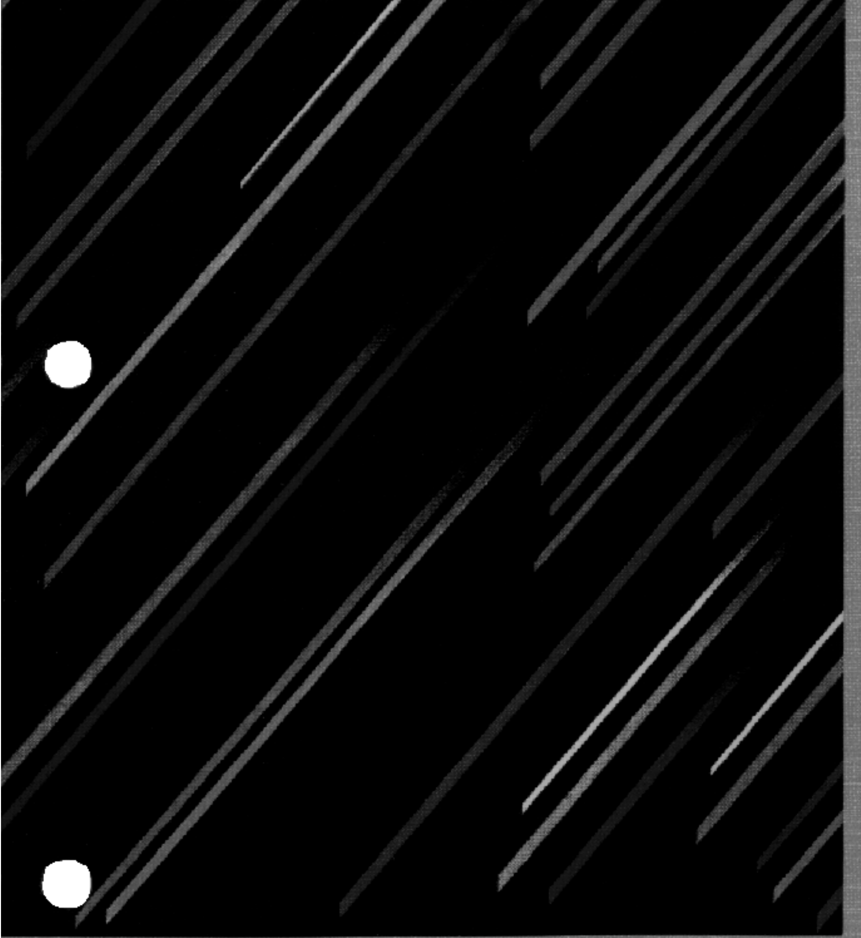
OS-9 Level Two Development System



TANDY®

OS-9 Level Two

Development System

OS-9 Level Two Software:
© 1987 Microware Systems Corporation.
Licensed to Tandy Corporation. All rights Reserved.

OS-9 Level Two Development System Software:
© 1987 Tandy Corporation.
All rights Reserved.

OS-9 Level Two Development System Documentation:
Interactive Debugger, Screen Editor, Relocating Macro Assembler, Utilities, Commands
© 1987 Tandy Corporation.
All rights Reserved.

Reproduction or use of any portion of this manual without express written permission from Tandy Corporation is prohibited. While reasonable efforts have been taken in the preparation of this manuals to assure its accuracy, Tandy Corporation assumes no liability resulting from any errors in or omissions from this manuals, or from the use of the information contained herein.

Tandy and the Tandy logo are registered trademarks of Tandy Corporation.

Motorola is a registered trademark of Motorola Inc.

10 9 8 7 6 5 4 3 2 1

Contents

This manual contains documents for:

- **Interactive Debugger**
  A program to aid in diagnosing system programs, testing machine language programs and to gain access to your computer's memory.

- **Screen Editor**
  A screen-oriented text editor for preparing letters, documents, and for writing OS-9 programs.

- **Relocating Macro Assembler**
  A full-featured macro assembler and linkage editor.

- **Utilities**
  Three utility programs: Make, to help maintain current version software; Touch, to update files; and VDD, a Virtual Disk Driver/RAM Disk Driver to create a high-speed storage in your systems RAM.

- **Commands**
  Twelve additional OS-9 commands to expand your system's capabilities.

Each document contains its own table of contents.

TERMS AND CONDITIONS OF SALE AND LICENSE OF TANDY COMPUTER EQUIPMENT AND SOFTWARE PURCHASED FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND RADIO SHACK FRANCHISEES OR DEALERS AT THEIR AUTHORIZED LOCATIONS

USA LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

A. CUSTOMER assumes full responsibility that this computer hardware purchased (the "Equipment"), and any copies of software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.

B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. LIMITED WARRANTIES AND CONDITIONS OF SALE

A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects **This warranty is only applicable to purchases of Tandy Equipment by the original customer from Radio Shack company-owned computer centers, retail stores, and Radio Shack franchisees and dealers at their authorized locations.** The warranty is void if the Equipment or Software has been subjected to improper or abnormal use If a manufacturing defect is discovered during the stated warranty period the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or a participating Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.

B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph Software is licensed on an "AS IS" basis, without warranty The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, a participating Radio Shack franchisee or Radio Shack dealer along with the sales document.

C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK

D. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK MAKES NO EXPRESS WARRANTIES, AND ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE IS LIMITED IN ITS DURATION TO THE DURATION OF THE WRITTEN LIMITED WARRANTIES SET FORTH HEREIN.**

E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER

III. LIMITATION OF LIABILITY

A. **EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE." IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE."**

NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.

B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software

C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.

D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER

IV. SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the TANDY Software on **one** computer, subject to the following provisions

A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software

B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software

C. CUSTOMER may use Software on a multiuser or network system only if either the Software is expressly labeled to be for use on a multiuser or network system, or one copy of this software is purchased for each node or terminal on which Software is to be used simultaneously

D. CUSTOMER shall not use make manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License Customer is expressly prohibited from disassembling the Software

E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use

F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.

G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby Radio Shack sells or conveys such Equipment to a third party for lease to CUSTOMER

B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and or licensor of the Software and any manufacturer of the Equipment sold by Radio Shack

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state

4/87

Interactive Debugger

Contents

Chapter 1 / Introduction 1-1
Calling Debug 1-1
Basic Concepts 1-1

Chapter 2 / Expressions 2-1
Constants 2-1
Special Names 2-2
Register Names 2-3
Operators 2-3
Forming Expressions 2-4
Indirect Addressing 2-4

Chapter 3 / Debug Commands 3-1
Calculator Commands 3-1
Dot and Memory Examine/Change Commands 3-2
Incrementing Dot 3-3
Decrementing Dot 3-3
Changing Dot 3-3
Changing Dot's Contents 3-4
Register Examine/Change Command 3-5
Breakpoint Commands 3-7
Setting Breakpoints 3-8
Removing Breakpoints 3-8
Program Setup and Run Commands 3-9
Goto Command 3-10
Link Command 3-10

Utility Commands 3-11
Clearing Memory 3-11
Displaying Memory 3-12
Searching Memory 3-12
Shell Command 3-12
Quitting Debug 3-13

Chapter 4 / Using Debug 4-1
Sample Program 4-1
Using Debug 4-4
Patching Programs 4-7
Patching OS-9 Component Modules 4-8

Chapter 5 / Debug Command Summary and Error Codes 5-1
Debug Command Summary 5-1
Dot Commands 5-1
Register Commands 5-2
Program Setup and Run Commands 5-2
Breakpoint Commands 5-2
Utility Commands 5-3
Debug Error Codes 5-3

Chapter 1

Introduction

Debug is an interactive debugger that aids in diagnosing system programs and testing machine-language programs for the 6809 microprocessor. You can also use it to gain direct access to the computer's memory. Debug's calculator mode can simplify address computation, radix conversion, and other mathematical problems.

Calling Debug

To run Debug, type the following command at the OS-9 system prompt:

DEBUG [ENTER]

Basic Concepts

Debug responds to 1-line commands entered from the keyboard. The screen shows the DB: prompt when Debug expects a command.

Terminate each line by pressing [ENTER]. Correct a typing error by using the backspace (←) key, or delete the entire line by pressing X while pressing [CLEAR].

Each command starts with a single character, which you can follow with text or one or two arithmetic *expressions*, depending on the command. You can use upper- or lowercase letters or a mixture. When you use the spacebar to insert a space before a specific *expression*, the screen shows the results in hexadecimal and decimal notation. For example, in the calculator mode, to obtain the hexadecimal and decimal notation for the hexadecimal expression **A+2**, type:

[SPACEBAR][A][+][2]

Debug displays:

```
DB: A+2
    $000C #00012
```

Chapter 2

Expressions

Debug's integral expression interpreter lets you type simple or complex expressions wherever a command calls for an input value. Debug expressions are similar to those used with high-level languages such as BASIC, except that some extra operators and operands are unique to Debug.

Numbers in expressions are 16-bit unsigned integers--the 6809's *native* arithmetic representation. The allowable range of numbers is 0 to 65535. Debug performs two's complement addition and subtraction correctly, but displays all results as positive numbers in decimal form.

Some commands require byte values. The screen shows an error message if the result of an expression is too large to be stored in a byte; that is, if the result is greater than 255. Some operands, such as individual memory locations and some registers, are only one byte long, and Debug automatically converts them to 16-bit *words* without sign extension.

Spaces, other than a space at the beginning of a command, do not affect evaluation of the expression. Use them as necessary between operators and operands to improve readability.

Constants

Constants can be in base 2 (binary), base 10 (decimal), or base 16 (hexadecimal). Binary constants require the prefix %. Decimal constants require the prefix #. Debug assumes all other numbers to be hexadecimal. They can have the optional prefix $. The following table shows examples of each type of constant:

Decimal	Hexadecimal	Binary
#100	64	%1100100
#255	FF	%11111111
#6000	1770	%1011101110000
#65535	FFFF	%1111111111111111

You can use character constants. Use a single quotation mark (') for 1-character constants and a double quotation mark (") for 2-character constants. Quotation marks produce the numerical value of the ASCII codes for the character(s) that follow. For example:

```
'A   =  $0041
'0   =  $0030
"AB  =  $4142
"99  =  $3939
```

Special Names

Dot (.) refers to Debug's current working address in memory. You can examine it, change it, update it, use it in expressions, and recall it. Dot eliminates a tremendous amount of memory address typing.

Dot-Dot (..) is the value of Dot before the last time it was changed. Use Dot-Dot to restore Dot from an incorrect value, or use it as a second memory address.

Register Names

Specify the MPU registers with a colon (:) followed by the mnemonic name of the register, as follows:

:A	=	Accumulator A
:B	=	Accumulator B
:D	=	Accumulator D
:X	=	X Register
:Y	=	Y Register
:U	=	U Register
:DP	=	Direct Page Register
:SP	=	Stack Pointer
:PC	=	Program Counter
:CC	=	Condition Codes Register

The values returned are the test program's registers, which are *stacked* when Debug is active. Debug increases 1-byte registers to a word when used in expressions.

> **Note:** When a break point interrupts a program, the SP register points at the bottom of the MPU register stack.

Operators

Operators specify arithmetic or logical operations to be performed within an expression. Debug executes operators in the following order:

-	(negative numbers)
& and !	(logical AND and OR)
* and /	(multiplication and division)
+ and -	(addition and subtraction)

Operators that are in a single expression and that have equal precedence (for example, + and -) are evaluated left to right. You can use parentheses, however, to override precedence.

Forming Expressions

An *expression* is composed of any combination of constants, register names, special names, and operators. The following are valid expressions:

```
#1024+#128
:X-:Y-2
.+20
:y*(:X+:A)
:U & FFFE
```

Indirect Addressing

Indirect addressing returns the data at the memory address, using a value (expression, constant, special name, and so on) as the memory address. The two Debug indirect addressing modes are:

<*expression*>	returns the value of a memory byte using *expression* as an address
[*expression*]	returns the value of a 16-bit word using *expression* as an address.

For example:

<200>	returns the value of the byte at Address 200
[:X]	returns the value of the word pointed to by Register X
[.+10]	returns the word value at Address Dot plus 10

Chapter 3

Debug Commands

This chapter describes Debug's available commands. Following the description for each command, there is an example. The left side of the example shows what you type, and the right side shows what the screen displays. Be sure to execute these examples in the order they appear so you obtain the screen display shown. Many of the examples' results depend on examples previously executed. Also, remember to press **[ENTER]** after each command.

Calculator Commands

The **[SPACEBAR]** *expression* command evaluates the specified *expression* and displays the result in both hexadecimal and decimal. For example:

You Type:	The Screen Shows:
[SPACEBAR]5000+200[ENTER]	$5200 #20992
[SPACEBAR]8800/2[ENTER]	$4400 #17408
[SPACEBAR]#100+#12[ENTER]	$0070 #00112

You can also use this command to convert values from one representation to another. For example:

You Type:	The Screen Shows:
[SPACEBAR]%11110000[ENTER]	$00F0 #00240
[SPACEBAR]'A[ENTER]	$0041 #00065
[SPACEBAR]#100[ENTER]	$0064 #00100
[SPACEBAR].[ENTER]	$0000 #00000

The examples show: (1) a conversion from binary to both hexadecimal and decimal, (2) a character constant conversion to hexadecimal and decimal ASCII, and (3) a decimal to hexadecimal conversion. The last example used indirect addressing to examine memory without changing Dot's value.

In addition, you can use indirect addressing to simulate 6809 indexed or indexed indirect instructions. The following example is the same as the assembly-language syntax [D,Y]:

You Type:	The Screen Shows:
[SPACEBAR][:D+:Y][ENTER]	$0110 *00272

Dot and Memory Examine/Change Commands

You can display the current value of Dot (the current memory address), using the DOT command. For example:

You Type:	The Screen Shows:
.	2201 B0

This shows that the present value of Dot is 2201. That memory address contains the value B0.

Incrementing Dot

You can use [ENTER] to increment the value of Dot and display its new value and contents:

You Type:	The Screen Shows:
[ENTER]	2202 05
[ENTER]	2203 C2
[ENTER]	2204 82

Decrementing Dot

Use the minus (-) key to decrement the value of Dot. As when you use the [ENTER] key, Debug displays both the new value and the contents of that address:

You Type:	The Screen Shows:
.[ENTER]	2204 82
-[ENTER]	2203 C2
-[ENTER]	2202 05

Changing Dot

You can enter an expression after the DOT command to change the value of Dot:

Debug evaluates the *expression*, and sets Dot to that value. For example:

You Type:	The Screen Shows:
. 500[ENTER]	0500 12

Debug displays the new value of Dot and its contents.

The DOT-DOT command (..) command restores Dot to its previous value:

You Type:	The Screen Shows:
.[ENTER]	0500 12
. 2000[ENTER]	2000 9C
..[ENTER]	0500 12

Changing Dot's Contents

You can change the contents of Dot with the EQUAL (=) command:

= expression

Debug evaluates *expression*, and stores the result at Dot. Debug then increments Dot and displays the next address and its contents.

The EQUAL command also checks Dot, after the new value is stored, to see that it changed to the correct value. If it did not, the screen shows an error message. This happens when you attempt to alter non-RAM memory. In particular, the registers of many 6800-family interface devices (such as PIAs and ACIAs) do not read the same as when written to.

For example:

You Type:	The Screen Shows:
.[ENTER]	2203 C2
=FF[ENTER]	2204 01
-[ENTER]	2203 FF

Note: The EQUAL command can change any memory location. Be careful when changing addresses so that you do not accidentally alter the Debug program, the program being tested, or OS-9.

Register Examine/Change Command

You can use any of several forms of the colon (:) REGISTER command to examine one or all registers or to change a specific register's contents.

The registers affected by these commands are actually images of the register values of the program under test. These values are stored on a stack when the program is not running. Although a *dummy* stack is established automatically when you start Debug, use the E command to give the register images valid data before using the G command to run the program. The *registers* are valid after breakpoints are encountered and are passed back to the program upon the next G command. (See the "Program Setup" and "GOTO Command" sections later in this chapter for information on the E and G commands.)

> **Note:** If you change the SP register, you move your stack and change register contents. In addition, Bit 7 of Register CC (the E flag) must always be set for the G command to work. If it is not set, Debug does not return to the program correctly.

This form of the REGISTER command displays the contents of a specific *register*:

: ***register***

Omitting *register* causes Debug to displays all register contents:

You Type:	The Screen Shows:
:PC[ENTER]	C499
:B[ENTER]	007E
:SP[ENTER]	42FD
:[ENTER]	PC=B265 A=01 B=0B CC=80 DP=0C SP=0CF4 X=FF0D Y=000B U=00AE

Use the following form of the REGISTER command to assign a new value to a register:

:*register expression*

Debug evaluates the *expression*, and stores the result in the specified *register*. If you specify 8-bit registers, the *expression* value must fit in one byte. Otherwise, Debug displays an error message and does not change the value of the register. Here is an example of this command:

You Type:	The Screen Shows:
:X #4096	:X #4096

Breakpoint Commands

The breakpoint capabilities of Debug let you specify addresses at which you want to suspend execution of the program under test and reenter Debug. When you encounter a breakpoint, the screen shows the values of the MPU registers and the DB: prompt. After the program reaches a breakpoint, you can examine or change registers, alter memory, and resume program execution. You can insert breakpoints at as many as 12 addresses.

The inserted breakpoints use the 6809 SWI instruction, which interrupts the program and saves its complete state on the stack. Debug automatically inserts and removes SWI instructions at the right times; so you do not *see* them in memory.

Because SWIs operate by temporarily replacing an instruction OP code, there are three restrictions on their use:

- You cannot use breakpoints in programs in ROM.
- You must position breakpoints at the first byte (OP code) of the instruction.
- You cannot use the SWI instruction in user programs for other purposes. (You can use SWI2 and SWI3.)

When you encounter the breakpoint during execution of the program under test, reenter Debug by typing : *register* [ENTER], where *register* is a mnemonic as discussed in Chapter 2. The screen shows the program's register contents.

Setting Breakpoints

Use the BREAKPOINT (B) command to insert breakpoints:

B *expression*

Debug evaluates the *expression*, and sets the breakpoint at that address. If you omit *expression*, Debug displays all present breakpoint addresses. Note in the following examples that the B . command sets a breakpoint at the address of Dot.

You Type:	The Screen Shows:
B 1C00[ENTER]	B 1C00
B 4FD3[ENTER]	B 4FD3
.[ENTER]	1277 39
B .[ENTER]	B .
B[ENTER]	1C00 4FD3 1277

Removing Breakpoints

Use the KILL (K) command to remove breakpoints:

K *expression*

Debug evaluates *expression* for the address at which to remove the breakpoint. Omitting *expression* causes Debug to remove all breakpoints. For example:

You Type:	The Screen Shows:
B[ENTER]	1C00 4FD3 1277
K 4FD3[ENTER]	
B[ENTER]	1C00 1277
K[ENTER]	
B[ENTER]	

Program Setup and Run Commands

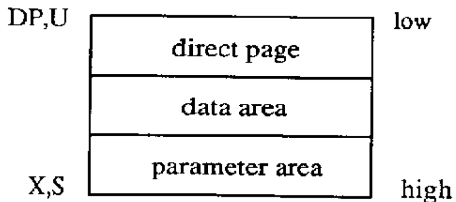
The ESTABLISH (E) command prepares Debug for testing a specific program module:

E *module-name*

This command's function is similar to that of the OS-9 Shell in starting a program. The E command does not, however, redirect I/O or override (#) memory size. The E command sets up a stack, parameters, registers, and data memory area in preparation for executing the program to be tested. The G command starts the program.

> **Note:** The E command allocates program and data area memory as appropriate. The new program uses Debug's current standard I/O paths, but can open other paths as necessary. In effect, Debug and the program become co-routines.

The E command is acknowledged by a register dump showing the program's initial register values. The G command begins program execution. The E command sets up the MPU registers as if you had just performed an F$CHAIN service request as shown in the following table:



D = *parameter area size*
PC = *module entry point absolute address*
CC = (F=0), (I=0) *interrupts disabled*

For example:

You Type:	The Screen Shows:
E myprog	**SP CC A B DP** **X Y PC** **0CF3 C8 00 01 0C** **0CFF 0D00 9214**

GOTO Command

To start (or resume) program execution, use the G command. The G command goes to (resumes) program execution after a breakpoint. If a breakpoint exists at the present program counter address, Debug does not insert that breakpoint. If you wish to suspend execution during each pass in a loop, you must insert two breakpoints in that loop.

> **Note:** Usually you use the E command before the first G command to set up the program to be tested. Debug initially sets up a default stack, so you can use G *expression* to start a program, using the results of the *expression* as a starting address.

Examples:

```
DB: G 4C00[ENTER]
DB: G :PC+100[ENTER]
DB: G [.][ENTER]
```

LINK Command

The LINK (L) command sets a link to the specified module:

L ***module-name***

If successful, LINK sets Dot to the address of the first byte of the program and displays it.

You can use L to find the starting address of an OS-9 memory module. For example:

You Type:	The Screen Shows:
L FPMATH [ENTER]	C00087

You can also use the LINK command to reset Dot to the first byte of a module:

You Type:	The Screen Shows:
L FPMATH [ENTER]	C000 87
. .+A10 [ENTER]	CA10 FF
L FPMATH [ENTER]	C000 87

Utility Commands

Clearing Memory

The CLEAR MEMORY (C) command performs a *walking bit* memory test and clears all memory between the two evaluated expressions:

C *expression1 expression2*

Expression1 specifies the starting address and *expression2* specifies the ending address, which must be higher. If any byte fails the test, the C command displays its address. You can test and clear random access memory only.

> **Note:** Use this command carefully. Be sure of the memory address you are clearing.

Some examples of this command are:

You Type:	The Screen Shows:
C . .+FF[ENTER]	
C 15FF 2000[ENTER]	**17E4** **17E7**

The first example clears all memory between the last value of Dot and Dot plus FF. Because Debug displayed a blank line (nothing), all memory tested good.

The second example indicates that there is bad memory at addresses 17E4 and 17E7.

Displaying Memory

The MEMORY command produces a screen-sized tabular display of the contents of memory in both hexadecimal and ASCII form:

M *expression1 expression2*

Expression1 specifies the starting address. *Expression2* specifies the ending address, which must be higher.

Each line's starting address displays on the left, followed by the contents of the subsequent memory locations. On the far right, Debug displays the ASCII representation of the same memory locations.

Debug substitutes periods (.) for nondisplayable characters.

Searching Memory

The SEARCH command searches an area of memory for a 1- or 2-byte pattern, beginning at Dot.

S *expression1* expression2

Expression1 specifies the ending address. *Expression2* is the data for which to search. If *expression2* is less than 256, Debug uses a 1-byte comparison. If it is greater than 256, Debug uses a 2-byte comparison.

If Debug finds a match, it sets Dot to the address at which the match occurred. If Debug does not find a match, it displays the DB: prompt.

Shell Command

To call the OS-9 shell from within Debug, use the $ command:

$ *shell-command*

This command executes the specified *shell-command* and returns to Debug. If you omit the *shell-command*, Debug calls the OS-9 Shell, which responds with prompts for one or more command lines.

You can also use the $ command to call the system utility programs and the assembler from within Debug. For example:

$DIR[ENTER]

displays the current directory.

Quitting Debug

The QUIT command lets you exit Debug and return to the OS-9 Shell. To exit Debug, type:

Q [ENTER]

The system returns you to OS-9.

> **Note:** Any modules you load using $load *module-name*,or any modules you link using L *module-name*, remain linked in memory. See the UNLINK command in the *OS-9 Level Two Operating System* manual for information about unlinking modules from memory.

Chapter 4

Using Debug

You use Debug primarily to test system memory and I/O devices, to *patch* the operating system or other programs, and to test hand-written or compiler-generated programs.

Sample Program

The simple assembly-language program shown here illustrates the use of Debug commands. This program prints HELLO WORLD and then waits for a line of input.

```
NAM  EXAMPLE

* Useful Numbers
PRGRM equ $10
OBJCT equu $01
STK                    equ 200

* Data Section
 csect
LINLEN RMB 2 LINE LENGTH
INPBUF RMB 80 LINE INPUT BUFFER
 endsect
```

```
* Program Section
 psect example,PRGRM+OBJCT,$81,0,STK,ENTRY

ENTRY EQU * MODULE ENTRY POINT
 LEAX OUTSTR,PCR OUTPUT STRING ADDRESS
 LDY #STRLEN GET STRING LENGTH
 LDA #1 STANDARD OUTPUT PATH
 os9 I$WritLn WRITE THE LINE
 BCS ERROR BRA IF ANY ERRORS
 LEAX INPBUF,U ADDRESS OF INPUT BUFFER
 LDY #80 MAX OF 80 CHARACTERS
 LDA #0 STANDARD INPUT PATH
 os9 I$ReadLn READ THE LINE
 BCS ERROR BRA IF ANY I/O ERRORS
 STY LINLEN SAVE THE LINE LENGTH
 LDB #0 RETURN WITH NO ERRORS
ERROR os9 F$Exit TERMINATE THE PROCESS

OUTSTR FCC /HELLO WORLD/ OUTPUT STRING
 FCB $0D END OF LINE CHARACTER
STRLEN EQU *-OUTSTR STRING LENGTH

 endsect End of PSect
```

Following is the listing (RMA output) for the Example program:

```
Microware OS-9 RMA - V1.1  87/03/16  17:33   example.a        Page   1
EXAMPLE -

00001                 NAM   EXAMPLE
00002
00003 * Useful Numbers
00004 0010            PRGRM  equ   $10
00005 0001            OBJCT  equ   $01
00006 00c8            STK    equ   200
00007
00008 * Data Section
00009 0000                   csect
00010 0000            LINLEN RMB  2            line length
00011 0002            INPBUF RMB  80           line input buffer
00012 0052                   endsect
00013
00014 * Program Section
00015                 psect example,PRGRM+OBJCT,$81,0,STK,ENTRY
00016
```

```
00017 0000          ENTRY  EQU  *              module entry point
00018 0000 308d0020        LEAX OUTSTR,PCR     output string address
00019 0004 108e000c        LDY  #STRLEN        get string length
00020 0008 8601            LDA  #1             standard output path
00021 000a=103f00          os9  I$WritLn       write the line
00022 000d 2512            BCS  ERROR          BRA if any errors
00023 000f 3042            LEAX INPBUF,U       address of input buffer
00024 0011 108e0050        LDY  #80            max of 80 characters
00025 0015 8600            LDA  #0             standard input path
00026 0017=103f00          os9  I$ReadLn       read the line
00027 001a 2505            BCS  ERROR          BRA if any I/O errors
00028 001c 109f00          STY  LINLEN         save the line length
00029 001f c600            LDB  #0             return with no errors
00030 0021=103f00   ERROR  os9  F$Exit         terminate the process
00031
00032 0024 48454c4c OUTSTR FCC  /HELLO WORLD/ OUTPUT STRING
00033 002f 0d              FCB  $0D            end-of-line character
00034 000c          STRLEN EQU  *-OUTSTR       string length
00035
00036 0030                 endsect             End of PSect
```

Following is the linkage map (Rlink output) for the Example program:

```
Linkage map for example File - /h0/CMDS/color/example

Section       Code IDat UDat IDpD UDpD File

example        0015 0000   0000   00  00 RELS/example.r
 dpsiz        udpd 0000
 end          udat 0000
 edata        idat  0000
 btext        code 0000
 etext        code 0045
os9defs_a      00450000 0000  00  00 ../LIB/sys.l
 I$ReadLn     cnst  008b
 I$WritLn     cnst  008c
 F$Exit       cnst  0006
               ------- ------- ------- ----
               00300000 0000  00  00
```

Note: This Psect Example has a value of $15, which is the offset from the beginning of the final module.

Following is the display created by using OS-9's DUMP command on the Example module:

```
OS9:dump /d0/cmds/example

Addr 0  1 2  3 4  5 6  7 8  9 A  B C  D E  F 0 2 4 6 8 A C E
---- ---- ---- ---- ---- ---- ---- ---- ---- ----------------
0000 87CD 0058 000D 11C1 3000 1500 C865 7861 .M.X...A0...Hexa
0010 6D70 6CE5 0030 8D00 2010 8E00 0C86 0110 mple.0.. .......
0020 3F8C 2512 3042 108E 0050 8600 103F 8B25 ?.%.0B...P...?.%
0030 0510 9F00 C600 103F 0648 454C 4C4F 2057 ....F..?.HELLO W
0040 4F52 4C44 0D00 0000 0000 0000 0065 7861 ORLD.........exa
0050 6D70 6C65 0091 A4B8                     mple..$8
```

Using Debug

Following is a sample session using the OS-9 Interactive Debugger:

First, run Debug by typing:

debug [ENTER]

The screen displays the Debug prompt DB:. To load the Example program module, type:

$load example [ENTER]

The dollar sign ($) tells Debug that you want to use an OS-9 system command and LOAD reads the example module from the current directory to your computer's memory.

You now need to tell Debug what module you want to use. Do so with the L (LINK) command. Type:

l example [ENTER]

Debug links to Example and displays the module's address:

```
C000 87
```

Redisplay the current address and its value using the DOT command. Type:

```
. [ENTER]
```

The screen shows:

```
C000 87
```

To display the contents of the entire module, use the M (display memory) command. Type:

```
m . .+57 [ENTER]
```

The screen displays:

```
C000 87CD 0058 000D 11C1 3000 1500 C865 7861 ...X....0....exa
C010 6D70 6CE5 0030 8D00 2010 8E00 0C86 0110 mpl..0.. .......
C020 3F8C 2512 3042 108E 0050 8600 103F 8B25 ?.%.0B...P...?.%
C030 0510 9F00 C600 103F 0648 454C 4C4F 2057 .......?.HELLO W
C040 4F52 4C44 0D00 0000 0000 0000 0065 7861 ORLD.........exa
C050 6D70 6C65 0091 A4B8 0000 FFFF 0000 0276 mple...........v
```

> **Note:** Psect of example program starts at an offset of $15 from the beginning linked module.

Prepare to run the Example program by typing:

```
e example [ENTER]
```

The screen displays the program's initial register values:

```
SP   CC   A    B    DP   X    Y    U    PC
2F3  A8   00   01   02   02FF 0300 0200 C015
```

To set a breakpoint at BCS ERROR, type:

```
b .+2f [ENTER]
```

Then, display the breakpoint by typing:

b [ENTER]

The screen displays:

C02F

To run the program, type:

g [ENTER]

The module displays **HELLO WORLD**. To complete the program, type a message and press **[ENTER]**, such as:

hello computer

Debug now encounters the breakpoint and displays the current register values:

```
BKPT:
  SP   CC   A    B    DP    X    Y    U    PC
02F3   A0   00   01   02   0202 000F 0200 C02F
```

You can display the module's data area by typing:

m :u :u+20 [ENTER]

The screen displays:

```
0200 D109 6865 6C6C 6F20 636F 6D70 7574 6572 ..hello computer
0210 0D86 A6A4 847F 8D06 A6A0 2AF6 8620 3410 ..........*.. 4.
0220 9E01 A780 9F01 3590 3432 860D 8DF0 304D ......5.42....0M
```

Display the relative data area at offset 2 by typing:

:u+2 [ENTER]

To step through the data area, press the **[ENTER]** one or more times. The screen displays the addresses and address values, such as:

```
0202 68
0203 65
0204 6C
0205 6C
0206 6F
```

To end the Debug session, type:

q [ENTER]

The **OS9**: prompt reappears on the screen.

Patching Programs

To *patch* a program (to change its object code), follow these steps:

1. Load the program into memory, using OS-9's LOAD command.
2. Use Debug's LINK, DOT, and EQUAL commands to link to and change the program in memory.
3. Save the new, patched version of the program on a disk file, using OS-9's SAVE command.
4. Update the program module's CRC check value, using OS-9's VERIFY command. Be sure to use the U option.
5. Set the module's execute status, using OS-9's ATTR command.

