system call
*   NOTE: This is already done via init module in most cases.
*   NOTE: MBinstall utility not in ROMBoot
*
-nx
*mbinstall
-x
*
* Create inetdb3 module
* Type ndbmod -? for help on parameters
*
ndbmod create inetdb3 11 400 0 160 0 0 0 100 0 400 65 256
*
* Initialize interface information
*   Change 192.168.0.5  to your system's IP address
*   Change 255.255.255.0 to your system's netmask
*
**ndbmod interface add enet0 address 192.168.0.5 netmask 255.255.255.0 binding
/spie0/enet
*
* Define domain and Addresses of DNS Servers
*   Change "MyDomain.com" to your domain name
*   Adjust addresses after "server" to match your DNS server's IP address
*   Remove second server entry if not requred
*
**ndbmod resolve MyDomain.com server 192.168.0.32 server 192.168.0.254
*
* Set hostname
*   Change "MyHostName" to the name of your system
*
**ndbmod hostname MyHostName
*
* Add locally defined Host names
*   Not required if DNS available
*   Room for approx 16 available
*   Add IP addresses and Hostnames as desired.
*
**ndbmod host add 192.168.0.5 Hawk5
**ndbmod host add 172.16.0.32 DNServe
*
* Add Default and other Routes
*   Change 192.168.0.254 to the IP address of your default router
*
**ndbmod route add Default 192.168.0.254
*
* Start SoftStax networking
*
ipstart
*
* Start Routing deaemon  (Not in ROM Boot)
*
*routed <>>>/nil &
*
```

```
* Start services Daemon(s)
*
*   Chd assumes /dd device (ramdisk) is part of Rom boot.
*   Setup an execution directory (optional)
*   Chx to the executiton directory (optional)
*   inetd: use once login and password file is available in a SYS directory
*   telnetd: with auto start of mshell (no login)
*   ftpd: with no login authentication
*
chd /dd
-nx
makdir /dd/CMDS
chx /dd/cmds
-x
*inetd <>>>/nil &
telnetd -f=mshell <>>>/nil &
ftpd -u <>>>/nil &
*
*   spfndpd: Start Hawk User state debugging daemon
*   spfnppd: Start Hawk Profiling Daemon (Not in ROM Boot)
*
spfndpd <>>>/nil &
*spfnppd <>>>/nil &
```

Starting `spfndpd` is only required if you are connecting to your target with Hawk during this session (before the next reboot). As noted in the comments, `telnetd` has been started so that a telnet session to your machine will immediately provide a “$” mshell prompt without any login required. Additionally, FTP has been started to not authenticate logins. While you will need to enter a username and password, they are not checked, therefor not requiring the login command or a password file. These daemon startup methods are provided for the initial startup of the system and loading of the optional attached hard drive and NOT recommended as the normal startup sequence.

For More Information

See the `MWOS/SRC/SYS/startspf` file for standard startup examples.

Test your target system by running Telnet or FTP from your Host system to the target. You can also Telnet or FTP from the console on your target machine to other machines on your network.

Establishing the Hard Disk Root Directory

At this point, your primary hard drive should be formatted and you should have established networking connections. The final step is to load the basic disk structure required on the Target system.

Prepare Image on Host machine

Step 1. Using Windows Explorer (or other disk browser) open the ports directory for your board.

Note
The board guide for your specific board contains the exact pathlist.

For example, the MVME162 is `MWOS/OS9/68040/PORTS/MVME162`. Generally, the CMDS directory is built up by starting with the 68000 CMDS and then overlaying additional CMDS from appropriate directories on top.

Step 2. Create a sub-directory called `Disk_Image`.

Step 3. Build the `Disk_Image/CMDS` directory.

- Right click on `MWOS/OS9/68000/CMDS` and select copy.
- Right click on the `Disk_Image` directory and select paste.
- Right click on `MWOS/OS9/68020/CMDS` and select copy.
- Right click on the `Disk_Image` directory and select paste.

  When the dialog box asks about replacing files, click on **yes to all**.
- Right click on `MWOS/OS9/68040/CMDS` and select copy.
- Right click on the `Disk_Image` directory and select paste replacing files with **yes to all**.

- Right click on `MWOS/OS9/68040/PORTS/MVME162/CMDS` and select copy.
- Right click on the `Disk_Image` directory and select paste.

Step 4. Build the `Disk_Image/SYS` directory.

- Right click on `MWOS/SRC/SYS` and select copy.
- Right click on the `Disk_Image` directory and select paste.
- Right click on `MWOS/OS9/SRC/SYS` and select copy.
- Right click on the `DISK_Image` directory and select paste.
- When the dialog box asks about replacing files, click on **yes to all**.

Add other files as desired.

Transfer Image to Target Machine

After preparing your image on the host machine, you can use FTP to transfer the files from the host to the root directory of your target machine disk. Following is an example of this process.

Step 1. Create a TAR Archive.

Using a Windows hosted `tar.exe` program create a tar archive of the `Disk_Image` directory by changing into the `Disk_Image` directory and running a command, on the host, similar to:

```
>tar -cvf tar.file CMDS SYS
```

```
CMDS/
CMDS/arp
CMDS/attr
CMDS/backup
CMDS/beam
CMDS/bfed
CMDS/binex...
```

You can also add additional files and directories with this command line.

For More Information

You can find an example of a Windows hosted tar program at the following url:

`http://www.reedkotler.com/RKTOOLS/rktools.html`

Step 2. FTP `tar.file` to the root directory of your target machine disk using binary mode.

Note

The procedure will vary slightly depending on which FTP software you are using.

From your Windows host machine, select **Start -> Run**. Type the following:

ftp <target machine>

The following displays in a DOS shell window. Enter the commands as shown.

```
Connected to <target machine>.
220 jimi.microware.com OS-9 ftp server ready
User (jimi.microware.com:(none)): <cr>
331 password required for (none)
Password: <cr>
230 user (none) logged in
ftp> bin
200 Type set to I.
ftp> hash
Hash mark printing On (2048 bytes/hash mark).
ftp> cd /h0
250 CWD command ok
```

```
ftp> put tar.file /pipe/tar.file
200 PORT command ok
150 Opening data connection for /pipe/tar.file
(172.16.4.207,4712).

#########
```

This starts the download to a named pipe on the target. The download pauses once the named pipe is full.

Step 3. On the OS-9 target machine, untar from the named pipe.

```
$ chd /h0
$ tmode nopause
$ tar -xvf /pipe/tar.file
drwxrwxrwx 0/0        0 Sep  2 11:40 1999 CMDS/
-rw-rw-rw- 0/0     6934 Jul 14 21:30 1999 CMDS/arp
-rw-rw-rw- 0/0     4284 Jul 14 09:34 1999 CMDS/attr ...
```

Note

Tar is included in the ROM boot on the MVME target machines.

Step 4. On the host, end the FTP session when the transfer is complete.

```
##################################################################################
##################################################################################
##################################################################################
226 Transfer complete
7055360 bytes sent in 164.60 seconds (42.86 Kbytes/sec)
ftp> quit
221 Goodbye
```

Step 5. On the target, set the file permissions of the files in `CMDS`:

```
$ chx /h0/CMDS
$ attr -ns CMDS SYS
$ dir -ru CMDS ! attr -nspeeprrz
d-ewrewr  CMDS/BOOTOBJS
d-ewrewr  CMDS/MAUIDEMO
d-ewrewr  CMDS/NOCSL
--ewrewr  CMDS/arp
--ewrewr  CMDS/attr
--ewrewr  CMDS/backup ...
```

Once the `CMDS` directory is on the hard drive, you can access the additional utilities by performing the `chx` command above.

Step 6. Correct the line termination of the files in `SYS`:

```
$ chd /h0/SYS
$ cudo -cdo *
```

This procedure converts the line terminations on text files from Windows (<cr><lf>) to OS-9 (<cr>).

For More Information

Chapter 4 describes customizing your boot so the SYS/startup file is run as the system boots up.

For More Information

The files in the SYS directory should be customized as desired. **Chapter 5** provides an overview of the networking startup methods.

Chapter 4: Boot and ROM Customizing

This chapter includes the following topics:

Overview

Each CPU Board Level Solution (BLS) package contains a version of OS-9 for 68K that can be booted from ROM. This chapter explains customizing the BLS and preparing a new boot for the CPU card. The basics for modifying the modules found in ROM are also covered.

For More Information

Chapter 2 describes booting the ROM-based system.

For More Information

Issues related to configuring the networking capabilities are covered in **Chapter 5**.

Once your development system is configured with the `MWOS` directory structure from a BLS or OEM package, you can customize the target system for your specific requirements.

Many of the examples in this chapter are based on the MVME162 BLS package. The board directory for the MVME162 is `MWOS/OS9/68040/PORTS/MVME162`.

For More Information

Refer to **Chapter 1** of this manual for an overview of the `MWOS` directory structure. The location of the `PORTS` directory for each BLS is supplied in ***OS-9 for 68K MVME Board Guide***.

PORTS Directory Organization

Within the `68000/PORTS`, `68040/PORTS`, `68060/PORTS` and `CPU32/PORTS` directories, sub-directories are created for each board to which OS-9 for 68K has been ported. The port directory for a BLS CPU board contains a number of subdirectories and files enabling you to adapt the board to your specific requirements. The MVME162 port directory contains the following subdirectories and files:

`BOOTLISTS`	Example boot module lists (bootlists) for making boots.
`BOOTS`	Makefiles for making a number of default boot images.
`DISK_IMAGE`	An optional directory for use in gathering the files to be transferred to systems configured with a hard drive or flash drive.
`CMDS`	Utilities for use with the board. The `BOOTOBJS` subdirectory and its subdirectories contain the board specific system modules.
`INIT`	Makefiles for creating different init module configurations.
`PCF`	PC File manager descriptor building.
`PIPE`	Pipe descriptor building.
`RBF`	Random Block File manager disk support build directory.
`ROM_CBOOT`	Ports using the CBOOT based ROM technology use this directory to build ROM images with or without ROMBUG and to change the ROM based bootfile for the system.
`ROM`	Ports using Modular ROM technology for their ROM code or a P2 loadable debugging and low level communication system build components from this directory.

`SBF`	Sequential Block File manager Tape support build directory.
`SCF`	Sequential Character File manager support build directory. Descriptors and board specific drivers (OEM Package only) are built from here.
`SCSI`	SCSI support build directory (OEM Package only).
`SPF`	Stacked Protocol File manager (SoftStax) configuration build directory.
`SYSMODS`	A variety of additional system modules specific to the board are built from here. Ticker, Real Time Clock, and snooper modules are examples. (OEM Package only).
`systype.d`	The `systype.d` file contains the hardware and software definitions to create the board specific modules. The definitions are an excellent source for information about how the hardware is set up by the ROMs included with the BLS. Any changes made to descriptors, `init` modules, and other user configurable modules normally start by editing the `systype.d` file and then running the appropriate makefiles to recreate the new OS-9 module.
`defsfile`	The `defsfile` in included by many source files to reference required `defs`.

BLS Makefiles

This is an example list of makefiles included in the MVME162 BLS along with the hierarcy of makes called from parent makefiles. The specific makefiles vary considerably between boards.

Table 4-1 MVME162 PORT Directory Makefiles

Makefile Called	Makefile Called	Makefile Called	Comments
`INIT/makefile`			
	`INIT/init_rom.mak`		
		`Init_rom`	
	`INIT/init_d0.make`		
		`Init_d0`	
	`INIT/init_h0.make`		
		`Init_h0`	
	`INIT/init_dd.make`		
		`Init_dd`	
`SYSMODS/makefile`			
	`SYSMODS/snoop.make`		
	`SYSMODS/clock.make`		Ticker and real time clock modules
`SCF/makefile`			
	`SCF/scf_descriptors.make`		Board's SCF descriptors
	`SCF/scf_drivers.make (OEM)`		Board's SCF drivers
`RBF/makefile`			
	`RBF/rbf_descriptors.make`		
	`RBF/rbf_teac_descriptors.make`		
	`RBF/rbf_vccs_descriptors.make`		
	`RBF/rbf_nvram_descriptors.make`		
`SBF/makefile`			
	`SBF/sbf_viper_descriptors.make`		
	`SBF/sbf_exabyte_descriptors.make`		
	`SBF/sbf_teac_descriptors.make`		
`SCSI/makefile`			

Table 4-1 MVME162 PORT Directory Makefiles (continued)

Makefile Called	Makefile Called	Makefile Called	Comments
`SPF/makefile`			
	`SPF/SP82596/makefile`		Ethernet descriptor and driver (OEM only)
	`SPF/ETC/makefile`		inetdb modules
`PCF/makefile`			
	`PCF/pcf_descriptors.make`		
`BOOTS/makefile`			
	`BOOTS/d0_bootfile.make`		
	`BOOTS/h0_bootfile.make`		
	`BOOTS/viper_tape_bootfile.make`		
`ROM_CBOOT/makefile`			
	`ROM_CBOOT/rom.make`		Nobug version of ROM
	`ROM_CBOOT/rombug.make`		Rombug version of ROM
	`ROM_CBOOT/rom_initext.make`		Builds the ROM init extension library
	`ROM_CBOOT/rom_booters.make`		Builds raw sysinit and booters image
	`ROM_CBOOT/rom_bootfile.make`		Makes ROM bootfile
`OEM Only Common makefiles`			
	`ROM_CBOOT/rom_common.make`		`Creates rom_common.l`
	`ROM_CBOOT/rom_serial.make`		`Creates rom_serial.l`
	`ROM_CBOOT/rom_port.make`		`Creates rom_port.l`
	`ROM_CBOOT/rom_descriptors.make`		`Creates rom_descriptors.l`
	`ROM_CBOOT/rom_image.make`		`Creates rom_image.l`

Note

On Windows cross-hosted development systems, makefiles must be executed with the `os9make` command.

The master makefile initiating all the others is `makefile`. Only the modules that require rebuilding are rebuilt. Bootfiles and ROM images are always reconstructed.

Modifying Bootlists

This section contains a bootlist file with enhanced descriptions of the modules available for use in an `OS9Boot` file. The process of making a new boot involves the following basic steps:

- Creating or modifying an appropriate bootlist file
- Generating a boot for the machine using one of the bootfile makefiles

The pathlist contained in the bootlist is for the `MWOS` directory structure and is relative to the root of the board's port directory. Modules customized for a particular system or CPU board go **down** to the local `CMDS/BOOTOBJS` directory, while generic system modules and VMEbus peripheral boards go **up and over** to reference the modules for use in the boot.

If you choose to move the modules for creating boots to a target's `/H0/CMDS/BOOTOBJS` directory, simply remove all relative pathlists before the `CMDS` part of the pathlist and then place the edited bootlist in an `/H0/BOOTLISTS` directory. The process described in **Chapter 3** for creating the hard drive disk image should have included all the modules needed for making a boot. As you customize some modules on your development system, additional modules may need to be transferred to the target system disk drive.

Note

If you choose to edit any of the bootlist files, do not add a blank line within the file. `os9gen` stops reading lines when it encounters a blank line.

Most utilities have a `-z` option that can be used with the bootlist files. A convenient way to verify that all the modules in your bootlist exist and have valid CRCs, is to execute `ident -qz=bootlists/xxx.bl` from the root of the CPU's port directory (`MWOS/OS9/68040/PORTS/ MVME162`). To add modules to the boot, simply un-comment the line by removing the asterisk (*) at the beginning of the line containing the reference to the module. To remove the module, add the asterisk (*) in the first character position on the line. When ready to generate a boot, use the `xxx_bootfile.make` makefile to generate a bootfile that corresponds to the bootlist you have edited in the `CMDS/BOOTOBJS/BOOTFILES` directory. You may also use the `os9gen` command as described later in this chapter.

The following BOOTLISTS/rom.bl file is the bootlist used for building the boot contained in the ROM image supplied with the MVME162 BLS.

Note

.stb modules named in the bootlists are only available to OEM package customer that have rebuilt the appropriate modules from source.

```
********************************
**  Bootlist for the MVME162
**
** Pathlists are relative to the MWOS/OS9/68040/PORTS/MVME162 or machine
** directories based on and parallel to the MVME162 Port directory.
** NOTE: .stb modules are only available when building modules from SRC.
** NOTE: remove/add leading comment "*" to add/remove modules in the OS-9 boot
*
```

The kernel and IOMan sections allow you to choose one of the four kernels available. The kernel is available in a standard and atomic version.

For More Information

Refer to the ***OS-9 for 68K Processors Technical Manual*** for more information on the differences between the kernels.

With each kernel, you can select the original colored memory allocator or a newer buddy allocator that allocates memory in powers of two. For example, if you ask for 33KB of memory, the system would actually allocate 64KB to the process. Most projects requiring the atomic kernel can be developed under the development kernel with its enhanced debugging and protection capabilities and then moved to an atomic based system once the code is known to function properly. Most users use the OS-9 Unified I/O system to communicate with peripherals and must include the IOMAN matching the kernel in use. Choose one kernel and matching IOMAN.

```
*
* OS-9 Kernel - select one variant:
* All modules named: kernel
*
* Development kernel - Standard memory allocator
../../../68040/CMDS/BOOTOBJS/dker040s
*../../../68040/CMDS/BOOTOBJS/STB/dker040s.stb
* Development kernel - Buddy memory allocator
*../../../68040/CMDS/BOOTOBJS/dker040b
*../../../68040/CMDS/BOOTOBJS/STB/dker040b.stb
* Atomic kernel - Standard memory allocator
*../../../68040/CMDS/BOOTOBJS/aker040s
*../../../68040/CMDS/BOOTOBJS/STB/aker040s.stb
* Atomic kernel - Buddy memory allocator
*../../../68040/CMDS/BOOTOBJS/aker040b
*../../../68040/CMDS/BOOTOBJS/STB/aker040b.stb
*
* Ioman: select one to match selected kernel above:
*
../../../68000/CMDS/BOOTOBJS/ioman_DEV
*../../../68000/CMDS/BOOTOBJS/STB/ioman_DEV.stb
*../../../68000/CMDS/BOOTOBJS/ioman_ATOM
*../../../68000/CMDS/BOOTOBJS/STB/ioman_ATOM.stb
*
```

Init Module

A selection of `init` modules are created by the makefiles for the port. Review the contents of the `init` modules and adjust the fields to match your requirements. The fields of the `init` modules are generally modified by adjusting the values in the `systype.d` file and remaking the `init` module. On resident OS-9 for 68K machines, the contents of the `init` module can be viewed and modified using the `moded` utility. The `init` modules included are those used by the initial bootable media for floppy (`d0`), hard disk (`h0`), tape, and ROM boots. Select only one `init` module.

