

Release

Notes

for

**OS-9 for 68K
Processors**

Version 3.0

microware



Copyright and Publication Information

Copyright ©1993 Microware Systems Corporation. All Rights Reserved. Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from Microware Systems Corporation.

This manual reflects version 3.0 of OS-9 for 68K processors.

Revision: A
Publication date: November 1993
Product Number: 1233-0011 / OPS6830PIRN

Disclaimer

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

Reproduction Notice

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

For additional copies of this software/documentation, or if you have questions concerning the above notice, please contact your OS-9 supplier.

Trademarks

OS-9, OS-9000, DAVID, FasTrak, and UpLink are registered trademarks of Microware Systems Corporation. SoftStax is a trademark of Microware Systems Corporation. All other product names referenced herein are either trademarks or registered trademarks of their respective owners.

Address

Microware Systems Corporation
1500 N.W. 118th Street
Des Moines, Iowa 50325
515-223-8000

Table of Contents

Chapter 1: New Features

9

10	Introduction
10	Processor Specific Kernels
10	Memory Allocation
10	Development/Atomic Kernel
11	New IOMan Module
11	Performance Improvements
12	Three Levels of Board Support
12	New Development Tools
13	New MWOS Directory Structure
13	About the Directory Structure
16	Development vs. Runtime
18	ISP, NFS, and Other Package's Directories
19	Kernel/IOMan
19	IRQ Polling Routines
19	CPU32-Family Kernels
19	Kernel/IOMan Split-up
20	Atomic/Development Environments
20	Atomic/Development Differences
21	System Calls Implemented in Kernel and IOMan
23	Memory Allocators
24	NVRAM and SHARED Colored Memory Support
25	Processor Versions and Kernel/IOMan Naming Conventions
27	System-state Time-Slicing
28	Supervisor Stack Usage



29	Process Descriptor Changes
30	General Enhancements
31	New System Calls
32	System Security Module (SSM) Module Changes
33	Cache Module
35	Init Module Changes
42	Floating Point Unit (FPU) Compatibility
44	68040 Users
44	68K Users
44	fbsp040 Support
46	New Trap Handler
47	New Boot ROM Features
47	Position Independence/Coexistence
48	NVRAM Configuration Enhancements

Chapter 2: Corrections and Enhancements

49

50	Utilities
50	Corrections
55	Enhancements
58	New Error Codes
60	RBF Changes
60	Corrections
61	Enhancements
62	Pipeman Changes
63	SCF Changes
64	SSM Changes
65	System Definitions Changes
65	Module Definitions Changes
66	Process Descriptor Changes
67	System Globals
69	Kernel/IOMan Changes

72	Driver Changes
72	RBTEAC Driver Changes
72	SCF Driver Changes
73	CBOOT/SCSI Driver Changes
74	Rombug

Chapter 3:OS-9 Version 3.0 Application Notes

75

76	Memory Management and Caching
76	Memory Management Units
77	System Caching
78	SSM Issues
78	Standard 68040 SSM Defaults
79	Note 1
80	Note 2
80	Note 3
80	Note 4
81	F\$FIRQ Application Notes
82	Differences Between the Fast and the Normal IRQ Services
82	Installing a Routine During the Init Routine
83	Removing a Routine During the TERMINATE Routine
84	Example Fast IRQ Service Routines in the Device Driver
86	Taking Control Of the Vector
87	Vector Table Method
87	Jump Table Method
90	Fast Dispatch Application Notes
90	Inserting Routines into the Fast Dispatch Queue
90	Using the D_FDisp Routine
91	Setting the Status of FstP_stat



93	Important Notes
94	Semaphores
94	Using Semaphores
96	Example Semaphore Structure
96	Example Semaphore Use
98	Non-Standard I/O Systems and the ROMIO File Example

Chapter 4:Compatibility Issues **101**

102	Introduction
103	Status Register (SR) and MSP/ISP Stack Registers
103	Device Drivers That Mask Interrupts
105	System-state Threads
106	Process Descriptor Changes
106	Process Descriptor Size
107	System-state Preemption
109	ROM Compatibility
110	Changes to System Definitions
112	Determining the OS-9 Release Level

Chapter 5:Documentation Changes **113**

114	Introduction
-----	--------------

Chapter 6:Known Bugs **117**

118	Introduction
118	ROMbug
118	F\$SysDbg
119	I\$Attach
119	Pre-V2.4 ROMs

119 Setime and the Startup File

Product Discrepancy Report

121

microware

Chapter 1: New Features

Introduction

With OS-9 Version 3.0 and the introduction of atomic OS-9, Microware provides significant improvements to the OS-9 operating system. These release notes document the differences between OS-9 Version 2.4 and Version 3.0. They also outline the variety of OS-9 kernel options which are now available.

Processor Specific Kernels

OS-9 kernels are now specific to each processor type. With Version 3.0, there are specific kernels for 68000, 68010, 68020, 68030, 68040, 68070, CPU32, and 68349 processors.

Memory Allocation

OS-9 now has two different memory allocations systems:

- The **standard colored memory allocator**
- A new **buddy memory allocator**

This choice of memory allocation schemes allows greater flexibility when creating specialized embedded systems.

Development/Atomic Kernel

The OS-9 kernels are now available in two forms:

- Development kernel
- Atomic kernel

The **development kernel** is the V3.0 version of the previously available kernels. It has full debugging support, multi-user protection mechanisms, parameter checking, and memory protection features.

The **atomic versions of the kernel** are specifically tailored to embedded systems. After you debug a system design, you can use the atomic kernel where space is at a premium and the added functionality of the development kernel is no longer required. All kernels are available as development or atomic versions with the standard or buddy memory allocator.

New IOMan Module

The unified I/O systems calls have been moved from the kernel modules into a separate module called **IOMan**. This allows you to create your own I/O system for small embedded systems. There are two IOMan modules available, one for use with the development kernel and one for the atomic kernels.

Performance Improvements

Every part of the kernel has been carefully analyzed and improved in both speed and determinism. (Specific timing and performance information is available from Microware upon request.)

System state preemption/time-slicing is now available.

Support for processors with MSP (master stack pointer) has been added. Using the MSP allows additional time saving in interrupt processing.

Three Levels of Board Support

OS-9 board support packs are now available in three levels of functionality:

- Embedded
- Disk-based
- Extended

The **embedded version** is directed toward small embedded systems. These systems may require only minimal support from an atomic kernel or may have the unified I/O systems available for serial and pipe support.

The **disk-based version** of OS-9 offers support for disk and tape systems. It is similar to previous packages for OS-9.

The **extended version** of OS-9 adds internet (ISP) and network file systems (NFS) client support to the package.

New Development Tools

A new Microware Operating Systems (MWOS) directory structure provides a unified directory structure for all Microware products. The new directory structure, along with new makefile technology and a wide range of new and improved development tools, is designed to allow ease of code development on a variety of host development systems.

Version 3.0 of OS-9 and Version 2.0 of ISP are now using the Ultra C compiler, assembler, and linker.

Version 3.0 introduces a wide range of new technologies that Microware will continue to improve and expand, providing you with real-time, flexible, and simple solutions to help you develop quality system software.

New MWOS Directory Structure

The directory structure introduced in OS-9 Version 3.0 represents a significant departure from its predecessor. Its design was influenced by a growing number of users developing not only under OS-9, but UNIX and DOS.

The new directory structure is designed to:

- Provide a consistent directory structure for all development platforms.
- Provide similar development environments for OS-9 and OS-9000.
- Allow code sharing between OS-9 and OS-9000.
- Make provisions for code and libraries optimized for 32 bit processors.
- Provide a clear division between the development and runtime directories.
- Allow for multiple ports from a common set of sources.

About the Directory Structure

The new structure is built under the `MWOS` directory. As you descend through the directories, the files become progressively more OS, CPU, and hardware dependent. A simplified module appears in [Figure 1-1 MWOS File Structure](#) and [Figure 1-2 MWOS/OS9 File Structure](#). For a more detailed examination, we suggest recursively walking down the directory structure of your newly installed product.

Figure 1-1 MWOS File Structure

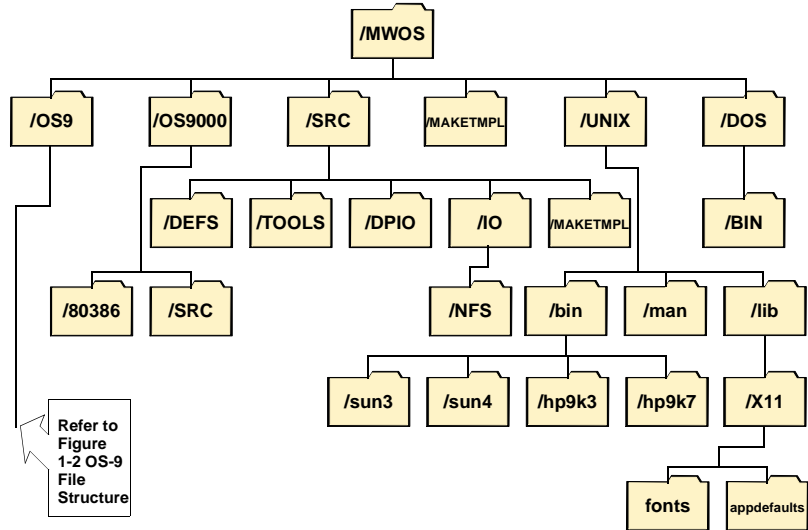
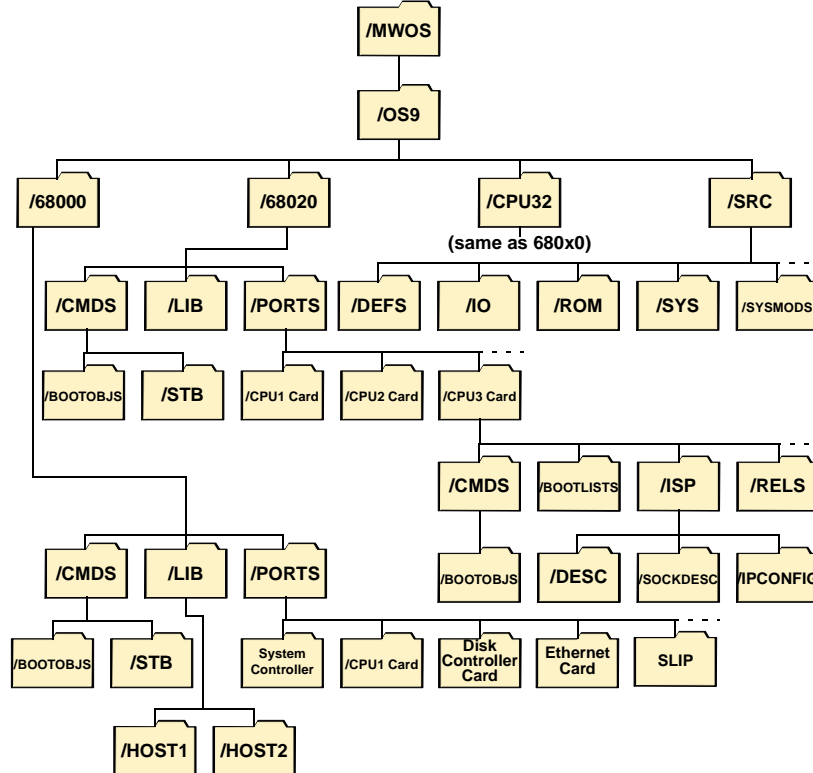


Figure 1-2 MWOS/OS9 File Structure



Sources particular to an operating system (OS) are kept in `MWOS/<OS>/SRC`. Sources common between all OSs are located in `MWOS/SRC`. The same logic applies to C header files and assembler defs. Ports for particular boards are kept under the `<OS>/<Processor family>/PORTS` directory.