Step 4 is essential because OS-9 cannot load the patched program into memory until the program's CRC check value is updated and correct.

The example that follows shows how the sample program is patched. In this case, the **ldy #80** instruction is changed to **ldy #32**.

OS9: debug	*call Debug*
Interactive Debugger	
DB: $load example	*call OS-9 to load the program*
DB: l example	*set dot to beg addr of program*
2000 87	*actual address will vary*
DB: . .+29	*add offset of byte to change*
2029 50	*current value is 00*
DB: =#32	*change to decimal 32*
202A 86	*next byte displayed*
DB: -	*back up 1 byte*
2029 20	*(change confirmed)*
DB: q	*exit Debug*
OS9: save temp example	*save in file called "temp"*
OS9: verify U temp newex	*update CRC and copy to "newex"*
OS9: attr newex e pe	*set execution status*
OS9: del temp	*delete temporary file*

Patching OS-9 Component Modules

Patching modules that are part of OS-9 (are contained in the OS-9 Boot file) is different than patching a regular program because you must use the COBBLER and OS9GEN programs to create a new OS-9 Boot file. This example shows how an OS-9 device descriptor module is permanently patched, in this case to change the uppercase lock of the device /TERM from *on* to *off*. This example assumes that a blank, freshly formatted diskette is in Drive 1 (/D1).

> **Note:** Always use a copy of your OS-9 System Disk when patching, in case something goes wrong.

```
OS9: debug                    call Debug

Interactive Debugger
DB: l term                    set dot to addr of TERM module
  CA82 87                     actual address will vary
DB: . .+13                    add offset of byte to change
  CA95 01                     current value os 01
DB: =0                        change value to 00 for "OFF"
  CA96 01
DB: -                         move back one byte
  CA95 00                     change confirmed
DB: q                         exit Debug
OS9: COBBLER /D1              write new bootfile on /D1
OS9: VERIFY </D1/OS9BOOT >/D0/TEMP U   update CRC value
OS9: DEL /D1/OS9BOOT          delete old boot file
OS9: COPY /D0/TEMP /D1/OS9BOOT   install updated boot file
```

You can now use the DSAVE command to build a new system disk.

Chapter 5

Debug Command Summary and Error Codes

Debug Command Summary

[SPACEBAR]*expression*	Evaluate; display in hexadecimal and decimal form

Dot Commands

.	Display Dot address and contents
..	Restore last Dot address; display address and contents
. *expression*	Set Dot to result of *expression*; display address and contents
= *expression*	Set memory at Dot to result of *expression*
-	Decrement Dot; display address and contents
[ENTER]	Increment Dot; display address and contents

Register Commands

:	Display all registers' contents
:*register*	Display the specified *register*'s contents
:*register expression*	Set *register* to the result of *expression*

Program Setup and Run Commands

E *module-name*	Prepare for execution
G	Go to the program
G *expression*	Goto the program at the address specified by the result of *expression*
L *module-name*	Link to the module named; display address

Breakpoint Commands

B	Display all breakpoints
B *expression*	Set a breakpoint at the result of *expression*
K	Kill all breakpoints
K *expression*	Kill the breakpoint at address specified by *expression*

Utility Commands

M *expression1 expression2*	Display memory dump in tabular form
C *expression1 expression2*	Clear and test memory
S *expression1 expression2*	Search memory for pattern
$ *text*	Call OS-9 Shell
Q	Quit (exit) Debug

Debug Error Codes

Debug detects several types of errors, and displays a corresponding error message and code number in decimal notation. The various codes and descriptions are listed here. Error codes other than those listed are standard OS-9 error codes returned by various system calls.

0 **Illegal Constant**: The expression includes a constant that has an illegal character or that is greater than 65,535.

1 **Divide by Zero**: You are trying to use a divisor of zero.

2 **Multiplication Overflow**: The product of the multiplication is greater than 65,535.

3 **Operand Missing**: An operator is not followed by a legal operand.

4 **Right Parenthesis Missing:** Parentheses are not correctly nested.

5 **Right Bracket Missing:** Brackets are not correctly nested.

6 **Right Angle Bracket Missing:** A byte-indirect is not properly nested.

7 **Incorrect Register:** A misspelled, missing, or illegal register name follows the colon.

8 **Byte Overflow:** You are trying to store a value greater than 255 in a byte-sized destination.

9 **Command Error:** A command is misspelled, missing, or illegal.

10 **No Change:** The memory location does not match the value assigned to it.

11 **Breakpoint Table Full:** Twelve breakpoints already exist.

12 **Breakpoint Not Found:** No breakpoint exists at the address given.

13 **Illegal SWI:** Debug encountered an SWI instruction in the user program at an address other than a breakpoint.

Index

! operator 2-3
" (quotation marks) 2-2
prefix 2-1 - 2-2
$ (SHELL) command 3-12 - 3-13
$ prefix 2-1 - 2-2
% prefix 2-1 - 2-2
& operator 2-3
. (DOT) command 2-2, 3-2 - 3-4, 4-5
.. (Dot-Dot) command 2-2, 2-3
* operator 2-3
+ operator 2-3
- operator 2-3
/ operator 2-3
:(REGISTER) command 3-5 - 3-6
: (register names) 2-3

accumulator 2-3
addition 2-3
addresses, specifying 3-6 - 3-8
addressing, indirect 2-4
ASCII codes 2-2
ASCII conversion 3-2

B (BREAKPOINT) command 3-7, 4-5 - 4-6
binary conversion 3-2
breakpoints 3-6 - 3-8

C (CLEAR MEMORY) command 3-11
changing register contents 3-5 - 3-6
character constants 2-2
clearing memory 3-11

COBBLER 4-8 - 4-9
codes, ASCII 2-2
colon (REGISTER) command 3-5 - 3-6
command line 1-2
commands
  $ (SHELL) 3-12 - 3-13
  B (BREAKPOINT) 3-7, 4-5 - 4-6
  C (CLEAR MEMORY) 3-11
  DOT 2-2 - 2-3, 4-5
  DOT-DOT 2-2, 3-3
  E (ESTABLISH) 3-5, 3-8-3-10, 4-5
  G (GOTO) 3-5, 3-9 - 3-10
  K (KILL) 3-8
  L (LINK) 3-10, 4-4 - 4-5
  M (MEMORY) 4-5 - 4-6
  OS-9 3-12 - 3-13
  REGISTER 3-5 - 3-6
  S (SEARCH) 3-12
content of registers 3-5
converting values 3-1
current working address 2-2

Debug prompt 1-1
decimal
  conversion 3-2
  notation 1-2
deleting a line 1-1
diagnosing programs 1-1
displaying memory 3-12
division 2-3
Dot 2-2
Dot-Dot 2-2
DSAVE 4-9
E (ESTABLISH) command 3-5, 3-8, 3-10, 4-5
ending a debug session 3-13
examining registers 3-5 - 3-6

execution
  of programs 3-9 - 3-10
  testing 3-8 - 3-9
exiting a debug session 3-13
expressions 2-3, 3-1
  displaying 1-2

G (GOTO) command 3-5, 3-9 - 3-10

hexadecimal
  conversion 3-2
  notation 1-2
indirect addressing 2-4
inserting breakpoints 3-6 - 3-7
integers 2-1
integral expression interpreter - 2-1

K (KILL) command 3-8
L (LINK) command 3-10, 4-4 - 4-5
line deleting 1-1
loading a program module 4-4
logical operators 2-3

M (MEMORY) command 4-5 - 4-6
memory
  clearing 3-11
  displaying 3-12
  searching 3-12
  testing 3-11
modes, indirect addressing 2-4
modules
  linking 3-10, 4-4 - 4-5
  loading 4-4
  patching 4-8 - 4-9
multiplication 2-3

negative arithmetic 2-1
negative numbers 2-3
notation (hexadecimal and decimal) 1-2

operands 2-1
operators 2-3
OS-9 Shell 3-12 - 3-13
OS9GEN 4-8

programs
  executing 3-9 - 3-10
  loading modules 4-4
  patching 4-7 - 4-9

quitting debug 3-13
quotation marks 2-2

REGISTER command 3-5 - 3-6
registers 2-3
  examining 3-5 - 3-6
resuming execution 3-9 - 3-10

S (SEARCH) command 3-12
sample session 4-4 - 4-9
searching memory 3-12
shell command 3-12 - 3-13
software interrupts 3-6 - 3-7
SP register 3-5
spaces 2-1
starting program execution 3-9 - 3-10
starting
  address 3-10
  Debug 1-1
subtraction 2-3
suspending execution 3-6 - 3-8

test
  execution 3-8 - 3-9
  memory 3-11
two's complement 2-1

value, assigning to a register 3-6
values, converting 3-1

working address 2-2

Screen Editor

Contents

Chapter 1 / Introduction 1-1
- Modes Of Operation 1-1
- Starting Scred 1-2
- Available Options 1-2

Chapter 2 / The Termset File 2-1
- Modifying the Termset File 2-1
  - The Termset File Format 2-2
  - Termset Fields 2-2

Chapter 3 / Command Mode 3-1
- Changing to the Edit Mode 3-1
- Changing to the Insert Mode 3-2
- Manipulating the Edit Buffer 3-3
  - Saving Text 3-3
  - Removing Text 3-4
  - Searching for Strings 3-5
  - Changing Strings 3-5
  - Using Wild Cards 3-6
  - Miscellaneous Commands 3-6
  - Exiting Scred 3-8

Chapter 4 / Edit Mode 4-1
- Getting Help 4-1
- Controlling the Cursor 4-2
- Scrolling the Screen 4-2
- Moving to a Specific Line 4-3
- Finding a String 4-3
- Replacing Strings 4-4
- Deleting Text 4-4
- Inserting or Replacing a Single Character 4-5
- Cutting and Pasting 4-5

Editing Lines 4-6
Displaying the Status Line 4-7

Chapter 5 / Insert Mode 5-1

Chapter 6 / Quick Reference 6-1
Command Mode 6-1
Edit Mode 6-3
Cursor Movement Commands 6-3
Cut and Paste Commands 6-5
Insert Mode 6-6

Chapter 1

Introduction

The OS-9 Level Two Screen Editor (Scred) is a powerful and simple to learn screen-oriented text editor. You can use Scred to prepare text for letters and documents or text to be used by other OS-9 programs such as the assembler and high level languages. Scred's features include:

- Adjustable screen and workspace size
- Continuously updated screen
- Cursor positioning by characters, words, and line-by-line
- Scrolling
- Cut and paste
- Change, find, and search strings
- Wild cards

Modes of Operation

Scred has three modes of operation: Command, Edit, and Insert. The Command Mode lets you execute Scred commands that affect files or the edit buffer. Scred starts up in Command Mode. The Edit Mode lets you modify or manipulate text within the edit buffer. The Insert Mode lets you enter new text into the edit buffer.

Starting Scred

To start Scred, type:

scred *filename* [ENTER]

If the file exists, Scred loads the file into the edit buffer, displays the beginning of the file, and enters Edit Mode.

If the file does not exist, Scred displays:

can't open *filename*
ERROR #216

and enters the Command Mode.

If you want to create a new file, type:

scred [ENTER]

This starts Scred in Command Mode, from which you can load a file or begin creating a new one by using the NEW command (see Chapter 3).

> **Note:** Scred uses a special file called *termset* to describe the attributes of a particular terminal. See Chapter 2, "The Termset File," for more information on this file.

Available Options

You can use several options on the command line when starting up Scred. These options specify the terminal type, buffer size, and so on. Use the following form when starting Scred with options:

scred *filename options* [ENTER]

The available options are:

-? Displays a list of the Scred options.

-b= *num*k Allocates *num*k bytes of memory for Scred's working buffer. The buffer's default size is 12 kilobytes. The "=" and "k" are optional parameters. For example, **-b32** is the same as **-b=32k**.

-e Configures Scred for terminals that have embedded video attributes, that is, terminals in which the attribute start flag uses one character position.

-g Configures Scred for special graphic-oriented terminals (terminals that do not support line feeds).

-l=*num* Specifies the number of lines to be displayed on the terminal screen. You can also set this option in the *termset* file. See Chapter 2, "The Termset File," for more information.

-t=*term* Specifies the terminal type. Use this option if your terminal type is different from the default terminal type as set in the *termset* file. See Chapter 2, "The Termset File," for more information.

-w=*num* Specifies the maximum number of characters per line to be displayed on the terminal screen. You can also set this option in the *termset* file. See Chapter 2, "The Termset File," for more information.

-z=*path* Sets the pathlist that Scred uses to find the *termset* file. See Chapter 2, "The Termset File," for more information.

Note: Since Scred normally checks the current window size, the -l and -w options are not often needed. If you use them, be certain you give valid values. Otherwise these options can interfere with screen formatting.

Examples

```
scred file1 -b=32k
```

This command starts up Scred with a 32k byte buffer.

```
scred file1 -l=24 -w=30
```

This command starts up Scred with a screen size of 24 lines by 30 characters.

Chapter 2

The Termset File

To operate properly, Scred must know the type of terminal you are using. Scred finds this information in a file named Termset. The Termset is a text file containing entries that describe a variety of terminals. The terminal types currently supported in Termset are:

- COCO (the default for windows)
- KT7
- VDG (for VDG screen)
- ANSI
- ABM85
- ABM85H

If you using other than the Coco terminal, use the -t option and specify the terminal name when starting Scred. If your terminal type is not currently supported in the Termset file, read the rest of this chapter for instructions to add your terminal to the file.

Scred looks for the Termset file in the directory /dd/sys, where dd is the default device for your system. If Scred doesn't find the file there, it looks in /h0/sys and then in /d0/sys. You can use the -z option of Scred to specify a different path for the Termset file.

Modifying the Termset File

To add a new terminal type to the Termset file, you can:

- Edit the Termset file using a text editor
- Use the Maketerm supplied on the Scred distribution diskette

Because Makefile is easier to use, it is the method shown in this chapter's examples.

The Termset File Format

The Termset file contains control code definitions for one or more types of terminals. Each text line in the file is a complete description list for a particular kind of terminal.

The first line of the Termset file contains the name and control code definitions for the default terminal type. This is the terminal type Scred uses if you do not use the -t option. The form is:

NAME:ccc:cov:dl:dc:cs:cel:il:sav:eav:sl:sw

Each field represents a different control code definition. Notice that each field is separated by a colon (:). Even if the terminal cannot perform a certain function, the colon must still be present to hold the function's position.

Termset Fields

The following list defines each field in a terminal type entry:

NAME **Terminal Name**
Specifies the identification name of the terminal described in the line. Use this name with the -t option to specify the terminal type for Scred to use. You must specify the name in all uppercase, although you can specify lowercase with the -t option on Scred's command line.

ccc **Cursor Control Code**
Positions the cursor to any location on the screen. This function is required. There are two parts to the Cursor Control Code : (1) one or more *position cursor* command characters, and (2) cursor coordinates. \X and \Y (or \X\X and \Y\Y) are cursor coordinates where X and Y refer to the column number and row number, respectively. The order in which you specify the cursor coordinates is dependent on your terminal's requirements.

This information should be supplied with the hardware specifications that come with your terminal.

Examples:

$1b[\Y\Y;\X\XH:
$1b$3d\Y\X:
$1bR\X\Y:

In the first example, the bracket character ([) has an ASCII value of $5B. You could use $5B in place of [to produce the same results.

cov **Cursor Offset Value**
Sets the offset value for the cursor coordinates. This value, specified in hexadecimal, is always added to the cursor X and Y coordinates. Many terminals use an offset of $20.

dl **Delete Line Control Character(s)**
Deletes the current line and causes lines below the deleted line to scroll up.

dc **Delete Character Control Character(s)**
Deletes the character under the cursor and shifts the remaining characters on the line to the left by one character position.

cs **Clear Screen**
Erases the entire screen, and returns the cursor to the home position.

cel **Clear to End of Line**
Erases all characters on the line from the current cursor position to the end of the line, including the character under the cursor.

il **Insert Line**
Creates a new blank line by scrolling the current and subsequent lines down one line.

sav **Start Alternate Video**
Displays all subsequent characters in reverse video, different intensity, or any similar mode that is visibly different from the normal video mode. This code is used when highlighting text.

eav **End Alternate Video**
Displays all subsequent characters in normal video mode.

You can specify 0-4 output control characters for the following fields: Delete Line, Delete Character, Clear Screen, Clear to End of Line, Insert Line, Start Alternate Video, and End Alternate Video.

sl **Screen Length**
Specifies, in hexadecimal, the number of lines to be displayed on the terminal screen. This field is optional. If you omit this value, Scred uses 24.

sw **Screen Width**
Specifies, in hexadecimal, the number of columns to be displayed on the terminal screen. This field is optional. If you omit this value, Scred uses 80.

Screen length and screen width are optional fields. If you omit them, Scred checks the size of the current screen (or part of the screen) and uses these values. For external terminals, Scred assumes a screen size of 24 lines by 80 columns. If you do specify a length and width, Scred uses these values and does not check on the size of the current screen.

Examples

Example 1

Create the following Termset entry:

ABM85:$1b$3d\eY\eX:$20:$1bR:$1bW:$1e$1bY:$1bT:$1bE:$1bj:$1bk:$18 :$50:

To create the above entry, type the following at the system prompt ($):

maketerm [ENTER]

The Maketerm utility prompts you to supply a value for each field in the Termset entry. If a Termset file does not exist, Maketerm creates it. If the file does exist, Maketerm appends the new entry to the end of the Termset file.

> **Note:** If a particular terminal does not have one of the requested features, simply press **[ENTER]** at the prompt.

Following are the prompts displayed by Maketerm and the responses needed to create the ABM85 entry:

terminal name: **ABM85 [ENTER]**
cursor positioning sequence: **$1b$3d\eY\eX [ENTER]**
cursor position offset: **$20 [ENTER]**
delete line sequence: **$1bR [ENTER]**
delete character sequence: **$1bW [ENTER]**
clear screen: **$1e$1bY [ENTER]**
clear to end of line: **$1bT [ENTER]**
insert line: **$1bE [ENTER]**
alternate video: **$1bj [ENTER]**
restore normal video: **$1bk [ENTER]**
screen length: **$18 [ENTER]**
screen width: **$50 [ENTER]**

Example 2

To create the following Termset entry:

TERM:$1bR\X\Y:$00:::$0e:::$1bj:$1bl:::

Type **maketerm [ENTER]**. The prompts and responses look like this:

terminal name: **TERM [ENTER]**
cursor positioning sequence: **$1bR\X\Y [ENTER]**
cursor position offset: **$00 [ENTER]**
delete line sequence: **[ENTER]**
delete character sequence: **[ENTER]**
clear screen: **$0e [ENTER]**
clear to end of line: **[ENTER]**
insert line: **[ENTER]**
alternate video: **$1bj [ENTER]**
restore normal video: **$1bl [ENTER]**
screen length: **[ENTER]**
screen width: **[ENTER]**

Chapter 3

Command Mode

The Command Mode lets you invoke commands that affect files or manipulate the entire edit buffer. Scred starts up in Command Mode if you do not specify a file on the command line. When you are in the Command Mode, Scred displays the > prompt in the lower left corner of the display screen.

Command Mode commands (except the GOTO command) are at least two characters long to distinguish them from the Edit and Insert Mode commands. You can use either the full name for the command, such as **edit**, or Scred's shortened form, **ed**. Commands that have short forms are shown as follows:

ed[it]

This means you can type either **ed** or **edit** for the EDIT command. Do not type the square brackets.

When entering commands in Command Mode, you can use the standard OS-9 control keys to backspace, delete lines and characters, and so on. Press **[ENTER]** after typing each command.

Changing to the Edit Mode

There are two methods in which you can enter Edit Mode from Command Mode:

1. Edit an existing file by typing at the > prompt:

   ol[d] ***filename*** **[ENTER]**

   If Scred can open the file, it then enters the Edit Mode.

2. If you have a file open and want to enter the Edit Mode, type:

   ed[it] [ENTER]

   You can also press **[CTRL][E]** to enter the Edit Mode.

   From the Edit Mode, you can change to the Command Mode by pressing **[CTRL][BREAK]**

Changing to the Insert Mode

You can enter Insert Mode from Command Mode by typing:

in[sert] [ENTER]

Create a new file by typing at the > prompt:

ne[w] ***filename*** **[ENTER]**

If Scred can create the file, it loads the file into the edit buffer and then enters the Edit Mode.

You can enter the Insert Mode from the Edit Mode by: (1) pressing **[ENTER]** to insert text before the cursor position, and (2) pressing the down arrow to insert a new line before the current line. You can then begin typing the new line.

Note: You cannot enter the Insert or Edit Modes if no file exists in the edit buffer.

Manipulating the Edit Buffer

Scred's edit buffer size is 12k bytes unless you use the -b option to specify a different value. If your file is larger than the edit buffer, Scred loads as much of the file as it can, while leaving approximately 2k free for changes and additions. With the 12k buffer size, Scred loads 10k of the file. The following commands show how to write, read, and insert files or sections of files.

Saving Text

The WRITE command writes the contents of the edit buffer and the remainder of the input file (if any) to the output file. WRITE then closes the file and clears the edit buffer. To write a file, type:

wr[ite] [ENTER]

When Scred saves a file, it creates an output file called Ed.tmp.*xxx*, where *xxx* is the process id number. If Scred can successfully create and write the entire output file, it deletes the current input file and renames the output file to the old name.

The UPDATE command writes out the changes you made to the edit buffer and re-enters the Edit Mode. To update a file, type:

up[date] [ENTER]

The ADD command lets you insert a specified file within the text of the edit buffer. Scred inserts the file directly before the current line. To add a file before the current line, type:

ad[d] *filename* [ENTER]

Note: There must be enough free space in the edit buffer for the extra text. If Scred runs out of space, it terminates with the message **file too large to add** and does not load any of the file.

The MORE command lets you read in the next section of the input file. Use this command when the file you are editing is too large to entirely fit in the edit buffer. The MORE command causes Scred to write the contents of the edit buffer between the top of the buffer and the current cursor position to the output file and read the next section of the input file into the edit buffer. To read the next section of a file, type

mo[re] [ENTER]

Removing Text

Scred lets you delete specified lines of text from the edit buffer or delete the entire buffer.

The DELETE command lets you delete specified lines from the edit buffer. To delete lines, type:

de[lete] *start-line end-line* [ENTER]

This command deletes text from *start-line* to *end-line*, inclusive.

The ABORT command erases the entire contents of the edit buffer and closes the file. To erase and close a file, type:

ab[ort] [ENTER]

The CLEAR command also erases the entire contents of the edit buffer but the file remains open. To clear the edit buffer, type:

cl[ear] [ENTER]

Searching for Strings

The FIND command prompts you to enter a *search mask* and then searches for that string. If Scred finds the string, it positions the cursor at the beginning of the first occurrence of the string and then enters Edit Mode. To find a string, type:

fi[nd] [ENTER]

The SEARCH command prompts you to enter a search mask and then searches for that string. In addition, SEARCH lets you search for that string between specified lines instead of through the entire file. If Scred finds the string, it displays the lines, including the line number, in which the string was found. To search for a string, type:

se[arch] *start-line end-line* [ENTER]

This command searches for the string beginning at *start-line* through *end-line*, inclusive. If you omit *start-line* and *end-line*, Scred searches the entire edit buffer.

Note: The SEARCH and FIND commands accept a *match first word only* character. By placing a ^ as the first character in the search string, Scred finds a match only if it finds the string at the beginning of the line.

Changing Strings

The CHANGE command replaces all occurrences of a string within the specified range of lines or over the entire edit buffer. To use the CHANGE command, type:

ch[ange] *start-line end-line* [ENTER]

If you omit *start-line* and *end-line*, Scred searches the entire edit buffer.

When you invoke the CHANGE command, Scred prompts you to enter a **Search mask:**. Enter the string you want to change. Scred then prompts you to enter a **Change mask:**. Enter the new string.

If Scred finds the search string, it displays the lines, including the line numbers, in which the changes occurred.

> **Note:** The CHANGE command accepts a *match first word only* character. By placing a ^ as the first character in the search string, Scred finds a match only if it finds the string at the beginning of the line.

Using Wild Cards

When entering the search string for the FIND, SEARCH and CHANGE commands, you can optionally use the wild card character "?". The wild card character matches any one character in the specified location. For example:

m????? [ENTER]

Scred matches all strings that begin with the letter "m" and are followed by five characters. Sample strings that would match are: "millio," "mister," and "my dog."

??_?? [ENTER]

In this example, Scred matches all five character strings with an underscore character (_) in the third character position. Some sample strings that match this string are: "SS_ID," "WA_86," and " _dj."

> **Note:** Scred matches spaces between words when searching for a wild card string.

Miscellaneous Commands

The GOTO command positions the cursor on a specified line and enters Edit Mode. To position the cursor, type:

g[oto] ***line-number*** **[ENTER]**

The CHD command changes the current working directory to the specified directory. You can specify either a relative or absolute path to the new directory. To change directories, type:

chd ***pathname*** **[ENTER]**

The DIR command displays the directory listing for the current directory. To obtain a listing, type:

dir [ENTER]

Scred can handle files with tabs in them. However, tabs are not a function of Scred. The TABS command lets you set tab stops at each *n* characters. To set the tab stops, type:

ta[bs] ***n*** **[ENTER]**

Scred sets tabs at every four characters by default.

Another feature of Scred is auto-indent. If you enter an indented line, Scred automatically aligns the next line with it.

The NOTAB command turns off the auto-indent function. To disable the auto-indent feature, type:

not[ab] [ENTER]

The AUTO INDENT command turns the feature back on. To enable the auto-indent feature, type:

au[to indent] [ENTER]

The $ command lets you execute a shell command line from within Scred. To execute an OS-9 command, type:

$*command-line* [ENTER]

For example, to list the contents of a file, type:

$list *filename* [ENTER]

When you use the SHELL command (**$ [ENTER]**), OS-9 starts a new shell (if your computer has enough free memory). In this way it can process several OS-9 commands. To return to the Scred > prompt, press **[CTRL][BREAK]**.

Exiting Scred

The EXIT command ends the current editing session. If a file exists, Scred saves the file to disk and returns to the OS-9 system. To exit Scred, type:

ex[it] [ENTER]

Chapter 4

Edit Mode

The Edit Mode lets you control and modify text in the edit buffer and on the screen display. You can enter Edit Mode from Command Mode by typing **ed [ENTER]** or by pressing **[CTRL][E]**. You can enter Edit Mode from Insert Mode by pressing **[CTRL][BREAK]**. When you enter Edit Mode, Scred displays the text of the file being edited.

Commands in this chapter, appear in uppercase as they appear on your keyboard. Unless specifically noted, you do not have to press **[SHIFT]** to invoke the commands.

Getting Help

You can display help information at any time while in Edit Mode. To do so, press **?**. Scred displays a list of commands at the top of the screen. The commands are divided into four groups:

- Cursor control keys
- Edit buffer controls
- CUT and PASTE commands
- Miscellaneous commands

Press the spacebar to review the display for each group. Press **q** to exit the help function.

Controlling the Cursor

The following table lists the keys Scred uses to position the cursor. When looking at this table, notice that the location of each key on the keyboard is related to the movement it performs.

Key	Action
I	moves the cursor up one line
, (comma)	moves the cursor down one line.
J	moves the cursor left one character
L	moves the cursor right one character
K	moves the cursor alternately to the beginning or end of the current line
H	moves the cursor one word to the left
;	moves the cursor one word to the right

Scrolling the Screen

Scred uses four keys to scroll the screen. The table below lists the keys and their descriptions. As before, notice the location of the keys on your keyboard.

Key	Action
U	scrolls the screen up continuously
M	scrolls the screen down continuously
O	scrolls the screen up
.	scrolls the screen down

The continuous scroll feature is useful when you want to quickly scan through a file. Use the space bar to pause and restart scrolling. Type any other character to terminate scrolling.

When scrolling down one screenful, the line at the bottom of the screen scrolls to the top of the screen. When scrolling up one screenful, the line at the top of the screen scrolls to the bottom of the screen.

Moving to a Specific Line

The GOTO command moves the cursor to the specified line within the edit buffer. To move the cursor to a specific line, press **G**. Scred prompts you to enter the line number with the prompt **goto:**. Enter the line number to which you want to move the cursor. Scred positions the cursor at the beginning of the specified line and positions that line on the third line of the screen.

Line 1 is the first line of the edit buffer. Any number higher than the last line number causes the last line to be selected.

Finding a String

The FIND command searches for a specified string and positions the cursor on the first character of that string. To invoke FIND, press **F**. Scred prompts you to enter a **Search mask:**. Type the string you want to find. If Scred finds the string, it positions the cursor on the first character of the string and positions the line in which the string occurred on the third line of the screen. If Scred cannot find the string, it displays the message, **find: no match**.

To find another occurrence of the same string, press **F** and press **[ENTER]** for the search mask. Scred moves the cursor to the next occurrence of the previously entered string.

Replacing Strings

The REPLACE command lets you substitute one string for another. To replace a string, press R [ENTER] and Scred prompts you to enter a Search string:. Enter the string you want to replace. Scred then prompts you to enter the Change string:. Enter the new string.

To replace the next occurrence of the search string with the same string, press R and press [ENTER] for both prompts.

Deleting Text

Scred offers a variety of ways to delete text. You can delete characters, words, and lines. The following table summarizes the key commands and their definitions.

Key	Action
[←]	deletes the character to the left of the cursor
[CTRL][;]	deletes the character under the cursor
[CTRL][A]	deletes one word to the left of the cursor
[CTRL][D]	deletes one word to the right of the cursor
[CTRL][C]	deletes from the current cursor position to the end of the line
[CTRL][Z]	deletes from the current cursor position to the beginning of the line
[CTRL][X]	deletes the current line

Note: If you accidentally delete text, you can recover by pressing [CTRL][F]. The [CTRL][F] command restores the current line to its original state.

Inserting or Replacing a Single Character

Scred easily lets you insert one character or substitute one character with another without having to enter Insert Mode.

The REPLACE CHARACTER command replaces the character under the cursor. To replace a character, type **X*character***. For example, typing **Xz** replaces the character under the cursor with a "z."

The INSERT CHARACTER command inserts a character in front of the character under the cursor. To insert a character, type **B*character***. For example, typing **Ba** inserts an "a" in front of the character under the cursor.

Cutting and Pasting

Scred's *cut and paste* feature lets you move a block of text and insert it at another location. Scred lets you move, delete, or duplicate blocks of text.

Before you move a block of text, you must mark the beginning point of the block. The SET command marks the starting line. To mark a line, move the cursor to the first line of the block of text you want to move, and press **S**. To mark in the middle of a line, first break the line into two lines, and then mark it. Scred displays the marked line in reverse video if your terminal has the capability.

Next, move the cursor to the last line of the text block you want to move. Use the CUT command to remove the text from the edit buffer. Scred places the text in its *paste* buffer.

You can add more text to the paste buffer by using the APPEND command. To use the APPEND command, mark the beginning of the text block using SET, and move the cursor to the end of the block. Press **A**, and Scred appends the text block to the text already in the paste buffer.

Use the PASTE command to return the contents of the paste buffer to the edit buffer. Scred pastes text on the line above the current line. Therefore, to paste the text, position the cursor one line below the line on which you want the text inserted, and press **P**.

You can also duplicate text by using the NON-DESTRUCTIVE CUT command. To do so, mark the beginning of the text block using the SET command and move the cursor to the last line of the text to be duplicated. Press **N** and Scred copies the text block into the paste buffer. The text in the edit buffer is untouched.

Scred also offers a NON-DESTRUCTIVE APPEND command. Mark the beginning of the text block (SET), and move the cursor to the last line of the text to duplicate. Press **V**, and Scred appends a copy of the text to the end of the paste buffer. The text in the edit buffer is untouched.

The ERASE command clears the paste buffer and returns its memory. Press **E** to erase.