```
*
* Init module: Select ONLY one.
* init_dd: Init module with /DD initial disk device and runs sysgo
* init_h0: Init module with /h0 initial disk device and runs sysgo
* init_d0: Init module with /D0 initial disk device and runs sysgo
* init_tape: Init module with /DD initial disk device and runs tapestart
* init_rom: Init module with NO initial disk device and runs sysgo
*
CMDS/BOOTOBJS/INITS/init_dd
*CMDS/BOOTOBJS/INITS/init_h0
*CMDS/BOOTOBJS/INITS/init_d0
*CMDS/BOOTOBJS/INITS/init_tape
*CMDS/BOOTOBJS/INITS/init_rom
```

Customization Modules

The following customization modules enable you to include caching for a CPU board (`cache040`), process address space protection (`SSM`), enable bus snooping (`Snoopxxx`), and use a variety of OS-9 P2 modules. The selected modules must also be included in the `init` module's P2 initialization list. Modules may be named in the `init` module that are not actually in the boot, enabling you to change the boot without changing an `init` module. The Atomic kernel does not support the SSM module. Choose all that are appropriate.

```
*
* Customization modules:
*
* Snooper Circuit Enable Module
* (if not present, the snooper is not enabled!)
*
CMDS/BOOTOBJS/snoop162
*CMDS/BOOTOBJS/STB/snoop162.stb
*
* Cache Control module
* (If not present, cache is disabled!)
*
../../../68040/CMDS/BOOTOBJS/cache040
*../../../68040/CMDS/BOOTOBJS/STB/cache040.stb
*
* MMU Control module
*
* ssm040 provides write-thru caching in supervisor state,
* ssm040_cbsup provides copy-back caching in supervisor state
* User state cache mode default (both versions) is write-thru,
* and this can be over-ridden via the CacheList entries in systype.d
*
../../../68040/CMDS/BOOTOBJS/ssm040
*../../../68040/CMDS/BOOTOBJS/STB/ssm040.stb
*
* FPSP/FPU Math emulation modules
*
* FPSP provides 68681/68682 compatibility for the 68040 CPU.
* FPSP is specifically for the 68040 and should not be used with
* 68040Ec and 680040LC processors.
* FPU is a general purpose math emulation module. It provides basic
* float and double support as required by the C libraries.
*
*../../../68040/CMDS/BOOTOBJS/fpsp040
*../../../68040/CMDS/BOOTOBJS/STB/fpsp040.stb
../../../68000/CMDS/BOOTOBJS/fpu
*../../../68000/CMDS/BOOTOBJS/STB/fpu.stb
```

Clock Modules

The `tkxxx` module is a driver providing the system with a periodic interrupt generator. The ticker is required for system time-slicing, sleep times, system date/time, and statistics information. The `rtclock` (`rtcxxx`) module is a driver used to access the system's time of day clock. Generally the time of day clock is a battery-backed chip from which the system's time is set upon reset or power up. Some systems do not have rtcxxx modules.

```
*
* System clock module:
* System ticker hardware driver
*
CMDS/BOOTOBJS/tk162
*CMDS/BOOTOBJS/STB/tk162.stb
*
* Battery backed time of day chip driver
* Real Time Clock module named: rtclock
* rtc162: rtclock module for MVME162
*
CMDS/BOOTOBJS/rtc162
*CMDS/BOOTOBJS/STB/rtc162.stb
```

SCF and Pipeman

The following section includes the Serial Character File manager (SCF) and Pipeman.

The `null` driver is used with the `nil` descriptor for SCF redirection to `/nil` and with Pipeman as the driver for the `/pipe` devices.

The `pipe` module is the standard descriptor for unnamed pipes used by the shell. It uses the default buffer size.

The serial ports section in the following code contains the `scxxx` driver and descriptors describing the serial ports on the board. `term` is generally the same port as used by RomBug.

The `tx` descriptors are generally used for terminals while `px` descriptors are used for printers or devices requiring no line editing.

Some boards have parallel ports. The drivers are generally named `scpxxx`, with the descriptor named `px`. Choose SCF along with all required drivers and descriptors.

```
*
* Sequential Character
* File Managers, Drivers and Descriptors:
* scf: Serial Character File Manager
* null: Null Driver
* nil: Null Driver's device descriptor
* pipeman: Pipe File Manager (used Null Driver)
* pipe: Pipe device descriptor
*
../../../68000/CMDS/BOOTOBJS/scf
*../../../68000/CMDS/BOOTOBJS/STB/scf.stb
../../../68000/CMDS/BOOTOBJS/null
../../../68000/CMDS/BOOTOBJS/nil
../../../68000/CMDS/BOOTOBJS/pipeman
*../../../68000/CMDS/BOOTOBJS/STB/pipeman.stb
../../../68000/CMDS/BOOTOBJS/pipe
*
* SCF Serial port Drivers and Descriptors
*
CMDS/BOOTOBJS/sc162
*CMDS/BOOTOBJS/STB/sc162.stb
CMDS/BOOTOBJS/term
CMDS/BOOTOBJS/t1
CMDS/BOOTOBJS/p1
```

Some boards such as the MVME167 also have parallel ports available. This example shows the driver and descriptor lines for the MVME167.

```
*
* parallel printer port
*
CMDS/BOOTOBJS/scp167
CMDS/BOOTOBJS/p
```

An example peripheral VME serial board is to be included in the system. Its driver and descriptors are included here with the onboard SCF devices. (These lines are not included in the mvme162 rom.bl bootlist.)

```
*
* MVME335 Serial Peripheral board
*
../../../68000/PORTS/MVME335/CMDS/BOOTOBJS/sc335
../../../68000/PORTS/MVME335/CMDS/BOOTOBJS/t10
../../../68000/PORTS/MVME335/CMDS/BOOTOBJS/t11
../../../68000/PORTS/MVME335/CMDS/BOOTOBJS/t12
../../../68000/PORTS/MVME335/CMDS/BOOTOBJS/t13
```

RBF

The next section includes the Random Block File manager (RBF), drivers and descriptors. The RAM driver and `r0` descriptors create a RAM disk of varying sizes. The `dd_r0` is the same device as `r0` but with the device name `/dd`.

You may choose to load the RAM driver in the boot but load the `r0` descriptors from disk after the machine is booted. This allows loading `r0` descriptors for different sized RAM disks without needing to remake the boot. The rom.bl bootlist contains an r0 and dd_r0 descriptor. The init module is set to use /dd. Once the networking has been started on the target, the /dd/SYS directory can be used to store `termcap`, `errmsg`, and other reference files normally looked for on the default device.

The next section is for loading the low level SCSI host adapter driver. The `scsixxx` driver is used by both RBF and SBF high level SCSI drivers to access peripherals on the SCSI bus.

External peripheral cards such as the MVME320 or MVME327 might be included in this area of the bootlist. Select all that are appropriate.

```
**
* Random Block File Manager,
* Drivers and Descriptors:
*
../../../68000/CMDS/BOOTOBJS/rbf
*../../../68000/CMDS/BOOTOBJS/STB/rbf.stb
../../../68000/CMDS/BOOTOBJS/ram
CMDS/BOOTOBJS/r0
CMDS/BOOTOBJS/dd_r0
*CMDS/BOOTOBJS/r0_3m
*CMDS/BOOTOBJS/dd_r0_3m
*
* SCSI Controller
*
../MVME162/CMDS/BOOTOBJS/scsi162
*../MVME162/CMDS/BOOTOBJS/STB/scsi162.stb
*
* SCSI Hard Drive Support
* RBVCCS driver and descriptors
*
../../../68000/CMDS/BOOTOBJS/rbvccs
*../../../68000/CMDS/BOOTOBJS/STB/rbvccs.stb
CMDS/BOOTOBJS/VCCS/h0
CMDS/BOOTOBJS/VCCS/h0fmt
*
* RBSCCS driver and descriptors (obsolete)
*
*../../../68000/CMDS/BOOTOBJS/rbsccs
*../../../68000/CMDS/BOOTOBJS/STB/rbsccs.stb
*CMDS/BOOTOBJS/SCCS/h0
*CMDS/BOOTOBJS/SCCS/h0fmt
*
* SCSI Floppy Drive Support
* RBTEAC driver and descriptors
*
*../../../68000/CMDS/BOOTOBJS/rbteac
*../../../68000/CMDS/BOOTOBJS/STB/rbteac.stb
*CMDS/BOOTOBJS/TEACFC1/d0
```

SBF

The Tape Manager section adds the Serial Block File manager (`SBF`) to the boot. A variety of SCSI tape drives are supported. Each of the drivers requires the `scsixxx` low level SCSI driver be available in memory for the device to be initialized.