A few examples may be useful here. Where performance is not an issue, it is the practice at Microware to compile OS-9 software products with a 68000 compiler, allowing execution on all Motorola 680X0 MPUs. Most utilities fall into this category and are found in `MWOS/OS9/68000/CMDS`. When there are significant performance benefits to be gained from compiling with a 32 bit compiler, such as with the ISP system modules, the executables are found in `MWOS/OS9/68020/CMDS`.

If you are doing a port of OS-9 to a new 68040 CPU card, you will find the kernel and any other processor specific modules in the `MWOS/OS9/68020/CMDS/BOOTOBS` directory. The remainder of the hardware-independent modules are in `MWOS/OS9/68000/CMDS/BOOTOBS`. CPU card specific components of the OS are found in `MWOS/OS9/<CPU>PORTS/<YOUR CPU>`.

Example boot driver source code is found in `MWOS/OS9/SRC/ROM`. Example high level driver sources are found in `MWOS/OS9/SRC/IO/<FILE MANAGER>/DRVR`.

Another useful example is for those doing cross development on a UNIX workstation. An OS-9 or OS-9000 targeted cross compiler will reside in `MWOS/UNIX/BIN/<YOUR WORKSTATION>` along with other cross-hosted utilities. Makefiles should target the appropriate OS and CPU. Those developing under DOS will find themselves in a similar environment.

Development vs. Runtime

The `MWOS` directory structure is oriented specifically toward software development. Whether the development occurs on a resident OS-9 system, a cross development environment, UNIX, or DOS, once the executable module have been created they are moved to their final locations on the target machine.

When you are developing an application on a resident development system, this might be simple a matter of copying a file from the `MWOS/OS9/<CPU>/CMDS` directory to the `/H0/CMDS` directory. It might involve downloading the modules into memory on a small target system, making a boot on a server to booted over ethernet, or creating a set of ROMs for a fully ROMed system.

Disk-based runtime systems are similar to their pre-V3.0 counterparts. Contents of system dependent directories are generally lifted to the root of the system drive, while (if desirable) a mirror image is kept with MWOS. For example, an OS-9 Version 2.4 MVME-147 development pack looked like this:

```

                                Directory of . 17:10:43
C          CMDS      DEFS      IO          ISP
LIB        SYS      SYSCACHE SYSMODES init.ramdisk
startup

```

Note that development (source) directories like `C`, `IO`, `SYSCACHE`, and `SYSMODS` appeared on the root.

An OS-9 Version 3.0 MVME-0147 Extended Board Support Pak (our example's nearest counterpart) would look like this:

```

                                Directory of . 17:10:43
CMDS      SYS      ETC      MWOS

```

All sources, header files, and libraries are now under the `MWOS` directory. Depending on the application, the executables found in `CMDS`, the system startup file(s) are found in `SYS`, and the network (Internet and/or NFS) database files are found in `ETC`. At the system administrator's option, these files may also be duplicated in `MWOS` so that they may be modified and tested prior to committing them for use on the development system.

Directories such as `USR`, `TFTPBOOT`, and other directories used on your OS-9 systems can continue to reside in their current location. The `SYS` and `LIB` directories may continue to reside on the root or on RAM disks if desired.

Please see the Ultra C documentation for additional information about the `MWOS` file structure. The sources provided in Microware Developer's Kits and BSP use pathlists for `defs` and `libs` which stay within the `MWOS` directory structure.

1

New Features



The sources and makefiles are designed to allow the relocating of the `MWOS` directory. Multiple `MWOS` directories may be created for different versions of OS-9 and OS-9000.

ISP, NFS, and Other Package's Directories

The ISP and NFS utilities and system modules are now located in the target system's `CMDS` and `CMDS/BOOTOBS` directories. This simplifies the startup procedures for both systems and allows utilities to be loaded as they are needed without long `PATH` searching.

The startup procedures for these packages still allow the utilities to be loaded at startup, but the practice is no longer required. You may choose to move the systems modules to the boot so that no loading is required.

Kernel/IOMan

This section contains information about changes to the kernel and IOMan.

IRQ Polling Routines

IRQ polling registers are now passed the vector offset (that is, `vector number * 4`) in register `d0.w` to aid in interrupt device recognition.

CPU32-Family Kernels

The kernels for the CPU32-family now set a system global (`D_MBAR`) to the address of the system's module block. This allows drivers, etc. to easily access the module block registers without having the overhead of reading the module block's `MBAR` register from CPU space.

Kernel/IOMan Split-up

The I/O portions of the kernel are now split out of the kernel into a separate module called **IOMan**. The standard I/O system has been retained; however, the I/O system is now optional for small systems. IOMan is a P2-style module; the name of the system's I/O manager (if any) is defined in the system's `Init` module.

Atomic/Development Environments

There are two distinct classes of the kernel/IOMan:

- The atomic version
- The development version

The **atomic version** is primarily designed for use in embedded systems, although it can be used in multi-user systems. The **development kernel/IOMan** is the full-featured version and is designed for use in embedded and multi-user environments.

Atomic/Development Differences

The main differences between the two versions of the kernel/IOMan are summarized as follows:

- User-state debugging (`F$DFork`, `F$DExec`, `F$DExit`) is not supported by the atomic kernel. Debugging on atomic systems is restricted to the capabilities of the system's ROM Debugger.
- Multi-user protection mechanisms are not implemented in the atomic environment. These mechanisms include features such as:
 - Validation of user parameters (for example, validation of user buffer pointers for data transfers).
 - External cache hardware is not supported. The atomic kernel only supports on-chip caches (if present).
 - No MMU support for memory protection
 - Validation of module/user ID permissions. For example, with an atomic kernel any user can link to any module, regardless of the access permissions specified in the module.

These kernel environments allow you to tailor the target system software to the requirements of the target application.

System Calls Implemented in Kernel and IOMan

The following summarizes system call implementation between the kernel and IOMan:

Table 1-1 System Calls Implemented in Kernel and IOMan

F\$Alarm	F\$AllPD	F\$AllPrc	F\$AProc
F\$Chain	F\$CmpNam	F\$CpyMem	F\$CRC
F\$DatMod	F\$DExec *	F\$DExit *	F\$DFork *
F\$DelPrc	F\$Exit	F\$Event	F\$FindPD
F\$FIRQ	F\$Fork	F\$FModul	F\$GBlkMp
F\$GModDr	F\$DPrDBT	F\$GPrDsc	F\$GProcP
F\$Gregor	F\$Icpt	F\$ID	F\$IRQ
F\$Julian	F\$Link	F\$Move	F\$Mem
F\$NProc	F\$PrsNam	F\$RetPD	F\$RTE
F\$Sema	F\$Send	F\$SetCRC	F\$SetSys
F\$SigMask	F\$SigReset	F\$Sleep	F\$SPrior
F\$SrqMem	F\$SRqCMem	F\$SRtMem	F\$SSvc

Table 1-1 System Calls Implemented in Kernel and IOMan (continued)

F\$STrap	F\$STime	F\$SUser	F\$SysDbg
F\$SysID ⁺	F\$Time	F\$TLink	F\$Trans
F\$UnLink	F\$UnLoad	F\$Wait	F\$UAcct
F\$VModul			

* These system calls (F\$DExec, F\$DExit, F\$DFork) are not available under the atomic kernel

+ This system call (F\$DSysID) has functional differences between the development and atomic kernels. See the relevant system call description in the **OS-9 Technical Manual** for details.

Table 1-2 IOMan System Calls

F\$AllBit	F\$DelBit	F\$IOQu	F\$IODEl
F\$Load	F\$PErr	F\$SchBit	I\$Attach
I\$Create	I\$ChgDir	I\$Close	I\$Delete
I\$Detach	I\$Dup	I\$GetStt	I\$MakDir
I\$Open	I\$Read	I\$ReadLn	I\$Seek
I\$SetStt	I\$SGetSt	I\$Write	I\$WritLn

Memory Allocators

You can choose one of two versions for the kernel's memory allocator:

- The buddy allocator
- The standard allocator

Which allocator you use is determined by the kernel version you select and the target application/environment.

The **buddy allocator** gives more deterministic operation in a real-time environment than the **standard allocator**, at the expense of memory efficiency.

The majority of systems use the **standard allocator**. This is the memory allocator used by the kernel since the original release of OS-9. It can allocate memory to a resolution of 16 bytes (for example, a request for 1025 bytes rounds up to 1040 bytes).

The **buddy allocator** uses a **binary-buddy** algorithm which maintains free memory in block sizes that are binary multiples. Under the buddy allocator, memory requests are rounded up to the next binary power of the request size (for example, a request for 1025 bytes rounds up to 2048 bytes). The binary-buddy technique results in faster memory allocation for the system, improving the overall system determinism. Because the buddy allocator is less efficient for system memory usage, the allocator is typically used in embedded (atomic kernel) applications where real-time performance is critical.

When you use the buddy allocator, the system memory lists (either in ROM or in the `init` module's colored memory lists) must have a start address on the same boundary as the size of the block.

For example, a 4M block started at address 0x00100000 must be described as a block with start address of 0x00000000 and an end address of 0x00800000. The memory search routine will correctly find the 4M block (that is, 0x00100000 to 0x00500000) provided the memory outside of the desired area correctly returns bus errors. If not, or if there are, for example, I/O devices in the areas, then you need to specify the desired region as more than one block. For example:

```
0x00100000 to 0x00200000
0x00200000 to 0x00400000
0x00400000 to 0x00500000
```

NVRAM and SHARED Colored Memory Support

Both memory allocators now support NVRAM (non-volatile RAM) and SHARED (shared RAM) colored types. These types of memory are used during system booting.

NVRAM memory is searched for modules. Any modules found are added to the system's module directory. NVRAM that does not contain modules is added to the system free memory pool.

SHARED memory is supported by allocating the system's data structure for the shared memory block out of the block itself. Currently, the number of SHARED blocks you can describe in the `Init` module memory list is limited to ten.

Processor Versions and Kernel/IOMan Naming Conventions

The kernel is now offered in processor-specific versions. If you attempt to run a kernel on an incorrect processor, a message indicating an **incompatible kernel** is issued and the system boot call will then fail.

The file names of the kernel reflect the kernel version, as follows:

`<environment>ker<processor><allocator>`

`<environment>` is one of the following:

d = development kernel

a = atomic kernel

`<processor>` is one of the following:

000 = 68000, 68008, 68EC000, 68301, 68302, 68303, 68306

010 = 68010

020 = 68020, 68EC020

030 = 68030, 68EC030

040 = 68040, 68EC040, 68LC040

070 = 68070

c32 = CPU32 and CPU32+ family processors

(For example, 68330, 68331, 68332, 68333, 68340, 68341, 68349 (no Cache support), 68360)

349=68349 (with Cache Support)

1

New Features



<allocator> is one of the following:

s = standard memory allocator

b = buddy memory allocator

The file names of IOMan reflect the two versions of IOMan available, as follows:

IOMan_<environment>

<environment> is one of the following:

DEV = development kernel environment

ATOM = atomic kernel environment

For example, the kernel called `dker030b` is the development kernel for the 68030/68EC030 with the buddy allocator.