Scred also lets you write sections of text to a file using the WRITE command. To do so, mark the beginning of the block (SET), and move the cursor to the last line of the block. Press **P**. Scred prompts you to enter an output filename. If you invoke the WRITE command without marking a text block, Scred writes the paste buffer to the output file. If Scred cannot create the file, it issues an error message.

Editing Lines

Scred allows you to use lines of up to 256 characters in length. However, because Scred does not wrap lines, you can see only a portion of the line if it is longer than the width of your screen. Scred offers an easy method of breaking and joining lines.

The BREAK command splits the line at the current cursor position. Scred inserts the break before the cursor. To break a line, press **[CTRL][B]**.

The JOIN command joins the current line with the one above. To join two lines, press **[CTRL][P]**.

Displaying the Status Line

The status line displays the line number, column number, amount of free space in the edit buffer, paste buffer size, current filename, and the current mode (Command, Edit, or Insert). To display the status line, press **[CTRL][G]**. Press the space bar to remove the status line from the screen.

The following sample status line shows the current cursor position to be Line 50, Column 0. There is more than 14k bytes free in the edit buffer and 51 bytes of text stored in the paste buffer. The filename is *Example*, and Scred is in the Edit Mode.

```
L:50  C:0  MB:14526  CB:51  F:Example  edit:
```

Chapter 5

Insert Mode

The Insert Mode lets you enter new text into the edit buffer. To enter the Insert Mode from the Command Mode, type **in [ENTER]**. To enter the Insert Mode from the Edit Mode, press **[ENTER]** or **[↑]**.

Scred inserts the new text before the current cursor position and stores it exactly as you type it. You can enter control characters. To enter control characters, press **[CTRL][V]** followed by the character you wish to enter. For example, to enter a Control-L into the edit buffer, press **[CTRL][V]**, then **[L]**.

Chapter 6

Quick Reference

The following tables provide a quick reference to the commands for the Command, Edit, and Insert Modes.

Command Mode

Command	Description
ab[ort]	Cancels all changes made to the current file, erases the entire edit buffer, and closes the current file.
ad[d] *filename*	Adds the text of the specified file to the edit buffer, starting at the line above the current cursor position.
au[to indent]	Tells Scred to automatically indent the next line after a carriage return in the previous line begun with a tab or space(s). Scred indents the new line to the same column position as the previous line. Scred starts up in auto-indent mode.
ch[ange][*start-line* [*end-line*]]	Replaces all occurrences of a string within the specified range of lines. Omitting a range value causes Scred searches the entire edit buffer.

Command	Description
chd *pathname*	Changes the current working directory.
cl[ear]	Erases all text in the edit buffer. Scred does not close the file.
de[lete] [*start-line* [*end-line*]]	Erases the specified range of lines from the edit buffer.
dir	Displays the directory listing for the current working directory.
ed[it]	Enters Edit Mode. You can also use [CTRL][E].
ex[it]	Writes the edit buffer to the output file and exits Scred.
fi[nd]	Searches for the first occurrence of a string. Enters the Edit Mode.
g[oto] *line*	Moves the cursor to the specified line number. Enters the Edit Mode.
in[sert]	Enters Insert Mode.
mo[re]	Saves the text in the edit buffer to the output file and reads in the next section of the input file.
ne[w] *filename*	Creates a new file with the specified filename and enters Insert Mode.
not[ab]	Turns off the auto-indent mode.
ol[d]	Clears the edit buffer, opens an existing file, and enters Edit Mode.

Command	Description
se[arch] [*start-line* [*end-line*]]	Searches for a string within the specified lines. If you omit the line numbers, Scred searches the entire edit buffer.
ta[bs] *n*	Sets the tab stops to every *n* characters.
up[date]	Writes changes to the output file and re-enters Edit Mode.
wr[ite]	Writes the contents of the edit buffer and the remainder of the input file, if any, to the output file.
$ [*command*]	Executes a shell command line.
[CTRL][G]	Displays the status line.

Edit Mode

Cursor Movement Commands

Command	Description
I	Moves the cursor up one line.
, (comma)	Moves the cursor down one line.
J	Moves the cursor left one character.
H	Moves the cursor left one word.
L	Moves the cursor right one character.
;	Moves the cursor right one word.
K	Moves the cursor to the beginning or end of the line.
R	Replaces a string.

Command	Description
U	Scrolls the text up. Press the space bar to stop and start. Press any other key to abandon.
M	Scrolls the text down. Press the space bar to stop and start. Press any other key to abandon.
O	Scrolls text up on page.
.	Scrolls text down one page.
G	Moves the cursor to the specified line.
F	Finds the first occurrence of a string.
X *char*	Replaces the character under the cursor with the specified character.
B *char*	Inserts the specified character before the cursor and advances the cursor.
[←]	Deletes the character to the left of the cursor.
[CTRL][;]	Deletes the character under the cursor.
[ENTER]	Enters Insert Mode.
[↓]	Moves the text in the edit buffer down one line and enters Insert Mode with the cursor on the new line.
[CTRL][BREAK]	Returns to Command Mode.
?	Displays help information.
[CTRL][A]	Erases one word to the left of the cursor.
[CTRL][D]	Erases one word to the right of the cursor.
[CTRL][F]	Cancels any changes made to the current line.

Command	Description
[CTRL][C]	Erases text from the cursor to the end of the line.
[CTRL][Z]	Erases text from the cursor to the beginning of the line.
[CTRL][X]	Erases the entire line.
[CTRL][B]	Splits the current line into two lines at the cursor position.
[CTRL][P]	Joins the current line with the line above.
[CTRL][G]	Displays the status line.

Cut and Paste Commands

Command	Description
S	**Set.** Marks the first line of a text block to be deleted, duplicated, or moved. If the starting mark is already set, **s** removes the mark.
C	**Cut.** Deletes the selected block of text from the edit buffer and stores it in the paste buffer.
N	**Non-destructive Cut.** Places the selected block of text in the paste buffer without altering the edit buffer.
P	**Paste.** Inserts the contents of the paste buffer at the line above the cursor.
A	**Append.** Deletes the specified block of text from the edit buffer and adds it to the end of the paste buffer.

Command	Description
V	**Non-destructive Append.** Appends the specified block of text to the paste buffer without altering the edit buffer.
E	**Erase.** Erases the content of the paste buffer and releases its memory space to the edit buffer.
W	**Write.** Writes the specified lines to the output file. If no lines are marked, Scred writes the paste buffer to the output file.

Insert Mode

Command	Description
[CTRL][V]*char*	Inserts the specified control character into the edit buffer.
[CTRL][BREAK]	Returns to Edit Mode.

Index

$ command 3-8
-? (list options) 1-3
-b (buffer size) 1-3, 3-3
-e (embedded video attributes) 1-3
-g (graphics terminals) 1-3
-l (display lines) 1-3
-t (terminal type) 1-3
-w (text width) 1-3
-z (termset path) 1-3, 2-1
> prompt 3-1
? (display help) 4-1
? (wild card) 3-6

ABM85 terminal 2-1
ABM85H terminal 2-1
ABORT command 3-4
ADD command 3-3
adding terminal type 2-1
allocating memory 1-3
ANSI terminal 2-1
APPEND command 4-5 - 4-6
auto-indent 3-7 - 3-8

BREAK command 4-7
buffer size 1-3, 3-3

CHANGE command 3-5 - 3-6
character
  deleting 4-4
  inserting 4-5
characters per line 1-3

CHD command 3-7
CLEAR command 3-4
clear screen code 2-3
clearing to end of line code 2-3
closing a file3-4
COCO terminal 2-1
command mode 1-1 - 1-2
commands 3-1
  $ 3-8
  ABORT 3-4
  ADD 3-3
  APPEND 4-5 - 4-6
  BREAK 4-7
  CHANGE 3-5 - 3-6
  CHD 3-7
  CLEAR 3-4
  CUT 4-5 - 4-6
  DELETE 3-4
  ERASE 4-6
  FIND 3-5, 4-3
  GOTO 3-7, 4-3
  INSERT 4-5
  JOIN 4-7
  MORE 3-4
  NEW 1-2
  NOTAB 3-7 - 3-8
  REPLACE 4-4 - 4-5
  SEARCH 3-5
  SET 4-5 - 4-6
  TABS 3-7
  UPDATE 3-3
  WRITE 3-3, 4-6
configuring terminals 1-3
creating a file 1-2, 3-2
cursor control 4-2
cursor control code 2-2 - 2-3
cursor offset value 2-3

cursor, moving 4-3
CUT command 4-5 - 4-6
cutting and pasting 4-5 - 4-6

DELETE command 3-4
deleting
  character codes 2-3
  line codes 2-3
  text 3-4, 4-4
directories
  changing 3-7
  listing 3-7
duplicating text 4-5 - 4-6

edit buffer, erasing 3-4
Edit mode 1-1, 3-1 - 3-2, 4-1 - 4-7
ERASE command 4-6
erasing the edit buffer 3-4
exiting Scred 3-8

features 1-1
file
  closing 3-4
  creating 1-2
  reading in 3-4
  saving 3-3
FIND command 3-5, 4-3

GOTO command 3-7, 4-3
graphics 1-3

help 4-1

indent, auto 3-7 - 3-8
INSERT command 4-5
Insert Line code 2-4

insert mode 1-1, 3-2, 5-1
  characters 4-5
  files 3-3

JOIN command 4-7

KT7 terminal 2-1

line character width 1-3
lines
  deleting 3-4
  moving to 4-3
listing
  directories 3-7
  options 1-3

Maketerm 2-1, 2-5 - 2-6
mask, search 3-5 - 3-6, 4-3
memory allocation 1-3
modes
  Command 1-1 - 1-2
  Edit 1-1, 3-1 - 3-2, 4-1 - 4-7
  Insert 1-1, 3-2, 5-1
MORE command 3-4

name of terminal 2-2
NEW command 1-2, 3-2
new text 5-1
NOTAB command 3-7 - 3-8

operation modes 1-1
options 1-2 - 1-3
OS-9 commands 3-8

paste 4-5 - 4-6

quitting Scred 3-8

reading files 3-4
replacing characters 4-5
REPLACE command 4-4 - 4-5
reverse video codes 2-4

saving text 3-3
screen, scrolling 4-2 - 4-3
screen length code 2-4
screen width code 2-4
scroll 4-2 - 4-3
SEARCH command 3-5
search mask 3-5 - 3-6, 4-3
SET command 4-5 - 4-6
shell 3-8
size of buffer 3-3
starting Scred 1-2 - 1-3
status line 4-7
string
  changing 3-5 - 3-6
  replacing 4-4
  searching for 3-5, 4-3
TABS command 3-7
terminal 2-1
  name 2-2
  type 1-3, 2-1
terminating Scred 3-8
termset file 1-2 - 1-3, 2-1 - 2-6
text
  deleting 3-4, 4-4
  duplicating 4-6
  entering 5-1
  saving 3-3
type of terminal 1-3, 2-1

UPDATE command 3-3

VDG terminal 2-1
video attributes 1-2

width of line 1-3
wild cards 3-6
WRITE command 3-3, 4-6

Relocating Macro Assembler

Contents

Chapter 1 / Introduction 1-1
Installation 1-2
Using the RMA 1-2
Available Options 1-3

Chapter 2 / General Information 2-1
Source File Format 2-2
The Label Field 2-2
The Operation Field 2-3
The Operand Field 2-3
The Comment Field 2-4
The Assembly Listing Format 2-4
Evaluation of Expressions 2-4
Expression Operands 2-5
Expression Operators 2-7
Symbolic Names 2-7
Symbolic Names for System Calls 2-8
The DEFS Directory 2-9
The LIB Directory 2-10

Chapter 3 / Macros 3-1
Macro Structure 3-2
Macro Arguments 3-3
Special Arguments 3-4
Automatic Internal Lables 3-5
Documenting Macros 3-7

Chapter 4 / Program Sections 4-1
Program Section Declarations 4-2

Chapter 5 / Program Section Directives 5-1
PSECT Directive 5-1
VSECT Directive 5-3
CSECT Directive 5-5

Chapter 6 / Assembler Directive Statements 6-1
End Statement 6-1
EQU and SET Statements 6-1
FAIL Statement 6-2
IF, ELSE, and ENDC Statements 6-3
NAM and TTL Statements 6-5
OPT Statement 6-5
PAG and SPC Statements 6-6
REPT and ENDR Statements 6-6
RMB Statement 6-7
USE Statement 6-8

Chapter 7 / Pseudo-Instructions 7-1
FCB and FDB Statements 7-1
FCC and FCS Statements 7-2
RZB Statement 7-3
OS9 Statement 7-3

Chapter 8 / Accessing the Data Area 8-1
Using Non-Initialized Data 8-1
Using Initialized Data 8-2

Chapter 9 / Using the Linker 9-1
Running the Linker 9-2
Available Options 9-2

Chapter 10 / Error Messages 10-1

Chapter 11 / Examples 11-1

Appendix A / 6809 Instructions and Addressing Modes A-1

Chapter 1

Introduction

The OS-9 Level Two Relocatable Macro Assembler (RMA) is a full-feature relocatable macro assembler and linkage editor designed to be used by advanced programmers or with compiler systems.

The RMA lets you assemble sections of assembly-language programs independently to create *relocatable object files*. The linkage editor, RLINK, takes any number of program sections and/or library sections, and combines them into a single executable OS-9 memory module. The RMA's features include:

- OS-9 modular, multi-tasking environment support
- Built-in functions for calling OS-9 system routines
- Position-independent, re-entrant code support
- Creating of standard subroutine libraries by allowing programs to be written and assembled separately and then linked together.
- Macro capabilities
- OS-9 Level Two compatibility
- Automatic resolution of global data and program references
- Conditional assembly and library source file support

This manual describes how to use the RMA and basic programming techniques for the OS-9 environment. However, this manual does not attempt to teach 6809 assembly language programming. If you are not familiar with 6809 programming, consult the Motorola 6809 programming manuals or an assembly-language programming book available at most bookstores and libraries.

Installation

The RMA distribution diskette contains a number of files that you will want to copy to a working system disk. After copying the files, store the original diskette in a safe place.

The files included on the distribution diskette are:

RMA	Relocatable Macro Assembler program. Copy this file to the system's execution directory (CMDS).
RLINK	Linkage Editor program. Copy this file to the system's execution directory (CMDS).
ROOT.a	Assembly-language source code file used as a front end section for programs that use initialized data. Copy this file to an RMA working data directory.

Using the RMA

RMA is a command program that you can run from the OS-9 Shell, from a Shell procedure file, or from another program. The basic format used to run the RMA is:

RMA *filename options* > *listing*

The *filename* argument represents the source text file. It is the only required argument.

The *options* argument lets you specify certain RMA features, such as the ability to generate a listing or object file. The list of available options is given in the next section.

The *listing* option tells the RMA to generate a program listing. The redirection symbol (>) lets you redirect the listing to a printer or a disk file, or even pipe the listing to another program. If you omit the redirection symbol, OS-9 prints the listing on your terminal screen.

Available Options

You specify options on the command line by using the prefix - or --. Use - to turn on an option and -- to turn off an option. The available RMA options are:

-o=*path*	Writes the relocatable output to the specified mass storage file. (Default=off)
-l	Writes the formatted assembler listing to standard output. When this option is off, OS-9 prints error messages only. (Default=off)
-c	Suppresses conditional assembly lines in assembler listings. (Default=on)
-f	Sends a top-of-form signal to the printer. (Default=off)
-g	Lists all code bytes generated. (Default=off)
-x	Suppresses macro expansions in assembler listings. (Default=on)
-e	Suppresses error messages in assembler listings. (Default=on)
-s	Prints the symbol table at the end of the assembly listing. (Default=off)
-d*n*	Sets the number of lines per page, for the listing, to *n*. (Default=66)

Note: You can override command line options by using the OPT statement with a source program. See the OPT statement for more information.

Examples

RMA prog5 -l -s -c >/p [ENTER]

This command line tells RMA to assemble the source program, Prog5. The -l and >/p options causes the RMA to write the formatted assembler listing to the printer. The -s option tells the RMA to print the symbol table. The -c option tells the RMA not to print any conditional assembly lines in the listing.

RMA sample -l -x --c >/h0/programs/sample.lst [ENTER]

This command line assembles the source program, *sample*, and sends the listing to the file Sample.lst on the hard disk. The -x option tells the RMA to suppress macro expansion in the listing. The --c option tells the RMA to print conditional assembly lines.

Chapter 2

General Information

The RMA is a two-pass assembler. During the first pass through the source file, it creates the symbol table. During the second pass, the RMA places the machine-language instructions and data values into the relocatable object file.

Writing and testing an assembly-language program using the RMA involves a basic edit, assemble, link, and test cycle. The RMA simplifies this process by letting you write programs in sections that you can assemble separately then link to form the entire program. With this method, if you must change one program section, you do not have to reassemble the entire program.

When using the RMA to develop assembly-language programs, follow these steps:

1. Create a source program file using a text editor, such as the OS-9 Level Two screen editor, Scred.
2. Run the RMA to translate the source file(s) to relocatable object module(s).
3. If the assembler reports any errors, correct the source files and reassemble.
4. Run RLINK to combine the relocatable object modules(s).
5. If RLINK reports any errors, correct the source files, reassemble, and relink.
6. Run and test your program. You can use the Interactive Debugger to help you with this step. Correct errors, if any.

You now have an executable assembly-language program.

Source File Format

The assembler reads its input from the specified source file. This source file contains variable-length lines of ASCII characters. You can create the source file using any text editor, such as Scred.

Each line of the source file is a text string terminated by an end-of-line character (carriage return). The maximum length for a line is 256 characters. Each line can have from one to four fields, which are:

- Label field (optional)
- Operation field
- Operand field (for some operations)
- Comment field (optional)

You can specify an entire line as a comment by placing an asterisk (*) as the first character of the line.

> **Note:** The assembler ignores any blank lines in the source file.

The Label Field

The label field begins at the first character position of the line. Some statements require labels (for example, EQU and SET); others must not have them (for example SPC and TTL).

If a label is present, the assembler usually defines the label as the program address of the first object code byte generated for the line. Exceptions occur in the SET, EQU, and RMB statements. In the SET and EQU statements, the assembler gives the label the value of the result of the operand field. In the RMB statement, it gives the current value of the data address counter.

The label must be a legal symbolic name consisting of from one to eight upper or lowercase characters. Letters, numbers, dollar signs

($), dots (.), and underline characters (_) are all allowed. The first character must be a letter. You must not define a label more than once in a program (except when using it with the SET directive).

If you follow the symbolic name in a label field with a colon (:), the RMA makes the name *globally* known to all modules that are linked together. In this way, you can execute a branch or jump to a label in another module. If you do not place a colon after the label, that label is known only in its own PSECT .

If a line does not contain a label, the first character must be a space.

The Operation Field

The operation field specifies the machine-language instruction or assembler directive statement mnemonic name. Use one or more spaces between it and the label field.

Some instructions include a register name (such as LDA, LDD, or LDU) in the operation field. In these cases, you cannot separate the register name from the rest of the field with a space. The RMA accepts instruction mnemonic names in either upper- or lowercase characters.

Instructions generate from one to five bytes of object code depending on the specific instruction and address mode. Some assembler directives (such as FCB and FCC) also cause the assembler to generate object code.

The Operand Field

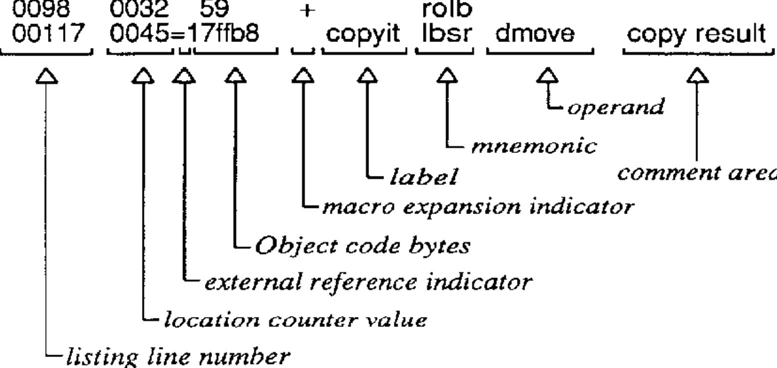
The operand field follows the operation field. You must separate the two fields by at least one space. Some instructions do not use the operand field; other instructions and assembler directives require an operand to specify an addressing mode, operand address, parameters, and so on.

The Comment Field

The comment field is the last field of a source statement. It is an optional field you can use to include a comment about the instruction. The RMA does not process this field but copies it to the program listing.

The Assembly Listing Format

If you specify the -l option with the RMA, the assembler generates a formatted assembly listing. The output listing uses the following format:



Evaluation of Expressions

Operands of many instructions and assembler directives can include numeric expressions in one or more places. The assembler can evaluate expressions of almost any complexity using a form similar to the algebraic notation used in programming languages such as BASIC and FORTRAN.

An expression consists of an *operand* and an *operator*. An operand is a symbolic name and an operator specifies an arithmetic or logical function. All assembler arithmetic uses 2-byte (16-bit binary

internally) signed or unsigned integers in the range of 0 to 65535 for unsigned numbers, or -32768 to +32767 for signed numbers.

In some cases, the assembler expects expressions to produce a value that fits in one byte, such as 8-bit register instructions. Such values must be in the range 0 to 255 for unsigned values and -128 to +127 for signed values.

If the result of an expression is outside its range, OS-9 returns an error message.

OS-9 evaluates expressions from left to right using the algebraic order of operations. That is, it performs multiplication and divisions before addition and subtraction. You can use parentheses to alter the natural order of evaluation.

Expression Operands

You can use the following items as operands within an expression:

decimal numbers — A positive or negative value containing one to five digits (values are in the range of -32768 through +32767). Examples:

```
100
-32767
0
12
```

hexadecimal numbers	The dollar sign ($) followed by one to four hexadecimal characters (0-9, A-F, or a-f). Examples: $EC00 $1000 $3 $0300
binary numbers	Percent sign (%) followed by one to 16 binary digits (0 or 1). Examples: %0101 %1111000011110000 %10101010 %11
character constants	Single quotation mark (') followed by any printable ASCII character. Examples: 'X 'c '5 'T
symbolic names	One to nine characters, the first character must be a letter. Legal characters are upper- and lowercase letters (A-Z, a-z), digits (0-9), and the special characters underscore (_), period (.), dollar sign ($), and "at" (@).
instruction counter	Placed at the beginning of the expression, the asterisk (*) represents the program instruction counter value.

Expression Operators

The following list shows the available operators in the order in which OS-9 evaluates them. Operators listed on the same line have identical precedence. OS-9 processes them left to right when they occur in the same expression.

Assembler Operators By Order of Evaluation

-	negative	^	logical NOT
&	logical AND	!	logical OR
*	multiplication	\	division
+	addition	-	subtraction

Logical operations are performed bit-by-bit for each bit of the operands.

Division and multiplication functions expect unsigned operands, but subtraction and addition accept signed (2's complement) or unsigned numbers. OS-9 returns an error if you attempt to divide by zero or multiply by a factor that results in a product larger that 65535.

Symbolic Names

A symbolic name consists of one to nine lower- or uppercase characters, decimal digits, or the special characters dollar sign ($), undescore (_), period (.), or the at (@). The first character in a symbolic name must be a letter. Some examples of legal symbolic names are:

```
HERE      there      SPL030    PGM_A
Q1020.1   t$integer  L.123.X   a002@
```

Note: The RMA does not convert lowercase characters to uppercase. The names file_A and FILE_A are unique names.

The following are examples of **illegal** symbol names:

2move	The first character is not a letter.
main.backup	There are more than nine characters.
lbl#123	The number sign (#) is not a legal character.

You define a name the first time you use it as a label in an instruction or directive statement. You can define a name only once in a program (except if it is a SET label). OS-9 returns an error message if you attempt to redefine a name.

If you use an undefined symbolic name in an expression, the RMA assumes the name is external to the PSECT. The RMA records information about the reference so the linker can adjust the operand accordingly.

> **Note:** You cannot use external names in operand expressions for assembler directives.

Symbolic Names for System Calls

A system-wide assembly language equate file called OS9defs.a defines the RMA symbolic names for all system calls. You can include this file when the RMA assembles hand-written or compiler-generated code by using the USE assembler directive (see Chapter 6). The RMA has a built-in macro that generates the system calls from the symbolic names.

Symbolic System names can also be resolved by using sys.l in the LIB directory with RLINK. This chapter contains additional information on the LIB Directory. Chapter 9 discusses RLINK.

The DEFS Directory

The OS9defs.a file contains the following groups of defined symbols:

System Service Request Code definitions
I/O Service Request Code definitions
File access modes
Signal codes
Status codes for GetStat/PutStat structure formats
Module definitions
    Universal module offsets
    Type-dependent module offsets
        System module
        File manager module
        Device driver module
        Program module
        Device descriptor module
Machine characteristics definitions
Error code definitions
System dependent error codes
Standard OS-9 error codes

To view the contents of the OS9defs.a file, which includes a brief description of each symbol name, use the OS-9 LIST command. For example, if your OS-9 Level Two Development Pack Diskette 1 is in the current drive, type:

```
list /d0/defs/os9defs.a
```

Or send the file to your printer by typing:

```
list /d0/defs/os9defs.a > /p
```

To include the OS9defs file with your source code when assembling a file, you can use the following statements:

```
ifp1
use os9defs.a
endc
```

However, OS9Defs.a provides the assembly source from which Sys.1 is created (see the following section "The LIB Directory"). In many cases, using Sys.l requires less memory and processes faster.

For programmers who prefer to use the OS-9 Level One ASM program for writing code, the DEFS directory contains four other files: Defsfile, Defsfile.dd, OS9defs, and Systype. These four files contain Level Two information but are in the format required by ASM.

Also included in the DEFS directory are Wind.h, Stdmenu.h, Mouse.h, and Buffs.h. These four files contain Level 2 data structures for window, menu, mouse, and buffer manipulation using the C language.

The LIB Directory

Two OS-9 library files are also included in the LIB directory on Diskette 1 of your Development Pack. The files are:

cgfx.l that provides Level Two graphics routines for the C language

sys.l the system library--defines the standard symbolic references (error messages, I$ and F$ system calls, and so on). Use with RLINK to resolve references rather than the USE instruction in your source code.

For instance, to link a program called Updn (see Chapter 11, "Examples"), you could type:

```
RLINK RELS/UPDN.R -l=/d0/lib/sys.l -o=/d0/cmds/updn
```

Chapter 3

Macros

At times, you might need to use an identical sequence of instructions more than once in a program, such as a routine to display messages to the screen. Instead of repeating the routine in your program, you can create a macro that you can call just like any other assembly-language instruction.

A *macro* defines a set of instructions with a name you assign. Using this name, you can call the macro as many times as you want. In addition, you can use macros to create complex constant tables and data structures. To define a macro, use the MACRO and ENDM directives. For example, the following macro performs a 16-bit left shift on the Register D:

```
dasl    MACRO
aslb
rola
ENDM
```

The MACRO directive marks the beginning of the macro definition. The name assigned to the macro is dasl. To use this new macro, specify dasl as an instruction as shown here:

```
ldd  12,s    get operand
dasl         double it
std  12,s    save operand
```

If the RMA encounters a macro name in the instruction field during the assembly process, it replaces the macro name with the machine instructions given in the macro definition. So, when the RMA encounters the dasl macro name in the instruction field, it outputs the codes for aslb and rola.

Normally, RMA does not expand macros on listings. However, you can use the -x option to cause it to do so.

> **Note:** Macros are similar to subroutines, but do not confuse the two. A macro duplicates the routine within your program every time you call it. It also allows some alteration of the instruction operands. A subroutine, however, appears only once within a program and cannot be changed. Also, you call a subroutine using the special instructions (BSR or JSR). Generally, using a macro instead of a subroutine produces longer but slightly faster programs.

Macro Structure

A macro definition consists of three sections: header, body, and terminator. The macro *header* marks the beginning of the macro and assigns the macro's name. The *body* of the macro contains the statements. The *terminator* indicates the end of the macro. The general format is as shown here:

```
name    MACRO     /* macro header */
          .
          .
        body      /*macro body */
          .
          .
        ENDM      /* macro terminator */
```

The *name* is required by the MACRO directive. It can be any legal assembler label. You can, if you wish, even redefine a 6809 directive, such as LDA or CLR, by defining a macro with the same name. This lets you use the RMA as a *cross-assembler* for non-6809 (8- or 16-bit) processors by either defining (or re-defining) instructions for the target CPU.

> **Note:** Redefinition of assembler directives, such as RMB, can cause unpredictable consequences. Redefine with care.

The *body* of the macro contains any number of legal RMA instruction or directive statements. You can even include references to previously defined macros. Calling another macro from within a macro is called *nesting*. For example:

```
times4  MACRO
        dasl
        dasl
        ENDM
```

This example shows the times4 macro calling the dasl macro twice. You can nest macros up to eight deep.

Note: You cannot define one new macro within another.

Macro Arguments

By using arguments with your macros, you can vary a macro each time you call it. You can use arguments to pass operands, register names, constants, variables, and so on, to the macro. A macro can have as many as nine arguments in the operand field. An argument consists of a backslash and an argument number (\1, \2, ...\9).

When the RMA expands the macro, the assembler replaces each argument with the corresponding text string argument specified in the macro call. When using arguments within the macro, you can only use them in the operand field. You can use arguments in any order and any number of times.

The following example macro performs the typical instruction sequence to create an OS-9 file:

```
create  MACRO
        leax    \1,pcr      get addr of filename string
        lda     #\2         set path number
        ldb     #\3         set file access mode
        os9     I$CREATE
        ENDM
```

The first argument, \1, supplies the filename string address. The second argument, \2, specifies the path number, and the third, \3, gives the file access-mode code. The following instruction shows how to call the create macro with its arguments:

```
create outname,2,$1E
```

RMA expands the create macro like this:

```
leax outname,pcr
lda  #2
ldb  #$1E
os9  I$CREATE
```

Note that if an argument string includes special characters such as backslashes or commas, you must enclose the string in double quotation marks. For example, the following instruction calls a macro called double and passes two arguments:

```
double count,"2,s"
```

To declare a null argument, omit the argument and use a comma to hold its place in the sequence (if necessary). The RMA creates an empty string. For example:

```
double count
```

or

```
double ,"2,s"
```

Special Arguments

RMA has two special argument operators that you might find useful when constructing complex macros. They are:

\L*n*	Returns the length of argument *n* in bytes
\#	Returns the number of arguments passed in a given macro call

Generally, you use these special operators with the RMA's conditional assembly statements to test the validity of arguments used in a macro call, or to customize a macro according to the actual arguments passed. You can use the FAIL directive if you want a macro to report errors that occur during execution. The following example is an expanded version of the create macro:

```
create  MACRO
        ifne    \# - 3      must have exactly 3 arguments
        FAIL    create: must have 3 arguments
        endc
        ifgt    \L1 - 29    filename can be 1 - 29 characters long
        FAIL    create: filename too long
        endc
        leax    \1,pcr      get addr of filename string
        lda     #\2         set path number
        ldb     #\3         set file access mode
        os9     I$CREATE
        ENDM
```

Automatic Internal Labels

At times, it might be necessary to use labels *within* a macro. You can specify macro-internal labels with \@. If there is more than one label, you can add an extra character or characters for uniqueness. For example, if you need two labels with a macro, you might use the names \@A and \@B. You can add the extra character(s) before the backslash or after the \@ symbol.

When the RMA expands the code, internal labels (\@) take the form \@*xxx* where *xxx* is a decimal number between 000 and 999. For example, the expansion of the labels \@A and \@B would be \001A and \001B. If the macro is called again, the expansion would be \002A and \002B, and so on.