```
* Tape Manager:
*
*../../../68000/CMDS/BOOTOBJS/sbf
*../../../68000/CMDS/BOOTOBJS/STB/sbf.stb
*
* Tape Drivers and Descriptors
*
* Archive Viper/DAT drives
*
*../../../68000/CMDS/BOOTOBJS/sbviper
*../../../68000/CMDS/BOOTOBJS/STB/sbviper.stb
*CMDS/BOOTOBJS/VIPER/mt0
*CMDS/BOOTOBJS/VIPER/mt1
*CMDS/BOOTOBJS/VIPER/dat
*
* TEAC Cassette
*
*../../../68000/CMDS/BOOTOBJS/sbteac
*../../../68000/CMDS/BOOTOBJS/STB/sbteac.stb
*CMDS/BOOTOBJS/TEACMT2/mt0
*
* Exabyte drive
*
*../../../68000/CMDS/BOOTOBJS/sbgiga
*../../../68000/CMDS/BOOTOBJS/STB/sbgiga.stb
*CMDS/BOOTOBJS/EXABYTE/mt2
*
```

Initial System Process

The next section selects the first process executed by the system. The boot's `init` module must reflect the name of the module selected here. `Sysgo` is a general purpose program that sets up the initial `CMDS` directory and tries to execute a `startup` file with the help of `mshell`. `tapestart` and `shell` are used by the tape distribution media to start the system and then create a system RAM disk from the second file on a tape. Including `shell or mshell` in the boot is useful when there is not a device from which to load the shell module at boot time. The `init` module can be modified to use `shell or mshell` as the initial process and to execute a `startup` file or sequence without using `sysgo`.

```
* Initial system process:
* sysgo: runs SYS/startup script, (re)forks mshell
* sysgo_nodisk: forks mshell (no startup file run)
* sysgo_tsmon: runs SYS/startup script, chains to tsmon
* sysgo_shell: runs startup script, (re)forks shell
* compatible with earlier OS-9 versions
* NOTE: sysgo modules require mshell except sysgo_shll
* which requires shell.
* NOTE: the init module can be configured to use shell
* or mshell as the initial process instead of sysgo.
* mshell: extended functionality shell (standard)
* shell: origininal small shell
* tapestart: used in tape based initial shipping boots
*
../../../68000/CMDS/BOOTOBJS/sysgo
*../../../68000/CMDS/BOOTOBJS/sysgo_nodisk
*../../../68000/CMDS/BOOTOBJS/sysgo_tsmon
*../../../68000/CMDS/BOOTOBJS/sysgo_shell
../../../68000/CMDS/mshell
*../../../68000/CMDS/shell
*../../../68000/CMDS/tapestart
```

Additional Modules and Utilities

This area includes additional modules such as the I/O and Math shared trap libraries. `csl` is used for Ultra C libraries. `cio`, `math`, and `math881` are used with programs compiled with the original Microware C Compiler and libraries. See the Ultra C/C++ documentation for more information on these modules. Additional utilities and applications may be added to the boot. Adding additional modules is particularly useful when using BootP to boot diskless systems or as in this case, when building an embedded boot for inclusion in the ROM.

```
**
* Additional system Support modules can be added here.
* csl: C Shared Library for Ultra C compiled binaries
* cio: C I/O library for Microware C compiled binaries
* math881: Math881 simulation library for Microware C
* compiled binaries
../../../68020/CMDS/csl
*../../../68020/CMDS/cio
*../../../68020/CMDS/math881
*
* OS Utilitities used during System configuration
* See utilities manual for usage.
*
../../../68000/CMDS/attr
../../../68000/CMDS/break
*../../../68000/CMDS/chown
*../../../68000/CMDS/copy
../../../68000/CMDS/date
*../../../68000/CMDS/dcheck
../../../68000/CMDS/deiniz
*../../../68000/CMDS/del
*../../../68000/CMDS/deldir
../../../68000/CMDS/devs
../../../68000/CMDS/dir
../../../68000/CMDS/diskcache
*../../../68000/CMDS/dsave
*../../../68000/CMDS/dump
*../../../68000/CMDS/echo
*../../../68000/CMDS/fixmod
../../../68000/CMDS/format
../../../68000/CMDS/free
*../../../68000/CMDS/frestore
*../../../68000/CMDS/help
../../../68000/CMDS/ident
../../../68000/CMDS/iniz
*../../../68000/CMDS/kermit
*../../../68000/CMDS/link
../../../68000/CMDS/list
*../../../68000/CMDS/lmm
```

```
../../../68000/CMDS/load
../../../68000/CMDS/makdir
../../../68000/CMDS/mdir
../../../68000/CMDS/mfree
*../../../68000/CMDS/os9gen
*../../../68000/CMDS/p2init
../../../68000/CMDS/partition
../../../68000/CMDS/pd
../../../68000/CMDS/printenv
../../../68000/CMDS/procs
../../../68000/CMDS/rename
../../../68000/CMDS/save
../../../68000/CMDS/setime
../../../68000/CMDS/tar
../../../68000/CMDS/tmode
*../../../68000/CMDS/tsmon
../../../68000/CMDS/unlink
```

Networking Modules

The remaining sections add networking modules to the boot. These modules can also be loaded from disk using the `loadspf` shell script in the `SYS` directory. The first section is the majority of the hardware independent system modules used in SoftStax.

For More Information

For more information SoftStax and the NFS Client package, see **Chapter 5**.

```
*
* SPF/Lancom Networking
*
* System MBuf Service
*
../../../68020/CMDS/BOOTOBJS/SPF/sysmbuf
*../../../68020/CMDS/BOOTOBJS/SPF/STB/sysmbuf.stb
*
* Pseudo Keyboard FM/Driver/Descriptors
* pkman: File Manager
* pkdvr: Driver
* pk: pkdvr descriptor
* pks: pkdvr (scf) descriptor
* NOTE: all required with Telnet and other applications needing
* Pseudo Keyboard fuctionality
*
../../../68020/CMDS/BOOTOBJS/SPF/pkman
*../../../68020/CMDS/BOOTOBJS/SPF/STB/pkman.stb
../../../68020/CMDS/BOOTOBJS/SPF/pkdvr
*../../../68020/CMDS/BOOTOBJS/SPF/STB/pkdvr.stb
../../../68020/CMDS/BOOTOBJS/SPF/pk
../../../68020/CMDS/BOOTOBJS/SPF/pks
*
* SPF/Lancom FM/Drivers/Descriptors
*
* SPF/Lancom Protocol Drivers/Descriptors
* spf: SoftStax File Manager
* spip: SPF IP driver module
* ip0: spip descriptor module
* sptcp: SPF TCP driver module
* tcp0:: sptcp descriptor module
* spudp: SPF UDP driver module
* udp0: spudp descriptor module
* spraw: SPF RAW driver module
* raw0: spraw descriptor module
* sproute: SPF Routing driver module
* route0: sproute descriptor module
*
../../../68020/CMDS/BOOTOBJS/SPF/spf
*../../../68020/CMDS/BOOTOBJS/SPF/STBspf.stb
../../../68020/CMDS/BOOTOBJS/SPF/spip
*../../../68020/CMDS/BOOTOBJS/SPF/STB/spip.stb
../../../68020/CMDS/BOOTOBJS/SPF/ip0
../../../68020/CMDS/BOOTOBJS/SPF/sptcp
*../../../68020/CMDS/BOOTOBJS/SPF/STB/sptcp.stb
../../../68020/CMDS/BOOTOBJS/SPF/tcp0
../../../68020/CMDS/BOOTOBJS/SPF/spudp
*../../../68020/CMDS/BOOTOBJS/SPF/STB/spudp.stb
../../../68020/CMDS/BOOTOBJS/SPF/udp0
../../../68020/CMDS/BOOTOBJS/SPF/spraw
*../../../68020/CMDS/BOOTOBJS/SPF/STB/spraw.stb
../../../68020/CMDS/BOOTOBJS/SPF/raw0
../../../68020/CMDS/BOOTOBJS/SPF/sproute
*../../../68020/CMDS/BOOTOBJS/SPF/STB/sproute.stb
../../../68020/CMDS/BOOTOBJS/SPF/route0
*
* Ethernet Support Driver/Descriptor
* Required for sp162/spie0 below
* spenet: Ethernet protocol driver
* enet: spenet descriptor
*
../../../68020/CMDS/BOOTOBJS/SPF/spenet
../../../68020/CMDS/BOOTOBJS/SPF/enet
*
```

Networking Configuration Modules

These modules are used to configure your system for the local network environment and the specific machine modules used to connect the network. Choose the proper inetdb module.

For More Information

For more information about building an inetdb module for your network, see **Chapter 5**.

The example shows adding an ethernet driver to the boot. Other communication drivers can also be included for SLIP or PPP networking connections. The inetdb2 module is generally used to specify information about the particular machine, such as IP addresses and hostname.

```
* Network specific modules
* netdb_local: resolve network info from inetdb modules
* netdb_dns: resolve from inetdbs then use DNS
* inetdb: local network info module
* NOTE: often inetdb is made with just network info so
* it can be used in all machines. inetdb2 etc. modules
* are created for machine specific info.
*
*../../../68020/CMDS/BOOTOBJS/SPF/netdb_local
../../../68020/CMDS/BOOTOBJS/SPF/netdb_dns
*../../../68020/CMDS/BOOTOBJS/SPF/inetdb
*
* System specific modules
*
* inetdb2: Machine lancom configuration Module
* sp162: Ethernet Hardware Driver
* spie0: Ethernet Hardware Descriptor
*
CMDS/BOOTOBJS/SPF/inetdb
CMDS/BOOTOBJS/SPF/inetdb2
CMDS/BOOTOBJS/SPF/sp162
CMDS/BOOTOBJS/SPF/spie0
```

Networking Utilities

The following section adds the SoftStax and LAN Communications Pak Client utilities to support remote connections with the system. These include the status program, resident configuration tools, `telnet`, `ftp`, and various daemons. On systems with a disk available, these utilities are generally not placed in the boot; instead they are loaded from the `CMDS` directory automatically by the shell as they are needed.