System-state Time-Slicing

V3.0 of OS-9 can perform time-slicing while in system-state. This improves the overall determinism of the state. In general, this will not be of concern unless you write your own file managers and/or system-state daemons. Refer to the section on ***System-State Time Slicing*** in the ***OS-9 Technical I/O Manual*** for further information. Also, note you can use a flag in the `Init` module to disable system-state time slicing (refer to the ***Init Module Changes*** section of this chapter for more information).

Supervisor Stack Usage

V3.0 kernels use the Master Stack Pointer (MSP) on those processors that have the Master/Interrupt Stack Pointer set. This change is implemented on the kernels for the following processors:

- 68020
- 68030
- 68040

This change improves the interrupt response time of the system. Refer to **Chapter 5 Compatibility Issues**, for a further discussion on the potential impact of this change.

Process Descriptor Changes

The process descriptor has been changed from a fixed 2K-byte structure to a more dynamic/configurable structure under OS-9 Version 3.0. The process descriptor is now composed of the following elements:

Table 1-3 Process Descriptor Changes

Element	Description
<code>Processdesc Body</code>	1K fixed size.
<code>FPU save area</code>	Optional save area allocated on systems with hardware FPU/floating point software emulation. Size is dependent upon the FPU requirements of the system.
<code>process stack area</code>	<p>You can now set the size of this area with an <code>Init</code> module field. The minimum value allowed is 1500 bytes, but you can increase it if needed.</p> <p>This stack area is used by the process when it is performing system calls (such as I/O operations), and thus systems that have C-language File Managers and/or Device Drivers that use the stack heavily may need to increase this value.</p>

System globals are available to indicate the size of process descriptors on a system. The process descriptor also has fields that indicate the size of the process descriptor and its stack size.

General Enhancements

The OS-9 operating system has been improved overall in terms of speed, size, and determinism. Areas of improvement include:

- `sleep(0)` and `sleep(a1)` are now much faster.
- Process scheduling has been improved.
- `pipeman` now performs direct I/O to the callers buffer whenever possible. Pipe transfers are now much faster.
- System-State Alarms are now called with the actual register packet image (and not a copy). This makes alarms faster and makes them use less stack space.
- Bitmap services are now faster.
- Table searching (process and path tables) for free entries is now much faster and more deterministic.
- Data modules are now created without a valid CRC. The module is created with a default CRC of 0, making data module creation much faster.
- Interrupt Response times have been improved for `F$IRQ` installed devices. Additionally, a new interrupt mechanism called the **fast interrupt scheme** (`F$FIRQ`) has been added to the kernel. See **Chapter 3, the `F$FIRQ` Application Notes** section, for further information.

New System Calls

The following system calls have been added to OS-9:

F\$FIRQ	Add/remove device from Fast Interrupt Polling Scheme.
F\$SigReset	Reset signal intercept context stack.
F\$SysID	Return system version/configuration information.
F\$Sema	Version 3.0 adds semaphore support for the following C calls: _os_sema_init() _os_sema_p() _os_sema_term() _os_sema_v()



For More Information

Refer to **Chapter 3, Application Notes**, and the **Ultra C Release Notes** for more information about semaphores.

System Security Module (SSM) Module Changes

The System Security Module (SSM) for the 68040 and 68030 now support the `CacheList` entries in the `Init` module. These `CacheList` entries allow tailoring of memory to cache mode for the system.



For More Information

Refer to the *OS-9 Technical Manual* for more information on `CacheList` entries.

When operating in an atomic kernel environment, no SSM-related calls (for example, `F$Protect`, `F$Permit`) are made by the kernel, thus the SSM module should NOT be used on an atomic system, **unless** the processor is a 68040. Atomic-68040 systems can use the SSM module (`ssm040`) to override the default (write-through) cache mode of user-state memory.



For More Information

See *Chapter 3, the System Caching* section, for further details.

Cache Module

There is a new cache module available for the 68349 processor. This module controls the system caching functions and uses the `Init` module's `M$Compat2` flags to determine which (if any) of the 68349 CIC banks to use as caches.

The file naming conventions for the cache modules have been changed to assist cross-development environments. The file names are now:

Table 1-4 Cache Modules

File Name	Description
<code>cache</code>	Dummy cache module for 68000/010/070 type processors
<code>cache020</code>	Cache module for 68020
<code>cache030</code>	Cache module for 68030
<code>cache040</code>	Cache module for 68040
<code>cache349</code>	Cache module for 68349



Note

These default cache modules support the on-chip caches of the processor. You can also support external hardware caches by creating **custom** versions of these modules. See the `syscache.a` source code for details.

Although the file names have changed, the `MODULE` name for the system cache module is unchanged (example: `SYSCACHE`).

Init Module Changes

The `Init` module was updated to reflect new features and removes obsolete features. These changes are reflected in the relevant definitions files, the `init.a` source, and the `moded.fields` file. The changes to `Init` are as follows:

Table 1-5 Init Module Changes

Field	Description
<code>M\$CacheList</code>	<p>The offset to the (optional) <code>CacheList</code> entries for the system.</p> <p>These entries define the caching mode to use on blocks of memory when that memory is accessed by user-state code. The System Security Module (SSM) uses these lists to override the default cache mode (write-through caching) when memory is allocated to a process. The current version of <code>moded</code> does not support direct manipulation of the actual list contents.</p>
<code>M\$Compat, Bit 0</code>	<p>This flag is OBSOLETE for OS-9 V3.0. It was defined as the slow irq flag for V2.X/V1.2 compatibility.</p> <p>Under V3.0, all interrupt service routines are required to adhere to the register usage conventions given in the <code>F\$IRQ/F\$FIRQ</code> system call descriptions (see the OS-9 Technical Manual for more information).</p>

Table 1-5 Init Module Changes (continued)

Field	Description
M\$Compat, Bit 6	<p>This flag was created solely as an aid to keep the system running while debugging.</p> <p>Set this flag to cause the system to continue when subjected to spurious interrupt conditions. If this flag is set, a counter in system globals (D_SpurIRQ) will be updated for every occurrence of a spurious interrupt. Spurious interrupts typically indicate a HARDWARE FAILURE; do NOT use this flag as a software work-around to hardware problems. The purpose of this flag is to allow investigation of the problem at hand.</p>
M\$Compat, Bit 7	<p>This flag was added to protect alarms from deletion by another process.</p> <p>Set this flag to cause alarms to be exclusively bound to the process that created them. When this flag is set, only the process that created the alarm may delete it. If not set, then any 0.N user or another process with the same user.group ID may delete a process's alarm.</p>

Table 1-5 Init Module Changes (continued)

Field	Description
M\$Compat2, Bits 4-7	<p>These flags control the enabling of the CIC cache banks for the 68349 kernel/cache module.</p> <p>Notes: Bit 7 was previously defined as the Don't Disable Data Cache while in I/O (DDIO) flag. This functionality is now obsolete.</p> <p>DMA-style drivers are now required to ensure that data cache coherency issues are NOT ignored. The <code>D_SnoopD</code> (all data caches are coherent) system global flag is available for drivers to inspect so that they can determine whether or not data cache flushing (via <code>F\$Cctl</code>) is required.</p>
M\$Events	<p>This field is now implemented. It contains the initial count of the number of events for the system.</p> <p><code>M\$Events</code> contains the initial counts for the system. Under the development kernel, the associated tables will expand when necessary. Under the atomic kernel the tables will NOT expand beyond the initial values.</p>

Table 1-5 Init Module Changes (continued)

Field	Description
M\$IOMan	<p>The offset to the system's I/O manager (if any). The default name for this module is IOMan. You can do any of the following:</p> <ul style="list-style-type: none"> • Use one of the standard IOMan modules supplied by Microware • Create your own • Your system can have no formal I/O Manager module <p>This module is called the same way as the P2-style modules named in the M\$EXtens list, and thus have the same conventions and restrictions.</p>
M\$MDirSz	<p>This field is now implemented. It contains the initial count of the number of modules for the system.</p> <p>M\$MDirSz contains the initial counts for the system. Under the development kernel, the associated tables will expand when necessary. Under the atomic kernel, the tables will NOT expand beyond the initial values.</p>

Table 1-5 Init Module Changes (continued)

Field	Description
M\$PreIO	<p>The offset to the list of modules (if any) called BEFORE the IOMan module is called. These modules are P2-style modules (they have the same conventions and restrictions as those modules named in the M\$Extens list).</p> <p>A customization module is intended to be used to complement or change the existing standard system calls used by OS-9. These modules will be searched for at startup, and if found will be executed in system state. Typically, the modules will be located in the boot file. The default name string to be searched is OS9PreIO.</p> <p>A typical use of the PreIO list would be to add system calls to the system that would be needed by the IOMan module's Init routine.</p>

Table 1-5 Init Module Changes (continued)

Field	Description
M\$PrcDescStack	<p>This field contains the size of the stack area allocated in a process descriptor. This stack is used when the process is in system-state (performing an I/O call).</p> <p>The default (minimum) size of this field is 1500 bytes. It is typically increased for systems that have heavy stack usage, such as C language File Managers and/or Device Drivers.</p>

Table 1-5 Init Module Changes (continued)

Field	Description
M\$SysConf	<p data-bbox="618 423 1105 479">This is a new control word, added so that various system options can be implemented.</p> <p data-bbox="618 499 953 526">Bit 0 System Table Expansion</p> <p data-bbox="618 546 1105 730">Set this flag to prevent the development kernel from expanding any system tables. Under the atomic kernel, table expansion is always disabled, so this flag allows simulation of the atomic kernel characteristics under a development environment.</p> <p data-bbox="618 751 896 777">Bit 2 CRC Check Disable</p> <p data-bbox="618 798 1105 854">Under the atomic kernel, set this flag to disable the CRC check of loaded modules.</p> <p data-bbox="618 874 968 900">Bit 3 System-state Time-slicing</p> <p data-bbox="618 921 1105 1135">This option allows the system to disable system-state time-slicing. It is provided primarily for systems that are running V2.X system-state code that cannot tolerate time-slicing, and cannot be modified according to the rules given in the <i>OS-9 Technical Manual</i>.</p> <p data-bbox="618 1156 858 1182">Bit 4 SSM Page Table</p> <p data-bbox="618 1203 1039 1291">Set this flag to cause the SSM to build a SINGLE page table for all user-state processes.</p> <p data-bbox="618 1312 1105 1564">This option is only available for the development kernel and it allows the simulation of the atomic environment for pagetables under a development environment. This option is only applicable for the 68040, as that is the only atomic target that uses SSM (the SSM code is used only for cache tuning issues).</p>

Floating Point Unit (FPU) Compatibility

The Ultra C release changed the way floating point math is handled. Ultra C only generates inline floating point calls, and inline floating point calls are used by the C library for its floating point routines. A floating point math emulation module (`fpu`) is used on systems that do not have floating point hardware. This module catches f-line exceptions and pretends to be the floating point hardware. Always generating inline floating point code allows a consistent interface for applications that may be running on machines either with or without floating point hardware.

If you are using a 68K processor without a math coprocessor, you must do the following to use the floating point math functions:

-
- Step 1. Add the `fpu` module to your bootfile.
 - Step 2. Add `fpu` to the extension module list in your `Init` module and remake it.
 - Step 3. Remake your bootfile and reboot the system.
-

If you are using a 68040 processor, you must do the following to use the floating point math functions:

-
- Step 1. Add the `fpsp040` module to your bootfile.
 - Step 2. Add `fpsp` to the extension module list in your `Init` module and remake it.