The following example shows a macro using internal labels:

```
testovr MACRO
        cmpd    #\1        compare to arg
        bls     @A         branch if in range
        orcc    #1         set carry bit
        bra     \@B        and skip next instruction
     @A andcc   #$FE       clear carry
     @B equ     *          continue with routine...
         .
         .
         .
        ENDM
```

If you call the testovr macro with the instruction:

```
testovr $80
```

RMA expands the labels in the following way:

```
        cmpd    #$80       compare to arg
        bls     @001A      branch if in range
        orcc    #1         set carry bit
        bra     @001B      and skip next instruction
@001A   andcc   #$FE       clear carry
@001B   equ     *          continue with routine...
         .
         .
         .
```

If you call the testovr macro a second time with:

```
testovr 240
```

RMA expands the labels in the following way:

```
        cmpd    #240       compare to arg
        bls     @002A      branch if in range
        orcc    #1         set carry bit
        bra     @002B      and skip next instruction
@002A   andcc   #$FE       clear carry
@002B   equ     *          continue with routine...
         .
         .
         .
```

Documenting Macros

Although macros are a useful programming tool, you should use them with care. Indiscriminate use can impair the readability of a program and make it difficult for other programmers to understand the program logic. Be sure to document your macros thoroughly.

Chapter 4

Program Sections

One of the most useful functions of the RMA is that it lets you write programs in segments that you can assemble separately. You can then use RLINK to combine the segments into one OS-9 memory module with a coordinated data storage area.

When writing a program in segments, you must divide it into sections for variable storage definitions (VSECTs) and sections for program statements (PSECTs). By using external names, the code in one segment can reference variables declared in another segment, or can transfer program control to labels in another segments. The assembler outputs a relocatable object file (ROF) for each program section. This object file contains the object code output plus information about the variable storage declarations for the linker to use.

RLINK reads relocatable object files, and assigns space in the data storage area. It also combines all the object code into a single executable memory module. To do this, RLINK must alter the operands of instructions to refer to the final variable assignments and must adjust program transfer control instructions that refer to labels in other segments.

The following shows a simplified memory map after the linker has processed three program segments (A, B, and C):

process data area

Segment A Variables
Segment B Variables
Segment C Variables

Executable Memory Module

Module Header
Segment A Object Code
Segment B Object Code
Segment C Object Code
CRC Check Value

Each section in the process data area corresponds to each program segment's VSECT. RLINK generates the module header and CRC check values. The Segment A Object Code is the *mainline*, or beginning, segment. Each object code segment corresponds to each program segment's PSECT.

Program Section Declarations

The RMA uses three section directives (PSECT, VSECT, and CSECT) to control the placement of object code and allocation of variable space in the program. The ENDSECT directive indicates the end of a section.

PSECT indicates the beginning of a relocatable object file. PSECT causes the RMA to initialize the instruction and data location counters, and assemble subsequent instructions into the ROF object code area.

VSECT causes the RMA to change the variable (data) location counters and to place information about subsequently declared variables in the appropriate ROF data description area. You declare VSECTs within PSECTs.

CSECT initializes a base value for assigning sequential numeric values to symbolic names. CSECTS are provided for convenience only. Their use is optional.

The RMA maintains the following counters within each section:

Directive	**Counter**
PSECT	instruction location counter
VSECT	initialized direct page counter non-initialized direct page counter initialized data location counter non-initialized data location counter

Because the source statements within a certain program section cause the linker to perform a function appropriate for the statement, the type of mnemonic allowed within a section is sometimes restricted. However, the following mnemonics can appear inside or outside any section: nam, opt, ttl, pag, spc, use, fail, rept, endr, ifeq, ifne, iflt, ifle, ifge, ifgt, ifpl, endc, else, equ, set, macro, endm, and endsect.

Chapter 5

Program Section Directives

PSECT Directive

The PSECT directive specifies the beginning of a program code section. You can specify only one PSECT for each assembly-language file. The PSECT directive initializes all assembler location counters and marks the start of the program segment. You must declare all instruction statements and VSECT data reservations (RMB) within the PSECT/ENDSECT block.

The syntax for the PSECT directive is:

PSECT ***name,typelang,attrrev,edition,stacksize,entry***

If the program section is to be a mainline segment, you can specify the name and five expressions as an operand list to PSECT. The RMA stores the values of the operand list in the relocatable object file for later use by the linker. If you omit the operand list, PSECT defaults to the name Program and all expressions default to zero. The following list describes the available expressions:

name — Used by the linker to identify the PSECT. The *name* can be up to 20 bytes long and can consist of any printable characters, except the space and comma. The *name* does not need to be unique; however, it is often easier to identify PSECTs when their names are distinct.

typelang — Used by the linker as the executable module type/language byte. If the PSECT is not a mainline segment, *typelang* must be zero.

attrrev Used by the linker as the executable module attribute/revision byte.

edition Used by the linker as the executable module edition byte.

stacksize Used by the linker as the amount of stack storage required by the PSECT. Specify *stacksize* as a word expression. The linker adds the value in all PSECTs that make up the executable module and adds the total to any data storage requirement for the entire program.

entry Used by the linker as the program entry point offset for the PSECT. Specify *entry* as a word expression. If the PSECT is not a mainline segment, this value must be zero.

Statements that you can use in a PSECT are: any 6809 mnemonic, fcc, fdb, fcs, fdb, rzb, vsect, endsect, os9, and end. Note that you cannot use RMB in a PSECT.

> **Note:** If you are familiar with the OS-9 Level I Interactive Assembler, note the following difference between the RMA's PSECT directive and the Interactive Assembler's MOD statement. The MOD statement directly outputs an OS-9 module header, but PSECT only sets up information for the linker. The linker creates the module header.

Example

```
* this program starts a basic09 process

         ifp1
         use ..../defs/os9defs.a
         endc

PRGRM    equ     $10
OBJCT    equ     $1
```

```
stk      equ 200
         psect rmatest,$11,$81,0,stk,entry

name     fcs      /basic09/
prm      fcb      $d
prmsize  *-prm

entry    leax     name,pcr
         leau     prm,pcr
         ldy      #prmsize
         lda      #PRGRM+OBJCT
         clrb
         os9      F$FORK
         os9      F$WAIT
         os9      F$EXIT
         endsect
```

VSECT Directive

The VSECT directive indicates the variable storage section, which can contain either initialized or non-initialized variable storage definitions. The VSECT directive causes the RMA to change the data location counters. The RMA offers two sets of counters for each VSECT: one set for direct page variables and another for variables that are normally index-register offsets into a process's data storage area.

The syntax for a VSECT directive is:

```
VSECT [DP]
```

If you specify the DP operand, the RMA uses the direct page counters. If you omit DP, the RMA uses the index register counters.

You can specify any number of VSECT blocks within a PSECT. Note, however, that the data location counters maintain their values from one VSECT to the next. Because the linker handles the actual data allocation, there is no facility to adjust the data location counters.

Statements that you can use within a VSECT are: rmb, fcc, fdb, fcs, fcb, rzb, and endsect. The fcc, fdb, fcb, fcs, and rzb directives place data into the initialized data area. Programs move initialized constants which appear inside a VSECT from the data section to the program section for accessing by the 6809 program counter relative addressing mode. Initialized constants can appear outside of a VSECT; however, if they do, the program cannot change them.

Example

```
        ifp1
        use ..../defs/os9defs.a
        endc

PRGRM   EQU     $10
OBJCT   EQU     $1
stk     EQU     200
        PSECT pgmlen,$11,$81,0,stk,start

* data storage declarations
        VSECT
temp    RMB     1
addr    RMB     2
buffer  RMB     500
ENDSECT

start   leax    buffer,u get address of buffer
        clr     temp
        inc     temp
        ldd     #500     loop count
loop    clr     ,x+
        subd    #1
        bne     loop
        os9     F$EXIT   return to OS9
        ENDSECT
```

CSECT Directive

The CSECT directive provides a method for assigning consecutive offsets to labels without resorting to EQUs.

The syntax for the CSECT directive is:

CSECT *expression*

If you specify an *expression*, the RMA sets the CSECT base counter to the specified value. If you do not include an *expression*, the RMA uses a base counter value of zero.

Example

```
* This CSECT assigns offsets of 0, 1, and 2 respectively.

        CSECT 0
R$CC    RMB 1       Condition code register
R$A     RMB 1       A accumulator
R$B     RMB 1       B accumulator
        ENDSECT
```

See the Defs file that is included in the OS9 Development diskette for more CSECT examples.

Chapter 6

Assembler Directive Statements

Directive statements give the assembler information that affects the assembly process, but they do not generate code. Read the descriptions in this chapter carefully. Some directives require labels, some allow optional labels, and a few cannot have labels.

END Statement

The END statement indicates the end of a program. The syntax for END is:

```
END
```

You cannot use a label with the END statement.

Because the RMA assumes the end of file when it encounters an end-of-file condition on the source file, the END statement is optional.

EQU and SET Statements

The EQU and SET statements let you assign a value to a symbolic name (label). The syntaxes for these statements are:

```
label EQU expression
label SET expression
```

The *label* is required. You can specify *expression* as an expression, a name, or a constant.

EQU lets you define symbols only once in the program. Usually, you use EQU to define program symbolic constants, especially those used with instructions. It is a standard programming practice to place all EQUs at the beginning of the program.

When using EQU, the *label* must be unique, and you must define the *expression* if you specify a name.

SET lets you redefine a symbol as many times as you want. Usually, you use SET to define symbols used to control the assembler operations, such as conditional assembly and listing control.

Example

```
FIVE      EQU    5
OFFSET    EQU    address-base
TRUE      EQU    $FF
FALSE     EQU    0
SUBSET    SET    TRUE

          ifne   SUBSET
          use    subset.defs
          else
          use    full.defs
          endc
SUBSET    set    FALSE
```

FAIL Statement

The FAIL statement forces the RMA to report an assembler error. Generally, you use FAIL with conditional assembly directives that test for various illegal conditions. The syntax for the FAIL statement is:

FAIL ***textstring***

The RMA displays the *textstring* operand in the same manner as normal RMA-generated error messages. Because the RMA assumes the entire line after the FAIL keyword to be the error message, you cannot specify a comment field.

Example

```
ifeq    maxval
FAIL    maxval cannot be zero
endc
```

IF, ELSE, and ENDC Statements

The IF, ELSE, and ENDC statements let you selectively assemble (or not assemble) one or more parts of a program, depending on the value of a variable or computed value. The syntaxes for these statements are:

```
IFxx    expression
        statements
ELSE
        statements
ENDC
```

When the RMA processes an IF statement, it makes the desired comparison. If the comparison result is true, the RMA processes the statements following the IF statement until it finds an ENDC or ELSE.

The ELSE statement is optional. If the RMA encounters an ELSE statement, it processes the statements following the ELSE if the result of the comparison is false.

The ENDC statement marks the end of a conditional program section.

There are several available IF statements:

IFEQ	True if operand equals zero
IFNE	True if operand does not equal zero
IFLT	True if operand is less than zero
IFLE	True if operand is less than or equal to zero
IFGT	True if operand is greater than zero
IFGE	True if operand is greater than or equal to zero
IFP1	True only during the assembler's first pass (no operand)

Examples

In the following example, IFEQ tests if the operand is equal to zero:

```
IFEQ   SWITCH
ldd    #0        assembled only if SWITCH=0
leax   1,x
ENDC
```

The following example adds the ELSE condition to the preceding program:

```
IFEQ   SWITCH
ldd    #0        assembled only if SWITCH=0
leax   1,x
ELSE
ldd    #1        assembled only if SWITCH does not equal 0
leax   -1,x
ENDC
```

You can use IF statements to test the result of a arithmetic evaluation as an operand. This example tests to see if the result of the subtraction of MIN from MAX is less than or equal to zero:

```
IFLE MAX-MIN
```

The IFP1 statement tells the RMA to process subsequent statements during the first pass only. You can use this for program sections that contain only symbolic definitions to be processed only once during the assembly. Because they do not generate actual object code output, the symbolic definitions are processed during Pass 1 only. The OS9Defs file is an example of a large section of such definitions. For example, you can use the following statements at the beginning of many source files:

```
IFP1
use    /do/defs/OS9Defs
ENDC
```

NAM and TTL Statements

The NAM and TTL statements let you define or redefine a program name or listing title line, respectively. The RMA prints this information on each listing page header.

The syntaxes for NAM and TTL are:

NAM *string*
TTL *string*

You cannot specify a label with these statements.

The RMA prints the program name, set by NAM, on the left side of the second line of each listing page. The RMA then prints a dash, and the title line, set by TTL. You can change the program name and listing title as often as you like.

Example

```
NAM Datac
TTL Data Acquisition System
```

This example prints the following information in the listing header:

```
Datac - Data Acquisition System
```

OPT Statement

The OPT statement lets you set or reset any of several assembler control options. The syntax is:

OPT *option*

The operand *option* can be any of the assembler options described in Chapter 1 of this manual. It consists of one character, except for the d and w options, which require a number. Do not specify - or -- in the OPT statement.

You cannot use the label or comment fields with the OPT statement.

Examples

The following statement suppresses the listing generation:

```
OPT l
```

The next example sets the line width to 72 charac-ers:

```
OPT w72
```

PAG and SPC Statements

The PAG and SPC statements let you improve the readability of a program listing by starting a new page or inserting blank lines. The syntaxes for the statements are:

PAG
SPC ***expression***

The PAG and SPC statements cannot have a label field.

The PAG statement causes the RMA to begin a new page in the listing. For Motorola compatability you can also use the alternate form, PAGE.

The SPC statements inserts blank lines in the listing. The operand *expression* specifies the number of blank lines to be inserted. The *expression* can be an expression, constant, or name. If you omit the *expression,* the RMA inserts one blank line.

REPT and ENDR Statement

REPT and ENDR let you repeat the assembly of a sequence of instructions a specified number of times. The syntaxes are:

REPT ***expression***
statements
ENDR

The operand *expression* specifies the number of times the assembly is to be repeated. The *expression* cannot include EXTERNAL or undefined symbols. You cannot nest REPT loops.

Example

```
* make module size exactly 2048 bytes
     REPT   2048-*-3     compute fill size w/crc space
     fcb    0
     ENDR
     emod

* 20-cycle delay
     REPT   5
     nop
     nop
     ENDR
```

RMB Statement

The RMB statement has two uses. When used within a VSECT, RMB declares storage for non-initialized variables in the data area. When used within a CSECT, RMB assigns a sequential value to the symbolic name given as its label. The syntax for RMB is:

label **RMB *expression***

When using RMB in a VSECT, specify a label that is assigned the relative address of the variable. In OS-9, the address must not be absolute and you should usually use indexed or direct page addressing modes to access variables. The linker assigns the actual relative address when processing the relocatable object file. It adds the operand, *expression* to the address counter to update them.

When using RMB in a CSECT, specify a label to which you assign the value of the current CSECT location counter. Doing this, then updates the counter by causing the program to add the result of the *expression* given.

USE Statement

The USE statement causes the RMA to temporarily stop reading the current input file. USE requests that OS-9 open and read input from the specified file/device until an end-of-file occurs. OS-9 then closes the new input file, and the RMA resumes processing at the statement following the USE statement. The syntax is:

```
USE pathlist
```

The *pathlist* specifies the new input file or device. You cannot specify a label with the USE statement.

You can nest as many USE statements as you can have open files at one time (usually 13, not including the standard I/O paths).

Example

To accept interactive input from the keyboard during the assembly of a disk file, use the following statement:

```
USE /term
```

Chapter 7

Pseudo-Instructions

Pseudo-instructions are special assembler statements that generate object code, but do not correspond to actual 6809 machine instructions. Their primary purpose is to create special sequences of data to be included in the program. Labels are optional on pseudo-instructions.

FCB and FDB Statements

The FCB and FDB pseudo-instructions generate sequences of constants within the program. The syntaxes for these pseudo-instructions are:

FCB *expression*, [*expression*,...]
FDB *expression*, [*expression*,...]

Expression can be any legal expression. You can specify more than one expression by separating them with commas.

FCB generates a sequence of single constants in the program. It reports an error if an *expression* has a value that is greater than 255 or less than -128.

FDB generates a sequence of double constants in the program. If FDB evaluates an *expression* with an absolute value of less than 256, the high-order byte is zero.

If FCB or FDB appears within a VSECT, the RMA assigns the data to the appropriate initialized data area (DP or non-DP). Otherwise, the RMA places the constant in the code area. If the constant contains an EXTERNAL reference, the program, using Root.a, must copy out and adjust the references.

Examples

```
FCB    1,20,'A
FCB    index/2+1,0,0,1

FDB    1,10,100,1000,10000
FDB    $F900,$FA00,$FB00,$FC00
```

FCC and FCS Statements

The FCC and FCS pseudo-instructions generate a series of bytes corresponding to the specified character *string*. The syntaxes are:

FCC ***string***
FCS ***string***

FCS is the same as FCC except that the most significant bit (the sign bit) of the last character in the string is set. This is a common OS-9 programming technique to indicate the end of a text string without using additional storage.

String must be enclosed in delimiters. You can use the following characters as delimiters:

! " # $ % & ( ) * + , - . /

The beginning and ending delimiters must be the same character. The delimiting character cannot appear in the character string.

FCC and FCS output bytes that are the literal numeric representation of each ASCII character in the character *string*.

If FCC or FCS appear in a VSECT, the RMA assigns the data to the appropriate initialized data area (DP or non-DP). Otherwise, the RMA places the constant in the code area.

Examples

```
FCC /this is the character string/
FCS ,01234567899,
FCS AA      null string
FCC $z$
FCS ""      null string
```

RZB Statement

The RZB pseudo-instruction fills memory with a sequence of bytes, each of which has a value of zero. The syntax is:

RZB *expression*

The *expression* is a 16-bit expression. The RMA evaluates the *expression* and places that number of zero bytes in the appropriate code or data section.

OS9 Statement

The OS9 pseudo-instruction is a convenient way to generate OS-9 system calls. The syntax is:

OS9 *expression*

The RMA uses the *expression* value as the request code. The following instruction sequence is the equivalent to the OS9 pseudo-instruction:

SWI2
FCB *operand*

The OS9Defs file contains the standard definitions of the symbolic names of all the OS-9 service requests. You can use these names with the OS9 pseudo-instruction to improve the readability and portability of assembly-language software.

Examples

```
OS9     I$READ     call OS-9 READ service request

OS9     F$EXIT     call OS-9 EXIT service request
```

Chapter 8

Accessing the Data Area

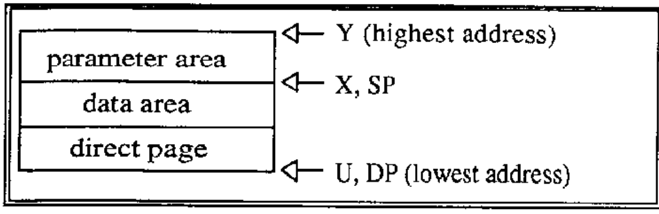
In general, the RMA assumes that the program will access data using indexed or direct page addressing modes. By convention, one index register contains the starting address of the data area, and the direct page register contains the page number of the lowest-address page of the data area. The RMA/RLINK system automatically adjust operands of instructions, using indexed and direct page addressing modes.

The RMA accesses the data area differently depending on whether or not your program uses initialized data. Initialized data is data that has an initial value that is modified by the program. You create initialized data with the FCB, FDB, FCC and similar directives used in a VSECT.

If you do not use initialized data, the RMA accesses program data using index registers--this is the method used by the OS-9 Level I Interactive Assembler.

Using Non-Initialized Data

Programs that do not used initialized data declare all data storage in VSECTs using RMBs. The following diagram shows how the RMA sets up the data memory area and registers for a new process:



When OS-9 executes a process, the MPU Registers contain the bounds of the data area. Register U contains the beginning address and Register Y contains the ending address. OS-9 sets the SP register to the ending address + 1, unless you use a parameter. The direct page register contains the page number of the beginning page. If you used no parameters, Y, X, and SP are the same value. The OS-9 Shell always passes at least an end-of-line character in the parameter area.

If Register U is maintained throughout the program, you can use constant-offset-indexed addressing.

You can write part of the program's initialization routine to compute the actual addresses of the data structure and store these addresses in pointer locations in the direct page. Then, obtain the addresses later using direct-page addressing mode instructions.

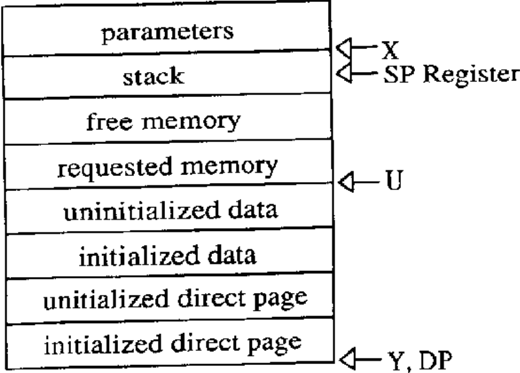
> **Note:** Because the memory addresses assigned to the program section and the address section are not a fixed distance apart, you cannot use program-counter relative addressing to obtain the address of objects in the data section.

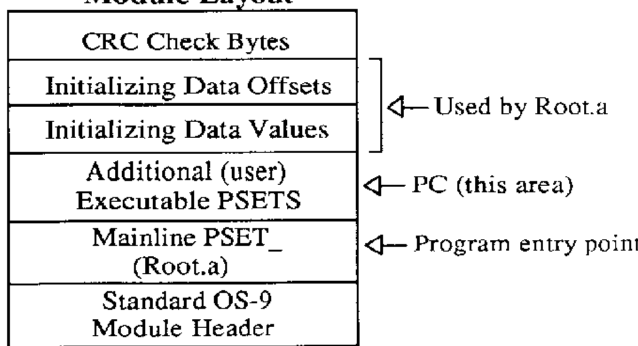
Using Initialized Data

If you plan to use initialized data, you need to copy the data from the initialized data section in the object module to the data storage area pointed to by the Register U. Do so by using the Root.a mainline module (object code that is directly executable by using the OS-9 F$FORK). The function of the Root.a mainline module is to use the initializing values and offsets of the initialized data location, stored in the object code module, to actually initialize variables. The linker automatically generates the initialization information area of the object code module based on information passed by the RMA in the relocatable object file.

Root.a sets Register Y to point to the same location to which Register U pointed. Register X points to the parameter area, and Register U points to the top of data allocated by the linker. The data-index

register choice is arbitrary, but use your choice consistently. To maintain compatibility with code produced by the C compiler, Register Y is used as the data pointer. For more information on Root.a, study the commented source code supplied on the distribution diskette. The following diagram shows how the RMA sets up the data area:





Chapter 9

Using the Linker

The Relocating Macro Assembler lets you write and assemble programs separately and then link them to form a single object code OS-9 module. The linker, RLINK, combines relocatable object files (ROF) into a single OS-9 format memory module. It also resolves external data and program references. Because RLINK allows references to occur between modules, you can write one program that references a symbol in another program.

If the RMA encounters an external reference during the assembly process, it sets up information denoting the existence of an internal reference. The RMA does not know the location of the external reference.

Because the RMA is a relocatable assembler, it produces relocatable object files that do not have absolute addresses assigned. The RMA assembles each section with the absolute address 0.

RLINK reads in all the relocatable object files and assigns each an absolute memory address for data locations and instruction locations for branching. OS-9 resolves any other addresses at execution time.

By using the RMA and RLINK, you can write programs in smaller sections that are easier to read and debug. In this way, if an error occurs, you need edit and reassemble only the module in which the error occurred. Then, you can relink the fixed module with the rest of the program.

Running the Linker

You call the linker, RLINK, with the following command line:

RLINK [*options*] *mainline* [*sub1*....sub*n*] [*options*]

All input files must be in relocatable object format (ROF).

Mainline specifies the pathlist of the mainline (first) segment from which RLINK resolves external references and generates a module header. It is the object of the *mainline* file to perform the initialization of data and the relocation of any initialized data references within the initialized data, using the information in the object module supplied by RLINK (See Chapter 7.) You indicate that a program is the mainline module by setting the *type/lang* value in the PSECT directive to a non-zero value.

The *sub1* and *subn* options represent any additional modules to be linked to the *mainline* module. The additional ROFs cannot contain a mainline PSECT notation (*type/lang>0*).

RLINK includes the *mainline* file and all sub-modules in the final linked object module, even if you did not reference the subroutine.

Available Options

You can use any of the following options on the RLINK command line:

-o=*path*	Writes the linker object (memory module) output to the file specified by the pathlist. RLINK assumes the last element in the pathlist to be the module name unless you use the -n option.
-n=*name*	Specifies *name* as the object file.

-l=*path* Specifies *path* as the library. A library file consists of one or more merged assembly ROFs. The assembler checks each PSECT in the file to see if it resolves any unresolved references. If so, RLINK includes the module in the final output module. Otherwise, RLINK skips the file. RLINK searches library files in the order in which you specify them on the command line. A library file cannot contain a mainline PSECT.

-E=*n* Sets the edition number in the final output module to *n*. You can also use -e (lowercase).

-M=*size* Sets the number of pages of additional memory for C.LINK to allocate to the data area of the final object module. If you omit this option, C.LINK adds up the total data stack requirements found in the PSECT of the input modules, and uses that value.

-m Prints the linkage map that indicates base addresses of the PSECTs in the final object module.

-s Prints the final addresses that RLINK assigned to symbols in the final object module.

-b=*ept* Links C-language functions so that they can be called from BASIC09. The argument *ept* specifies the name of the function to which control is transferred when BASIC09 executes a RUN command.

-t Allows static data to appear in a BASIC09 callable module. RLINK assumes that the C function being called and the calling BASIC09 program provide a sufficiently large static storage data area pointed to by Register Y.

Chapter 10

Error Messages

When the RMA detects an error during assembly, it prints an error message in the listing just before the source line containing the error. In some cases, the RMA might report more than one error for a source line. If you do not use the -l option to produce the listing, the RMA still prints the error messages and the problem source line. At the end of the listing, the RMA reports the total number of errors and warnings in the assembly summary statistics.

The RMA prints all error messages, the associated source line, and the assembly summary to the assembler's error path. You can redirect this output using the shell redirection symbol. For example:

```
RMA sourcefile -o=sourcefile >>src.error
```

During the initial stages of assembly, you might find it useful to suppress generation of both the listing and object code (by omitting the -l and -o options). Doing this lets the RMA perform a quick assembly just to check for errors. In this way, you can find and correct many errors before printing a lengthy listing.

Some errors stop execution on a line. In these cases, the RMA might not detect all errors that occur on one line; so, make changes carefully.

The following list shows the RMA error messages and a description for each message.

Bad label

The label field contains an incorrectly formed label.

Bad Mnemonic

The mnemonic field contains a mnemonic that the RMA does not recognize or a mnemonic that is not allowed in the current program section.

Bad number

The numeric constant definition contains a character that is not allowed in the current radix.

Bad operand

The operand field is missing an expression or contains an incorrectly formed operand expression.

Bad operator

The operator contains an incorrectly formed arithmetic expression.

Bad option

An option is not recognized or is incorrectly specified.

Bracket Missing

A bracket is missing from an expression.

Can't open file

The RMA encountered a problem when opening an input file.

Can't open macro work file

The RMA cannot open a macro work file.

Comma expected

The RMA cannot find an expected comma.

Conditional nesting error

Program contains mismatched IF and ELSE/ENDC conditional assembly directives.

Constant definition

The statement contains an incorrectly formed constant definition.

DP section???

Direct Page assignments have exceeded 256 bytes.

ENDM without MACRO

The RMA encountered an ENDM statement without a matching MACRO statement.

ENDR without REPT

The RMA encountered an ENDR statement without a matching REPT statement.

Fail *message*

The RMA encountered a FAIL directive.

File close error

An error occurred closing a file.

Illegal addressing mode

The specified addressing mode cannot be used with the instruction.

Illegal external reference

You cannot use external names with assembler directives. If an operand expression contains an external name, the RMA can only perform binary plus and minus operations.

Illegal index register

You cannot use the specified register as an index register.

Label missing

The statement is missing a required label.

Macro arg too long

More than 60 characters (total) were passed to the macro.

Macro file error

The RMA experienced problems when trying to access the macro work file.

Macro nesting too deep

You can nest macros up to eight levels deep.

Nested MACRO definitions

You cannot define a macro within another macro definition.

Nested REPT

You cannot nest repeat blocks.

New symbol in pass two

This indicates an assembler symbol lookup error. This error can be caused by a symbol table overflow or bad memory.

No input files

You must specify an input file.

No param for arg

A macro expansion is attempting to access an argument that was not passed by the macro call.

Phasing error

A label has a different value during Pass 2 than it did during Pass 1.

Redefined name

The name appears more than once in the label field (other than on a SET directive).

Register list error

The legal register names allowed in tfr, exg, and pul are: A, B, CC, DP, X, Y, U, S, and PC.

Register size mismatch

The registers specified in the tfr and exg instructions must be the same size.

Symbol lost?

This indicates an assembler symbol lookup error. This error can be caused by a symbol table overflow or bad memory.

Too many args

You can pass up to nine arguments to a macro.

Too many object files

You can specify the -o option to the RMA only once on the command line.

Too many input files

You can specify a maximum of 32 input files.

Undefined org

The * (program counter org) cannot be accessed within a VSECT.

Unmatched quotes

A quotation mark is missing.

Value out of range

A byte expression value is less than -128 or greater than 255.