```
*
* SPF/Lancom Utilities
*
* SPF Startup/Configuration Utilitities
* mbinstall: install sysmbuf p2 module
* ipstart: start spf system
* ifconfig: configure spf/lancom devices
* route: control/display routing entries
* ndbmod: build/modify Inetdb module
* netstat: display lancom information
* idbdump: display inetdb modules
* hostname: set/display system hostname
*
../../../68020/CMDS/mbinstall
../../../68020/CMDS/ipstart
*../../../68020/CMDS/ifconfig
../../../68020/CMDS/route
../../../68020/CMDS/ndbmod
../../../68020/CMDS/netstat
../../../68020/CMDS/idbdump
../../../68020/CMDS/hostname
*
```

```
* SPF Applications
* arp:
* bootpd: Bootp server
* ftp: Ftp user program
* ftpd: FTP daemon (or use inetd)
* ftpdc: FTP daemon child (use w/ftpd or inetd)
* inetd: Master Daemon
* ping: User "system up?" utility
* routed: routing Daemon
* telnet: Telnet user program
* telnetd: Telnet daemon (or use inetd)
* telnetdc: Telnet daemon child (use w/telnetd or inetd)
* tftpd: TFTP server daemon
* tftpdc: TFTP server daemon child (required w/tftpd)
*../../../68020/CMDS/arp
*../../../68020/CMDS/bootpd
../../../68020/CMDS/ftp
../../../68020/CMDS/ftpd
../../../68020/CMDS/ftpdc
*../../../68020/CMDS/inetd
*../../../68020/CMDS/ping
../../../68020/CMDS/routed
../../../68020/CMDS/telnet
../../../68020/CMDS/telnetd
../../../68020/CMDS/telnetdc
*../../../68020/CMDS/tftpd
*../../../68020/CMDS/tftpdc
```

NFS

This section includes the system modules and utilities used to support NFS client and NFS server functionality on the OS-9 target system.

For More Information

Refer to **Chapter 5** for more information about creating the descriptors loaded in these sections.

```
**
* NFS Client Utilities
* nfsc: nfs client daemon (required with nfs FM)
* mount: mount nfs served devices
* rpcdbgen: generate rpcdb module
* rpcdump: display rpcdb contents
* nfsstat: nfs status program
* rpcinfo: display rpc information from specific calls
*
*../../../68020/CMDS/nfsc
*../../../68020/CMDS/mount
*../../../68020/CMDS/rpcdbgen
*../../../68020/CMDS/rpcdump
*../../../68020/CMDS/nfsstat
*../../../68020/CMDS/rpcinfo
*
* NFS Server application Modules
* exportfs: export file systems
* portmap: port mapping daemon (required)
* nfsd: nfs daemon (required)
* mountd: mount request servicing daemon
* showmount: show systems that have mounted locally exported devices
*
*../../../68020/CMDS/exportfs
*../../../68020/CMDS/portmap
*../../../68020/CMDS/nfsd
*../../../68020/CMDS/mountd
*../../../68020/CMDS/showmount
```

Remote Hawk Debugging

This section includes modules that support the Hawk user-state debugging facility. SPF networking is required in addition to these modules.

For More Information

Refer to ***Using Hawk*** for more information about Hawk debugging.

```
*    Hawk Debugging
*
*    User State
*     SPF Networking is required in addition to these modules
*    spfndpd:     Network Debugger Protocol (NDP) Daemon
*    spfndpdc:    NDP Server Daemon Child
*    ndpio:       NDP I/O Server
*    spfnppd:     Hawk Profiler Daemon
*    spfnppdc:    Hawk Profiler Daemon Child
*
*../../../68000/CMDS/spfndpd
*../../../68000/CMDS/spfndpdc
*../../../68000/CMDS/ndpio
*../../../68000/CMDS/spfnppd
*../../../68000/CMDS/spfnppdc
*
```

Making Boots

There are three basic ways of booting the OS-9 operating system.

The first is to use one of several external sources to load the OS-9 Boot into memory. The bootp network booter, hard disk, floppy, and tape booters are examples of external source booters available in the OS-9 PROM.

The second is to use the embedded boot concept where the OS-9 system image is contained in the ROM/FLASH image on the board.

The third method is to load a boot image via the console port and ROMBUG. This is the most time consuming method if the image is loaded via the console port.

I/O based Booters

The `d0_bootfile.make`, `h0_bootfile.make`, and `viper_tape_bootfile.make` makefiles located in the `<port>/BOOTS` directory are used to create OS-9 bootfiles in the `CMDS/BOOTOBJS/BOOTFILES` subdirectory. The same makefile can be used to `ident` the boot you just made by typing the command `os9make -f=h0_boofile.make ident.` A bootfile is the image of an OS9Boot. Once it is created, there are a variety of methods used to make the file available to the target machine.

Hard disk boot	`os9gened` RBF boot disk
Floppy boot	`os9gened` RBF boot disk
Tape boot	`tapegened` SBF boot tape
Network boot	OS-9, Windows, or UNIX BootP server

Hard Disk Boot

Hard Disk Boot Method 1

Step 1. Create the boot by modifying the `<port>/BOOTLISTS/h0.bl` bootlist file to select the desired modules to be part of your `OS9Boot` image.

Step 2. Change into the `BOOTS` directory and enter **os9make -f=h0_bootfile.make**. This creates the bootfile image `<port>/CMDS/BOOTOBJS/BOOTFILES/h0.bf`.

Step 3. Once the image is built, ftp the file to the target system's hard disk root directory. Be sure to use binary mode when ftping the file.

Step 4. Use **os9gen** on the target machine to "gen" the `OS9Boot` on the hard drive using the following command:

os9gen /h0fmt -eb=<buffer size> h0.bf

Note

<buffer size> should be large enough to hold the entire bootfile.

Note

The use of the format enabled descriptor `h0fmt` is required in order to write the sector 0 (zero) bootrecord information as part of the `OS9gen` procedure of creating the `OS9Boot` on the drive.

Hard Disk Boot Method 2

Step 1. Transfer the files used for creating the boot to the target machine as described in **Chapter 3**.

Step 2. Perform the **os9gen** command on the `/h0fmt` device.

Step 3. From the root directory (/h0) execute the following command:

```
os9gen /h0fmt -eb=<buffer size> -z=bootlists/h0.bl
```

Note

Be sure to have a secondary means of booting the target machine in the event the new boot is flawed and fails to boot the system.

Floppy Boot

From a running OS-9 for 68K machine you can create a floppy disk boot. The `booter` attempts to read a number of OS-9 RBF disk formats. However, the OS-9 Universal format is recommended since it is currently the default used by most /d0 device descriptors. The following examples show two methods of making a boot on floppy disk.

Step 1. Modify the `<port>/BOOTFILES/d0.bl` bootlists as desired.

Step 2. Change to the `<port>/BOOTS` directory and execute the following command:

```
make -f=d0_bootfile.make
```

Step 3. FTP the `<port>/CMDS/BOOTOBJS/BOOTFILES/d0.bl` file to a running OS-9 machine using binary transfer mode and placing the file in the `/h0/cmds/bootobjs/bootfiles` directory as `d0.bf`

Step 4. From the OS-9 machine, on the root directory, execute the following command:

```
os9gen -eb=<buffer size> /d0 CMDS/BOOTOBJS/BOOTFILES/d0.bf
```

or alternatively, if you have moved the files needed for creating a boot to the target system, you can enter the command:

```
os9gen /d0 -eb=<buffer size> -z=bootlists/d0.bl
```

In either case, the boot put on the floppy may be for a diskless system, disk-based system using `/d0` or `/h0` as the initial system disk, or a system using an NFS mounted disk for the system disk.

`os9gen` Create and Link the OS9Boot File

Syntax

```
os9gen [<opts>] <devname> {<path>}
```

Description

`os9gen` creates and links the `OS9Boot` file required on any disk from which OS-9 for 68K is to be bootstrapped. Following are some examples of how you can use `os9gen`:

- Make a copy of an existing boot file.
- Add modules to an existing boot file.
- Create an entirely new boot file for a different system.

To use the `os9gen` utility, type `os9gen` and the name of the device on which to install the `OS9Boot` file. `os9gen` creates a working file called `TempBoot` on the device specified. Each file specified on the command line is opened and copied to the `TempBoot` file.

Note

Only super users (`0.n`) may use this utility. Also, you can only use `os9gen` on format-enabled devices.

After all input files are copied to `TempBoot`, any existing `OS9Boot` file on the target device is renamed `OldBoot`. If an `OldBoot` file is already present, `os9gen` deletes it before renaming `OS9Boot`.

`TempBoot` is then renamed `OS9Boot`. Its starting address and size are linked in the disk's identification sector (`LSN 0`) for use by the OS-9 bootstrap firmware.

Options

`-?`	Display the options, function, and command syntax of `os9gen`.
`-b=<num>`	Assign `<num>k` of memory for `os9gen`. Default memory size is 4K.
`-e`	Extended Boot. Allows you to use large (greater than 64K) and/or non-contiguous files. **Note:** Bootstram ROMS must support this feature.
`-q=<file>`	Quick Boot. Set sector zero pointing to `<file>.`
`-r`	Remove the pointer to the boot file. This file is not deleted.
`-x`	Search the execution directory for pathlists.
`-z`	Read the file names from standard input.
`-z=<file>`	Read the file names from `<file>`.

If your boot file is non-contiguous or larger than 64KB, use the `-e` option.

Note

Your bootstrap ROMs must support this feature. If they do not, you should not use this option.

If you use the `-z` option, `os9gen` first uses the files specified on the command line and then the file names from its standard input, or from the specified pathlist, one pathlist per line. If the names are entered manually, no prompts are given and the end-of-file key (usually `<escape>)` is used to terminate input.

To determine what modules are necessary for your boot file, use the `ident` utility with the `OS9Boot` file that came with your system.

The `-q` option updates information in the disk's Identification Sector by directing it to point to a file already contained in the root directory of the specified device.

The `-q` option is useful when restoring the `OldBoot` file as the valid boot on the disk. `os9gen` renames the specified file to be `OS9Boot` and saves the current boot as described previously.

The `-r` option removes the pointer to the boot file but does not delete the file. This is useful if you delete the bootfile from your disk (using the `del` command). Deleting the bootfile from the file structure **does not** remove the bootfile pointers from the disk's Identification Sector. You can also use it to make a disk non-bootable without deleting the actual bootfile.

Examples

This command manually installs a boot file on device `/d1`, which is an exact copy of the `OS9Boot` file on device `/d0`.