Step 3. Remake your bootfile and reboot the system.

`fpu` supports the following platforms:

- 68000/10/20/30/70
- CPU32
- EC/LC68040 (Mask 3E23G or later)
- MC68040 (We recommend using `fpsp040` instead. `fpsp040` supports the MC68040 only.)

`fpu` has the following restrictions:

- You cannot install `fpu` on a system where an emulation module (`fpu`, `fpsp040`) is already installed. Attempting such a reinstallation causes `fpu` to return the error `000:231` (`E$KwnMod`).
- `P2INIT` can only be used to install `fpu` on a pre-V3.0 system. For systems running OS-9 V3.0, `fpu` must be in the bootfile and installed via the extensions list in the `Init` module. Attempts to install `fpu` on a system running V3.0 cause `fpu` to return error `000:224` (`E$IPrCID`).



For More Information

See Chapter 6 of the *OS-9 Technical Manual* for more information about the math module.

68040 Users

The following instructions are supported in hardware by the 68040:

fabs	fadd	fbcc	fcmp	fdbcc	fdiv
fmove	fmovem	fmul	fneg	fnop	frestore
fsave	fsccl	fsqrt	fsub	ftrapcc	ftst

The `fpsp040` emulation module for the 68040 provides software support for the rest of the floating point instructions, as detailed in the *M68040 Users Manual*, page D3.



Note

The math emulation module makes the trigonometric functions and `fintrz` much faster.

68K Users

The math emulation module for 68K coprocessors provides software support for the following instructions:

fmove	fsinh	fintrz	fsqrt	ftanh	fatan
fasin	fsin	ftan	fetox	flogn	flog10
fabs	fcosh	fneg	facos	fcos	fdiv
fmod	fadd	fmul	fsub	fcmp	ftst
fmovem	fbcc	fsccl			

fpsp040 Support

`fpsp040` is now available for 68040 users. The FPSP support provides significant speed improvements, as well as improving the execution performance.



Note

FPU404 is not supported for OS-9 Version 3.0.

New Trap Handler

With the release of the Ultra C compiler, the `cio` module was replaced by the `cs1` module. For older utilities compiled with the Microware Version 3.2 compiler, the `cio` utility trap handler must be in the execution directory or preloaded into memory. By using special I/O techniques, `cio` allows these utilities to be much smaller. Most of these utility programs fail to execute if `cio` is missing. `cio` is typically loaded into memory at startup.

Standard utilities compiled with the Ultra C compiler use the `cs1` trap handler. While `cio`'s main emphasis was largely on I/O functions, `cs1`'s main emphasis is on large commonly used functions. Some systems which have a combination of old and new utilities may require both trap handlers to be present on the system.

New Boot ROM Features

This section contains information about new features in the OS-9 Boot ROMs.

Position Independence/Coexistence

Boot ROMs are now capable of running from a base address other than where they were linked. This allows coexistence with other ROM-based debuggers and diagnostic routines (for example, with Motorola's **Bug** ROMs for CPUs such as the MVME147, MVME167, and MVME162).

ROMs supplied in the OS-9 Board Support Packages (BSPs) for these CPUs can be ROM-booted by the Motorola **Bug** ROMs and they can run stand-alone. Several changes were made to accomplish this:

- `boot.a` was modified to calculate a relocation factor (**execution address-linked address**). This relocation factor is saved for `CBOOT` and `ROMBUG` initialization. It is added to applicable vector table entries when `RAMVects` is defined. For a vector table entry to qualify for relocation, the upper 16 bits of the linked entry address must match the upper 16 bits of the ROM's base link address.
- The `CBOOT` and `ROMBUG` routines were modified to use the relocation factor supplied by `boot.a` to correctly access initialized data, preset the static data areas, and access `boot.a`'s internal jump table.
- `mbugboot.a` was added to implement a ROM-bootable module which could be recognized by Motorola's **Bug** ROMs.

NVRAM Configuration Enhancements

For CPUs which implemented a battery-backed OS-9 configuration area (MVME147, MVME162, and MVME167), the format of that area is now common across all CPUs. This format is defined in `nvr.am.d` and `nvr.am.h`.

The configuration area is now checksummed and contains a version code for upward/downward compatibility. Support routines to process these areas are contained in:

- `nvr.am.m` (a macro definition file)
- `nvr.am.a` (for access from `sysinit.a` and CBOOT routines)
- The `nvr.am` module in the CBOOT `sysboot.1` library

These changes were implemented in the OS-9 Board Support Packages for the above CPUs, but OEMs can use the MVME147 example port as a guide for their own use as desired.

Chapter 2: Corrections and Enhancements

Utilities

This section details corrections and enhancements made to the OS-9 utilities.

Corrections

The following are corrections to OS-9 utilities:

Table 2-1 OS-9 for 68K Utility Corrections

Utility	Correction
<code>binex</code>	<code>binex</code> now checks for write errors to keep from encoding large amounts of data without outputting anything.
<code>cfp</code>	<code>cfp</code> no longer limits input lines to 255 characters; the limit is 512. <code>cfp</code> now examines the user's environment variable <code>SHELL</code> to determine what shell to use to execute commands.
<code>cmp</code>	<code>cmp</code> no longer reports that two files are the same when only one is a zero length file and the <code>-s</code> option is used.
<code>copy</code>	<code>copy</code> no longer fails to copy files from a CDFM device.

Table 2-1 OS-9 for 68K Utility Corrections (continued)

Utility	Correction
del	<p>You can now use <code>del</code> with both the <code>-z</code> and <code>-p</code> options.</p> <p><code>del</code> now correctly handles the use of both the <code>-f</code> and <code>-x</code> options.</p>
delmdir	<p><code>delmdir</code> no longer fails to delete directories from NFS devices.</p>
devs	<p><code>devs</code> no longer attempts to read memory from address zero under any circumstances.</p> <p><code>devs</code> can now handle variable sized device tables.</p>
diskcache	<p><code>diskcache</code> no longer requires an equal sign (=) and <code>k</code> character on the <code>-t</code> option. Also, the minimum size for <code>-t</code> is 10K.</p> <p><code>diskcache</code> now automatically attaches/detaches the device when appropriate.</p> <p><code>diskcache</code> now frees all the memory it allocates if a failure occurs.</p> <p><code>diskcache</code> can now disable CRC checking of the cache (<code>-c</code>) and statistical maintenance (<code>-i</code>).</p>
dsave	<p><code>dsave</code> no longer attempts to fork commands with their first letter in upper case.</p> <p><code>dsave</code> now supports CDFM devices as the source of the <code>dsave</code>.</p>
dump	<p><code>dump</code> now unlinks from the module before exiting when you use the <code>-m</code> option.</p>

Table 2-1 OS-9 for 68K Utility Corrections (continued)

Utility	Correction
format	<p>format no longer displays unused variables from output.</p> <p>format now checks for a media size change after physical format on autosizing devices.</p>
free	<p>free now works correctly when the sector size of the device is greater than 1024 bytes.</p> <p>free now works correctly on an NFS device.</p>
frestore	<p>frestore can now de-archive from a file on a NFS device (see -f).</p> <p>frestore now processes its command line pathlist specification correctly.</p> <p>frestore now restores an archive with zero length files at the end of media correctly.</p>
fsave	<p>fsave can now archive to a file on an NFS device (see -f).</p> <p>fsave can now perform backups from a directory that does not have write permission.</p> <p>fsave can now accept pathlists greater than eighty characters for the -f option.</p> <p>fsave no longer limits its buffer size to 64K bytes.</p> <p>fsave no longer fails to create a valid backup if the last data block is filled entirely with zero length files.</p>
help	<p>help now contains directory names that are consistent with OS-9 conventions.</p>

Table 2-1 OS-9 for 68K Utility Corrections (continued)

Utility	Correction
ident	ident now displays the module size when you use the <code>-q</code> option.
irqs	irqs can now print the information from the fast IRQ table and handle variable sized device tables.
make	<p>make now searches the entire directory for source files to make target files rather than reading the directory in sequential order and accepting the first potential source file found.</p> <p>make no longer ignores a command sequence if it is followed by a line with a tab and a carriage return.</p> <p>make now prints the extensions correctly when a new rule is added and the <code>-d</code> option is used.</p>
maps	maps now has updated system call names to reflect those in V3.0.
mdir	<p>mdir now displays the entire module directory rather than the first 512 modules.</p> <p>mdir no longer prints the column titles when an unformatted listing is requested (<code>-u</code>).</p>
moded	moded no longer contains directory names that are inconsistent with OS-9 conventions.
os9gen	os9gen can now accept blank lines in the middle of a file given to the <code>-z</code> option.
procs	procs was updated with system calls from V3.0 of OS-9.

Table 2-1 OS-9 for 68K Utility Corrections (continued)

Utility	Correction
rename	rename no longer fails when used on an NFS device.
shell	shell no longer limits the total size of its initial command line arguments to 256 characters. shell no longer contains directory names that are inconsistent with OS-9 conventions.
tapegen	tapegen now properly reads an entire device block when the <code>-c</code> option is used.
tmode	tmode no longer truncates device names greater than thirty-two bytes. tmode no longer fails when used on a UCM device.
touch	touch no longer fails to update the date of a file on a NFS device.
uMacs	uMacs now accepts key bindings with characters of any case.
xmode	xmode no longer fails when used on a UCM device.

Enhancements

The following are enhancements to OS-9 utilities:

Table 2-2 OS-9 for 68K Utility Enhancements

Utility	Enhancement
attr	<p>attr can now be given options that begin with <code>-g</code> for group permissions. This was done so the OS-9 version will work when called from an OS-9000-like makefile. Since there is no implementation of group permissions for OS-9, a warning is emitted when they are used.</p> <p>For example,</p> <pre>\$attr -grpr disk_file Warning '-g' options are ignored on OS-9. ----r-wr disk_file</pre>
backup	<p>backup no longer clears its buffer. This was done to make backup faster.</p>
cmp	<p>cmp no longer clears its buffer. This was done to make cmp faster.</p>
copy	<p>copy no longer clears its buffer. This was done to make copy faster.</p>
dsave	<p>dsave now has a <code>-t</code> option that you can use to omit the execution of <code>tmode nopause</code> and <code>tmode pause</code>.</p>

Table 2-2 OS-9 for 68K Utility Enhancements (continued)

Utility	Enhancement
fixmod	<p>fixmod now has a <code>-n</code> option. This option allows you to change the name stored in the module header. It may only be used on files that contain a single module and must be used in conjunction with <code>-u</code>.</p> <p>For example,</p> <pre data-bbox="615 656 858 753"> \$chd /h0/CMDS \$copy dir ls \$fixmod -u -n=ls ls </pre> <p>Module dir - Fixing header parity - Fixing module CRC</p> <p>fixmod is now capable of performing operations on modules from both OS-9/68000 and OS-9000/80386.</p>
ident	<p>ident can now be used on modules for both OS-9/68000 and OS-9000/80386.</p>

Table 2-2 OS-9 for 68K Utility Enhancements (continued)

Utility	Enhancement
make	<p>make now has modified rules and options to support Ultra C (<code>-mode</code> option).</p> <p>make now has an <code>include</code> facility so it can read other make files as if they were part of the main makefile.</p> <p>make now has improved algorithms for rule definition.</p> <p>make now allows white space between the <code>-f</code> option and its argument.</p> <p>make now has a <code>-r</code> option to display the entire list of rules that are used on a given makefile.</p> <p>make now has a <code>-o</code> option that can be used to override the assumption that an object code target file requires a ROF file.</p>
moded	moded can now work on a socket descriptor.