Chapter 11

Examples

The chapter contains two assembly language programming examples:

- **LSIT**, to list files
- **UpDn**, to convert input case to either upper or lower

```
**********
* LSIT UTILITY COMMAND
* A "LIST" Command for poor typists
* Syntax: lsit <path>
* Lsit copies input from path to standard output
* NOTE: This command is similar to the
* LIST command.  Its name was changed
* to allow easy assembly and testing
* since LIST normally is already in memory.

PRGRM       equ    $10
OBJCT       equ    $01
STK         equ    200

            csect
IPATH       rmb    1              input path number
PRMPTR      rmb    2              parameter pointer
BUFSIZ      rmb    200            size of input buffer
            endsect

            psect list,PRGRM+OBJCT,$81,0,STK,LSTENT

BUFFER      equ    200            allocate line buffer
READ.       equ    1 file         access mode
```

```
LSTENT   stx     PRMPTR      save parameter ptr
         lda     #READ.      select read access mode
         os9     I$Open      open input file
         bcs     LSIT50      exit if error
         sta     IPATH       save input path number
         stx     PRMPTR      save updated param ptr

LSIT20   lda     IPATH       load input path number
         leax    BUFFER,u    load buffer pointer
         ldy     #BUFSIZ     max bytes to read
         os9     I$ReadLn    read line of input
         bcs     LSIT30      exit if error
         lda     #1          load st. out path #
         os9     I$WritLn    output line
         bcc     LSIT20      repeat if no error
         bra     LSIT50      exit if error

LSIT30   cmpb    #E$EOF      at end of file?
         bne     LSIT50      branch if not
         lda     IPATH       load input path number
         os9     I$Close     close input path
         bcs     LSIT50      ..exit if error
         ldx     PRMPTR      restore param ptr
         lda     0,x
         cmpa    #$0D        end of param line?
         bne     LSTENT      ..no; list next file
         clrb
LSIT50   os9     F$Exit      ..terminate
         endsect
```

```
* This is a program to convert characters from lower to
* upper case (by using the u option), and upper to lower
* (by using no option). To use type:
*     updn u (for lower to upper) < input  > output

            nam    updn

            opt    l
            ttl    ASSEMBLY LANGUAGE EXAMPLE

PRGRM       equ    $10
OBJCT       equ    $01
stk         equ    250
            psect updn,PRGRM+OBJCT,$81,0,stk,entry

            vsect
temp        rmb    1
uprbnd      rmb    1
lwrbnd      rmb    1
            endsect

entry       lda    ,x+          search parameter area
            anda   #$df         make upper case
            cmpa   #'U          see if a U was input
            beq                 upper branch to set uppercase
            cmpa   #$0D         carriage return?
            bne    entry        no; go get another char

            lda    #'A          get lower bound
            sta    lwrbnd       set it in storage area
            lda    #'Z          get upper bound
            sta    uprbnd       set it in storage area
            bra    start1       go to start of code

upper       lda    #'a          get lower bound
            sta    lwrbnd       set it in storage
            lda    #'z          get upper bound
            sta    uprbnd       set it in storage

start1      leax   temp,u       get storage address
            lda    #0           standard input
            ldy    #$01         number of characters
```

```
loop     os9     I$Read      do the read
         bcs     exit        exit if error
         ldb     temp        get character read
         cmpb    lwrbnd      test char bound
         blo     write       branch if out
         cmpb    uprbnd      test char bound
         bhi     write       branch if out
         eorb    #$20        flip case bit
write    stb     temp        put it in storage
         inca    reg 'a'     standard out
         os9     I$WritLn    write the character
         deca                return to standard in
         bcc     loop        get char if no error
exit     cmpb    #E$EOF      is it an EOF error
         bne     exit1       not eof, leave carry
         clrb                clear carry, no error
exit1    os9     F$Exit      error returned, exit
         endsect
         clrb
```

Appendix A

6809 Instructions and Addressing Modes

	Direct	Extended	Index	Immed	Accum	Inher	Relat	Regis
ABX						X		
ADCA	X	X	X	X				
ADCB	X	X	X	X				
ADDA	X	X	X	X				
ADDB	X	X	X	X				
ADDD	X	X	X	X				
ANDA	X	X	X	X				
ANDB	X	X	X	X				
ANDCC				X				
ASL	X	X	X					
ASLA						X		
ASLB						X		
ASR	X	X	X					
ASRA						X		
ASRB						X		
(L)BCC							X	
(L)BCS							X	
(L)BEQ							X	
(L)BGE							X	
(L)BGT							X	
(L)BGI							X	
(L)BHS							X	
BITA	X	X	X	X				
BITB	X	X	X	X				

	Direct	Extended	Index	Immed	Accum	Inher	Relat	Regis
(L)BLE							X	
(L)BLO							X	
(L)BLS							X	
(L)BLT							X	
(L)BMI							X	
(L)BNE							X	
(L)BPL							X	
(L)BRA							X	
(L)BRN							X	
(L)BSR							X	
(L)BVC							X	
(L)BVS							X	
CLR	X	X	X		X			
CMPA	X	X	X	X				
CMPB	X	X	X	X				
CMPD	X	X	X	X				
CMPS	X	X	X	X				
CMPU	X	X	X	X				
CMPX	X	X	X	X				
CMPY	X	X	X	X				
COM	X	X	X		X			
CWAI				X				
DAA						X		
DEC	X	X	X	X	X			
EORA	X	X	X	X				
EORB	X	X	X	X				
EXG								X
INC	X	X	X		X			
JMP	X	X	X					
JSR	X	X	X					
LDA	X	X	X	X				
LDB	X	X	X	X				
LDD	X	X	X	X				
LDS	X	X	X	X				
LDU	X	X	X	X				

	Direct	Extended	Index	Immed	Accum	Inher	Relat	Regis
LDX	X	X	X	X				
LDY	X	X	X	X				
LEAS			X					
LEAU			X					
LEAX			X					
LEAY			X					
LSL	X	X	X		X			
LSR	X	X	X		X			
MUL						X		
NEG	X	X	X		X			
NOP						X		
ORA	X	X	X	X				
ORB	X	X	X	X				
ORCC				X				
PSHS								X
PSHU								X
PULS								X
PULU								X
ROL	X	X	X		X			
ROR	X	X	X		X			
RTI						X		
RTS						X		
SBCA	X	X	X	X				
SBCB	X	X	X	X				
SEX						X		
STA	X	X	X					
STB	X	X	X					
STS	X	X	X					
STU	X	X	X					
STX	X	X	X					
STY	X	X	X					
SUB	X	X	X	X				
SUBA	X	X	X	X				
SUBB	X	X	X	X				
SWI						X		

	Direct	Extended	Index	Immed	Accum	Inher	Relat	Regis
SWI2						X		
SWI3						X		
SYNC						X		
TFR								X
TST	X	X	X		X			

Index

! (logical OR) 2-7
& (logical AND) 2-7
* (comment) 2-2
* (multiplication) 2-7
+ (addition) 2-7
- (negative) 2-7
- (subtraction) 2-7
-x option (expand macro listings) 3-2
\ (division) 2-7
\ (macro argument) 3-3 - 3-5
\# (macro argument) 3-4
\@ (macro labels) 3-5 - 3-6
\L (macro argument) 3-4
^ (logical NOT) 2-7

additional memory, setting 9-3
arguments, macros 3-3 - 3-5
assembling programs 6-3 - 6-4
assembler control options 6-5 - 6-6
assembler error reporting 6-2 - 6-3
assembly location counters 5-1
assembly-language programs, developing 2-1
attribute/revision byte 5-2

base counter, CSECT 5-5
binary numbers 2-6
blank lines 2-2
body, macro 3-2 - 3-3

C compiler, compatibility 8-3
C-language functions, linking 9-3
cgfx.1 file 2-10
character
  constants 2-6
  string 7-2 - 7-3
code bytes, listing 1-3
colon (global name) 2-3
comment field 2-2, 2-4
comments 2-2
comparison statement 6-3 - 6-4
conditional assembly lines, suppressing 1-3
consecutive offsets to labels 5-5
constants
  character 2-6
  generating 7-1 - 7-2
counter, instruction 2-6
CSECT 4-2 - 4-3, 5-5, 6-7

data
  area 8-2
  location counters 5-3
  storage 8-1 - 8-2
data, initialized 8-2 - 8-3
decimal numbers 2-5
defined symbols 2-9
DEFS directory 2-9 - 2-10
Defs file 5-5
device or file, opening 6-8
direct page
  addressing 8-2
  register 8-2
  variable 5-3

directives
  CSECT 5-5
  ENDSECT 4-2
  FAIL 3-5
  VSECT 5-3 - 5-4
directive statements 6-1 - 6-8
directory
  DEFS 2-9 - 2-10
  LIB 2-10
division functions 2-7
double constants 7-1 - 7-2

edition number option 9-3
ELSE statement 6-3 - 6-4
END statement 6-1
ENDC statement 6-3 - 6-4
ENDR statement 6-6 - 6-7
ENDSECT 4-2, 5-1
entry point offset 5-2
EQU statement 6-1 - 6-2
error, assembler 6-2 - 6-3
error messages, suppressing 1-3
executable module edition byte 5-2
expanding macro listings 3-2
expression evaluation 2-4 - 2-8
external
  data references 9-1 - 9-3
  names 4-1
  program references 9-1 - 9-3
EXTERNAL reference 7-1

FAIL
  directive 3-5
  statement 6-2 - 6-3
FCB statement 7-1 - 7-2, 8-1
FCC statement 7-2 - 7-3, 8-1
FCS statement 7-2 - 7-3

FDB statement 7-1 - 7-2, 8-1
features 1-1
field 2-2
file or device, opening 6-8
files to copy 1-2
filling memory 7-3
final address, printing 9-3
format of assembly listings 2-4

global names 2-3

header, macro 3-2
hexadecimal numbers 2-6

IF statement 6-3 - 6-4
IFEQ 6-3 - 6-4
IFGE 6-3 - 6-4
IFGT 6-3 - 6-4
IFLE 6-3 - 6-4
IFLT 6-3 - 6-4
IFNE 6-3 - 6-4
IFP1 6-3 - 6-4
illegal symbolic names 2-8
index-register offsets 5-3, 8-1
initialized data 8-1 - 8-3
input file, reading 6-8
instruction counter 2-6
instructions
  repeating 3-1 - 3-7, 6-6 - 6-7
integers 2-5

keyboard input 6-8

label field 2-2
labels (macro) 3-5 - 3-6
LIB directory 2-8, 2-10
library path, specifying 9-3

line
  fields 2-2
  maximum length 2-2
lines-per-page, setting 1-3
linking C-language functions 9-3
linkage map, printing 9-3
linker, starting 9-2
linker output 9-2
linking programs 9-1 - 9-3
listing
  blank lines 6-6
  code bytes 1-3
  page 6-6

macro
  arguments 3-3 - 3-5
  body 3-2 - 3-3
  expansion, suppressing 1-3
  header 3-2
  labels 3-5 - 3-6
  terminator 3-2
MACRO directive 3-1
macros, documenting 3-7
memory, filling 7-3
MOD statement 5-2
multiplication functions 2-7

NAM statement 6-5
name, defining 6-5
  PSECT 5-1
  symbolic 2-6, 2-7 - 2-10
nesting macros 3-3
non-initialized data 8-1 - 8-2
null argument (macro) 3-4

numbers
  binary 2-6
  decimal 2-5
  hexadecimal 2-6
numeric expressions 2-4

opening files or devices 6-8
operand 2-4
Operand field 2-2, 2-3 - 2-4
operands, expression 2-5 - 2-6
operation field 2-2, 2-3
operator 2-4
  precedence 2-7
OPT statement 1-3, 6-5 - 6-6
options
  assembler 1-3, 6-5 - 6-6
  RLINK 9-2 - 9-3
OS9 statement 7-3 - 7-4
output, linker 9-2

PAG statement 6-6
page, starting new 6-6
page number 8-2
pointer locations 8-2
precedence of operators 2-7
printing the symbol table 1-3
printer, top-of-form 1-3
program segment 5-1
program statements 4-1
programs, assembling 6-3 - 6-4
PSECT 4-1 - 4-3, 5-1 - 5-3, 9-3
  statements 5-2

reading input files 6-8
register name 2-3
relocatable object file 2-1, 4-1, 9-1
repeating instructions 6-6 - 6-7

REPT statement 6-6 - 6-7
RLINK 2-8, 4-1, 9-1 - 9-3
  options 9-2 - 9-3
  output 9-2
  starting 9-2
RMB statement 6-7
Root.a mainline module 8-2
RZB statement 7-3

segments 4-1
setting lines-per-page 1-3
SET statement 6-1 - 6-2
single constants 7-1 - 7-2
source file 2-2
SPC statement 6-6
specifying
  library path 9-3
  name, RLINK 9-2
stack storage 5-2
starting
  RLINK 9-2
  RMA 1-2
statements
  ELSE 6-3 - 6-4
  END 6-1
  ENDC 6-3 - 6-4
  ENDR 6-6 - 6-7
  EQU 6-1 - 6-2
  FAIL 6-2 - 6-3
  FCB 7-1 - 7-2
  FCC 7-2 - 7-3
  FCS 7-2 - 7-3
  FDB 7-1 - 7-2
  IF 6-3 - 6-4
  NAM 6-5
  OPT 6-5 - 6-6
  OS9 7-3 - 7-4

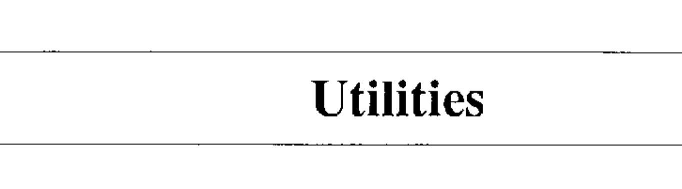
PAG 6-6
REPT 6-6 - 6-7
RMB 6-7
RZB 7-3
SET 6-1 - 6-2
SPC 6-6
TTL 6-5
USE 6-8
statements
directive 6-1 - 6-8
PSECT 5-2
VSECT 5-4
static
data 9-3
storage area 9-3
storage, declare 6-7
storage file s 1-3
string, character 7-2 - 7-3
structure, macro 3-2 - 3-3
suppressing
conditional lines 1-3
error messages 1-3
macro expansions 1-3
symbol table 2-1
printing 1-3
symbolic name 2-2 - 2-3, 2-6, 2-7 - 2-8, 6-1
assigning a sequential value 6-7
symbols
defining 2-9, 6-2
redefining 6-2
Sys.1 file 2-10
system calls
symbolic names 2-8 - 2-10
generating 7-3 - 7-4

terminator, macro 3-2
text string 7-2 - 7-3
title line 6-5
top-of-form signal 1-3
TTL statement 6-5
type/language byte 5-1

USE statement 6-8

variable storage
  definitions 4-1
  section 5-3 - 5-4
VSECT 4-1 - 4-3, 5-1, 6-7, 7-1 - 7-2, 8-1
  directive 5-3 - 5-4
  statements 5-4

writing
  assembler listing 1-3
  output 1-3



Utilities

Contents

Chapter 1 / Introduction 1-1

Chapter 2 / Make Utility 2-1
- Using Make 2-1
  - Examples 2-3
- What is a Makefile? 2-3
  - Built-in Rules and Definitions 2-4
  - Macros 2-5
  - Special Macros 2-6
  - Reserved Macros 2-7
  - Commands 2-7
  - Comments 2-8
  - Long Lines 2-8
- How Make Works 2-8
- Notes about Make 2-9
- Examples of Makefiles 2-10
  - Example 1 2-10
  - Example 2 2-11
  - Example 3 2-11
  - Example 4 2-12
  - Example 5 2-13

Chapter 3 / Touch Utility 3-1
- Examples 3-2

Chapter 4 / Virtual Disk/RAM Disk Driver 4-1
- Initalizing VDD 4-1

Chapter 1

Introduction

The OS-9 Level Two Development Pack includes three utilities:

- **Make:** Helps maintain the current version of software by keeping track of modifications to program source to determine the need for recompiling, reassembling, or relinking files.
- **Touch:** Updates the modification date of specified files.
- **Virtual Disk Driver/RAM Disk Driver:** Creates a high-speed storage system in your computer's RAM that simulates a disk drive.

Chapter 2

Make Utility

The Make utility helps maintain the current version of software. It uses built-in knowledge of OS-9 compilers, file types, and file naming conventions to maintain up-to-date versions of your programs as you develop them. By keeping track of modifications to program source, make can determine the need to recompile, reassemble, and/or relink the files necessary to create an object file.

Using Make

The syntax for Make is as follows:

make *options target1* [*target2*] [*macros*]

The *target1* argument specifies the program that Make is to create. Make accepts multiple arguments *(target2, target3,...,*and so on). The *macros* argument lets you specify macros that Make uses when creating the new target program.

The *options* argument can be one of the following:

-? Displays the usage of Make.

-b Turns off built-in rules governing implicit file dependencies. Use this option if you are quite explicit about your makefile dependencies and do not want Make to assume anything.

-d Turns on the Make debugger and gives a complete listing of the macro definitions, a listing of the files as it checks the dependency list, and all the file modification dates.

-f[=]*path*	Specifies *path* as the makefile. If you omit this option, Make searches for the file named Makefile in the current directory.
-f	causes Make to use the standard input instead of a makefile.
-i	Ignores errors. If you omit this option, Make stops execution if an error code is returned after executing a command line in a makefile.
-n	Displays commands to standard output but does not execute them.
-s	Executes command without echo (silent mode). If you omit this option, Make echoes commands in the makefile to standard output.
-t	Touches the files. Make opens the file for update and then closes it. This updates the modification dates without executing the commands.
-u	Causes Make to execute the makefile commands.
-x	Uses the cross-compiler/assembler.
-z	Reads a list of Make targets from standard input.
-z=*path*	Reads a list of Make targets from *path*.

You can include options on the command line when you run Make or include them in the makefile. You can also define one or more macros on a command line instead of a makefile or to override a macro definition in a makefile. Enclose in quotes any macro definitions that contain spaces or other delimiters. See the following section "Macros".

Examples

```
make -f/d0/source/test.make -i test
```

This Make command creates a program called Test using the makefile /d0/source/test.make. Make ignores any errors that occur.

```
make -s myprog
```

Make uses the file Makefile in the current directory as the makefile for the program Myprog. Make does not echo commands during execution.

What is a Makefile?

A makefile is a special type of procedure file that describes the *dependencies* between files that make up the target program. The makefile contains a sequence of entries that specifies dependencies and commands to resolve the dependencies. A *dependency* entry begins with the target name of the file or module followed by a colon (:). This is then followed by a list of files that are prerequisites to building the target file. This is called a *dependency list.*

In addition to the dependency entry, the makefile can contain commands on how to update a particular target file (if it needs to be updated). Make updates a target file only if it depends on files that are newer than the target file. If Make cannot find the file, it assumes a date of -01/00/00 00:00, indicating that the file needs updating. If you do not specify update instructions, Make attempts to create a command line to perform the operation. Make recognizes a command line because it begins with one or more spaces.
The following is a sample makefile:

```
program: xxx.r yyy.r
   cc xxx.r yyy.r -xf=program
xxx.r: xxx.c /d0/defs/oskdefs.h
   cc: xxx.c -r
yyy.r: yyy.c /d0/defs/oskdefs.h
   cc: yyy.c -r
```

This makefile specifies that the target file program is made up of two relocatable files (.r suffix): *xxx.r* and *yyy.r*. These files are dependent upon *xxx.c* and *yyy.c*, respectively, and both files are dependent on the file *oskdefs.h*.

If either *xxx.c* or */d0/defs/oskdefs.h* has a more recent modification date than *xxx.r*, Make executes the command cc xxx.c -r. Likewise, if either *yyy.c* or */d0/defs/oskdefs.h* has a more recent modification date than *yyy.r*, Make executes the command cc yyy.c -r. If either of the former commands is executed, Make also executes the command cc xxx.r yyy.r -xf=program.

Built-in Rules and Definitions

Make uses the following conventions when determining file types or in defining its rules:

Source Files	Files with a suffix of either .a, .c, .f, or .p are source files in assembly, C, Fortran, and Pascal, respectively.
Relocatable Files	Make determines a file to be relocatable if it has the suffix .r. Relocatable files are made from source files and are assembled or compiled, if necessary, during a make.
Object Files	Make determines a file to be an object file if the file does not have a suffix. An object file is made from a relocatable file and is linked, if necessary, during a make.
Default Compiler	Make's default compiler is cc.
Default Assembler	Make's default assembler is the Relocatable Macro Assembler (RMA).

Default Linker	Make's default linker is cc. You should only use the default linker with programs that use Cstart.
Default Directory	Make uses the current directory (.) for all files.

Macros

You can use macros within a makefile or on the command line. Use the following form to specify a macro:

macro-name=expansion

Make then substitutes every occurrence of *macro-name* with the *expansion.*

Macro names are prefixed with the dollar sign character ($). If you want to specify a macro name longer than a single character, you must enclose the name in parentheses. For example, $R refers to the macro R and $(PFLAGS) refers to the macro PFLAGS. The macro names $(B) and $B refer to the same macro, B. The macro name $BR refers to the B macro also, followed by the character R.

> **Note:** If you define a macro in your makefile and then redefine it on the command line, the command line definition overrides the definition in the makefile. You might find this feature useful for compiling with special options.

Special Macros

Make provides the following special macros:

Macro	Definition
SDIR=*path*	Make searches the directory, specified by *path*, for all implicitly defined source files. If you do not define SDIR within the makefile, Make searches the current directory.
RDIR=*path*	Make searches the directory, specified by *path*, for all implicitly defined relocatable files. If you do not define RDIR within the makefile, Make searches the current directory.
ODIR=*path*	Make searches the directory, specified by *path*, for all files that have no suffix or relative pathlist (object files). The default is the current execution directory.
CFLAGS=*options*	Make uses the specified compiler *options* to generate command lines.
RFLAGS=*options*	Make uses the specified assembler *options* to generate command lines.
LFLAGS=*options*	Make uses the specified linker *options* to generate command lines.

Reserved Macros

Make expands the following macros when a command line associated with a particular file dependency is forked. You might find these macros useful when you need to be explicit about a command line but have a target program with several dependencies. You can use these macros only in a makefile command.

Macro	Expands to:
$@	The name of the file to be made by the command
$*	The prefix of the file to be made
$?	The list of files that were found to be newer than the target file on a given dependency line

Commands

You can specify more than one command for any dependency. Make forks each command separately unless it is continued from the previous command (see Long Lines).

If you start a command line with the @ symbol, Make does not echo to standard output. If you start a command line with a hyphen (-), Make ignores any error codes returned on that line.

If your system runs out of memory while executing a command, you can redirect the output of Make into a procedure file and execute the procedure file.

Do not mix comments and commands.

Comments

You can specify an entire line as a comment by placing an asterisk (*) as the first character in that line. You can place comments at the end of a line by preceding the comment with the pound sign character (#).

Make ignores blank lines within a makefile.

Long Lines

If you use lines longer than 256 characters or lines wider than your screen, you need to place a space followed by a backslash (\) at the end of each line to be continued. The continuation line must have a space or tab as its first character.

For example:

```
Files : aaa.r bbb.r ccc.r ddd.r eee.r fff.r ggg.r \
  hhh.r iii.r jjj.r
```

Make ignores leading spaces and tabs on non-command lines and continuation lines.

How Make Works

Make starts by using the makefile to set up a table of dependencies. When Make encounters a name on the left side of a colon, Make first checks to see if it has encountered the name before. If Make has, it connects the lists and continues. It treats every item on the right side of the colon as a unique structure.

After reading the entire makefile, Make determines the target file (the main file to be made) on the list. It then makes a second pass through the subtable. It looks for object files that have no relocatable files in their dependency lists and for relocatable files that have no source files in their dependency lists.

If Make needs to find any source files or relocatable files to complete the dependency lists, it looks for them in the directory specified by the macros SDIR and RDIR (or RDIR's default .). Make looks in these directories for files with the same name as their dependent file. For example, if no source file is found for *program.r*, Make searches the specified directory (RDIR or .) for *program.a* (or .c, .p, .f).

Make does a third pass through the list to get the file dates and compare them. When Make finds a file that is newer than its dependent file, it generates the necessary command or executes the command given. Since OS-9 only stores the time down to the closest minute, Make remakes a file if its date matches one of its dependents.

> **Note:** When Make generates a command line for the linker, it looks at its list and uses the first relocatable file that it finds, but only the first one. For example:
>
> **prog: x.r y.r z.r**
>
> generates the following:
>
> **cc x.r**
>
> It does **not** generate **cc x.r y.r z.r** or **cc prog.r**

Notes about Make

If an object has more than one dependency, Make links the dependency lists together. If the first dependency lists multiple objects, then all the objects on that dependency line share the same set of dependencies. This might or might not be correct, depending on the situation. In the following example, the first makefile is correct, and the second one creates some extra dependencies:

First makefile:

```
x.r: defs.h
x.r y.r z.r: defs2.h
```

Second makefile:

```
x.r y.r z.r: defs2.h
x.r: defs.h
```

The first makefile specifies that *xr* is dependent on *defs.h* and *defs2.h*. It specifies *y.r* and *z.r* as dependent on *defs2.h*.

The second makefile specifies that all three .r files are dependent on *defs2.h*, and seems to specify only *x.r* as dependent on *defs.h*. Because the second makefile lists all three .r files on the same dependency line, they implicitly share in any future dependencies for any of the individual files. Therefore, *x.r*, *y.r*, and *z.r* are all implicitly dependent on *defs.h*.

> **Note:** The Make language is very specific. Therefore, you need to be careful when you use dummy files with names like print. Unless a file is specifically an object file or you use the -b option to turn off the implicit rules, use a suffix for your dummy files (i.e. print.file and xxx.h for header files).

Examples of Makefiles

Example 1

```
program: xxx.r yyy.r
  cc xxx.r yyy.r -xf=program
xxx.r yyy.r: /d0/defs/oskdefs
```

This example shows a shorter version of the makefile shown earlier in this chapter. This example makes use of Make's awareness of file dependencies. Because the makefile makes no mention of C-language files, Make looks in the directory specified by the macro definition SDIR=*path* (in this case, the default of the current directory) and adjusts the dependency list accordingly. Make also generates a command line to compile *xxx.r* and *yyy.r* if one or both need updating.

Example 2

```
program:
```

This simple makefile uses only one source file. Make assumes the following from this simple command:

1. Because *program* has no suffix, Make assumes that it is in an object file and therefore needs to rely on relocatable files to be made.
2. Because there is no dependency list given, Make creates an entry in the table for *program.r.*
3. After creating an entry for *program.r,* Make creates an entry for a source file connected to the relocatable file.

If Make finds the file *program.a,* it checks the dates on the various files and generates one or both of the following commands, if required:

```
rma program.a -o=program.r        ( + RFAGS if used)
cc program.r                      ( + LFLAGS if used)
```

Example 3

```
* beginning
ODIR = /d0/cmds
RDIR = rels
UTILS = attr copy load dir backup dsave
SDIR = ../utils/sources

utils.files:   $(UTILS)
   touch utils.files

*end
```

In this example, Make looks in the *rels* directory for *attr.r, copy.r, load.r,* etc and looks in *../utils/sources* for *attr.c, copy.c, load.c* and so on. Make then generates the proper commands to compile and/or link any of the programs that need to be made. If one of the files in the *utils* directory is made, then Make forks the command **touch util.files** to maintain a current overall date.

Example 4

```
* beginning
ODIR = /h0/cmds
RDIR = rels
CFILES = domake.c doname.c dodate.c domac.c
RFILES = domake.r doname.r dodate.r
R2 = ../test/domac.r
RFLAGS = -q
make:   (RFILES)     (R2)      getfd.r
   linker
$(RFILES): defs.h
$(R2): defs.h
   cc  $*.c -r=../test
print.file:     (CFILES)
   list $? >/p
   touch print.file
* end
```

This example is a makefile to create Make. This makefile looks for the .r files (listed in RFILES) in the directory specified by RDIR (rels). The only exception is *../test/domac.r*, which has a complete pathlist specified.

Even though *getfd.r* does not have any explicit dependents, Make checks its dependency on *getfd.a*. All of the source files are found in the current directory.

Notice that you can use this makefile to make listings as well. By typing **make print.file** on the command line, Make expands the macro $? to mean all of the files that were updated since the last time *print.file* was updated. If you keep a dummy file called *print.file* in your directory, it only prints out the newly made file. If no *print.file* exists, Make prints all the files.

Example 5

See the makefile in the SOURCES directory of Disk 2 in the OS-9 Level Two Development Pack. This complete makefile is for use with the updn.a and lsit.a examples in Chapter 11 of the "Relocatable Macro Assembler" section of this manual.

Chapter 3

Touch Utility

The Touch utility updates the last modification date of a file. This command is especially useful when used inside a makefile with Make. Associated with every file is the date that the file was last modified. The Touch utility simply opens a file and closes it, thereby updating the time that the file was last modified with the current date.

If Touch cannot find the specified file, it creates the file with the current date as the modification date.

The syntax for Touch is:

touch *options filename*

The *options* include any of the following:

-?	Displays the usage of Touch
-c	Does not create a file if Touch cannot find the specified file
-q	Does not stop execution if an error occurs
-x	Searches the execution directory for the file
-z	Reads the filenames from standard input
-z=*path*	Reads the filenames from *path*

Examples

touch -c /h0/doc/program

Touch searches for the specified file but does not create it if it does not exist.

touch -cz

Touch reads the filenames from standard input. If it cannot find a specified file, Touch does not create it. **[CTRL][BREAK]** at the beginning of a line signals Touch to terminage.

touch -z=filelist

Touch reads filenames from filelist, a file containing 1 filename on each line.

Virtual Disk/RAM Disk Driver

The Virtual Disk Driver is a high-speed, general storage/retrieval system that uses your computer's memory to simulate a fast disk device. You can use the VDD to store frequently used files (such as OS9DEFS) and programs to cut down on floppy disk access time. The Virtual Disk Driver uses two to six pages of system address space and allocates the amount of RAM specified in the descriptor (R0).

The VDD system consists of two modules: R0 (the VDD descriptor) and RAM (the driver).

Initializing VDD

You can initialize the Virtual Disk Driver by issuing an I$Attach call for R0 or by opening or creating a file on R0. You can also use INIZ to perform the I$Attach call. The syntax for INIZ is as follows:

```
iniz r0
```

> **Note:** Do not use I$Open and I$Create to initialize VDD even though they both do an implicit I$Attach, because the I$Close call does an implicit I$Detach. If an I$Attach call is not made before the file is opened, all data in the RAM disk is lost when the file is closed.

When VDD is initialized, it obtains information about the total amount of memory it is to allocate and the system memory block size from the descriptor. VDD then initializes Sector zero, the bit map, and the root directory. Once the Ram Disk is initialized, you can treat R0 like any other disk device.

R0 is a standard RBF device descriptor. You can choose the amount of RAM used by VDD by changing the default sectors per track (module offset $1B). To do so, use the debugger or reassemble R0 with the desired alteration. The size that VDD uses can be changed by altering the number of surfaces (module offset $19).

Your development diskettes contain three versions of R0, a 96 kilobyte version, a 128 kilobyte version, and a 192 kilobyte version. You can only use one version at a time.

Index

access time, disk 4-1
arguments, macros 2-1
assembler 2-4
  options 2-6

blank lines (Make) 2-8

CFLAGS (Make macro) 2-6
comments (Make) 2-8
compiler 2-4
  options (Make) 2-6
cross-compiler/assembler (Make) 2-2
create file (Touch) 3-1

date, modification 3-1
debugger (Make) 2-1
default
  assembler 2-4
  compiler 2-4
  directory 2-5
  linker 2-5
dependencies 2-3, 2-9
dependency, line 2-7
directory 2-5
  search 2-6
disk
  access time 4-1
  driver, virtual 4-1- 4-2
displaying commands (Make) 2-2
dollar sign (Make) 2-5

echoing commands (Make) 2-2
error (Touch) 3-1
error codes 2-7
errors, ignoring (Make) 2-2
execution directory, searching (Touch) 3-1

file
  dates 2-3
  dependencies 2-1, 2-3, 2-9
  prefix 2-7
file touching 2-2
filename 2-7
filenames (Touch) 3-1
files
  object 2-4
  relocatable 2-4
  source 2-4

initializing VDD 4-1

leading spaces (Make) 2-8
LFLAGS (Make macro) 2-6
line width 2-8
linker 2-5
  options (Make) 2-6

macros 2-2, 2-5 - 2-7
  arguments 2-1
  reserved 2-7
  special 2-6
maintain program 2-1 - 2-13

Make 2-1 - 2-13
  conventions 2-4 - 2-5
  debugger 2-1
  language 2-10
  macros 2-5 - 2-7
  options 2-1 - 2-2
  targets 2-2
makefile 2-2
  path 2-2
modification date
  Make 2-4
  Touch 3-1
modifications 2-1 - 2-13

object files 2-4
ODIR (Make macro) 2-6
options
  assembler 2-6
  Make 2-1 - 2-2
  compiler 2-6
  linker 2-6

procedure file 2-3
program maintenance 2-1 - 2-13

RAM driver 4-1 - 4-2
RDIR (Make macro) 2-6
redirecting output (Make) 2-7
relocatable files 2-4
reserved macros (Make) 2-7
RFLAGS (Make macro) 2-6

SDIR (Make macro) 2-6
searching the directory 2-6
source files 2-4

standard input
  Make 2-2
  Touch 3-1
standard output (Make) 2-7

target file, updating 2-3
Touch 3-1 - 3-2
touching files 2-2

virtual disk 4-1 - 4-2

Commands

Contents

Chapter 1 / Introduction 1-1

Chapter 2 / Command Reference 2-1
BINEX 2-1
DUMP 2-3
EXBIN 2-6
LOGIN 2-7
MODPATCH 2-10
MONTYPE 2-14
PARK 2-16
SAVE 2-18
SLEEP 2-19
TEE 2-21
TSMON 2-23
VERIFY 2-24

Chapter 1

Introduction

The CMDS directory of Disk 1 in the OS-9 Level Two Development System contains several commands to help in system operations. These commands and their functions are:

Command	Function
BINEX	Converts a binary file into an S-Record file
DUMP	Displays the physical data contents of a file or device in both ASCII and hexadecimal form
EXBIN	Converts an S-Record file into its binary form
LOGIN	Provides login security on timesharing systems
MODPATCH	Modifies modules residing in memory
MONTYPE	Sets a system for the specified type of monitor
PARK	Moves the heads of a hard disk in preparation for moving the drive unit
SAVE	Creates a file and writes a copy of the specified memory module(s) into the file
SLEEP	Suspends a process for a specified time
TSMON	Supervises idle terminals and initiates login
TEE	Copies standard input to multiple devices
VERIFY	Checks module header parity and CRC values

Chapter 2

Command Reference

BINEX

Syntax: binex *filename1 filename2*

Function: Converts a binary file into an S-Record file.