```
$ os9gen /d1 /d0/os9boot
```

The following three methods manually install a boot file on device `/d1`. The boot file on `/d1` is a copy of the `OS9Boot` file on device `/d0` with the addition of modules stored in the files `/d0/tape.driver` and `/d2/video.driver`:

Manual Bootfile Installation Method 1

```
$ os9gen /d1 /d0/os9boot /d0/tape.driver /d2/video.driver
```

Manual Bootfile Installation Method 2

```
$ os9gen /d1 /d0/os9boot -z
/d0/tape.driver
/d2/video.driver
[ESCAPE]
```

Manual Bootfile Installation Method 3

```
$ os9gen /d1 -z
/d0/os9boot
/d0/tape.driver
/d2/video.driver
[ESCAPE]
```

You can automatically install a boot file by building a **bootlist** file and using the `-z` option to either redirect `os9gen` standard input or use the specified file as input:

```
$ build /d0/bootlist                Create file bootlist
? /d0/os9boot                       Enter first file name
? /d0/tape.driver                   Enter second file name
? /d2/video.driver                  Enter third file name
?                                   * V1.2 of video driver
                                    * Comment line
? [RETURN]                          Terminate build
$ os9gen /d1 -z </d0/bootlist       Redirects standard input
$ os9gen /d1 -z=/d0/bootlist        Reads input from pathlist
```

Note

`os9gen` treats any input line preceded by an asterisk `(*)` as a comment.

The following command makes the `OldBoot` file the current boot and saves the current `OS9Boot` file as `OldBoot`:

```
$ os9gen /d1 -q=oldboot
```

Note

`os9gen` is an OS-9 hosted utility.

Tape Booting

An OS-9 for 68K machine can be used to create a tape boot. The tape boot can optionally initialize a RAM disk from an image of the disk created prior to creating the tape. The `tapegen` utility is used to create the boot tape and the `tapestart` utility is used to initialize the RAM disk from the tape. The `init_tape init` module uses this method.

Tape startup Sequence

The tape booting procedure operates in a manner similar to normal disk booting. The `tapeboot` code in the supplied ROMs reads a header block (equivalent to Sector 0 of a disk) from the tape. This block contains the location and size of the bootstrap file on the tape. It allows the booting code to locate and read the bootstrap file into system memory.

The format of the boot tape header block also allows the specification of additional files on the tape for application-specific purposes. In the standard distribution media, this feature allows the `RAM disk image` to be stored on tape. When the system is booted, an application program is executed. This application program reads the `RAM disk image` from tape and writes it into the RAM disk itself.

Supplied Utilities

Two utility programs are supplied on the distribution media to support the concepts described above. These utilities are `tapegen` and `tapestart`.

The `tapegen` utility creates the **bootable** tape. `tapegen` is a standard utility performing a function similar to the `os9gen` utility. Both utilities place the bootstrap file onto the media and mark the media identification block with information regarding the bootstrap file. In addition, `tapegen` can optionally place **initialized data** on the tape for application-specific purposes.

To use the **initialized data** feature, use the following procedure:

Step 1. Create a RAM disk descriptor that sets the RAM disk size as required on the target machine.

Step 2. Load the descriptor.

Step 3. Initialize the descriptor with `iniz`.

Step 4. Fill the RAM disk with the files desired on the target machine.

Step 5. Save the raw image of the RAM disk to tape.

Using this method, `startup` script files, `termcap`, and `errmsg` files are available on a target where there is no other **disk** installed. The `-i` option of `tapegen` is then used to point to the RAM disk on the host.

Note

The target boot must contain the ram driver and a descriptor with a device size equal to that of the initialized RAM disk image on tape. When edition #24 or greater of the RAM driver is used with the target boot, the descriptor must also be format enabled as well.

`tapegen` Put Files on Tape

Syntax

```
tapegen [<opts>] <filename> <filename>
```

Description

The `tapegen` utility creates the **bootable** tape. `tapegen` is a standard utility performing a function similar to the `os9gen` utility. Both utilities place the bootstrap file onto the media and mark the media identification block with information regarding the bootstrap file. In addition, `tapegen` can optionally place **initialized data** on the tape for application-specific purposes.

Options

`-?`	Displays the options, function, and command syntax of `tapegen`.
`-b=<bootfile>`	Installs an OS-9 boot file.
`-bz`	Reads boot module names from standard input.
`-bz=<bootlist>`	Reads boot module names from the specified bootlist file.
`-c`	Checks and displays header information.
`-d=<dev>`	Specifies the tape device name. The default is `/mt0`.
`-o`	Takes the tape drive off-line when finished.
`-t=<target>`	Specifies the name of the target system.
`-i=<file>`	Installs an initialized data file on the tape. This is usually a RAM disk image.
`-v=<volume>`	Specifies the name of the tape volume.
`-z`	Reads filenames from standard input.
`-z=<file>`	Reads filenames from the specified file.

Examples

The following example makes a bootable tape. The disk image is derived from the /dd device.

```
$ tapegen -b=OS9Boot.tape -i=/dd@
   "-v=OS-9/68K Boot Tape" -t=MySystem
```

This example makes a bootable tape with no initialized data file. The header information is displayed after writing the tape.

```
$ tapegen -b=OS9Boot.h0 -c
```

Note

`tapegen` is an OS-9 hosted utility.

`tapestart` Start System from Tape

Syntax

```
tapestart [<opts>] [<device name>] [<opts>]
```

Description

The `tapestart` utility supplied on the distribution media is an application-specific program used to initialize the RAM-disk contents. For tape booting configurations, `tapestart` allows the RAM disk to be fully initialized **prior** to forking the standard `SysGo` module.

Function

`tapestart` reads the header block from the tape, determines the position and size of the initialized data file, and copies this data to the specified device.

Note

When creating boot tapes with initialized data files and using the supplied `tapestart` utility, the size of the initialized data file **must** be the same as the size of the device to which the data is being written.

To use the `tapestart` utility, type `tapestart` followed by any desired options.

Options

`<device name>`	Specifies the RBF device to initialize. This defaults to whatever is specified in the `init` module.
`-?`	Displays the options, function, and command syntax of `tapestart`.

`-d=<tdevname>`	Specifies the tape device name. The default is `/mt0`.
`-o`	Forces the tape drive off-line when finished. **Note:** On some drives, this ejects the tape.

Examples

The following example writes the initialized data file on `/mt0` to the system's default device.

```
$ tapestart
```

This example writes the initialized data file on `/mt2` to `/r0`.