New Error Codes

A number of new error codes have been defined. Refer to **Appendix C** of the **OS-9 Technical Manual** for a complete list of error codes.

Error codes are categorized as follows:

Table 2-3 OS-9 for 68K Error Codes

Range	Description
000:001 - 000:031	Miscellaneous Errors
000:032 - 000:047	Ultra C Related Errors
000:102 - 000:163	<p>Processor Exception Errors</p> <p>Error codes in this range are reserved to indicate that a processor related exception occurred on behalf of the program. Only those listed within this range can occur on behalf of the user program. All other numbers between 100 - 163 are reserved. Unless the program provides for special handling of the exception condition (<code>F\$STrap</code>), the error is fatal and the program terminates. The listed errors that fall between 100 - 163 represent the hardware exception vector plus 100.</p>
000:164 - 000:176	Miscellaneous Errors
000:177	Semaphore Error

Table 2-3 OS-9 for 68K Error Codes (continued)

Range	Description
000:200 - 000:239	Operating System Errors These errors are normally generated by the kernel or file managers.
000:240 - 000:255	I/O Errors These error codes are generated by device drivers or file managers.
001:000 - 001:001	Complier Errors
006:100 - 006:206	RAVE Errors
007:001 - 007:029	Internet Errors
008:001 - 008:017	ISDN Errors

RBF Changes

Corrections

The problem in `GetStat (FDInf)` when multiple processes open on same path (this resulted in stale data problems) has been corrected.

If the bitmap pointer was at the end of the bitmap and the request size (for bitmap allocation) was greater than 64K bytes, the disk root directory may have been corrupted. This has been corrected.

Changing file size (via `SS_Size`) on a file with multiple open paths no longer causes an infinite loop in RBF or a corrupted disk.

`SS_FD` and `SS_FDInf` calls now validate the user buffer via `F$ChkMem`.

`GetStat (SS_Ready)` now always returns a count of 1 (as per documentation) instead of 0.

When the file segment list becomes full, RBF no longer fails to return the last segment when the file is deleted.

Passing illegal modes to `Delete` could cause disk corruption. RBF now protects against this.

It was possible for a process to receive an `E$Full` error prematurely when multiple processes were attempting bitmap allocation at the same time. This has been corrected.

Enhancements

RBF now maintains a bitmap sector offset field in the drive table to improve bitmap allocation times.

RBF now supports a **write-protect** flag in the descriptor control word. This flag, if set, defines that the media is not writable by RBF.

RBF is now pre-emptable when possible.

2

Corrections and Enhancements



Pipeman Changes

SS_FD and SS_FDInf now validate the user's bugger via F\$ChkMem.

Pipe transfer routines are MUCH faster; pipeman now performs direct user buffer I/O whenever possible.

Pipeman is now pre-emptable when possible.

Pipeman's file manager definitions in the Path Descriptor have been reorganized for alignment and speed issues.

SCF Changes

`SetStat (SS_Opt)` now updates the device special characters in the driver static storage. This prevents the requirement that some type of I/O operation be performed before the `SS_Opt` update takes effect.

`Open` and `Create` now update `V_LPRC`. This prevents the requirement that some type of I/O operation take place on the path before the special signals (Interrupt, Quit) are recognized.

`Open` and `Create` now set up the `Pause` conditions. This prevents (the rare) case of `Pause` being ignored when a path is opened to a device that has no `Pause` set in the descriptor, output is performed to that path and a `Pause` char is typed (previously some type of Input operation had to take place before the `Pause` request was recognized).

SCF is now pre-emptable when possible.

SSM Changes

The following changes apply to the `ssm851` and `ssm040` modules:

- Error checking problems in `F$Permit` have been corrected.
- SSM has been enhanced to utilize the `CacheList` entries in the `Init` module.
- Problems with `F$GSPUMP` and `Size = 0` have been fixed.
- Range overflow problems (where address + size overflows) have been fixed.
- `SSM040` can be used under the Atomic kernel so that cache mode changes (detailed in the `CacheList` entries in the `Init` module) can be implemented. This affects User-state memory allocation.
- The `SSM040` module can create a single user-state page-table (that is, simulation of the Atomic environment under the Development kernel). The `Init` module's `M$SysConf` flag controls this.

System Definitions Changes

The following changes appear in `Sys.l` (and `usr.l`, where applicable), as well as the relevant C header Files.

The following error codes associated with processor exceptions are now defined:

<code>E\$CProto</code>	Co-processor protocol violation
<code>E\$StkFmt</code>	Stack format error
<code>E\$UnIRQ</code>	Uninitialized interrupt
<code>E\$FPUnData</code>	Unsupported data type
<code>E\$MMUConf</code>	MMU configuration error
<code>E\$MMUAccess</code>	Access level violation error

X-windows error codes and status codes are now defined.

`SS_VolStat` Status code was added for NFS.

RAVE error codes are now defined.

The GFM status code `SM_GetMode` was added.

`SS_MIDI` status codes were added.

ISDN status codes were added.

New system calls (`F$Sema`, `F$FIRQ`) were added.

New error code (`E$BSig`) were added for semaphores.

Module Definitions Changes

The following changes were made to the Module definitions.

The `DT_ISDN` (ISDN Device Type) was added.

2



`M$HdExt` and `M$HdExtSz` have been added to the module header. These provide a pointer to and size of a module header extension area. This release of the Kernel does NOT support this concept.

The `Init` module contains the following new fields:

<code>M\$CachList</code>	This offset is a pointer to the CacheList entries (usually defined in the system's <code>sysType.d</code> file).
<code>M\$SysConf</code>	System configuration flags. These flags allow tailoring of the system configuration.
<code>M\$NumSigs</code>	This is the maximum number of signals that a process can have queued. It is NOT implemented in this release.
<code>M\$IOMan</code>	Offset to system's I/O Manager (typically <code>IOMan</code>) name, if any.
<code>M\$PreIO</code>	Offset to a list of extension modules that will be called PRIOR to calling the <code>M\$IOMan</code> list.
<code>M\$PrcDescStack</code>	This field sets the stack size for process descriptors.

Process Descriptor Changes

The following changes were made to the process descriptor.

An `fpu` emulation state flag was added (`P$EmuState`).

A `P$Preempt` field was added. Setting the field to non-zero prevents system-state time slicing for the process.

`P$State` has a new state (`DeadChild`), for assistance with `F$Wait`.

`P$ProcSiz` indicates the size of this process descriptor. The size of a process descriptor is now set at system start-up, depending on FPU and process descriptor stack requirements.

`P$StackSiz` indicates the size of the stack area in the process descriptor.

`P$SigIRet` is used by the kernel to determine the state of the process with respect to signal intercept handler calls.

System Globals

The following system globals were added:

<code>D_FPUSize</code>	The maximum size of the FPU state save frame.
<code>D_FPUMem</code>	fpu emulation static memory pointer.
<code>D_SpurIRQ</code>	Spurious interrupt counter. This counter is used in conjunction with the <code>B_SpuIRQ</code> flag (<code>Init</code> module, <code>Compat</code> flags).
<code>D_Preempt</code>	Setting this to non-zero prevents system-state time-slicing for the system.
<code>D_AllocType</code>	Indicates the memory allocator type that the kernel is using.
<code>D_FDisp</code>	Pointer to the routine to install a fast dispatch routine.
<code>D_KerTyp</code>	Indicates the size of a device table entry. It is set by <code>IOMan</code> .

2 Corrections and Enhancements



D_DevCnt

The count of device table entries (maximum) for the system. This value may be larger than the value specified in the `Init` module, due to memory allocator rounding of memory requests.

D_MBAR

This global is applicable for the CPU32-family kernels. It indicates the address of the MPU's Module Block.

Kernel/IOMan Changes

The V3.0 kernel is divided as follows:

IOMan	All I\$ service requests calls now reside in IOMan. All F\$ service requests that are I/O related reside in IOMan.
Kernel	All non-I/O related F\$ service requests.



For More Information

See the *OS-9 Technical Manual, Appendix D (the system calls)*, for a complete list of which module contains which system calls.

The **disable data cache while in I/O** flag (the `Init` module, `Compat2` flag) is no longer used for that purpose. DMA-style drivers are now required to maintain data cache integrity, using `F$CCtl` calls and the `D_SnoopD` system global. (`D_SnoopD` is set from the `Compat2` flags of the `Init` module).

The **slow IRQ** flag (`Init` module, `Compat` flag) is no longer used. IRQ routines run under the `F$IRQ` polling system **MUST** conform to the register usage conventions of the V2.X series.

Support was added for the 68040 processor.

Support was added for the 68EC/LCXXX series processors.

Setting the time after forking a number of processors will now initialize the process descriptor time fields (`procs` no longer prints ???).

`SetStat` (`SS_Opt`) now checks the user buffer with `F$ChkMem`.

Problems with remote data initialization (68040, Initial Release only) have been repaired.

The new system global `D_FPUSize` contains the maximum size of an FPU state frame. This allows User-supplied FPU emulation to **resize** the FPU state frame buffer.

It is now much more difficult to unlink active program and trap modules.

The 68040 version of the kernel is now more robust in terms of support for 68040 chip-ERRATA items.

The Send Signal routine was considering signal value of 32 (`S$Deadly`) to be deadly signal. Only signals LESS than 32 are deadly (effectively, signals 2-31: Signal 1 is WAKE-UP, signal 0 is KILL).

`F$Chain` now correctly validates the **open path count**.

`F$Dfork` now validates the register buffer with `F$ChkMem`.

`EVInfo` requests now validate the user buffer with `F$ChkMem`.

Problems with `D_Forks` (count of forked processes) have been fixed.

Problems with Fork on systems with no RAM at 0 have been fixed.

A problem with activating `SysDbg` has been fixed.

`Chain` now clears the `fpu initialized` flag (in `P$EmuState`). This was an issue only for FPU emulation.

It is now possible for a cyclic alarm to delete itself.

It is no longer required that the current process be 0.0 when `F$SysDbg` calls are made from system-state. This allows (for example) drivers to make the call without forcing the current process to masquerade as user 0.0.

The kernel can now be made to ignore spurious interrupts. See the `Init` module's `Compat` flag.

Exceptions in alarm routines are now handled more robustly.

The kernel now supports the Master Stack Pointer (MSP) on processors that have the MSP/ISP stack pointers (68020, 68030, 68040).

Problems with SNAN exceptions with the 68882 FPCP have been fixed, so user-state exception handlers can now continue the process.

There is now a kernel specifically for the 68349. This is the same kernel as the CPU32-style kernel, except the on-chip caches of the 68349 are supported according to the `Init` module's `M$Compat2` flags.

Driver Changes

This section describes changes to the RBTEAC, SCF, and CBOOT/SCSI drivers.

RBTEAC Driver Changes

The RBTEAC driver has been modified in an attempt to operate on all Teac SCSI floppy drivers which were available at the time of testing.



For More Information

See the *OS-9 Board Support Packages Manual* for more detail on the tested controller/drive/firmware combinations and the corresponding jumper settings.

SCF Driver Changes

Many of the standard SCF drivers were modified so as to not disturb the Master/Interrupt stack bit in the status register when masking interrupts. This is a compatibility issue that is explained in **Chapter 5, Compatibility Issues**, of these release notes.