Parameters:

filename1	The name of the file to convert
filename2	The name of the file in which to store the converted code

Notes:

- Binex converts the specified OS-9 binary file (*filename1*) to an S-Record file and gives the new file the name specified by *filename2*. If filename1 is a non-binary load module file, OS-9 prints a warning message and asks you if BINEX should proceed anyway. Press Y to continue with the conversion. Pressing any other key causes BINEX to terminate.

- When you run BINEX, the program asks you for a program name and a starting load address. It stores this information in a header record. Although OS-9 is position independent and does not require absolute addresses, S-Record files do. The following example illustrates a BINEX command, its prompts, and possible user input.

  ```
  binex /d0/cmds/scanner scanner.s1 [ENTER]

  Enter starting address for file:
  $100 [ENTER]
  Enter name for header record:
  scanner [ENTER]
  ```

- To download the Scanner.s1 file to a device (such as a PROM programmer) using serial port /T1, type:

  ```
  list scanner.s1 >/t1 [ENTER]
  ```

- An S-Record is a type of text file that contains records representing binary data in hexadecimal character form. Most commercial PROM programmers, emulators, logic analyzers, and similar RS-232 devices can directly accept this Motorola-standard format. You can also use S-Record files to transmit data over data links that can only handle character-type data or to convert OS-9 assembler- or compiler-generated programs to load on non-OS-9 systems.

Example:

To convert a binary file named Zap to an S-Record file named Zap.sr, type:

```
binex /d0/cmds/zap /d1/sr/zap.sr
```

DUMP

Syntax: dump [*name*]

Function: Displays the physical data contents of the specified file or device in both ASCII and hexadecimal form .

Parameter:

name Either a file pathlist or a device name

Notes:

- If you do not specify a file or device, DUMP displays the standard input path (the keyboard). Dump writes output to the standard output path (the video display).
- Use DUMP to examine the contents of non-text files.
- The DUMP display adjusts to the type of screen you are using. In 32- and 40-column screens, DUMP displays eight bytes per line. In 80-column screens, DUMP displays 16 bytes per line.
- DUMP displays data in both hexadecimal and ASCII character format. If data bytes have non-displayable values, DUMP displays them as periods (.).
- The addresses displayed by DUMP are relative to the beginning of the file. Because memory modules are position-independent and are stored in files exactly as they exist in memory, the addresses shown on the dump correspond to the relative load addresses of memory-module files.

Examples:

To display keyboard input in hex on the screen, type the following command. Press [CTRL][BREAK] to return to the shell.

dump [ENTER]

Then, to display the contents of the diskette in Drive /D1, type:

dump @/d1 [ENTER]

The @ symbol causes OS-9 to treat the entire disk as a file.

Sample output, 32 columns:

DUMP SYS/password >/P [ENTER]

```
     0 1 2 3 4 5 6 7  0 2 4 6
ADDR 8 9 A B C D E F  8 A C E
==== +-+-+-+-+-+-+-+- + + + +
0000 2C2C302C3132382C ,,0,128,
0008 2F44302F434D4453 /D0/CMDS
0010 2C2D2C5348454C4C ,.,SHELL
0018 0D55534552312C2C .USER1,,
0020 312C3132382C2E2C 1,128,.,
0028 2E2C5348454C4C0D .,SHELL.
0030 55534552322C2C32 USER2,,2
0038 2C3132382C232C23 ,128,.,.

     0 1 2 3 4 5 6 7  0 2 4 6
ADDR 8 9 A B C D E F  8 A C E
==== +-+-+-+-+-+-+-+- + + + +
0040 2C5348454C4C0D55 ,SHELL.U
0048 534552332C2C332C SER3,,3,
0050 3132382C232C2E2C 128,.,.,
0058 5348454C4C0D5553 SHELL.US
0060 4552342C2C342C31 ER4,,4,1
0068 32382C2E2C2E2C53 28,.,.,S
0070 48454C4C0D       HELL.
```

The first column indicates the starting address. The next eight columns (00-EF) display data bytes in hexadecimal format. The final column (0-E) displays data byes in ASCII format. The display shows non-ASCII as periods in the ASCII character display section.

Sample output, 80-columns:

DUMP SYS/password >/P [ENTER]

```
ADDR  0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 2 4 6 8 A C E
---- ---- ---- ---- ---- ---- ---- ---- ----  ----------------
0000 2C2C 302C 3132 382C 2F44 302F 434D 4453  ,,0,128,/D0/CMDS
0010 2C2E 2C53 4845 4C4C 0D55 5345 5231 2C2C  ,.,SHELL.USER1,,
0020 312C 3132 382C 2E2C 2E2C 5348 454C 4C0D  1,128,.,.,SHELL.
0030 5553 4552 322C 2C32 2C31 3238 2C2E 2C2E  USER2,,2,128,.,.
0040 2C53 4845 4C4C 0D55 5345 5233 2C2C 332C  ,SHELL.USER3,,3,
0050 3132 382C 2E2C 2E2C 5348 454C 4C0D 5553  128,.,.,SHELL.US
0060 4552 342C 2C34 2C3A 3238 2C2E 2C2E 2C53  ER4,,4,128,.,.,S
0070 4845 4C4C 0D                             HELL.
```

EXBIN

Syntax: exbin *filename1 filename2*

Function: Converts an S-Record file into its binary form

Parameters:

filename1 The name of the file to convert

filename2 The name of the file in which to store the converted code

Notes:

- EXBIN is the inverse operation of BINEX. It assumes the file specified by *filename1* is an S-Record format text file and converts it to a pure binary form in the file specified by *filename2*. The load addresses of each data record must describe contiguous data in ascending order.
- EXBIN does not generate or check for the proper OS-9 module headers, the header CRC check value, or the module CRC check value required to load the binary file. Use the IDENT or VERIFY commands to check the validity of the modules.

Examples:

- To convert an S-Record file named Program.s1 to a binary file named Program and store it in the commands file of the current diskette, type:

  exbin program.s1 cmds/program [ENTER]

LOGIN

Syntax: login

Function: Provides login security on timesharing systems. LOGIN automatically adjusts its output for 32- or 80-column displays.

Parameters: None

Notes:

- The timesharing monitor, TSMON, automatically calls LOGIN. You can also use LOGIN after initial login to change a terminal's user.
- LOGIN requests your name and password, which it checks against a validation file. If the information is correct, LOGIN sets up your system priority, ID, and working directories according to information stored in the file. Then, LOGIN executes the initial program (usually shell) specified in the password file.
- The LOGIN process terminates if you cannot supply a correct user name and password after three attempts.
- The validation file is /DD/SYS/password. The file contains one or more variable-length text records, one for each user name. Each record has the following fields (the file uses commas as delimiters):

  User name. The name can be a maximum of 32 characters, including spaces. If the name field is empty, any name matches.

  Password. The password can be a maximum of 32 characters, including spaces. If the password field is blank, the system does not require the record's owner to type a password.

User index. This is the user ID number. It can be in the range 0 to 65535 (0 is the *superuser* or system manager). Both the file security system and the system-wide user ID use this number to identify all processes initiated by the user. The system manager should assign a unique ID to each potential user.

Priority. This is the initial process (CPU time) priority. It can be in the range of 1 to 255.

Execution Directory. This is a pathlist showing the name and location of the initial execution directory (usually /D0/CMDS).

Working Directory. This is a pathlist showing the name and location of the initial data directory (the specific user's directory). The initial data directory is usually the ROOT directory.

Execution Program. This is the name of the initial program to execute (usually shell). Do not use shell command lines, such as DIR or DCHECK, as initial program names.

- Here is the system default validation file:

```
,,0,128,/D0/CMDS,.,SHELL
USER1,,1,128,.,.,SHELL
USER2,,2,128,.,.,SHELL
USER3,,3,128,.,.,SHELL
USER4,,4,128,.,.,SHELL
```

In this sample, the superuser's record, the first entry, contains no user name or password. The ID number is 0, the initial process priority is 128, the execution directory is /D0/CMDS, and the ROOT directory is the initial data directory. The initial program to execute is shell. The second entry is the same except the user's name is the default USER1.

- To use LOGIN, type:

  login [ENTER]

  Prompts ask for your name and (optionally) a password. If you answer correctly, the system completes your login. LOGIN initializes the user number, working execution directory, the working data directory, and executes a specified program. It displays the date, time, and process number. LOGIN adjusts its output format for 80- or 32-column displays.

- To kill the shell that called LOGIN, use EX. For example:

  ex login [ENTER]

- Use the OS-9 text editor to edit Password and add users.

- Logging off the system terminates the program specified in the password file. For most programs (including shell) logging off involves typing an end-of-file character (**[CTRL][BREAK]**) as the first character on a line.

- If Motd exists in the SYS directory, LOGIN displays its contents (after a successful login).

Examples:

Following is possible user input and the screen display during LOGIN.

[ENTER]

OS-9 Timesharing system
Level II RS VR. 02.00.01
87/04/10 08:35:44

User name?: superuser [ENTER]
Password: secret [ENTER] (your entry does not appear on the screen)

Process #07 logged on 87/04/10 08:36:01
Welcome!

LOGIN then displays a message of the day from the Motd file.

MODPATCH

Syntax: modpatch [*options*] *filename* [*options*]

Function: modifies modules residing in memory. MODPATCH reads a file and executes the commands in the file to change the contents of one or more modules.

Parameters:

filename The name of a file containing instructions for MODPATCH

options One of the following options that change MODPATCH's function

Options:

-s Silent mode, does not display patchfile command lines as they are executed.

-w Does not display warnings, if any

-c Compares only, does not change the module

Notes:

- Before using MODPATCH, you must create a patchfile to supply the data to control MODPATCH's operation. This file contains single-letter commands and the appropriate module addresses. The commands are:

l *modulename*	Link to the module specified by *modulename*.
c *offset origval newval*	Change the byte at the offset address specified by *offset* from the value specified by *origval* to the new value specified by *newval*. If the original value does not match origval, MODPATCH displays a message.
v	Verify the module--update the modules CRC . If you plan to save the patched module to a file that the system can load, you must use this command.
m	Mask IRQ's . Turns off interrupt requests (for patching service routines).
u	Unmask IRQ's. Turns on interrupt requests (for patching service routines).

- You can use the BUILD command or any word processing program to create patchfiles.
- Module byte addresses begin at 0. MODPATCH changes values pointed to by an offset address (offset from 0) rather than an absolute memory address.

- To view the contents of a memory module, use SAVE and DUMP to copy the module to a file and display its contents. Also use SAVE to copy the patched module to a disk file.

- Changing a memory module might not produce an immediate effect. You have to duplicate the initialization procedure for that module. This means, if the module loads during bootup, you have to create a new boot file that includes the changed module, then reboot using the new boot file.

- To use the patched module in future system boots, use SAVE to store the module in the MODULES directory of your system disk. You can then use OS9GEN to create a new system disk using the patched module. If you are using the patched module to replace another module, rename the original module and then give the patched module the original name.

- If you patch a module that is loaded during the system boot, you can use COBBLER to make a new system boot that uses the patched module.

Examples:

The following example shows the commands, the screen prompts, and the entries you make to patch the standard 40-column term window descriptor to be an 80-column screen rather than the standard 40-column screen:

```
OS9: build termpatch [ENTER]
? l term [ENTER]
? c 002c 28 50 [ENTER]
? c 0030 01 02
? v [ENTER]
? [ENTER]

OS9: modpatch termpatch [ENTER]
```

To change the size, columns, and colors of Device Window W1, create the following procedure file and name it W180:

```
l w1
c 0030 01 02
c 002c 1b 50
c 002d 0b 18
```

If the W1 module is not already in memory, load it from the MODULES directory of your system disk. Then, before initializing W1, run MODPATCH:

modpatch w180 [ENTER]

Next, initialize W1:

iniz w1 [ENTER]
shell i=/w1& [ENTER]

Press **[CLEAR]** to display the new window with 80 columns, 24 lines, and a white background.

MONTYPE

Syntax: montype *type*

Function: Sets your system for the type of monitor you are using

Parameters:

type A single letter indicating the monitor type:

- c for composite monitors or color televisions
- r for RGB monitors
- m for monochrome monitors or black and white televisions

Notes:

- Different types of color monitors display colors differently. For the best results, set your system to the type of monitor you are using.
- If you are using a monochrome monitor or black and white television, you can obtain a sharper image by setting your monitor type to monochrome.
- Include the MONTYPE command in your system's Startup file to automatically boot in the proper monitor mode.
- If you do not use MONTYPE, the system defaults to c (composite monitor).

Example:

To set your system for an RGB monitor, type:

montype r [ENTER]

To add a MONTYPE command to your existing Startup file, first use BUILD to create the new command. For example:

```
build temp [ENTER]
montype r [ENTER]
[ENTER]
```

Next, append the file to Startup. Type:

```
merge startup temp > startup.new [ENTER]
```

Delete the temp file:

```
del temp [ENTER]
```

To enable the system to use Startup.new when booting, rename the original Startup file:

```
rename Startup Startup.old
```

Then rename Startup.new:

```
rename Startup.new Startup
```

PARK

Syntax: park *drive*

Function: Moves the heads of a hard disk to the innermost tracks in preparation for moving the drive unit.

Parameters:

drive	The hard disk drive for which you want to park the heads

Notes:

- Jarring your hard disk can cause its recording heads to bump against the highly polished surface of the recording media, destroying stored data. Such jarring can easily happen when you move your hard disk drive.

  PARK moves all of your disk's recording heads onto the innermost tracks where information is not stored, and where such inadvertent bumping cannot destroy data.

- Always use PARK before relocating your hard disk or anytime you think it might be bumped or jiggled.
- After running PARK, turn off the system. Wait at least 15 seconds before turning on the power again. When you do turn your system on, the hard disk is immediately ready for use.
- Your hard disk is a precision instrument, built to extremely close tolerances. Always handle it carefully, even after parking its heads.

Example:

To park the heads of your hard disk, type:

park /h0 [ENTER]

SAVE

Syntax: save *filename modname* [...]

Function: Creates a file and writes a copy of the specified memory module(s) into the file

Parameters:

filename Is the name of the file you want to create

modname Secifies one or more modules to include in the file

Notes:

- The module name(s) must exist in the module directory when SAVEd. SAVE gives the new file all access permission except public write.
- SAVE's default directory is the current data directory. Generally, you should save executable modules in the default execution directory.
- You can use SAVE to create a file of the commands you use most often so that you can load all of these commands using only one filename.

Examples:

To save a module named Wcount into a newly created file called Workcount in the /D0/CMDS directory, type:

save /d0/cmds/workcount wcount [ENTER]

The following command saves four modules (add, sub, mul and div) into the new file called /D1/Math_pack.

save /d1/math_pack add sub mul div [ENTER]

SLEEP

Syntax: sleep *tickcount*

Function: Puts a process to *sleep* for the specified number of clock ticks

Parameters:

Tickcount Can be any number in the range 1 to 65535

Notes:

- If you give SLEEP a value larger than 65535, OS-9 reduces the value by mod 65536. For example, 65536, and all the multiples of 65536, become 0. A tick count of 95000 becomes an actual tick count of 29464.

  In other words, if you give SLEEP a value higher than 65535, it reduces tickcount by subtracting the closest multiple of 65536 that is lower than your value.

- Use SLEEP to generate time delays or to *break up* jobs requiring a large amount of CPU time. The duration of a tick is 16.66 milliseconds.

- A tick count of 1 causes the process to *give up* its current time slice. A tick count of 0 causes the process to sleep indefinitely. (A signal sent to the process awakens it.)

Examples:

The following command puts the process *to sleep* for 25 ticks (416.50 milliseconds):

sleep 25 [ENTER]

The following command sequence causes LIST to start running as a child process invoked from the shell, and as a background task. SLEEP then puts the shell to sleep indefinitely. When LIST attempts to find the file Nothing, which does not exist, it terminates and sends a signal (the error status), which wakes up the shell.

```
list startup sys/motd nothing & sleep 0
```

A sample screen display follows:

```
&004
setime </term

WELCOME TO COLOR COMPUTER OS-9
-004
ERROR #216

OS9:
```

If an error does not occur, the shell continues to sleep. (Use [BREAK] to wake the shell. Any keys you pressed while the shell was asleep are then displayed.

TEE

Syntax: tee *pathlist* or *devname* [...]

Function: Copies standard input to multiple devices

Parameters:

pathlist Is one or more paths for the input data to follow

devname Is one or more devices to which the system directs the input data

Options: TEE can send output to any number of devices specified by *devname*.

Notes: TEE is a filter that copies all text lines from its standard input path to the specified output paths.

Examples:

The following command line uses a pipeline and TEE to send the output listing of DIR simultaneously to the terminal, the printer, and a disk file:

```
dir e ! tee />p /d0/dir.listing
```

Here, a pipeline takes the output of DIR E and sends it to the terminal and TEE. TEE in turn sends the output to the printer and to a file called /D0/Dir.listing.

In the following example, the pipeline and TEE send the output of an assembler listing to a file (Pgm.list) and to the printer.

asm pgm.src l ! tee pgm.list /p [ENTER]

The next example broadcasts a message to the terminal.

echo WARNING SYSTEM DOWN IN 10 MINUTES ! tee />t1 [ENTER]

TSMON

Syntax: tsmon [*devname*]

Function: Supervises idle terminals and initiates the login sequence for timesharing applications

Parameter:

devname Is the device for which you want login and supervision capabilities

Notes:

- If you specify a devname, TSMON opens standard I/O paths for that device. When you enter a carriage return, TSMON automatically calls the LOGIN command. If the LOGIN fails because the user cannot supply a valid user name or password, control returns to TSMON. The LOGIN command and its password file must be present for TSMON to work correctly. (See the LOGIN command description.)
- Logging Off the System: Most programs terminate when you enter an end-of-file marker (**[CTRL][BREAK]**) as the first character on a command line. Pressing **[CTRL][BREAK]** causes your terminal to log off the system and to return to TSMON. TSMON runs the login sequence again when you press **[ENTER]**.

Examples:

The following command line activates /T1.

tsmon /t1& [ENTER]

The command must run concurrently in order to keep /TERM active.

VERIFY

Syntax: verify [u] <*filename1* [>*filename2*]

Function: Checks to see if the module header parity and CRC value of one or more modules on a file are correct

Parameters:

filename1	Is the name of the module to be checked
filename2	Is the name for the verified module created with the u option

Options:

u (update) copies the module(s) to a new module with the header and parity and CRC values replaced with VERIFY's computed values

Notes:

- VERIFY reads module(s) from the standard input and sends output to the standard output. It sends messages to the standard error path.
- VERIFY is dependent on the input redirection command. **If you fail to use the redirection symbol, VERIFY causes the system to lock.** To gain control of the system, press **[BREAK]**. You must always redirect the input path. If you use the u option, you must also redirect the output to the new file you want to create.
- Using the u (update) option causes VERIFY to copy the module(s) to the standard output path with the module's header parity and CRC values replaced with new computed values. VERIFY, with the update option, does not set the execute flag in the file attributes. Use ATTR to do this.

- If you do not use the u option, VERIFY does not copy the module to standard output. VERIFY displays a message indicating whether the module's header parity and CRC match those computed by VERIFY.

Examples:

Because the following command line uses the u option, VERIFY copies the edit module to a new module, Newedit, with the header parity and CRC values replaced with VERIFY's computed values.

```
verify u </d0/cmds/edit >/d0/cmds/newedit [ENTER]
```

The next command line checks the edit module. Because the command does not specify the u option, VERIFY only displays a summary message.

```
verify <edit [ENTER]
```

A possible screen display is:

```
Header parity is correct
CRC is correct
```

In the next command line, VERIFY checks Myprogram2, an invalid module. Because the command does not specify the u option, VERIFY does not copy the module to standard output, but displays a message.

```
verify <myprogram2 [ENTER]
```

The screen displays:

```
Header parity is INCORRECT!
CRC is INCORRECT!
```

Index

ASCII, file contents 2-3

binary file
  converting to S-Record 2-1 - 2-2
  converting from S-Record 2-6
BINEX command 2-1 - 2-2
BUILD command 2-11, 2-15

changing modules 2-10 - 2-13
clock ticks 2-19
COBBLER command 2-12
command list 1-1
commands, patchfile 2-11
contents, file 2-3 - 2-5
converting binary to S-Record/binary 2-1 - 2-2
converting S-Record to binary 2-6
CPU time 2-8
CRC value 2-24 - 2-25

devices, copying input to 2-21 - 2-22
DIR command 2-21
directory
  execution 2-8
  SYS 2-9
  working 2-8
DUMP command 2-3 - 2-5, 2-12

emulators 2-2
EX command 2-9
EXBIN command 2-6

execution
  directory 2-8
  program 2-8

file contents 2-3 - 2-5
File conversion, binary/S-Record 2-1 - 2-2
File, displaying contents 2-3 - 2-5
filter, TEE 2-21 - 2-22

hard disk, parking heads 2-16
hexadecimal form, file contents 2-3

input, standard 2-21 - 2-22

LIST command 2-20
logic analyzers 2-2
LOGIN command 2-23 , 2-7 - 2-9

memory modules, saving 2-18
modifying module 2-10 - 2-13
module header parity, checking 2-24 - 2-25
modules
  modifying 2-10 - 2-13
  saving from memory 2-18
  updating 2-24
MODULES directory 2-12
monitor, setting type 2-14 - 2-15
monochrome monitors 2-14
MONTYPE command 2-14 - 2-15

non-text files 2-3

OS9GEN command 2-12

parity, module header 2-24 - 2-25
PARK command 2-16 - 2-17
password (LOGIN) 2-7

patchfile commands 2-11
paths 2-21 - 2-22
pipeline (TEE) 2-21 - 2-22
priority 2-8
process, putting to sleep 2-19
PROM programmers 2-2

reference, commands 1-1
RGB monitors 2-14

S-Record
  from binary 2-1 - 2-2
  to binary 2-6
SAVE command 2-12, 2-18
saving memory module 2-18
security, login 2-7 - 2-9
shell, killing 2-9
SLEEP command 2-19 - 2-20
standard
  I/O paths 2-23
  input 2-21 - 2-22
Startup file 2-15
suspending processes 2-19
SYS directory 2-9

TEE command 2-21 - 2-22
television 2-14
terminal, supervising 2-23
tick count 2-19
timesharing 2-7 - 2-9, 2-23
TSMON command 2-23
types of monitor 2-14 - 2-15

updating modules 2-24
user index (LOGIN) 2-8

validation file (LOGIN) 2-7
VERIFY command 2-24 - 2-25

working directory 2-8

Technical Reference

OS-9 Technical Reference

Contents

Chapter 1 System Organization 1-1
I/O System Modules 1-1
Color Computer OS-9 Modules 1-2
Kernel, Clock Module, and INIT 1-2
Input/Output Modules 1-3
I/O Manager 1-3
File Managers 1-3
Device Drivers 1-3
Device Descriptors 1-4
Shell 1-4

Chapter 2 The Kernel 2-1
System Initialization 2-1
System Call Processing 2-4
OS9Defs and Symbolic Names 2-4
Types of System Calls 2-4
Memory Management 2-5
Memory Use 2-5
Color Computer OS-9 Typical Memory Map 2-7
Memory Management Hardware 2-7
Multiprogramming 2-12
Process Creation 2-12
Process States 2-13
Execution Scheduling 2-14
Signals 2-15
Interrupt Processing 2-16
Logical Interrupt Polling System 2-17
Virtual Interrupt Processing 2-19

Chapter 3 Memory Modules 3-1
Module Types 3-1
Module Format 3-1
Module Header 3-2
Module Body 3-2
CRC Value 3-2
Module Headers: Standard Information 3-3
Sync Bytes 3-3
Module Size 3-3
Offset to Module Name 3-3
Type/Language Byte 3-4
Attributes/Revision Level Byte 3-4
Header Check 3-5

Module Headers: Type-Dependent Information 3-5
Executable Memory Module Format 3-6

Chapter 4 OS-9's Unified Input/Output System 4-1
I/O System Modules 4-1
The I/O Manager 4-2
File Managers 4-3
File Manager Structure 4-3
Create, Open 4-4
Makdir 4-4
ChgDir 4-4
Delete 4-5
Seek 4-5
Read 4-5
Write 4-6
ReadLn 4-6
WriteLn 4-6
GetStat, PutStat 4-6
Close 4-7
Interfacing with Device Drivers 4-7
Device Driver Modules 4-8
Device Driver Module Format 4-10
OS-9 Interaction With Devices 4-11
Suspend State (Level Two Only) 4-13
Device Descriptor Modules 4-15
Path Descriptors 4-18

Chapter 5 Random Block File Manager 5-1
Logical and Physical Disk Organization 5-1
Identification Sector (LSN 0) 5-2
Disk Allocation Map Sector (LSN 1) 5-3
ROOT Directory 5-3
File Descriptor Sector 5-3
Directories 5-5
The RBF Manager Definitions of the Path Descriptor .. 5-5
RBF-Type Device Descriptor Modules 5-8
RBF Record Locking 5-10
Record Locking and Unlocking 5-11
Non-Shareable Files 5-12
End-of-File Lock 5-12
Deadlock Detection 5-13
RBF-Type Device Driver Modules 5-13
The RBF Device Memory Area Definitions 5-13
RBF Device Driver Subroutines 5-16

Chapter 6 Sequential Character File Manager 6-1
SCF Line Editing Functions 6-1
Read and Write 6-1
Read Line and Write Line 6-2
SCF Definitions of the Path Descriptor 6-2
SCF-Type Device Descriptor Modules 6-6
SCF-Type Device Driver Modules 6-9
SCF Device Driver Subroutines 6-10

Chapter 7 The Pipe File Manager (PIPEMAN) 7-1

Chapter 8 System Calls 8-1
Calling Procedure 8-1
I/O System Calls 8-2
System Call Descriptions 8-2
User Mode System Calls Quick Reference 8-3
System Mode Calls Quick Reference 8-5
User System Calls 8-7
I/O User System Calls 8-44
Privileged System Mode Calls 8-66
Get Status System Calls 8-112
Set Status System Calls 8-130

Appendices A-1
A Memory Module Diagrams A-1
B Standard Floppy Disk Format B-1
C System Error Codes C-1

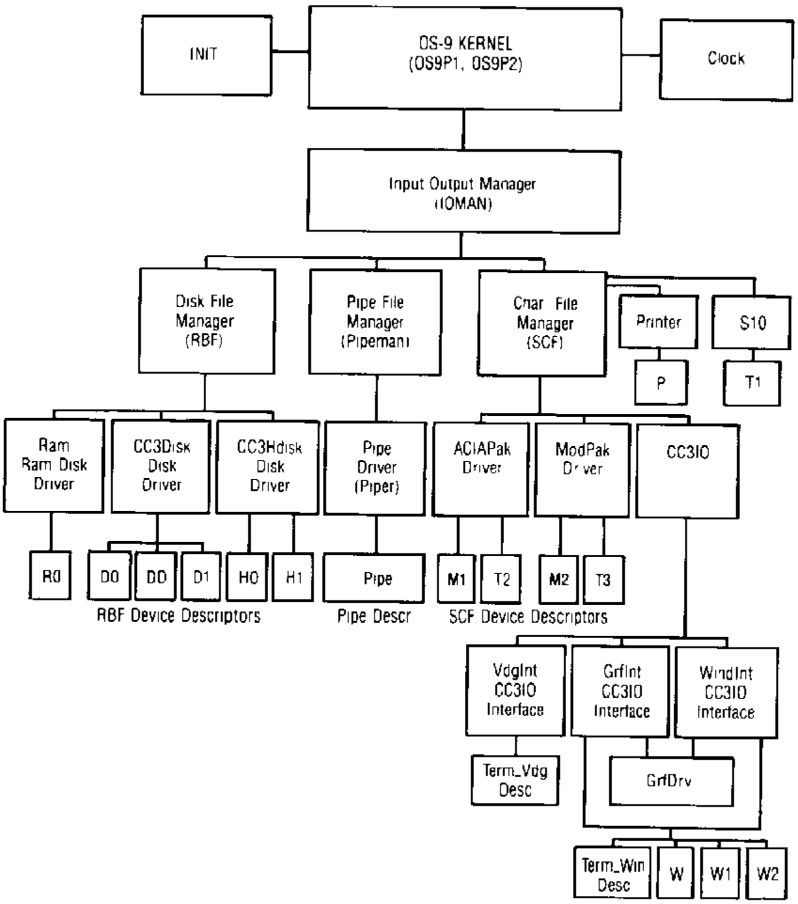
Index

Chapter 1

System Organization

OS-9 is composed of a group of modules, each of which has a specific function. The following illustration shows the major modules. Actual module names are capitalized.

I/O System Modules



OS-9 COMPONENT MODULE ORGANIZATION

Color Computer OS-9 Modules

IOMAN	Input/output management
INIT	System initialization table
CLOCK	Software routine time management
RBF	Random block file management
SCF	Sequential character file management
PIPEMAN	Pipe file management
CC3DISK	Color Computer disk driver
CC3IO	Color Computer input/output driver

The VDGINT (video display generator) provides both interface functions and low-level routines for Color Computer 2 VDG compatibility.

The GRFINT interface provides high-level graphics code interpretation and interface functions.

The WINDINT interface contains all the functions of GRFINT, along with additional support for Multiview functions. If you are using Multiview, exclude GRFINT from the system.

Both WINDINT and GRFINT use the low-level driver GRFDRV to perform the actual drawing on bitmap screens.

Term_VDG uses CC3IO and VDGINT. TERM_WIN and all window descriptors (W, W1, W2, and so on) use CC3IO, WINDINT, GRFINT, and GRFDRV modules.

Kernel, Clock Module, and INIT

The system's first level contains the *kernel, clock module,* and *INIT*.

The kernel provides basic system services, such as multitasking and memory management. It links all other OS-9 modules into the system.

The clock module is a software handler for the real-time clock hardware.

INIT is an initialization table used by the kernel during system startup. This table loads initial tasks and specifies initial table sizes and initial system device names. It is loaded into RAM (random access memory) by the OS-9 bootstrap module Boot. Boot also loads the OS9P2 and INIT modules during system startup.

There are two ways to run boot:

- Using the DOS command with Color Disk BASIC, Version 1.1, or later.
- Pressing the reset button after OS-9 is running.

Input/Output Modules

The remaining modules make up the OS-9 I/O system. They are defined briefly here and are discussed in detail in Chapter 4.

I/O Manager

The system's second level (the level below the kernel) contains the input/output manager, IOMAN. The I/O manager provides common processing for all input/output operations. It is required for performing any input/output supported by OS-9.

File Managers

The system's third level contains the *file managers*. File managers perform I/O request processing for similar classes of I/O devices. There are three file managers:

RBF manager — The random block file manager processes all disk I/O operations.

SCF manager — The sequential character file manager handles all non-disk I/O operations that operate one character at a time. These operations include terminal and printer I/O.

PIPEMAN — The pipe file manager handles *pipes*. Pipes are memory buffers that act as files. Pipes are used for data transfers between processes.

Device Drivers

The system's fourth level contains the *device drivers*. Device drivers handle basic I/O functions for specific I/O controller hardware. You can use pre-written drivers, or you can write your own.

Device Descriptors

The system's fifth level contains the *device descriptors*. Device descriptors are small tables that define the logical name, device driver, and file manager for each I/O port. They also contain port initialization and port address information. Device descriptors require only one copy of each I/O controller driver used.

Shell

The shell is the command interpreter. It is a program and not a part of the operating system. The shell is fully described in the *OS-9 Commands* manual.

Chapter 2

The Kernel

The kernel is the core of OS-9. The kernel supervises the system and manages system resources. Half of the kernel (called OS9P1) resides in the boot module. The other half of the kernel (called OS9P2) is loaded into RAM with the other OS-9 modules.