```
$ tapestart -d=/mt2 /r0
```

BootP Booting

The target system can be booted from a Windows or UNIX hosted BootP server if available. The BLS and OEM packages also contain an OS-9 for 68K Bootstrap Protocol Server (BootP) that enables you to create a BootP server for your target system from a separate disk based OS-9 system.

For More Information

Refer to ***Using LAN Communications*** for information about the OS-9 BootP server and its installation.

Creating the Boot

Use an existing makefile (for example `BOOTS\h0_bootfile.make`) and the corresponding bootlist file (for example `BOOTLISTS\h0.bl`) to create an appropriate boot for your target system. You can either make changes directly to an existing bootlist file and then use the corresponding makefile to build your bootfile, or you can make a copy of these files (for example copy to `bootp_bootfile.make/bootp.bl`) and then make your changes to the new files.

Note

If you choose to create new makefile and/or bootlist files, be sure to update the following definitions in the new makefile:

- `MAKER` (new makefile name)
- `OFILE` (new bootfile name)
- `FILES` (new bootlist name)

Also update the following definition in `BOOTS\makefile`:

- `TRGTS` (add new makefile name)

Once you have created an appropriate boot, move the bootfile to the directory used by your BootP server.

Note

For TFTP booting, please refer to the **Customizing ROM images** section.

Booting the boot

After resetting the target system, select the **ie** boot menu item to boot boards such as the MVME162, MVME167, MVME172, and MVME177 using the I82596 Ethernet controller. If automatic booting is desired, the sequenced order of booters should be entered and the `rombug` boot menu disabled through the use of the `reconfig` command.

In addition, if the debugger is also disabled, upon power up or reset, the system continues trying the backup sequence forever. Booters may be specified more than once in the booter preference sequence.

Customizing ROM images

Making new or additional ROMs is another method used to modify the boot on the target board. For developers of ROMed target systems, this is usually done once the code has been debugged and proven functional. This is a good way for BLS users to provide an **always present** backup boot when the primary boot method is from an attached hard disk.

Modifying the ROM Bootfile

The ROM Bootfile uses the `rom.bl` bootlist in your board's `<port>/BOOTLISTS` directory to select files/modules for inclusion in the boot image.

Makefiles for rebuilding the ROM boot are located in the `ROM_CBOOT` directory of your board's port directory. The makefiles create merged components in the `CMDS/BOOTOBJS/NOBUG` or `CMDS/BOOTOBJS/ROMBUG` subdirectories of your board's port directory depending on whether or not you choose to include rombug in your ROM image. The example concentrates on the ROMBUG version.

Step 1. Edit the `BOOTLISTS/rom.bl` file to add or change the modules to include in your boot. Adjust the size of your boot to the available size of the PROM/FLASH memory on your board. The ROMBUG section of the PROM will use the first 128k of ROM space. Therefore, on a board with 1 Meg of ROM, there will be $E0000 / #917504 bytes of ROM available for the boot.

Step 2. To determine the size of your boot, change to the root of your board's port directory (`MWOS/OS9/68040/PORTS/MVME162` for our example 162 system).

Step 3. Use the command **os9merge -z=BOOTLISTS/rom.bl >boot.tmp** to create a boot.

Step 4. Look at the size of the file using the **dir** command.

You can also use **os9ident -q -z=BOOTLISTS/rom.bl** to quickly identify modules within your boot.

Step 5. Once you are satisfied with the revised bootlists, change to the `ROM_CBOOT` directory and rebuild the ROM image.

os9make -f=rombug.make causes the ROMBUG version of the ROM image to be rebuilt and placed in the `<PORT>/CMDS/BOOTOBJS/ROMBUG/rombugger` file.

The `rom_bootfile.make` makefile calls the padrom utility to pad the size of the bootfile to a known size. Adjust the makefile if you wish to pad to a size different than $E0000 bytes.

The **makefile rom_booters** generally builds the ROMBUG section of the ROM image and padroms that section to 128k. The two pieces are then combined by `rombug.make` to build the 1 meg ROM image used in the PROMS provided with the BLS.

The C based rombooter used on the MVME boards will find any number of modules contiguous with the OS-9 kernel, which must be the first module in any boot. If a board contains 4 meg of FLASH memory, the booter would allow a boot of up to 4 meg minus 128k for ROMBUG. If the FLASH memory was in two separate, noncontiguous banks, the first bank would set the maximum embedded boot size. However, the system’s init module could be modified to request that the OS-9 kernel search the second FLASH bank for additional modules after the kernel has taken control of the system.

For More Information

Refer to ***OS-9 for 68K MVME Board Guide*** for information about the layout of each CPU’s RomBug ROMs.

initext File

To customize the ROM's initial startup code, Microware provides a file called `initext.a` (initialization extension). This file enables BLS users to create a bootstrap ROM performing special initialization of the system during the initial startup of the bootstrap process, without requiring you to customize and re-make the core of the distributed bootstrap ROMs.

The majority of users do not need to modify the `initext` code installed in their ROMs. A typical application for this code is initializing custom hardware that would otherwise interfere with the booting process. For example, hardware that asserts an interrupt to the CPU on power-up and must be accessed to clear the interrupt.

The modification of the `initext` code is optional. If you need to modify the code, take the following steps to include the modified code in the ROMs:

Step 1. Examine the supplied `initext.a` file to gain an insight into when the code is called and what functions you can perform. This is an actual file you can modify according to your requirements.

Step 2. Modify `initext.a` to meet your requirements.

Step 3. Re-make your customized version using one of the make commands. If your make changes to other modules to be contained in the ROMs, performing a full make may be required to update those modules. In the `ROM_CBOOT` directory you can simply type **os9make** or you can run only the version you prefer utilizing one of the following commands.

os9make -f=rom.make

os9make -f=rombug.make

Step 4. Make a new set of boot ROMs or program into the board's FLASH memory if available.

For More Information

See ***OS-9 for 68K MVME Board Guide*** for more information.

Note

OEM package customers have many more options for customizing ROM images.

Download Booting

This method is used most often during the development phase of a project. It requires operator intervention to load the boot code at the appropriate memory location and time. The sample session provided below shows downloading S-Records using RomBug's download facility.

For More Information

For more information on the RomBug download command see the ***Using RomBug*** manual.

Once the memory for the `ml` boot has been allocated, the boot may also be moved into place by using another CPU board in the same backplane to copy the code into place.

Step 1. At power up or after reset, type **g <Return>** at the `rombug` debugger prompt (assuming it is enabled) to receive the boot menu.

Step 2. Select the Boot Manually Loaded Bootfile Image option by typing **ml <Return>**. You are then prompted for the size of the boot file to be downloaded.

Step 3. Enter the size of the bootfile. The booter allocates enough memory to hold the boot and responds with the beginning address of the allocated memory in the yes/no/quit question displayed below.

```
BOOTING PROCEDURES AVAILABLE ---------------- <INPUT>
Boot from SCSI(SCCS) hard drive ------------ <hs
Boot from Viper tape drive ----------------- <vs>
Boot from Teac SCSI floppy drive ----------- <fs>
Boot from BOOTP i82596 LANC ---------------- <ie>
Boot from BOOTP backplane ------------------ <bp>
Boot from a non-volatile (Static) RAM disk - <sd>
Load Bootfile from ROM --------------------- <lr>
Boot from ROM ------------------------------ <ro>
Boot Manually Loaded Bootfile Image -------- <ml>
Reconfigure the boot system ---------------- <rc>
Restart the system ------------------------- <q>

Select a boot method from the above menu: ml<return>
Enter loadfile size<cr>: 85442<return>

Is the loadfile image ready at 0x8310:
(<yes>/<no>/<quit>)?
```

Step 4. Press the abort button on the CPU to return to the debugger.

```
<Aborted>
dn: 00000000 00000000 00000000 00000000 00000005 00000001 FFA009FE 00007000
an: FFA19076 0000422A FFA18400 FFF45004 00008010 00006FF0 00000010 00006F3A
pc: FFA021A0 sr:2704 (--SI-7--Z--)t:OFF msp:FFE10000 usp:00000010 ^isp^
0xFFA021A0 >67F8  beq.b 0xFFA0219A
RomBug:
```

Step 5. Set `.r7` as the default relocation register

```
RomBug: @7<ret>
```

Step 6. Load the download memory address into `.r7`

```
RomBug: .r7 8310<ret>
```

Step 7. Set a download I/O delay value of 20 in `.d0`

```
RomBug: .d0 20<ret>
```

Step 8. Execute the `dl` (download) command.

```
RomBug: dl<ret>
00008400
00008600
00008800
```

Step 9. At this point, start the download from the host system. The download starts and displays the address at 512 byte intervals until the final S-record has been received.

```
0001C800
0001CA00
0001CC00
0001CE00
0001D000
load done
```

Step 10. Type **g <Return>** to resume at the download booter's prompt. Pressing **<Return>** reprints the prompt. Typing **y <Return>** causes the download booter to start the code that was downloaded to bring up the system.