CBOOT/SCSI Driver Changes

The CBOOT and SCSI drivers written in the C language were modified to be compiled with the Ultra C compiler in backward compatibility mode. This was a first step towards conforming to ANSI specifications for all C code. In general, high-level SCSI drivers (such as `rbvccs` and `rbteac`) were compiled with full optimization and are smaller. Low-level SCSI drivers (dealing directly with hardware) cannot be compiled with optimization until the source becomes ANSI compliant. In a future release, all C drivers will be modified for ANSI compliance and compiled with full optimization.

2

Corrections and Enhancements



Rombug

Rombug now correctly handles and displays the Master and Interrupt stack pointer registers. This is a compatibility issue that is explained in **Chapter 5, Compatibility Issues**, of these release notes.

Chapter 3: OS-9 Version 3.0

Application Notes

Memory Management and Caching

With the release of the atomic kernel, you need to be aware of some memory management (MMU) and caching issues if your system processor provides these features. The general concepts of MMU and cache have not changed dramatically since OS-9 Version 2.4, but a clear understanding of the software components that deal with these issues will ensure smooth integration of your target system.

This section covers the following topics:

- Memory Management Units
- System Caching
- SSM Issues
- Standard 68040 SSM Defaults

Memory Management Units

MMU hardware is, in general, used to provide a protection mechanism under OS-9 so that user-state modules do not interfere with each other and the operating system. If the MMU hardware provides memory-mapping capabilities, these are usually not provided. The standard module name for the module that controls the MMU is **SSM** (for System Security Module). Microware provides standard SSM modules for the following systems:

- 68020 (using 68851 PMMU)
- 68030 (68851-style built-in MMU)
- 68040 (built-in MMU)

As stated previously, the SSM software only controls the MMUs protection mechanisms. However, on 68040 systems, SSM software also controls the modes of caching (refer to the ***MC68040 User Manual*** for full details of these modes).

SSM modules provide the following functional system call routines:

F\$Protect	F\$Permit	F\$ChkMem
F\$AllTsk	F\$DelTsk	F\$GSPUMp

These functional routines replace the kernel's (dummy) default routines for these system calls (except F\$GSPUMp, which is an unknown service request on non-SSM systems).

System Caching

The standard module name for the cache control module on OS-9 is `syscache`. `syscache` is a module that controls the processor's on-chip caches and can be customized to manage any external caches that the system might have.

The `syscache` module provides a functional F\$CCt1 routine (the kernel's default routine is a dummy routine) that:

- Performs operations on the cache control registers (non-68040 systems), or
- Executes the relevant cache instructions (68040 systems) according to the requested action.

On 68030 systems, the `Init` module's `Compat` flag is available to indicate whether cache-burst operation should be performed.

On 68040 systems, the modes of caching are determined by the SSM module used (system state), and interact with the `CacheList` entries in the `Init` module (user state).

For Version 3.0 kernels, the **atomic kernel environment** option makes it necessary for you to pay attention to certain system integration issues. These issues are explained in the following sections.

SSM Issues

For non-68040 systems, leave the SSM module out of the bootfile/bootroms. For 68040 systems, you can use the `SSM040` module for user-state cache mode support, using the `CacheList` entries of the `Init` module.

Standard 68040 SSM Defaults

The SSM module shipped by Microware for the 68040 has the following default characteristics. These may need some modification, depending on the setup of the MMU by the ROM startup code and the systems requirements.

The defaults are:

Table 3-1 Standard 68040 SSM Defaults

Register	Use	Set By
ITTO	Not used.	
ITT1 (Note 1)	Cache enable (write-through), entire memory map.	SSM

Table 3-1 Standard 68040 SSM Defaults (continued)

Register	Use	Set By
DTT0 (Note 2)	System specific I/O region, cache-inhibited, serialized access.	System ROM code (OEM specific)
DTT1 (Notes 1, 3, and 4)	System RAM, cache-enabled (write-through), entire memory map.	SSM

Note 1

The Cache List facility of the `Init` module applies to **user-state** access. For system-state, the caching mode can be either write-through or copy back, according to the module(s) used:

SSM040	Supervisor write-through.
SSM040_cbsup	Supervisor copy-back.

Note 2

The cache-inhibited I/O region is essential for correct I/O device operation. Many systems assume that I/O will only be accessed using system calls (`I$` calls), and thus, if user-state programs are to access this region directly, one of two methods must be used:

- Ensure that the function code checking flags are disabled in `DTT0` by the ROM code, or
- Add the I/O region (with the same attributes as `DTT0`) to the `Init` module cache lists and have the application program perform `F$Permit/F$Protect` calls to the desired region.

Note 3

`DTT1` describes the entire region by default. The `DTT0` set up will override `DTT1` if a hit occurs in the region that `DTT0` describes.

Note 4

You can further refine (override) areas that hit in `DTT1` using the Cache List entries in the `Init` module. The Cache List entries apply to **user-state** access only, and allow applications (for example) to have a shared (non-cachable) region with another processor, while also running copy-back caching on-board and default (write-through) caching to bus memory.

F\$FIRQ Application Notes

Atomic OS-9 features a fast IRQ system. This fast IRQ system provides a faster interrupt response context than the normal IRQ polling scheme, which is provided via the `F$FIRQ` system call.

This chapter covers the following topics:

- Differences between the fast and the normal IRQ services.
- Installing a routine during the driver's `Init` routine.
- Removing a routine during the driver's `TERMINATE` routine.
- Using fast IRQ service routines in the device driver.
- Taking complete control of the vector.



Note

The `F$FIRQ` system call is documented in the *OS-9 Technical Manual—System Calls*.

Differences Between the Fast and the Normal IRQ Services

The following are the differences between the fast and the normal IRQ services:

- The fast IRQ service saves few registers. It saves only the `d0` and `a2` registers. If you use any other register, you must save and restore it.
- The fast IRQ service performs no set up to catch exceptions. Therefore, you cannot generate any exceptions when in a fast IRQ service routine.
- If you make an operating system call while in a Fast IRQ routine, you must ensure the following:
 - You must save and restore registers changed by the system call (especially `d1` for the error return case).
 - You must disable system-state time-slicing by incrementing `D_Preempt` for the duration of the call.

Installing a Routine During the Init Routine

To install a routine during the device driver's `Init` routine, you need to do the following:

-
- Step 1. Get the vector number from the device descriptor.
 - Step 2. Specify the routine to install.
 - Step 3. Use the `F$FIRQ` system call to install the device on the `FIRQ` table.
-

For example, to install the `IRQSvc` routine into the `FIRQ` table, you could use the following:

```
(a1) = device descriptor pointer
(a2) = device static storage pointer
(a6) = system global data pointer

move.b    M$Vector(a1),d0    get vector # from
                             descriptor
moveq.l   #0,d1              reserved value: must be 0
lea.l    IRQSvc(pc),a0      routine to install
* pass (a2) as ptr to our static
os9      F$FIRQ             Install device on FIRQ
                             table
```



Note

This example does not include error checking.

Removing a Routine During the TERMINATE Routine

Removing a routine during the device driver's `TERMINATE` routine is similar to installing a routine. You need to do the following:

-
- Step 1. Get the vector number from the descriptor.
 - Step 2. Indicate the removal of the routine from the `FIRQ` tables.
 - Step 3. Use the `F$FIRQ` system call to remove the device from the `FIRQ` table.
-

The following example illustrates this:

```
(a1) = device descriptor ptr
(a2) = device static storage ptr
(a6) = system global data ptr

move. B   m$Vector(a1),d0   get vector # from
                             descriptor
suba.a   a0,a0               indicate removal from FIRQ
                             tables
* d1 is not inspected when deleting
* pass (a2) the same as when device added
OS9      F$FIRQ              remove device from FIRQ
                             table
```

Example Fast IRQ Service Routines in the Device Driver

When you do not request operating services, using the device driver's fast IRQ service routines is quite simple. Basically, you need to do the following:

-
- Step 1. Get a pointer to the hardware.
 - Step 2. Clear the interrupt.
 - Step 3. Return with carry clear.
-

Requesting operating system services (such as performing a system call) while using the device driver's fast IRQ service routines requires an additional step. You need to:

-
- Step 1. Save any registers that the system call may return changed. In particular, you should save `d1`, as it is always possible to receive an error.
 - Step 2. Disable system-state timeslicing.
 - Step 3. Perform the operating system request.
 - Step 4. Restore system-state timeslicing.
 - Step 5. Restore the registers saved in step 1.
 - Step 6. Return.
-

The following code shows how you could implement both examples:

```
d0.w = interrupting vector offset (vector number * 4)
(a2) = device static storage ptr
(a6) = system global data ptr

* the interrupt service routine for F$FIRQ functions can
* only modify the following registers: d0, a2, ccr.
* All other registers mused MUST be preserved.
*
* Exit Cases:
* Return from Interrupt Context: cc = carry clear
* If carry is clear on return, the kernel will continue
* interrupt service by calling the normal F$IRQ polling
* table system to check for further interrupts pending
* on the vector.
*
```

In the first example, no operating system services are requested. It is a direct IRQ service which immediately returns to the interrupted context.

```
IRQSvc: movea.l  HWAddr(a2),a2      get ptr to
                                     hardware
        move.b  #IrqClr,IrqSt(a2) clear interrupt
        rts                                     return (carry
                                               clear)
```

In the second example, the operating system F\$Send service is requested. However, it is still a direct IRQ service which immediately returns to the interrupted context.

```

IRQSvc:  move.l   d1,-(a7)           save d1 reg (if OS
                                           error return)
         move.w   V_WAKE(a2),d0     get process ID to
                                           wake up
         clr.w    V_WAKE(a2)       signal wake-up sent
         movea.l  HWAddr(a2),a2    get ptr to hardware
         move.b   #IrqClr,IrqSt(a2) clear interrupt
         moveq.l  #S$Wake,d1      signal to send
         addq.l   #1,D_Preempt(a6) system state
                                           timeslicing off
         OS9      F$Send          send signal (no error
                                           check)
         subq.l   #1,D_Preempt(a6) system state
                                           timeslicing on
         move.l   (a7)+,d1        restore d1, clear
                                           entry
         rts                               return

```

In both examples, if Carry is set at routine end, the normal F\$IRQ system polling table will service additional devices on the vector.

Taking Control Of the Vector

The normal (F\$IRQ) and fast (F\$FIRQ) polling schemes provide an efficient and relatively fast mechanism for dispatching to interrupt service routines. These mechanisms provide adequate response time for most situations. However, when the system requires even faster response, you can directly install the interrupt service routine across the vector itself or in the OS-9 Jump Table.

Installing the routine in the vector table is slightly faster than using the Jump Table method, but it has the following disadvantages:

- If the system vectors are in ROM, then the entry cannot be updated.
- Vector monitoring (using ROM debuggers) will be lost for that vector, as the IRQ routine will be ahead of the debugger's routine.