The kernel's main functions are:

- System initialization after reset
- Service request processing
- Memory management
- Multiprogramming management
- Interrupt processing

I/O functions are not included in the list because the kernel does not directly process them. Instead, it passes I/O system calls to the I/O Manager for processing.

System Initialization

After a hardware reset, the kernel initializes the system. This involves:

1. Locating modules loaded in memory from the OS-9 Boot file.
2. Determining the amount of available RAM.
3. Loading any required modules that were not loaded from the OS-9 Boot file.

OS-9 Level Two cannot install new system calls using the OS-9 Level One system call F$SSvc. F$SSvc does not work with a Level Two user program because of the separation of system and user address space.

OS9P3 can be used to tailor the system to fit specific needs. The following listing is an example of how to use the OS9P3 module.

```
Microware OS-9 Assembler 2.1  11/18/83   16:06:01   Page 001

OS-9 Level TWO V1.2, part 2 - OS-9 System Symbol Definitions

00001
00002
00003

00011  ******************************************************************
00012  *
00013  *              Module Header
00014  *
00015  00C1           Type     set  Systm+Objct
00016  0081           Revs     set  ReEnt+1
00017  0000 87CD005E           mod  OS9End,OS9Name,Type,Revs,Cold,256
00018  000D 4F533970  OS9Name  fcs  "OS9p3"
00019
00029  0012 01                 fcb  1  edition number
00030                          use  defsfile
00031  0002           level    equ  2
00032                          opt  -c
0033                           opt  f
00041
00042  ********************************************************
00043  *
00044  *              Routine Cold
00045  *
00046  *
00047
00048  0013 318D0004  Cold     leay SvcTbl,pcr   get service routine
00049  0017 103F32             OS9  F$SSvc   install new service
00050  001A 39                 rts
00051
00052
00053  ********************************************************
00054  *
00055  *              Service Routines Initialization Table
00056  *
00057
00058  0025           F$SAYHI    equ  $25 set up new call
00059  *      Add this to the user os9defs file.
00060
```

```
00061  001B            SvcTbl   equ   *
00062  001B  25                 fcb   F$SAYHI
00063  001C  0001               fdb   SayHi-*-2
00064  001E  80                 fcb   $80


Microware OS-9 Assembler 2.1  11/18/83   16:06:01   Page 002
OS-9 Level TWO V1.2, part 2 - OS-9 System Symbol Definitions

00068        *
00069        *Service call Say Hello to user
00070        *
00071        *Input:  U = Registers ptr
00072        *             R$X,u = Message ptr (if 0 send default)
00073        *             Max message length = 40 bytes.
00074        *
00075        *Output:  Message sent to standard error path of user.
00076        *
00077        *Data:    D.Proc
00078        *
00079
00080  001F  AE44        SayHi    ldx   R$X,u      get mess. address
00081  0021  2619                 bne   SayHi6     bra if not default
00082  0023  109E50               ldy   D.Proc     get proc descr ptr
00083  0026  EE24                 ldu   P$SP,y     get caller's stack
00084  0028  33C8D8               leau  -40,u      room for message
00085  002B  96D0                 lda   D.SysTsk   system's task num
00086  002D  E626                 ldb   P$TASK,y   caller's task num
00087  002F  108E0028             ldy   #40        set byte count
00088  0033  308D0012             leax  Hello,pcr  destination ptr
00089  0037  103F38               OS9   F$Move     mess into user mem
00090  003A  30C4                 leax  0,u
00091  003C  108E0028    SayHi6   ldy   #40        get max byte count
00092  0040  DE50                 ldu   D.Proc     get proc desc ptr
00093  0042  A6C832               lda   P$PATH+2,u path num of stderr
00094  0045  103F8C               OS9   I$WritLn   write mess line
00095  0048  39                   rts
00096
00097  0049  48656C6C    hello    fcc   "Hello there user."
00098  005A  0D                   fcb   $D
00099
00100  005B  5104B6               emod        module CRC
00101
```

```
00102  005E            OS9End    equ    *
00103
00104                                   end

00000 error(s)
00000 warning(s)
$005E 00094 program bytes generated
$0000 00000 data bytes allocated
$2884 00372 bytes used for symbols
```

System Call Processing

System calls are used to communicate between OS-9 and assembly-language programs for such functions as memory allocation and process creation. In addition to I/O and memory management functions, system calls have other functions. These include interprocess control and timekeeping.

System calls use the SWI2 instruction followed by a constant byte representing the code. You usually pass parameters for system calls in the 6809 registers.

OS9Defs and Symbolic Names

A system-wide assembly-language *equate file*, called OS9Defs, defines symbolic names for all system calls. This file is included when assembling hand-written or compiler-generated code. The OS-9 assembler has a built-in *macro* to generate system calls. For example:

```
OS9 I$Read
```

is recognized and assembled as equivalent to:

```
SWI2
FCB I$Read
```

The OS-9 assembly macro OS9 generates an SWI2 function. The label I$Read is the label for the code $89.

Types of System Calls

System calls are divided into two categories, *I/O calls* and *function calls*.

I/O calls perform various input/output functions. The kernel passes calls of this type to the I/O manager for processing. The symbolic names for I/O calls begin with I$. For example, the Read system call is called I$Read.

Function calls perform memory management, multi-programming, and other functions. Most are processed by the kernel. The symbolic names for function calls begin with F$. For example, the Link function call is called F$Link.

The function calls include *user calls* and privileged *system mode* calls. (See Chapter 8, "System Calls", for more information.)

Memory Management

Memory management is an important operating system function. Using memory modules, OS-9 manages the logical contents of memory and the physical assignment of memory to programs.

All programs that are loaded must be in the memory module format. This format allows OS-9 to maintain a module directory of all the programs in memory. The directory contains information about each module, including its name and address and the number of processes using it. The number of processes using a module is called the module's *link count.*

When a module's link count is zero, OS-9 deallocates its part of memory and removes its name from the module directory.

Memory modules are the foundation of OS-9's modular software environment. Advantages of memory management are:

- Automatic runtime linking of programs to libraries of utility modules
- Automatic sharing of re-entrant programs
- Replacement of small sections of large programs into memory for update or correction

Memory Use

OS-9 reserves some space at the top and bottom of RAM for its own use. The amount depends on the sizes of system tables that are specified in the INIT module.

OS-9 pools all other RAM into a free memory space. As the system allocates or deallocates memory, it dynamically takes it from or returns it to this pool. RAM does not need to be contiguous because the memory management unit can dynamically rearrange memory addresses.

The basic unit of memory allocation is the 256-byte *page*. OS-9 always allocates memory in whole numbers of pages.

The data structure that OS-9 Level Two uses to keep track of memory allocation is a 256-byte *bit map*. Each bit in this table is associated with a specific page of memory. A cleared bit indicates that the page is free and available for assignment. A set bit indicates that the page is in use (that no RAM is free at that address). OS-9 Level Two always allocates memory in 8192-byte increments. This is the smallest memory block that the memory management hardware supports.

OS-9 automatically allocates memory when any of the following occurs:

- Program modules are loaded into RAM
- Processes are created
- Processes execute system calls to request additional RAM
- OS-9 needs I/O buffers or larger tables

OS-9 also has inverse functions to deallocate memory allocated to program modules, new processes, buffers, and tables.

In general, memory for program modules and buffers is allocated from high addresses downward. Memory for process data areas is allocated from low addresses upward.

Following, is a memory map of a typical system. Actual memory sizes and addresses can vary depending on the exact system configuration.

Color Computer OS-9 Typical Memory Map

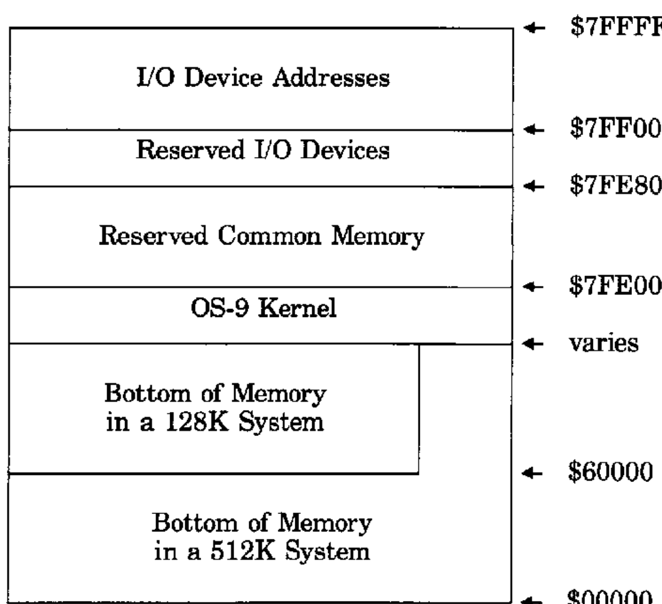


Figure 2.1

Note: The high two pages of every logical address space contain the defined areas I/O Device Addresses, Reserved I/O Devices, and Reserved Common Memory.

Memory Management Hardware

The 8-bit CPU in the Color Computer 3 can directly address only 64 kilobytes of memory using its 16 address lines (A0-A15). The Color Computer 3's Memory Management Unit (MMU) extends the addressing capability of the computer by increasing the address lines to 19 (A0-A18). This lets the computer address up to 512 kilobytes of memory ($0-$7FFFF).

The 512K address space is called the *physical address space*. The physical address space is subdivided into 8K *blocks*. The six high order address bits (A13-A18) define a *block number*.

OS-9 creates a *logical address space* of up to 64K for each task by using the FORK system call. Even though the memory within a logical address space appears to be contiguous, it might not be—the MMU translates the physical addresses to access available memory. Address spaces can also contain blocks of memory that are common to more than one map.

The MMU consists of a multiplexer and a 16 by 6-bit RAM array. Each of the 6-bit elements in this array is an MMU task register. The computer uses these task registers to determine the proper 8-kilobyte memory segment to address.

The MMU task registers are loaded with addressing data by the CPU. This data indicates the actual location of each 8-kilobyte segment of the current system memory. The task registers are divided into two sets consisting of eight registers each. Whether the task register select bit (TR bit) is set or reset, determines which of the two sets is to be used.

The relation between the data in the task register and the generated addresses is as follows:

Bit	D5	D4	D3	D2	D1	D0
Corresponding Memory Address	A18	A17	A16	A15	A14	A13

Figure 2.2

When the CPU accesses any memory outside the I/O and control range (XFF00=XFFFF), the CPU address lines (A13-A15) and the TR bit determine what segment of memory to address. This is done through the multiplexer when SELECT is low. (See the following table.)

When the CPU writes data to the MMU, A0-A3 determine the location of the MMU register to receive the incoming data when SELECT is high. The following diagram illustrates the operation of the Color Computer 3's memory management:

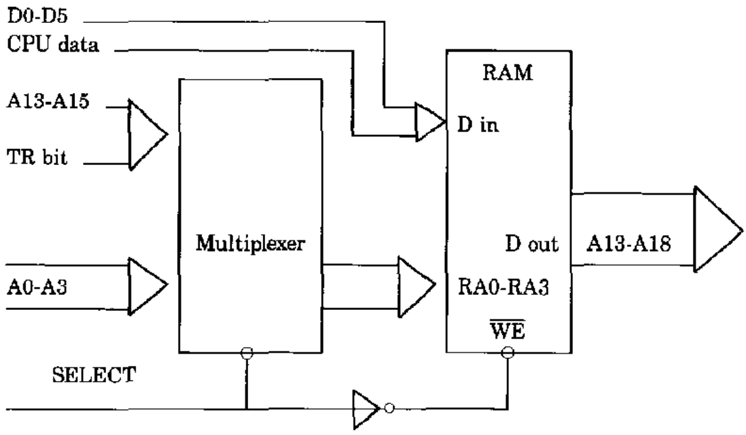


Figure 2.3

The system uses the data from the MMU registers to determine the block of memory to be accessed, according to the following table:

TR Bit	A15	A14	A13	AddressRange	MMU Address
0	0	0	0	X0000-X1FFF	FFA0
0	0	0	1	X2000-X3FFF	FFA1
0	0	1	0	X4000-X5FFF	FFA2
0	0	1	1	X6000-X7FFF	FFA3
0	1	0	0	X8000-X9FFF	FFA4
0	1	0	1	XA000-XBFFF	FFA5
0	1	1	0	XC000-XDFFF	FFA6
0	1	1	1	XE000-XFFFF	FFA7
1	0	0	0	X0000-X1FFF	FFA8
1	0	0	1	X2000-X3FFF	FFA9
1	0	1	0	X4000-X5FFF	FFAA
1	0	1	1	X6000-X7FFF	FFAB
1	1	0	0	X8000-X9FFF	FFAC
1	1	0	1	XA000-XBFFF	FFAD
1	1	1	0	XC000-XDFFF	FFAE
1	1	1	1	XE000-XFFFF	FFAF

Figure 2.4

The translation of physical address to 8K-blocks is as follows:

Range		Block	Range		Block
From	To	Number	From	To	Number
00000	01FFF	00	40000	41FFF	20
02000	03FFF	01	42000	43FFF	21
04000	05FFF	02	44000	45FFF	22
06000	07FFF	03	46000	47FFF	23
08000	09FFF	04	48000	49FFF	24
0A000	0BFFF	05	4A000	4BFFF	25
0C000	0DFFF	06	4C000	4DFFF	26
0E000	0FFFF	07	4E000	4FFFF	27
10000	11FFF	08	50000	51FFF	28
12000	13FFF	09	52000	53FFF	29
14000	15FFF	0A	54000	55FFF	2A
16000	17FFF	0B	56000	57FFF	2B
18000	19FFF	0C	58000	59FFF	2C
1A000	1BFFF	0D	5A000	5BFFF	2D
1C000	1DFFF	0E	5C000	5DFFF	2E
1E000	1FFFF	0F	5E000	5FFFF	2F
20000	21FFF	10	60000	61FFF	30
22000	23FFF	11	62000	63FFF	31
24000	25FFF	12	64000	65FFF	32
26000	27FFF	13	66000	67FFF	33
28000	29FFF	14	68000	69FFF	34
2A000	2BFFF	15	6A000	6BFFF	35
2C000	2DFFF	16	6C000	6DFFF	36
2E000	2FFFF	17	6E000	6FFFF	37
30000	31FFF	18	70000	71FFF	38
32000	33FFF	19	72000	73FFF	39
34000	35FFF	1A	74000	75FFF	3A
36000	37FFF	1B	76000	77FFF	3B
38000	39FFF	1C	78000	79FFF	3C
3A000	3BFFF	1D	7A000	7BFFF	3D
3C000	3DFFF	1E	7C000	7DFFF	3E
3E000	3FFFF	1F	7E000	7FFFF	3F

Figure 2.5

In order for the MMU to function, the TR bit at $FF90 must be cleared and the MMU must be enabled. However, before doing this, the address data for each memory segment must be loaded into the designated set of task registers. For example, to select a standard 64K map in the top range of the Color Computer 3's 512K RAM, with the TR bit set to 0, the following values must be preloaded into the MMU's registers:

MMU Location Address	Data (Hex)	Data (Bin)	Address Range
FFA0	38	111000	70000-71FFF
FFA1	39	111001	72000-73FFF
FFA2	3A	111010	74000-75FFF
FFA3	3B	111011	76000-77FFF
FFA4	3C	111100	78000-79FFF
FFA5	3D	111101	7A000-7BFFF
FFA6	3E	111110	7C000-7DFFF
FFA7	3F	111111	7E000-7FFFF

Figure 2.6

Although this table shows MMU data in the range $38 to 3F, any data between $0 and $3F can be loaded into the MMU registers to select memory addresses in the range 0 to $7FFFF, as illustrated by Figure 2.5.

Normally, the blocks containing I/O devices are kept in the system map, but not in the user maps. This is appropriate for time-sharing applications, but not for process control. To directly access I/O devices, use the F$MspBlk system call. This call takes a starting block number and block count, and maps them into *unallocated* spaces of the process's address space. The system call returns the logical address at which the blocks were inserted.

For example, suppose a display screen in your system is allocated at extended addresses $7A000-$7DFFF (blocks 3D and 3E). The following system call maps them into your address space:

```
ldb     #2          number of blocks
ldx     #3D         starting block number
os9     F$MapBlk    call MapBlk
stu     IOPorts     save address where mapped
```

On return, the U register contains the starting address at which the blocks were switched. For example, suppose that the call returned $4000. To access extended address $7A020, write to $4020.

Other system calls that copy data to or from one task's map to another are available, such as F$STABX and F$Move. Some of these calls are system mode privileged. You can unprotect them by changing the appropriate bit in the corresponding entry of the system service request table and then making a new system boot with the patched table.

Multiprogramming

OS-9 is a multiprogramming operating system. This means that several independent programs called *processes* can be executed at the same time. By issuing the appropriate system call to OS-9, each process can have access to any system resource.

Multiprogramming functions use a hardware real-time clock. The clock generates interrupts 60 times per second, or one every 16.67 milliseconds. These interrupts are called ticks.

Processes that are not waiting for some event are called *active processes*. OS-9 runs active processes for a specific system-assigned period called a time slice. The number of time slices per minute during which a process is allowed to execute depends on a process's priority relative to all other active processes. Many OS-9 system calls are available to create, terminate, and control processes.

Process Creation

A process is created when an existing process executes a Fork system call (F$Fork). This call's main argument is the name of the program module that the new process is to execute first (the *primary module*).

Finding the Module. OS-9 first attempts to find the module in the module directory. If it does not find the module, OS-9 usually attempts to load into memory a mass-storage file in the execution directory, with the requested module name as a filename.

Assigning a Process Descriptor. Once OS-9 finds the module, it assigns the process a data structure called a *process descriptor*. This is a 64-byte package that contains information about the process, its state (see the following section "Process States"), memory allocations, priority, queue pointers, and so on. OS-9 automatically initializes and maintains the process descriptor. The process itself cannot access the descriptor; it has no need to do so.

Allocate RAM. The next step is to allocate RAM for the process. The primary module's header contains a storage size. OS-9 uses this size unless the Fork system call requests a larger area. OS-9 then attempts to allocate a memory area of the specified size from the free memory space. The memory space does not need to be contiguous.

Proceed or Terminate. If OS-9 can perform all of the previous steps, it adds the new process to the active process queue for execution scheduling. If it cannot, it terminates the creation; the process that originated the Fork is informed of the error.

Assign Process ID and User ID. OS-9 assigns the new process a unique number called a *process ID*. Other processes can communicate with the process by referring to its ID in various system calls.

The process also has a *user ID*, which is used to identify all processes and files belonging to a particular user. The user ID is inherited from the parent process.

Process Termination. A process terminates when it executes an Exit system call (F$Exit) or when it receives a *fatal* signal. The termination closes any open paths, deallocates memory used by the process, and unlinks its primary module.

Process States

At any instant a process can be in one of three states:

- Active—The process is ready for execution.
- Waiting—The process is suspended until a *child process* terminates or until it receives a signal. A child process is a process that is started (execution is begun by) another process—a *parent process*.

- Sleeping—The process is suspended for a specific period of time or until it receives a signal.

Each state has its own queue, a linked list of *descriptors* of processes in that state. To change a process's state, move its descriptor to another queue.

The Active State. Each active process is given a time slice for execution, according to its priority. The scheduler in the kernel ensures that all active processes, even those of low priority, get some CPU time.

The Wait State. This state is entered when a process executes a Wait system call (F$Wait). The process remains suspended until one of its *child* processes terminates or until it receives a *signal.* (See the "Signals" section later in this chapter.)

The Sleeping State. This state is entered when a process executes a Sleep system call (F$Sleep), which specifies the number of ticks for which the process is to remain suspended. The process remains asleep until the specified time has elapsed or until it receives a wakeup signal.

Execution Scheduling

The OS-9 scheduler uses an algorithm that ensures that all active processes get some execution time.

All active processes are members of the *active process queue*, which is kept sorted by process *age*. Age is the number of process switches that have occurred since the process's last time slice. When a process is moved to the active process queue from another queue, its age is set according to its priority—the higher the priority, the higher the age.

Whenever a new process becomes active, the ages of all other active processes increase by one time slice count. When the executing process's time slice has elapsed, the scheduler selects the next process to be executed (the one with the next highest age, the first one in the queue). At this time, the ages of all other active processes increase by one. Ages never go beyond 255.

A new active process that was terminated while in the system state is an exception. This process is given high priority because it is usually executing critical routines that affect shared system resources.

When there are no active processes, the kernel handles the next interrupt and then executes a CWA1 instruction. This procedure decreases interrupt latency time (the time it takes the system to process an interrupt).

Signals

A *signal* is an asynchronous control mechanism used for inter-process communication and control. It behaves like a software interrupt. It can cause a process to suspend a program, execute a specific routine, and then return to the interrupted program.

Signals can be sent from one process to another process by the Send system call (F$Send). Or, they can be sent from OS-9 service routines to a process.

A signal can convey status information in the form of a 1-byte numeric value. Some *signal codes* (values) are predefined, but you can define most. The signal codes are:

0	=	Kill (terminates the process, is non-interceptable)
1	=	Wakeup (wakes up a sleeping process)
2	=	Keyboard terminate
3	=	Keyboard interrupt
4	=	Window change
128-255	=	User defined

When a signal is sent to a process, the signal is saved in the process descriptor. If the process is in the sleeping or waiting state, it is changed to the active state. When the process gets its next time slice, the signal is processed.

What happens next depends on whether or not the process has set up a *signal intercept trap* (signal service routine) by executing an Intercept system call (F$Icpt).

If the process has set up a signal intercept trap, the process resumes execution at the address given in the Intercept call. The signal code passes to this routine. Terminate the routine with an RTI instruction to resume normal execution of the process.

Note: A wakeup signal activates a sleeping process. It sets a flag but ignores the call to branch to the intercept routine.

If it has not set up a signal intercept trap, the process is terminated immediately. It is also terminated if the signal code is zero. If the process is in the system mode, OS-9 defers the termination. The process dies upon return to the user state.

A process can have a signal pending (usually because the process has not been assigned a time slice since receiving the signal). If it does, and another process tries to send it another signal, the new signal is terminated, and the Send system call returns an error. To give the destination process time to process the pending signal, the sender needs to execute a Sleep system call for a few ticks before trying to send the signal again.

Interrupt Processing

Interrupt processing is another important function of the kernel. OS-9 sends each hardware interrupt to a specific address. This address, in turn, specifies the address of the device service routine to be executed. This is called *vectoring* the interrupt. The address that points to the routine is called the *vector*. It has the same name as the interrupt.

The SWI, SWI2, and SWI3 vectors point to routines that read the corresponding pseudo vector from the process's descriptor and dispatch to it. This is why the Set SWI system call (F$SSWI) is local to a process; it only changes a pseudo vector in the process descriptor.

Hardware Vector Table

Vector	Address
SWI3	$FFF2
SWI2	$FFF4
FIRQ	$FFF6
IRQ	$FFF8
SWI	$FFFA
NMI	$FFFC
RESTART	$FFFE

FIRQ Interrupt. The system uses the FIRQ interrupt. The FIRQ vector is not available to you. The FIRQ vector is reserved for future use. Only one FIRQ generating device can be in the system at a time.

Logical Interrupt Polling System

Because most OS-9 I/O devices use IRQ interrupts, OS-9 includes a sophisticated polling system. The IRQ polling system automatically identifies the source of the interrupt, and then executes its associated user- or system-defined service routine.

IRQ Interrupt. Most OS-9 I/O devices generate IRQ interrupts. The IRQ vector points to the real-time clock and the keyboard scanner routines. These routines, in turn, jump to a special IRQ polling system that determines the source of the interrupt. The polling system is discussed in the next section, "Logical Interrupt Polling System."

NMI Interrupt. The system uses the NMI interrupt. The NMI vector, which points to the disk driver interrupt service routine, is not available to you.

The Polling Table. The information required for IRQ polling is maintained in a data structure called the *IRQ polling table*. The table has a 9-byte entry for each device that might generate an IRQ interrupt. The table size is permanent and is defined by an initialization constant in the INIT module. Each entry in the polling table is given a number from 0 (lowest priority) to 255 (highest priority). In this way, the more important devices (those that have a higher interrupt frequency) can be polled before the less important ones.

Each entry has six variables:

Polling Address — Points to the status register of the device. The register must have a bit or bits that indicate if it is the source of an interrupt.

Flip Byte — Selects whether the bits in the device status register indicate active when set or active when cleared. If a bit in the flip byte is set, it indicates that the task is active whenever the corresponding bit in the status register is clear.

Mask Byte — Selects one or more interrupt request flag bits within the device status register. The bits identify the active task or device.

Service Routine Address — Points to the interrupt service routine for the device. You supply this address.

Static Storage Address — Points to the permanent storage area required by the device service routine. You supply this address.

Priority — Sets the order in which the devices are polled (a number from 0 to 255).

Polling the Entries. When an IRQ interrupt occurs, OS-9 enters the polling system via the corresponding RAM interrupt vector. It starts polling the devices in order of priority. OS-9 loads the status register address of each entry into Accumulator A, using the device address from the table.

OS-9 performs an exclusive-OR operation using the flip byte, followed by a logical-AND operation using the mask byte. If the result is non-zero, OS-9 assumes that the device is the source of the interrupt.

OS-9 reads the device memory address and service routine address from the table, and performs the interrupt service routine.

> **Note:** If you are writing your own device driver, terminate the interrupt service routine with an RTS instruction, **not** an RTI instruction.

Adding Entries to the Table. You can make entries to the IRQ (interrupt request) polling table by using the Set IRQ system call (F$IRQ). Set IRQ is a *privileged system call*, OS-9 can execute it only in the system mode. OS-9 is in system mode whenever it is running a device driver.

> **Note:** The code for the interrupt polling system is located in the I/O Manager module. The OS9P1 and OS9P2 modules contain the physical interrupt processing routines.

Virtual Interrupt Processing

A virtual IRQ, or VIRQ, is useful with devices in Multi-Pak expansion slots. Because of the absence of an IRQ line from the Multi-Pak interface, these devices cannot initiate physical interrupts. VIRQ enables these devices to act as if they were interrupt driven. Use VIRQ only with device driver and pseudo device driver modules. VIRQ is handled in the Clock module, which handles the VIRQ polling table and installs the F$VIRQ system call. Since the F$VIRQ system call is dependent on clock initialization, the CC3GO module forces the clock to start.

The virtual interrupt is set up so that a device can be interrupted at a given number of clock ticks. The interrupt can occur one time, or can be repeated as long as the device is used.

The F$VIRQ system call installs VIRQ in a table. This call requires specification of a 5-byte packet for use in the VIRQ table. This packet contains:

- Bytes for an actual counter
- A reset value for the counter
- A status byte that indicates whether a virtual interrupt has occurred and whether the VIRQ is to be re-installed in the table after being issued

F$VIRQ also specifies an initial tick count for the interrupt. The actual call is summarized here and is described in detail in Chapter 8.

Call:	OS9 F$VIRQ
Input:	(Y) = *address of 5-byte packet*
	(X) = 0 to delete entry, 1 to install entry
	(D) = *initial count value*
Output:	none
	(CC) carry set on error
	(IS) *appropriate error code*

The 5-byte packet is defined as follows:

Name	Offset	Function
Vi.Cnt	$0	Actual counter
Vi.Rst	$2	Reset value for counter
Vi.Stat	$4	Status byte

Two of the bits in the status byte are used. These are:

Bit 0 - set if VIRQ occurs
Bit 7 - set if a count reset is required

When making an F$VIRQ call, the packet might require initialization with a reset value. Bit 7 of the status byte must be either set or cleared to signify a reset of the counter or a one-time VIRQ call. The reset value does not need to be the same as the initial counter value. When OS-9 processes the call, it writes the packet address into the VIRQ table.

At each clock tick, OS-9 scans the VIRQ table and subtracts one from each timer value. When a timer count reaches zero, OS-9 performs the following actions:

1. Sets Bit 0 in the status byte. This specifies a Virtual IRQ.
2. Checks Bit 7 of the status byte for a count reset request.
3. If bit 7 is set, resets the count using the reset value. If bit 7 is reset, deletes the packet address from the VIRQ table.

When a counter reaches zero and makes a virtual interrupt request, OS-9 runs the standard interrupt polling routine and services the interrupt. Because of this, you must install entries on both the VIRQ and DIRQ polling tables whenever you are using a VIRQ.

Unless the device has an actual physical interrupt, install the device on the IRQ polling table via the F$IRQ system call before placing it on the VIRQ table.

If the device has a physical interrupt, use the interrupt's hardware register address as the polling address for the F$IRQ call. After setting the polling address, set the flip and mask bytes for the device, and make the F$IRQ call.

If the device is totally VIRQ-driven, and has no interrupts, use the status byte from the VIRQ packet as the status byte. Use a mask byte of %00000001, defined as Vi.IFlag in the defs file. Use a flip byte value of 0. The following examples show how to set up both types of VIRQ calls. The first example is taken from an ACIA-type driver that has a physical interrupt found in a status register, but that cannot be accessed by the processor if used in the Multi-Pak. The second example is for a device with no physical interrupt handling, all interrupts are handled through the VIRQ.

```
* VIRQ Example #1 - Device Driver possessing real IRQ's

     *    Copyright 1985,1986 by Microware Systems
     *    Corporation. Reproduced Under License

     use defsfile

     * actual mask byte for hardware interrupt
     IRQReq set %10000000 Interrupt Request
       .
       .

     * offset to the actual hardware status register
     Status equ 1

     * VIRQ countdown value
     VIRQCNT equ 1      do the VIRQ on every tick
       .
       .

     *****************
     * Static storage offsets
     *
      org V.SCF room for scf variables
       .
       .

     VIRQBUF rmb 5 buffer for fake interrupt from clock
       .
       .
     MEM equ . Total static storage requirement

     ***************
     * Module Header
      mod MEND,NAM,DRIVR+OBJCT,REENT+1,ENT,MEM
      fcb UPDAT.

      fcb Edition Current Revision
     ************************
     * Driver entry jump table
     ENT lbra INIT
      lbra READ
      lbra WRITE
      lbra GETSTA
```

```
 lbra PUTSTA
  bra TRMNAT

* Actual mask information for F$IRQ call for the
* hardware interrupt MASK fcb 0 no flip bits
* fcb IRQReg Irq polling mask
* fcb 10 (higher) priority

***************
* Init
*   Initialize the device
*   Includes setting up the IRQ and VIRQ entries
*
INIT
    .
    .

* Install IRQ polling Table Entry first
*   Use the hardware status register and the hardware
*   mask
 ldd V.PORT,U get port address in D
 add #Status point to hardware status byte
 leax MASK,PCR get the hardware interrupt mask
 leay MIRQ,PCR address of interrupt service routine
 OS9 F$IRQ Add to IRQ polling table
 bcs INIT9 error - return it

* Install VIRQ in Clock Module second
*
 leay VIRQBUF,U get the 5 byte VIRQ buffer pointer
 lda #$80 get reset flag for repeated VIRQ's
 sta Vi.Stat,y put it into buffer
 ldd #VIRQCNT get count for number of ticks for the VIRQ
 std Vi.Rst,y put in initial reset value
 ldx #1 put onto table
 os9 F$VIRQ make the service request
 bcs INIT9 Error - return it
   .
   .

INIT9 rts

READ
   .
```

```
  .
WRITE
GETSTA
PUTSTA
  .
  .

***************
* Subroutine TRMNAT
*    Terminate device, including removal from tables
*
TRMNAT
  .
  .
* remove from VIRQ table first
 ldx #0 remove from VIRQ table
 leay VIRQBUF,U get address
 os9 F$VIRQ remove modem from VIRQ table
* next remove from IRQ table
 ldx #0
 os9 F$IRQ remove modem from polling tbl
 rts

********************
* MIRQ
*   process Interrupt
*
MDIRQ
  .
  .
< actual interrupt service routine >
  .
 rts

 emod Module Crc
MEND equ *
```

```
* VIRQ Example #2 - Device Driver without hardware interrupts

    ***************
    * STATIC STORAGE DEFINITION
    *
```

```
*
  .
  .

VIRQBF rmb 5 buffer for VIRQ
DMEM equ .

*****************
* Module Header
*
 mod DEND,DNAM,DRIVR+OBJCT,REENT+REV,DENT,DMEM
 fcb UPDAT. mode byte

 fcb 3 EDITION BYTE


* Driver entry table
DENT lbra INIT initialize
 lbra READ
 lbra WRITE
 lbra GETSTAT get status
 lbra SETSTAT set status
 lbra TERM terminate

* Mask information packet for F$IRQ call
* NOTE: uses the virtual interrupt flag, Vi.IFlag, for
* the maskbyte
*
DMSK fcb 0 no flip bits
 fcb Vi.IFlag polling mask for VIRQ
 fcb 10 priority

*****************
* INITIALIZE STORAGE AND CONTROLLER
* Includes setting up the IRQ and VIRQ table entries
*
INIT

* set up IRQ table entry first
* NOTE: uses the status register of the VIRQ buffer for
* the interrupt status register since no hardware status
* register is available
*
 leay VIRQBF+Vi.Stat,U get address of status byte
```

```
 tfr y,d put it into D reg
 leay DIRQ,PCR get address of interrupt routine
 leax DMSK,PCR get VIRQ mask info
 os9 F$IRQ install onto table
 bcs INIT9 exit on error

* now set up the VIRQ table entry
 leay VIRQBF,U point to the 5-byte packet
 lda #$80 get the reset flag to repeat VIRQ's
 sta Vi.Stat,y save it in the buffer
 ldd #VIRQCNT get the VIRQ counter value
 std Vi.Rst,y save it in the reset area of buffer
 ldx #1 code to install the VIRQ
 os9 F$VIRQ install on the table
 bcs INIT9 exit on error
   .
   .

INIT9 rts
   .
   .

 READ
 WRITE
 GETSTAT
 PUTSTAT
   .
**********************
* TERM - terminate the device and remove entries from
* tables
TERM
   .
* remove from VIRQ table first
 ldx #0 get zero to remove from table
 leay VIRQBF,U get address of packet
 os9 F$VIRQ
* then remove from IRQ table
 ldx #0 get zero to remove from table
 os9 F$IRQ
   .
   .
  rts

**********************************************************
```

```
* DIRQ - interrupt service routine
*
* NOTE: The service routine must be sure to reset the
* status byte of the VIRQ packet so that the interrupt
* looks as if it is cleared.
*
DIRQ
  .
  .
 lda VIRQBF+Vi.Stat,U get status byte
 anda #$FF-Vi.IFlag mask off interrupt bit
 sta VIRQBF+Vi.Stat,U put it back
  .
  .
 rts

 EMOD
DEND equ   *

END
```

Chapter 3

Memory Modules

In Chapter 2, you learned that OS-9 is based on the concept that memory is modular. This means that each program is considered to be an individually named *object*.