```
RomBug: g<return>
<ret>
(<yes>/<no>/<quit>)? yes<return>
A valid OS-9 bootfile was found.
-t -np
*
* OS-9/68000 - Version 3.0
* Copyright 1984, 1993 by Microware Systems Corporation
*
* The commands in this file are highly system dependent and should
* be modified by the user.
*
setime <>>>/term  ;* start system clock
 yy/mm/dd hh:mm:ss [am/pm]
Time:
```

Note

This booter can also be used in conjunction with other CPU cards on the VME Bus to load the image at the allocated address rather than downloading the boot image via RomBug.

ROM Customization (OEM Package)

The Embedded OS-9 for 68k (OEM) Package includes the ability to customize the booters, initial memory search lists, and rebuild additional sections of the CBOOT ROMs.

For More Information

Refer to ***OS-9 for 68K Processors OEM Installation Manual*** for more information.

Chapter 5: Configuring Your System for Networking

This chapter provides pointers to the appropriate OS-9 documentation for configuring your OS-9 system for networking:

- ***Using LAN Communications***

  Chapter 2: LAN Communications Overview provides an overview of the network modules.

  Appendix A: Configuring LAN Communications describes the network configuration that may need to occur as well as a walk-through of the LAN Communications portion of the bootlist file.

- ***Using Network File System/Remote Procedure Call***

  Appendix A: Getting Started With Network File System/Remote Procedure Call provides an overview, configuration description, and bootlist walk-through of the NFS modules.

Note

To view these documents, select `View Documentation` from the Microware OS-9 for 68K CD-ROM.

RadiSys.
MICROWARE SOFTWARE

Chapter 6: Developing Your System in Hawk

This chapter provides pointers to the appropriate OS-9 documentation for using the Hawk Integrated Development Environment for your OS-9 project:

- ***Getting Started with Hawk*** (includes an example Hawk project).
- ***Using Hawk*** (includes a description of the Hawk interface and instructions for debugging over a SLIP connection).

Note
To view these documents, select `View Documentation` from the Microware OS-9 for 68K CD-ROM.

RadiSys.
MICROWARE SOFTWARE

Appendix A: SCSI Information

This appendix includes the following topics:

- **Overview**
- **SCSI Software Configurations — Implementation Notes**

Overview

This appendix provides information regarding SCSI software and hardware configurations. The first section contains details about:

- SCSI IDs
- The `rbsccs` and `rbvccs` device drivers
- The differences between `rbsccs` and `rbvccs` device descriptors and other important device descriptor fields
- How to convert a `rbsccs` drive to a `rbvccs` drive

The second section deals with the configuration of SCSI peripherals. It contains configuration information for:

- Embedded SCSI hard disk support
- Embedded SCSI floppy disk support
- Embedded SCSI tape support

SCSI Software Configurations — Implementation Notes

SCSI IDs

The default descriptors provided with the Board Support Packages map devices to specific SCSI IDs. Take care to set the device IDs on each of the devices. Failure to set the IDs correctly is the most common problem in the initial setup. The following are valid SCSI IDs:

Table A-1 Valid SCSI IDs

SCSI ID	Type	Peripheral Device
7	Initiator	None (host CPU) or SCSI host adaptor
6	Target	TEAC FD235 HS or JS (FC-1 Controller) embedded SCSI floppy drive. LUN 0-3 are available but must be the same type (HS or JS). LUN0 is the default drive for booting.
5	Target	Default value for second tape drive.
4	Target	Primary Tape Device (Archive 2150S, 2060S Viper Cartridge tape, or TEAC MT-2ST Cassette tape).
3	Target	Reserved.
2	Target	Reserved.

Table A-1 Valid SCSI IDs (continued)

SCSI ID	Type	Peripheral Device
1	Target	Default value for second CCS hard drive.
0	Target	Primary Disk Device (usually CCS Winchester drive).

Note

The OS-9 for 68K SCSI implementation currently supports only single initiator mode of operation. Placing additional initiators on the bus could be fatal to the system

Device Drivers: rbsccs/rbvccs

The current version of OS-9 for 68K RBF supports logical sector sizes other than 256 bytes. In the past, due to device constraints, if the physical sector size of the device was other than 256 bytes, the device driver (`rbsccs`) had to manage the logical to physical mapping of the drive. The newer device driver (`rbvccs`) is now the only driver provided with this version of OS-9 for 68K. This driver assumes the logical/physical mapping of the drive is 1:1 at the sector size determined from the SCSI drive during initialization. The implications of this, especially the effects on the device descriptors, are explained below.

Note
For those systems still using RBSCCS formatted disks, the RBSCCS driver currently in use on the 3.X systems can be used under Microware OS-9 for 68K. Any drives being freshly formatted should use `rbvccs`.

Note
`rbsccs` and `rbvccs` are not directly compatible. It is not sufficient to just change the driver name to make the change. If the drive was created under `rbsccs` with a physical sector size other than 256 bytes, you must reformat the drive before using it with `rbvccs.elow`.

`rbvccs` drives allow only a single **LOGICAL UNIT** per **DEVICE ID**. The device address should be:

```
   Device Address  +  SCSI ID
```

The provided descriptor generators take this address into account. This is important if you use the `moded` utility to change the SCSI IDs in a descriptor. This method of providing unique addresses allows multiple processes to access separate drives simultaneously, without locking. It results in a significant performance increase for **SCSI disconnect** capable systems.

Differences Between rbsccs and rbvccs Device Descriptors

While `rbsccs` is not in the Microware OS-9 for 68K package, the differences between `rbsccs` and `rbvccs` device descriptors are summarized here.

`rbsccs:`	Logical sector size is always 256. Sector size field = `0`: Assume physical sector size is 256. Sector size field = `n`: Assume physical sector size is `n`.
`rbvccs:`	Sets logical and physical sector size to same value. Sector size field = `0:` Use current device sector size. Sector size field = `n`: Set sector size to `n`, must be a format enabled descriptor (`h0fmt`).

The `rbvccs` descriptors have the sector size set at 0, indicating the drive should be queried to determine the sector size and the driver uses the current sector size of the drive.

Other Important Device Descriptor Fields

Other important device descriptor fields are summarized here.

Drive Number:	Use this field to assign a unique **LOGICAL DRIVE NUMBER** for each disk device the driver controls. If the driver supports multiple **LOGICAL UNITS** (for example, `rbteac` and `rbsccs`), this number should be unique for each unit on the controller. Set this field to `0` for drivers supporting a single **LOGICAL UNIT** per drive (for example, `rbvccs`). The number selects which drive table entry the driver uses for the drive.
Controller ID:	Indicates the actual hardware address of the SCSI device on the bus. `moded` refers to this field as SCSI controller ID.

Logical Unit:		Indicates the actual LUN of the device on the controller device. It has nothing to do with the drive number field. This field is set to `0` for most embedded controller drives.
SCSI Options:		Controls the options used on the SCSI bus. Set the appropriate bit(s) to turn on the options as follows:
	Bit 0	Disconnect allowed.
	Bit 1	Enable target mode (not currently supported).
	Bit 2	Synchronous transfers.
	Bit 3	Enable SCSI parity (not normally used).
	Bits 4 - 31	Reserved for future use by Microware.

Note

Use the SCSI options bits 0 and 2 with care. In general, only enable the synchronous field for drives explicitly stated as synchronous capable. The same is true for the disconnect bit. Some devices do not support disconnect. Consult the device manuals prior to enabling these options. (Default descriptors have these options disabled.)

Converting a rbsccs Drive for Use with rbvccs

If the drive was formatted under `rbsccs` with a sector size of 256, `rbvccs` and its associated descriptor should work; no conversion is needed.

If the drive was formatted under `rbsccs` with a sector size other than 256, use the following steps to convert the drive:

Step 1. Back up the drive. Because you are going to format the drive, all information on it is lost. You can use any method using standard utilities for this operation (for example, `fsave`). You can use a second Winchester for this purpose; `rbvccs` and `rbsccs` can be used simultaneously in the system.

Step 2. Create a descriptor for the device being converted. You can either modify `systype.d` and use the descriptor generator or copy the supplied descriptor and make the necessary changes. Remember the `h0` and `h0fmt` descriptors must agree.

Step 3. Load `rbvccs`, the new descriptor, and any required programs into memory.

Step 4. Format the drive you are converting, using the new descriptor and driver.

Step 5. Restore the drive's contents (for example, `frestore`).

Step 6. Install a new boot file containing the `rbvccs` driver and the new `h0.vccs` descriptor.

Note

Under the current release of OS-9 for 68K, when sector sizes are changed on some devices, the correct capacity in sectors is not available until after the drive has been physically formatted. When format reports the number of sectors, make a quick check of them. If you find an overly large disparity in the capacity, format the drive a second time at the same sector size. This sets up the drive with the correct capacity. Also, remember many drives are specified with unformatted capacity. The formatted capacity is always somewhat less than the unformatted capacity. It is common for a drive formatted at 256 bytes/sector to have less formatted capacity than the same drive formatted at 512 bytes/sector. The formatted capacity of a given drive depends on many factors; consult the individual drive manuals if questions arise.

Embedded SCSI Hard Disk Support

Software Driver:	`rbvccs`
Controller/Driver:	In general, any SCSI embedded drive supporting CCS Rev 4B. Examples include: Imprimis Wren III, IV, V, VI, and VII Imprimis Swift Series Seagate 225N, 138N, 157N Syquest SQ555 (44 megabyte removable)
SCSI ID:	0
Drive LUN:	0 (fixed)
Host Parity:	Disabled

Note

rbsccs was the deblocking version of the driver. It supported a logical sector size of 256 bytes and variable physical sector sizes (as defined in the device descriptor's sector size field). This driver was mainly provided for existing installations that have initialized the media under the pre-Version 2.4 OS-9 for 68K RBF.

rbvccs is the only supported driver for new installations. It supports variable logical/physical sector sizes on a one-to-one basis. You should use this driver with Version 2.4 OS-9 for 68K or greater RBF.

Note

Removable media devices **must** have the **removable hard disk** flag set in the device descriptor. If you use removable media and this flag is not set, RBF and the driver may not detect when the media platter has been changed. This may result in damage to the media's file structure.

Embedded SCSI Floppy Disk Support

Software Driver:	rbteac
Controller/Driver:	TEAC FD-235 HS/JS
SCSI ID:	6
Drive LUN:	0 (optional drives may be connected as LUNs 1 - 3)
Host Parity:	Disabled

Note

The HS version of the drive supports double density (1 Megabyte unformatted capacity - **DD**) and high-density (2 Megabyte unformatted capacity - **HD**) media. The JS version of the drive supports DD and HD formats, as well as extra density (4 Megabyte unformatted - **ED**) media.

Embedded SCSI Tape Support

Software Driver:	`sbviper`, `sbteac`, `sbgiga`
Controller/Driver:	Archive QIC tape drives (`sbviper`) TEAC MT-2ST/N50 and N60 tape cassette (`sbteac`) Exabyte 8200 and 8mm tape cassettes (`sbgiga`)
SCSI ID:	4
Drive LUN:	0 (fixed)
Host Parity:	Disabled

Module Locations

Drivers are found in `MWOS/OS9/68000/CMDS/BOOTOBJS`.

Descriptors are found in `MWOS/OS9/68020/PORTS/<cpu>/CMDS/BOOTOBJS/<subdirectory>.`

Table A-2 Module Locations

Drive Type	Driver Name	Subdirectory for Descriptors
Hard disk drive	`rbvccs`	`vccs`
Hard disk	`rbsccs`	`sccs`
Embedded floppy drive	`rbteac`	`teacfci`
Cartridge/DAT tape drive	`rbviper`	`viper`
Cassette tape drive	`sbteac`	`teacfci`
8mm tape drive	`sggiga`	`exabyte`