The following examples show how to implement both methods. In both cases, the register input is:

```
(a1) = device descriptor ptr
(a2) = device driver static ptr
(a6) = system global data ptr
```

Vector Table Method

```
moveq.l #0,d0          swap reg
move.b  M$Vector(a1),d0 get vector
asl.l   #2,d0          multiply by 4 for longword entries
ifeq   (CPUType-68000)*(CPUType-68010)*(CPUType-68070)*(CPUType-68302)
* MPU with vectors always at address 0.
suba.l  a3,a3          vectors always at address 0
else
* MPU with VBR usage.
movec   vbr,a3        locate vectors
endc   CUType
move.l  (a3,d0.w),    VectSave(a2) save original vector
lea.l   IRQSvc(pc),a0 point at service routine
move.l  a0,(a3,d0.w)  install routine in vector table
```

Jump Table Method

```
moveq.l #0,d0          sweep reg
move.b  M$Vector(a1),d0 get vector #
subq.l  #2,d0          jump table entries biased at bus error
start
mulu.w  #10,d0         each entry is 10 bytes long
movea.l D_VctJump(a6),a3 get address of jump table
move.l  6(a3,d0.w),VectSave(a2) save original pointer
lea.l   IRQSvc(pc),a0  point at IRQ service routine
move.l  a0,6(a3,d0.w)  install it in jump table
```

In both examples, the interrupt service routine is being installed before the kernel's IRQ polling system. Interrupt service routines that run in this environment must be aware of the following consequences of this method:

- You **must** save any registers used by the routine and restore them before exiting the routine.
- If the routine elects to return to the interrupted context, it must exit via an `rte` (not `rts`, as in `F$IRQ` or `F$FIRQ` installed routines).



Note

The jump table method leads to the vector offset being pushed on the stack before calling the routine; thus, the direct exit methods are:

Table 3-2 Direct Exit Methods

Vector Table		Jump Table	
<code>rte</code>	<code>return</code>	<code>addq.l #4,a7</code>	Toss stacked vector offset
		<code>rte</code>	<code>return</code>

- Returning directly to the interrupted context will delay any fast dispatch routines installed by the IRQ service routine or debugger vector monitoring. It is generally better to service the interrupt and then call the original handler for any further processing.
- When the driver's terminate routine is run, be sure to restore the original routine pointer/vector.

- Keep stack use to a minimum. The kernel's routines use the `Init` module's `IRQ Stack` field, and because your routine is ahead of the kernel, the stack switch has not taken place.
- Avoid exceptions in the service routine. The kernel has no way of knowing that your service routine is running and thus the kernel will deliver the exception condition to the currently active process (which may have no relationship to your routine). That process will terminate with the exception error.
- If you perform operating system service requests, disable system-state timeslicing (`D_Preempt`) during the operating system request.

Fast Dispatch Application Notes

You can now run subroutines that you want to run with your interrupt service routine outside of the context of the interrupt service routine. The interrupt service routine places a dispatch entry in the fast dispatch queue located with `D_FDispQ` to be called when the interrupt service routine finishes. Then, at the end of the interrupt service routine, the system checks this dispatch queue and immediately runs any subroutines that are in the queue until the queue is empty.

This section covers the following topics:

- Inserting subroutines into the fast dispatch queue.
- Important notes.

Inserting Routines into the Fast Dispatch Queue

You can insert subroutines into the fast dispatch queue using two different methods:

- Using the `D_FDisp` routine.
- Setting the status of `FstP_status`.

Using the `D_FDisp` Routine

You can use the `D_FDisp` routine to insert fast dispatch routines into the table. The `D_FDisp` routine can be called by any system-state routine by passing the pointer to the fast dispatch routine block to insert in the `(a0)` register and then `jrs`-ing to it.

This puts the subroutine at the end of the queue. You should be sure to preserve all the registers used by the subroutine.

D_FDisp does not return any values.

Setting the Status of FstP_stat

The following is a description of the fast dispatch queue:

* Fast Dispatch Description

```
org 0
FstP_next      do.l 1 next element in queue
FstP_routine   do.l 1 pointer to dispatched routine
FstP_data      do.l 1 pointer to static data
FstP_status    do.l 1 status information
FstP_size      equ . size of FstP
```

To place a process in the queue, set the status of FstP_Queued. The system will call the process and take it out of the queue while the program is executing. It will also set the FstP_InUse bit. The following is an example of inserting a program in the fast dispatch queue:

```
fdisp: movem.l  d0/d7/a0/a1/a6,-(sp)    save regs
        move.w   sr,d7                save irq masks
        ori.w    #IntMask,sr          disable interrupts
        lea.l    D_FDispSys(a6),a0    get system fast
                                        dispatch process block
                                        ptr
        bset.b   #FstP_Queued,FstP_status(a0) is it
                                        already in queue?
        bne.s   FDisp_30              continue if so
*activate system block
*
lea.l    seth(pc),a1                  going to seth
move.l   a1,FstP_routine(a0)          set (seth) routine
                                        ptr into system block
move.l   a6,FstP_data(a0)             set (system global)
                                        data ptr
move.l   #0,FstP_next(a0)             there is no next!
bclr.b   #FstP_InUse,FstP_status(a0) clear inuse bit
bra.s    FDisp_10                     go insert
* entry point for caller's via D_FDisp
```

3 OS-9 Version 3.0 Application Notes



```
*
* a0 = ptr to fast dispatch block to insert
*
FastDisp: movem.l  d-/d7/a0/a1/a6,-(sp)  save regs
            move.w  sr,d7                save irq masks
            ori.w   #IntMask,sr         disable interrupts
            sysglob a6 get system      global data ptr
* common insert routine
*
FDisp_10  move.l  D_FDispQ(a6),d0      queue is there
bne.s    FDisp_20                      go find end
move.l   a0,D)FDispQ(a6)              set block as queue head
bra.s    FDisp_30                      exit
FDisp_20 movea.l  d0,a1                copy queue pointer
move.l   FstP_next(a1),d0             any next?
bne.s    FDisp_20  .yes;              find end
move.l   a0,FstP_next(a1)            insert block
FDisp_30 move.w  d7,sr                restore irq masks
move.l   (sp)+,d0/d7/a0/a1/a6        restore regs
rts                                           return
ends
```

Important Notes

Please read the following notes concerning the use of the fast dispatch:

- You are executing on a system stack of an unknown size. Therefore, you need to be careful so that you do not run out of stack.
- Interrupts will be unmasked to the level that they were before the last interrupt occurred.
- You must save all registers that you use.
- No other processes will run until the routines in the fast dispatch queue are completed.
- You must clear `FstP_InUse` and set `FstP_Queued` to put the routines in the queue.
- The developer must determine the ordering of the queue when inserting the routine yourself. If you use the `D_FDisp` method, `FIFO` queues are used.

Semaphores

Semaphores provide another form of process synchronization. Semaphores are useful for providing process' exclusive access to shared resources. They are similar to events in the way they provide applications with mutually exclusive access to data structures. Semaphores differ from events in that they are strictly binary in nature, and hence, are more efficient in their implementation.

This section covers the following topics:

- Using semaphores
- Example semaphore use



For More Information

Refer to the **Ultra C** manual for information about the C bindings used with semaphores.

Using Semaphores

A semaphore has two states:

- | | |
|----------|--|
| Reserved | When a semaphore is reserved, any process that attempts to reserve the semaphore will wait. This includes the process that has the semaphore reserved. |
| Free | When a semaphore is free, any process may try to reserve the semaphore. |

To acquire exclusive access to a resource, a process may use the `_os_sema_p()` C binding to reserve the semaphore. If the semaphore is already busy, the process is suspended and placed at the end of the semaphore's wait queue.

To release exclusive access to a resource, a process may use the `_os_sema_v()` C binding to release the semaphore. When the owner process releases the semaphore, the first process in the semaphore's queue is activated and retries the reserve operation on the semaphore.

A single semaphore system call provides all of the semaphore's functionality. One of the parameters to the system call indicates which operation is being performed on the semaphore. The other parameter to the system call is a pointer to the semaphore structure. The semaphore C-language structure is defined below. Unlike events, there is no system call provided to create a semaphore. Since semaphores are typically used to protect specific resources, it is useful to declare the semaphore structure as part of the resource structure.

A typical application using semaphores would create a data module containing the memory for the intended resource and its associated semaphore. Using a data module for implementing semaphores allows applications to use the operating system's module protection mechanisms for protecting the semaphore.

Once the semaphore data module has been created and initialized, additional processes within the application may use the semaphore by linking to the semaphore data module. The semaphore data module must be created with appropriate permissions to allow the other processes within the application to link to and use the semaphore and its resources.

Example Semaphore Structure

The following is the semaphore structure:

```

/* Semaphore structure definition */
typedef struct semaphore {
    u_int32 s_value,          /*semaphore value (free/busy status)*/
           s_lock;          /*semaphore structure lock (use
                           count)*/
    Pr_desc s_qnext,        /*wait queue for process descriptors*/
           s_qprev;        /*wait queue for process descriptors*/
    u_int32 s_length,       /*current length of wait queue*/
           s_owner,        /*current owner of semaphore*/
           s_rewerved[6];  /*reserved space*/
} semaphore, *Semaphore;

```

Example Semaphore Use

The following example shows how to use semaphores.

```

#ifndef _SEMAPORE_H
#include <semaphore.h>
#endif

#ifndef _MODULE_H
#include <module.h>
#endif

register Semaphore sema;
Semaphore locate_semaphore();

/* link/create the semaphore */
sema = locate_semaphore();
while (1) {

    /* perform semaphore "P" operation (reserve the semaphore) */
    if ((err = _os_sema_p(sema)) != SUCCESS)
        exit(_errmsg(err, "could not perform P operation - "));

    /*Enter critical section */

    /* perform semaphore "V" operation (release semaphore) */
    if ((err = _os_sema_v(sema)) != SUCCESS)
        exit(_errmsg(err, "could not perform V operation - "));
}
/* terminate usage of the semaphore */

```

```
_os_sema_term(sema);
}

#define ATTR_REV 0x8001 /* semaphore data-module's attribute
                        revision value */

/* locate_semaphore - link or create semaphore module
(initialize it). */
Semaphore locate_semaphore()
{
    register Semaphore sema;
    register mh_com *semamod;
    static char *semaname = "semaphore";
    mh_com *modlink();
    mh_com *_mkdata_module();
    /* attempt to link to the semaphore */
    if ((semamod = modlink(semaname, MT_DATA)) == ((mh_com*)-1))
    {
```

Non-Standard I/O Systems and the ROMIO File Example

For atomic systems that wish to perform I/O without using the standard OS-9 I/O system (as provided by IOMan), a number of choices can be made to provide I/O services for the system. These choices are:

- Embed the I/O routines into the application code itself.
- Provide a custom I/O system module.
- Use the system's ROM I/O routines.

To demonstrate the feasibility of non-standard I/O systems, an example file, called `romio.a`, is provided (both in object and source form) to show how this can be achieved.

The `romio.a` file provides an arbitrarily compatible I/O system similar to IOMan and uses the system's console device routines to perform I/O. You can, for example, run the standard `procs` utility on an atomic/ROMIO system and achieve the same output as an atomic or development kernel running IOMan. With a carefully designed I/O system module, you can develop and debug an application in a full development environment, and then download the binary forms of the application without redesigning/recompiling on to an atomic/custom-I/O target.

If your application lends itself to this type of environment, you should study the supplied `romio.a` source code to see how it can be adapted to your desired purposes. When you use this example as a basis for your application, be aware that `romio.a` was designed to be generic and thus uses standard ROM entry points to call the low-level polled I/O routines of the boot ROM.

As most of these routines typically mask interrupts to level 7 while performing I/O, real-time response will suffer while in your ROM routines. The choices you have in this case are typically:

- Modify your ROMs so that interrupt masking is removed, **or**,
- Modify `romio.a` to include non-masking versions of your ROM I/O routines.

3 OS-9 Version 3.0 Application Notes



Chapter 4: Compatibility Issues

Introduction

OS-9 Version 3.0 provides a high degree of compatibility with the V2.x series of releases. However, there are some changes in V3.0 that may need minor attention by software developers. The following sections discuss issues that must be addressed when you update to V3.0.

Status Register (SR) and MSP/ISP Stack Registers

OS-9 Version 3.0 introduces the usage of the **Master Stack Pointer** (MSP) for processors that have the MSP/ISP supervisor registers (68020, 68030, and 68040).

On MSP-kernels, the MSP is used for the process stack when the process is in system-state (for example, performing a system call). On these kernels, the Interrupt Stack Pointer (ISP) is only used for Interrupt Servicing. You must modify any system-state code that assumes the ISP is the only active system-state stack pointer so that no accidental stack switch occurs. The most common places where system-state code performs operations on the status register (SR) that may affect the MSP flag are:

- Device drivers that mask interrupts
- System-state threads

Device Drivers That Mask Interrupts

Often, device drivers that need to mask interrupts form an **SR image**, and then use that image during the critical section. The image formed **MUST** take into account the context in which the image will be used. Usually the value of the SR when the **SR image** is computed can be used as a base value to select the appropriate stack. Thus, the following code example is incorrect for masking interrupts in a driver:

```
moveq.l  #0,d0                clear register
move.b   PD_M$IRQLvl(a1),d0  get irq level
lsl.w    #8,d0                move level to correct
                                SR position
```

4 Compatibility Issues



```
bset.l    #SupvsrBit+8,d0    force system-state
move.w   d0,MaskIRQ(a2)    save SR image for later
```

This example is incorrect in that it forces the ISP register to become active when the image is loaded into the SR (status register) at some later stage as shown here:

```
subq.l   #4,a7                make SR save space (keep
                             stack LONG aligned)
move.w   sr,0(a7)            save current IRQ level
move.w   MaskIRQ(a2),sr     mask interrupts.
```

----perform useful work----

```
move.w   0(a7),sr           restore SR
addq.l   #4,a7                toss scratch
rts                      return
```

As the above code may be entered with either the MSP on 68020/030/040 processors or ISP (for other processors) active, the setup example is incorrect. Using the mask derived in the above manner causes an accidental stack switch.

The correct way to form the image is shown in the following example:

```
moveq.l  #0,d0                clear register
moveq.l  #0,d1                clear register
move.w   sr,d1                copy current
                             IRQ/Trace/Stack flags
andi.w   #IntEnab,d1         clear interrupt flags
move.b   M$IRQLvl(a1),d0     get device irq level
lsl.w    #8,d0                move to correct position
                             in SR image
or.w     d0,d1                for sr image
move.w   d1,MaskIRQ(a2)     save it
```

This code preserves the state of the MSP bit from the current SR in the saved `MaskIRQ` value.

System-state Threads

System-state threads are those processes created by system-state code through methods other than `F$Fork`. These are typically created for special purposes such as I/O system processes. The SBF file manager uses system-state threads for its asynchronous activities.

When these processes are created, the process creating the thread process creates the register image (in the process descriptor) for the new process. When this is being done, it is important that the **current** active supervisor stack selection is propagated to the new process. Thus, this example is incorrect:

```
move.w  #0x200,R$sr(a5) <--forces ISP, system-state
```

The correct way to set the SR image is:

```
move.w  sr,d0           copy current SR
andi.w  #Supervis+MasterSP,d0  retain MSP and
                               Supervisor flags
move.w  d0,R$sr(a6)     set SR image in
                               process desc. reg
                               image
```

Process Descriptor Changes

The format and size of the process descriptor has been changed for OS-9 Version 3.0. The changes should have minimal or no effect on user-state or system-state code. Code that examines or affects process descriptor information should take the following points into account:

- The size of a process descriptor
- System-state preemption

Process Descriptor Size

With OS-9 Version(s) 2.x, process descriptor size was fixed at 2048 bytes. With V3.0, this has been changed. The process descriptor size is now determined at system-startup time and is based upon a number of variables, such as the process descriptor stack size given in the `Init` module and the presence/absence of Floating Point hardware and/or emulation software.

The new layout of a process descriptor is defined as:

- A 1024 byte fixed part
- A variable sized section

The 1024 byte fixed part is essentially the first 1K of the V2.x process descriptor. Any new fields introduced for Version 3.0 have been added to previously reserved sections.

The variable sized section is used for Floating Point support (if needed) and Process Stack space. Future releases of OS-9 may add other information to this part.

Code that needs to examine the variable parts of the process descriptor can access pointers/fields in the fixed part that provide the information needed to examine the variable fields. For example, to locate the **top** of the process descriptor, the following changes would occur:

Under V2.4:

```
lea.l    2048(a4),a0    find top of proc desc
```

Under V3.0:

```
moveq.l  #0,d0         clear register
move.w   P$ProcSiz(a4),d0 get size of this proc
                                     desc
lea.l    (a4,d0.1),a0   find top of proc desc
```

System-state Preemption

V3.0 introduces system-state preemption. If the system must run system-state code that cannot be time-sliced, there are two methods that you can use:

-
- Step 1. Set the `Init` module's `M$SysConf` flag to disable system-state time slicing. This gives the system the same system-state characteristics as under V2.x (that is, ALL system-state preemption is OFF).

This is the least desirable way as it causes adverse affects on system determinism.

- Step 2. Set the `P$Preempt` field of the process descriptor to a non-zero (preferably 1) value. This causes the system to ignore system-state time slicing for THIS process only.

This is the preferable method as it degrades system determinism the least.

4 Compatibility Issues



You can set `P$Preempt` either at process startup time or only during critical code sections.



For More Information

See the ***OS-9 Technical I/O Manual*** for further details on system-state time slicing.

ROM Compatibility

The boot ROM interface between the ROM code and the kernel has NOT been changed for V3.0. However, the MSP register was not correctly supported in V2.x versions of the ROM debugger (`r_ombug`). Note that the debuggers do not use the MSP register, nor does any of the boot ROM code supplied by Microware. The MSP support issue is only important when the system calls back into the debuggers (either via `F$SysDbg`, the ABORT switch, or because of a fatal system exception).

While Microware recommends that you use the latest ROM booting code for V3.0, it may be possible for your system to boot V3.0 while running V2.x boot ROMs. To accomplish this, you need:

- A **non-MSP** supporting processor (not a 68020, 68030, or 68040),

OR:

- A 68020/030/040 system that had:
 - Boot ROMs that only had booter code (no debugger)
 - No abort switch

Changes to System Definitions

The system structure changes to V3.0 were made so that the impact to code is minimized. The following comments are provided as a guide to the impact of V3.0 and your system requirements:

- V2.X user-state code should run as is under V3.0. The only possible problems would arise from user-state processes that make assumptions about the size of process descriptors. Code that needs to examine the variable section of the process descriptor needs to:
 - Get the fixed part of the process descriptor.
 - Extract the size information.
 - Get the remaining part of the process descriptor.
- The I/O system device table size (maintained by `IOMan`) has been changed.

Any code that needs to examine the device table entries should use the `D_DevSiz` (size of a device table entry) and `D_DevCnt` (number of device table entries) system globals to gain information about the device table.

- The interrupt polling table entry size has been changed. You must relink system-state code that walks these table entries.
- The obsolete path descriptor field (`PD_CNT`) has been removed from the path descriptor definitions.

The correct field to use is `PD_COUNT`. If needed, you can recode system-state code to use the correct field.

Microware recommends that, at a minimum, you relink all your system-state code for V3.0. This ensures that you account for any structure/definitions changes. If labels that you used under

V2.x become unresolved, that indicates that some change is required in the algorithm using that label (for example, someone assuming that a process descriptor is 2048 bytes in size).

Determining the OS-9 Release Level

As you use different releases of an operating system, determining which version you are using may become an issue because features and services provided by different releases must be accounted for if code is to successfully operate under many releases.

To help you detect different release levels of OS-9, V3.0 introduces the `F$SysID` system call. Some of the later versions in the OS-9 v2.x series supported forms of `F$SysID`, but these were undocumented.



For More Information

Refer to the *OS-9 Technical Manual* for details about `F$SysID`.

You can use `F$SysID` to determine the following:

- If `F$SysID` causes an error #208 (Unknown Service Request), the kernel release is at V1.0, V1.1, V1.2, or V2.0.
- If no error occurs, the release is at V2.1 or later. If `d6=0`, the kernel release is V2.1, V2.2, V2.3, or V2.4. If `d6!=0`, the kernel release is given in the register.

Chapter 5: Documentation Changes

Introduction

The OS-9 Version 3.0 software changes described in these release notes necessitated wide-spread changes to OS-9 documentation.

Major revisions to the documentation reflect:

- The new MWOS directory structure
- Makefile changes
- Installation changes
- New OS-9 packages: Embedded, Disk-based, Extended

The new and updated manuals include the following:

Table 5-1 OS-9 for 68K Documentation

Manual Title	Revision	Product Code
OS-9 Technical Manual	A	TEC68NARAMO
OS-9 Technical I/O Manual	A	TIO68NARAMO
Using OS-9	A	USR68NARAMO
OS-9 OEM Installation Manual	A	IST68OERAMO
Using OS-9/Internet	C	INT68NARCMO
OS-9 ROM Debugger User's Manual	B	RBG68NARBMO
Installing the Network File System (N.F.S.)	A	NFSINSTPAMO

Table 5-1 OS-9 for 68K Documentation (continued)

Manual Title	Revision	Product Code
OS-9 PC File Manager (PCF) Manual	B	PCF68NARBMO
OS-9 Utilities Reference Manual	A	UTL68NARAMO



Note

The OS-9 Utilities were previously included in the *Using Professional OS-9* manual. New utilities have been added and are now a separate manual.

The OS-9 system calls are now in Appendix D of the *OS-9 Technical Manual*.

5 Documentation Changes



Chapter 6: Known Bugs

Introduction

The following are known bugs in the OS-9 Version 3.0 release.

ROMbug

The ROMBug disassembler incorrectly displays the target location of the branch in `DBCC` instructions. When using symbols, the target location is **some location in another modules data area**. When not using symbols, the target offset value is off by `0x10000`.

F\$SysDbg

Under certain circumstances, the caller's registers may be corrupted in the register display when you invoke `F$SysDbg`. The problem occurs when:

ROMbug is monitoring the bus error vector.

A bus error occurs during the call into ROMbug (for example, probing a card for **debugger enabled** and the card does not exist).

When the bus error display occurs, the registers displayed are correct.

When you type `g` to enter ROMbug, the registers displayed differ from those displayed at the exception.

I\$Attach

I\$Attach is not fully re-entrant. Driver `Init` routines should avoid sleeping when multiple attaches can occur on a device. For example, the following might fail (when the device is not attached to the system):

```
$iniz /device & iniz /device
```

The work around is:

```
$iniz /device; iniz /device
```

Pre-V2.4 ROMs

Version 3.0 kernels will fail to locate the `Init` module (or any other loaded bootfile modules) if the ROMs installed are pre-V2.4.

Setime and the Startup File

There is a problem with the `Setime` utility which causes a `Setime` command within a procedure file (such as `Startup`) to fail when requesting the time from a user.

The workaround is to redirect the standard I/O paths for the `Setime` command line in the procedure file.

Thus, the following will fail if placed in a procedure file:

```
setime          ;* set system time
```

The following will work correctly:

```
setime <>>>/term;* ask for time from system console
```

Note also that this problem does NOT affect `Setime` operation when using the time stored in a battery-backed clock. Thus, `setime -s` will work as expected from within a procedure file.

6 Known Bugs