You also learned that each program loaded into memory must be in the module format. This format lets OS-9 manage the logical contents of memory, as well as the physical contents. Module types and formats are discussed in detail in this chapter.

Module Types

There are several types of modules. Each has a different use and function. These are the main requirements of a module:

- It cannot modify itself.
- It must be position-independent so that OS-9 can load or relocate it wherever space is available. In this respect, the module format is the OS-9 equivalent of *load records* used in older operating systems.

A module need not be a complete program or even 6809 machine language. It can contain BASIC09 I-code, constants, single subroutines, and subroutine packages.

Module Format

Each module has three parts: a *module header*, a *module body*, and a *cyclic-redundancy-check value* (CRC value).

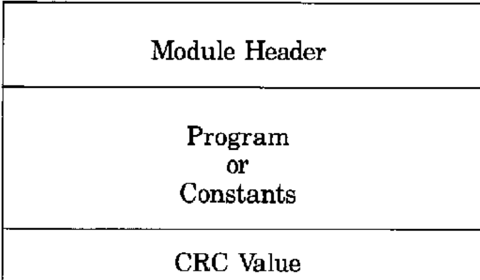


Figure 3.1

Module Header

At the beginning of the module (the lowest address) is the module header. Its form depends upon the module's use.

The header contains information about the module and its use. This information includes the following:

- Size
- Type (machine code, BASIC09 compiled code, and so on)
- Attributes (executable, re-entrant, and so on)
- Data storage memory requirements
- Execution starting address

Usually, you do not need to write routines to generate the modules and headers. All OS-9 programming languages automatically create modules and headers.

Module Body

The module body contains the program or constants. It usually is pure code. The module name string is included in this area. Figure 3.2 provides the offset values for calculating the location of a module's name. (See "Offset to Module Name".)

CRC Value

The last three bytes of the module are the Cyclic Redundancy Check (CRC) value. The CRC value is used to verify the integrity of a module.

When the system first loads the module into memory, it performs a 25-bit CRC over the entire module, from the first byte of the module header to the byte immediately before the CRC. The CRC polynomial used is $800FE3.

As with the header, you usually don't need to write routines to generate the CRC value. Most OS-9 programs do this automatically.

Module Headers: Standard Information

The first nine bytes of all module headers are defined as follows:

Relative Address	Use
$00,$01	Sync bytes ($87,$CD)
$02,$03	Module size
$04,$05	Offset to module name
$06	Module type/Language
$07	Attributes/Revision level
$08	Header check

Figure 3.2

Sync Bytes

The sync bytes specify the location of the module. (The first sync byte is the start of the module.) These two bytes are constant.

Module Size

The module size specifies the size of the module in bytes (includes CRC).

Offset to Module Name

The offset to module name specifies the address of the module name string relative to the start of the module. The name string can be located anywhere in the module. It consists of a string of ASCII characters with the most significant bit set on the last character.

Type/Language Byte

The type/language byte specifies the type and language of the module.

The four most significant bits of this byte indicate the type. Eight types are pre-defined. Some of these are for OS-9's internal use only. The type codes are given here (0 is not a legal type code):

Code	Module Type	Name
$1x	Program module	Prgrm
$2x	Subroutine module	Sbrtn
$3x	Multi-module (for future use)	Multi
$4x	Data module	Data
$5x-$Bx	User-definable module	
$Cx	OS-9 system module	Systm
$Dx	OS-9 file manager module	FlMgr
$Ex	OS-9 device driver module	Drivr
$Fx	OS-9 device descriptor module	Devic

Figure 3.3

The four least significant bits of Byte 6 indicate the language (denoted by x in the previous Figure). The language codes are given here:

Code	Language
$x0	Data (non-executable)
$x1	6809 object code
$x2	BASIC09 I-code
$x3	PASCAL P-code
$x4-$xF	Reserved for future use

Figure 3.4

By checking the language type, high-level language runtime systems can verify that a module is the correct type before attempting execution. BASIC09, for example, can run either I-code or 6809 machine language procedures arbitrarily by checking the language type code.

Attributes/Revision Level Byte

The attributes/revision level byte defines the attributes and revision level of the module.

The four most significant bits of this byte are reserved for module attributes. Currently, only Bit 7 is defined. When set, it indicates the module is re-entrant and, therefore, *shareable*.

The four least significant bits of this byte are a revision level in the range 0 to 15. If two or more modules have the same name, type, language, and so on, OS-9 keeps in the module directory only the module having the highest revision level. Therefore, you can replace or patch a ROM module, simply by loading a new, equivalent module that has a higher revision level.

> **Note:** A previously linked module cannot be replaced until its link count goes to zero.

Header Check

The header check byte contains the one's complement of the Exclusive-OR of the previous eight bytes.

Module Headers: Type-Dependent Information

More information usually follows the first nine bytes of a module header. The layout and meaning vary, depending on the module type.

Module types $Cx-$Fx (system module, file manager module, device driver module, and device descriptor module) are used only by OS-9. Their formats are given later in the manual.

Module types $1x through $Bx have a general-purpose executable format. This format is often used in programs called by F$Fork or F$Chain. Here is the format used by these module types:

Executable Memory Module Format

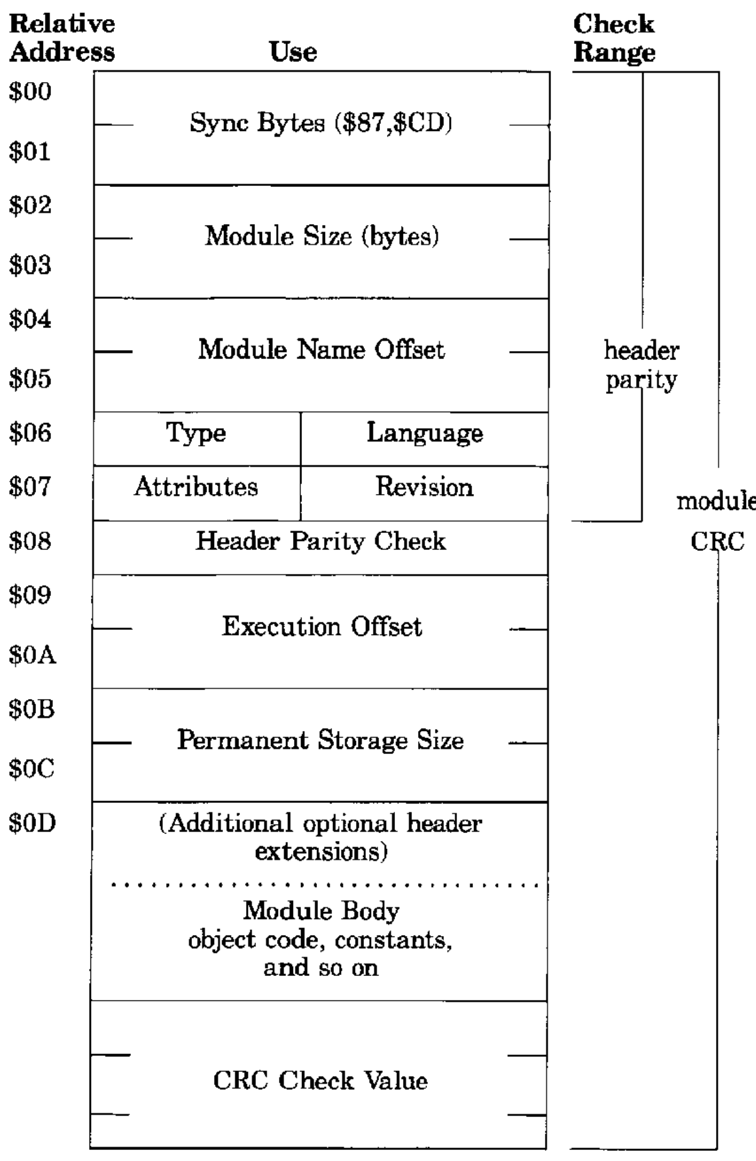


Figure 3.5

As you can see from the preceding chart, the executable memory has four extra bytes in its header. They are:

$09,$0A	Execution offset
$0B,$0C	Permanent storage size

Execution Offset. The program or subroutine's offset starting address, relative to the first byte of the sync code. A module that has multiple entry points (such as cold start and warm start) might have a branch table starting at this address.

Permanent Storage Size. The minimum number of bytes of data storage required to run. Fork and Chain use this number to allocate a process's data area.

If the module is not directly executed by a Fork or Chain system call (for instance a subroutine package), this entry is not used by OS-9. It is commonly used to specify the maximum stack size required by re-entrant subroutine modules. The calling program can check this value to determine if the subroutine has enough stack space.

When OS-9 starts after a single system reset, it searches the entire memory space for ROM modules. It finds them by looking for the module header sync code ($87,$CD).

When OS-9 detects the header sync code, it checks to see if the header is correct. If it is, the system obtains the module size from the header and performs a 24-bit CRC over the entire module. If the CRC matches, OS-9 considers the module to be valid and enters it into the module directory. All ROM modules that are present in the system at startup are automatically included in the system module directory.

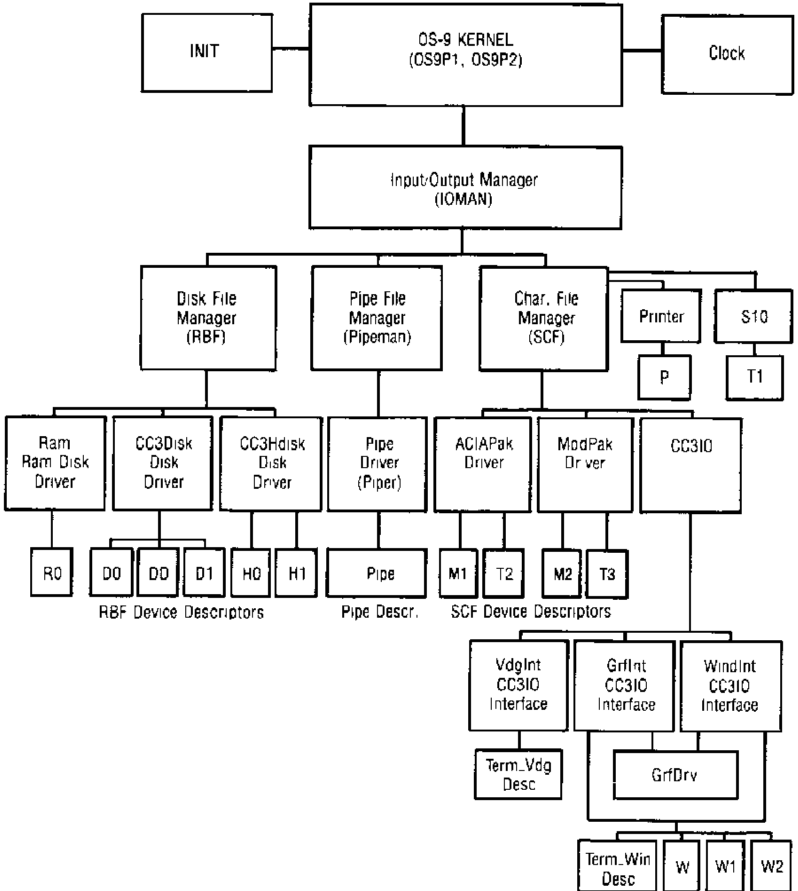
After the module search, OS-9 links to the component modules it found. This is the secret to OS-9's ability to adapt to almost any 6809 computer. It automatically locates its required and optional component modules and rebuilds the system each time it is started.

Chapter 4

OS-9's Unified Input/Output System

Chapter 1 mentioned that OS-9 has a unified I/O system, consisting of all modules except those on the kernel level. This chapter discusses the I/O modules in detail.

I/O System Modules



OS-9 COMPONENT MODULE ORGANIZATION

The VDG Interface performs both interface and low level routines for VDG Color Computer 2 compatible modes and has limited support for high res screen allocation.

The GrfInt Interface provides the standard code interpretations and interface functions.

The WindInt Interface, available in the Multi-view package, contains all the functionality of GrfInt, along with additional support features. If you use WindInt, do not include GrfInt.

Both WindInt and GrfInt use the low-level driver GrfDrv to perform drawing on the bit-map screens.

Term_VDG uses CC3IO/VdgInt while Term_win and all window descriptors use CC3IO/(WindInt/GrfInt)/GrfDrv modules.

The I/O system provides system-wide, hardware-independent I/O services for user programs and OS-9 itself. All I/O system calls are received by the kernel and passed to the I/O manager for processing.

The I/O manager performs some processing, such as the allocation of data structures for the I/O path. Then, it calls the file managers and device drivers to do most of the work. Additional file manager, device driver, and device descriptor modules can be loaded into memory from files and used while the system is running.

The I/O Manager

The I/O manager provides the first level of service of I/O system calls. It routes data on I/O process paths to and from the appropriate file managers and device drivers.

The I/O Manager also maintains two important internal OS-9 data structures—the *device table* and the *path table*. Never modify the I/O manager.

When a path is opened, the I/O manager tries to link to a memory module that has the device name given or implied in the pathlist. This module is the *device descriptor*. It contains the names of the device driver and file manager for the device. The I/O manager saves the names so later system calls can be routed to these modules.

File Managers

OS-9 can have any number of *file manager modules*. Each of these modules processes the raw data stream to or from a class of device drivers that have similar operational characteristics. It removes as many unique characteristics as possible from I/O operations. Thus, it assures that similar devices conform to the OS-9 standard I/O and file structure.

The file manager also is responsible for mass storage allocation and directory processing, if these are applicable to the class of devices it serves.

File managers usually buffer the data stream and issue requests to the kernel for dynamic allocation of buffer memory. They can also monitor and process the data stream, for example, adding line-feed characters after carriage-return characters.

The file managers are re-entrant. The three standard OS-9 file managers are:

- Random block file manager: The RBF manager supports random-access, block-structured devices such as disk systems and bubble memories. (Chapter 5 discusses the RBF manager in detail.)
- Sequential Character File Manager: The SCF manager supports single-character-oriented devices, such as CRTs or hardcopy terminals, printers, and modems. (Chapter 6 discusses SCF in detail.)
- Pipe File Manager (PIPEMAN): The pipe manager supports interprocess communication via *pipes*.

File Manager Structure

Every file manager must have a branch table in exactly the following format. Routines that are not used by the file manager must branch to an error routine, that sets the carry and loads Register B with an appropriate error code before returning. Routines returning without error must ensure that the carry bit is clear.

```
* All routines are entered with:
* (Y) = Path Descriptor pointer
* (U) = Caller's register stack pointer

EntryPt equ *
 lbra Create
 lbra Open
 lbra MakDir
 lbra ChgDir
 lbra Delete
 lbra Seek
 lbra Read
 lbra Write
 lbra ReadLn
 lbra WriteLn
 lbra GetStat
 lbra PutStat
 lbra Close
```

Create, Open

Create and Open handle file creating and opening for devices. Typically, the process involves allocating any required buffers, initializing path descriptor variables, and establishing the path name. If the file manager controls multi-file devices (RBF), directory searching is performed to find or create the specified file.

Makdir

Makdir creates a directory file on multi-file devices. Makdir is neither preceded by a Create nor followed by a Close. File managers that are incapable of supporting directories need to return carry set with an appropriate error code in Register B.

ChgDir

On multi-file devices, ChgDir searches for a directory file. If ChgDir finds the directory, it saves the address of the directory (up to four bytes) in the caller's process descriptor. The descriptor is located at P$DIO+2 (for a data directory) or P$DIO+8 (for an execution directory).

In the case of the RBF manager, the address of the directory's file descriptor is saved. Open/Create begins searching in the current directory when the caller's pathlist does not begin with a slash (/). File managers that do not support directories should return the carry set and an appropriate error code in Register B.

Delete

Multi-file device managers handle file delete requests by initiating a directory search that is similar to Open. Once a device manager finds the file, it removes the file from the directory. Any media in use by the file are returned to unused status. In the case of the RBF manager, space is returned for system use and is marked as available in the free cluster bit map on the disk. File managers that do not support multi- file devices return an error.

Seek

File managers that support random access devices use Seek to position file pointers of an already open path to the byte specified. Typically, the positioning is a logical movement. No error is produced at the time of the seek if the position is beyond the current "end of file".

Normally, file managers that do not support random access ignore Seek. However, an SCF-type manager can use Seek to perform cursor positioning.

Read

Read returns the number of bytes requested to the user's data buffer. Make sure Read returns an EOF error if there is no data available. Read must be capable of copying pure binary data, and generally performs no editing on the data. Generally, the file manager calls the device driver to actually read the data into the buffer. Then, the file manager copies the data from the buffer into the user's data area to keep file managers device-independent.

Write

The Write request, like Read, must be capable of recording pure binary data without alteration. The routines for Read and Write are almost identical with the exception that Write uses the device driver's output routine instead of the input routine. The RBF manager and similar random access devices that use fixed-length records (sectors) must often preread a sector before writing it, unless they are writing the entire sector. In OS-9, writing past the end of file on a device expands the file with new data.

ReadLn

ReadLn differs from Read in two respects. First, ReadLn terminates when the first end-of-line (carriage return) is encountered. ReadLn performs any input editing that is appropriate for the device. In the case of SCF, editing involves handling functions such as backspace, line deletion, and the removal of the high-order bit from characters.

WriteLn

WriteLn is the counterpart of ReadLn. It calls the device driver to transfer data up to and including the first (if any) carriage return encountered. Appropriate output editing can also be performed. For example, SCF outputs a line feed, a carriage return character, and nulls (if appropriate for the device). It also pauses at the end of a screen page.

GetStat, PutStat

The GetStat (get status) and PutStat (put status) system calls are wildcard calls designed to provide a method of accessing features of a device (or file manager) that are not generally device independent. The file manager can perform specific functions such as setting the size of a file to a given value. Pass *unknown* status calls to the driver to provide further means of device independence. For example, a PutStat call to format a disk track might behave differently on different types of disk controllers.

Close

Close is responsible for ensuring that any output to a device is completed. (If necessary, Close writes out the last buffer.) It releases any buffer space allocated in an Open or Create. Close does not execute the device driver's terminate routine, but can do specific end-of-file processing if you want it to, such as writing end-of-file records on disks, or form feeds on printers.

Interfacing with Device Drivers

Strictly speaking, device drivers must conform to the general format presented in this manual. The I/O Manager is slightly different because it only uses the Init and Terminate entry points. Other entry points need only be compatible with the file manager for which the driver is written. For example, the Read entry point of an SCF driver is expected to return one byte from the device. The Read entry point of an RBF driver, on the other hand, expects Read to return an entire sector.

The following code is part of an SCF file manager. The code shows how a file manager might call a driver.

```
****************
*   IOEXEC
*     Execute Device's Read/Write Routine
*
*   Passed:    (A) = Output character (write)
*              (X) = Device Table entry ptr
*              (Y) = Path Descriptor pointer
*              (U) = Offset of routine (D$Read,
                     D$Write)
*   Returns:   (A) = Input char (read)
*              (B) = Error code, CC set if error
*   Destroys B,CC

IOEXEC pshs a,x,y,u save registers
 ldu V$STAT,x get static storage for driver
 ldx V$DRIV,x get driver module address
 ldd M$EXEC,x and offset of execution entries
 addd 5,s offset by read/write
 leax d,x absolute entry address
 lda ,s+ restore char (for write)
 jsr 0,x execute driver read/write
 puls x,y,u,pc return (A)=char, (B)=error

 emod Module CRC
Size equ * size of sequential file manager
```

Device Driver Modules

The device driver modules are subroutine packages that perform basic, low-level I/O transfers to or from a specific type of I/O device hardware controller. These modules are re-entrant. So, one copy of the module can concurrently run several devices that use identical I/O controllers.

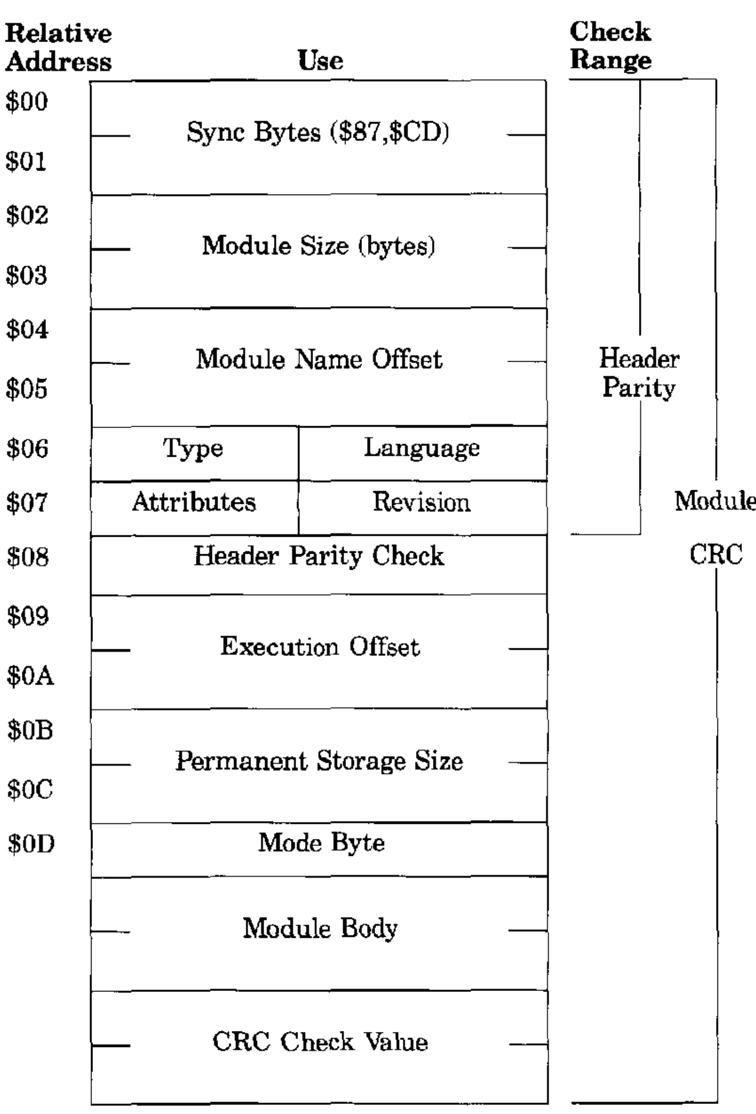
Device driver modules use a standard module header, in which the module type is specified as code $Ex (device driver). The execution offset address in the module header points to a branch table that has a minimum of six 3-byte entries.

Each entry is typically an LBRA to the corresponding subroutine. The file managers call specific routines in the device driver through this table, passing a pointer to a path descriptor and passing the hardware control register address in the 6809 registers. The branch table looks like this:

Code	Meaning
+$00	Device initialization routine
+$03	Read from device
+$06	Write to device
+$09	Get device status
+$0C	Set device status
+$0F	Device termination routine

(For a complete description of the parameters passed to these subroutines, see the "Device Driver Subroutines" sections in Chapters 5 and 6.)

Device Driver Module Format



$0D Mode Byte - (D S PE PW PR E W R)

OS-9 Interaction With Devices

Device drivers often must wait for hardware to complete a task or for a user to enter data. Such a wait situation occurs if an SCF device driver receives a Read but there is no data is available, or if it receives a Write and no buffer space is available. OS-9 drivers that encounter this situation should suspend the current process (via F$Sleep). In this way the driver allows other processes to continue using CPU time.

The most efficient way for a driver to awaken itself and resume processing data is by using interrupt requests (IRQs). It is possible for the driver to sleep for a number of system clock ticks and then check the device or buffer for a ready signal. The drawbacks to this technique are:

- It requires the system clock to always remain active.
- It might require a large number of ticks (perhaps 20) for the device to become ready. Such a case leaves you with a dilemma. If you make the program sleep for two ticks, the system wastes CPU time while checking for device ready. If the driver sleeps 20 ticks, it does not have a good response time.

An interrupt system allows the hardware to report to the CPU and the device drivers when the device is finished with an operation. Using interrupts to its advantage, a device driver can set up interrupt handling to occur when a character is sent or received or when a disk operation is complete. There is a built-in polling facility for pausing and awakening processes. Here is a technique for handling interrupts in a device driver:

1. Use the Init routine to place the driver interrupt service call (IRQSVC) routine in the IRQ polling sequence via an F$IRQ system call:

```
ldd V.Port,u get address to poll
leax IRQPOLL,pcr point to IRQ packet
leay IRQSVC,pcr point to IRQ routine
OS9 F$IRQ add dev to poll sequence
bcs Error abnormal exit if error
```

2. Ensure that driver programs waiting for their hardware, call the sleep routine. The sleep routine copies V.Busy to V.Wake. Then, it goes to sleep for some period of time.

3. When the driver program wakes up, have it check to see whether it was awakened by an interrupt or by a signal sent from some other process.

   Usually, the driver performs this check by reading the V.Wake storage byte. The V.Busy byte is maintained by the file manager to be used as the process ID of the process using the driver. When V.Busy is copied into V.Wake, then V.Wake becomes a flag byte and an information byte. A non-zero Wake byte indicates that there is a process awaiting an interrupt. The value in the Wake byte indicates the process to be awakened by sending a wakeup signal as shown in the following code:

```
        lda V.Busy,u        get proc ID
        sta V.Wake,u        arrange for wakeup
        andcc #^IntMasks    prep for interrupts
Sleep50 ldx #0              or any other tick time
                            (if signal test)
        OS9 F$Sleep         await an IRQ
        ldx D.Proc          get proc desc ptr if
                            signal test
        ldb P$Signal,x      is signal present?
                            (if signal test)
        bne SigTest         bra if so if signal
                            test
        tst V.Wake,u        IRQ occur?
        bne Sleep50         bra if not
```

   Note that the code labeled "if signal test" is only necessary if the driver wishes to return to the caller if a signal is sent without waiting for the device to finish. Also note that IRQs and FIRQs must be masked between the time a command is given to the device and the moving of V.Busy and V.Wake. If they are not masked, it is possible for the device IRQ to occur and the IRQSVC routine to become confused as to whether it is sending a wakeup signal or not.

4. When the device issues an interrupt, OS-9 calls the routine at the address given in F$IRQ with the interrupts masked. Make the routine as short as possible, and have it return with an RTS instruction. IRQSVC can verify that an interrupt has occurred for the device. It needs to clear the interrupt to retrieve any data in the device. Then the V.Wake byte communicates with the main driver module. If V.Wake is non-zero, clear it to indicate a true device interrupt and use its contents as the process ID for an F$Send system call. The F$Send call sends a wakeup signal to the process. Here is an example:

```
        ldx V.Port,u get device address
        tst ?? is it real interrupt from device?
        bne IRQSVC90 bra to error if not
        lda Data,x get data from device
        sta 0,y
        lda V.Wake,u
        beq IRQSVC80 bra if none
        clr V.Wake,u clear it as flag to main
        routine
        ldb #S$Wake,u get wakeup signal
        OS9 F$Send send signal to driver
IRQSVC80 clrb clear carry bit (all is well)
        rts
IRQSVC90 comb set carry bit (is an IRQ call)
        rts
```

Suspend State (Level Two only)

The Suspend State allows the elimination of the F$Send system call during interrupt handling. Because the process is already in the active queue, it need not be moved from one queue to another. The device driver IRQSVC routine can now wake up the suspended main driver by clearing the process status byte suspend bit in the process state. Following are sample routines for the Sleep and IRQSVC calls:

```
        lda D.Proc get process ptr
        sta V.Wake,u prep for re-awakening

        enable device to IRQ, give command, etc.

        bra Cmd50 enter suspend loop

  Cmd30 ldx D.Proc get ptr to process desc
```

```
        lda P$State,x get state flag
        ora #Suspend put proc in suspend state
        sta P$State,x save it in proc desc
        andcc #^IntMasks unmask interrupts
        ldx #1 give up time slice
        OS9 F$Sleep suspend (in active queue)
Cmd50 orcc #IntMasks mask interrupts while
changing state
  ldx D.Proc get proc desc addr (if signal
test)
  lda P$Signal,x get signal (if signal test)
  beq SigProc bra if signal to be handled
  lda V.Wake,u true interrupt?
  bne Cmd30 bra if not
  andcc #^IntMasks assure interrupts unmasked
```

Note that D.Proc is a pointer to the process descriptor of the current process. Process descriptors are always allocated on 256-byte page boundaries. Thus, having the high order byte of the address is adequate to locate the descriptor. D.Proc is put in V.Wake as a dual value. In one instance, it is a flag byte indicating that a process is indeed suspended. In the other instance, it is a pointer to the process descriptor which enables the IRQSVC routine to clear the suspend bit. It is necessary to have the interrupts masked from the time the device is enabled until the suspend bit has been set. Making the interrupts ensure that the IRQSVC routine does not think it has cleared the suspend bit before it is even set. If this happens, when the bit is set the process might go into permanent suspension. The IRQSVC routine sample follows:

```
ldy V.Port,u get dev addr
tst V.Wake,u is process awaiting
IRQ?
beq IRQSVCER no exit

clear device interrupt
exit if IRQ not from this device

lda V.Wake,u get process ptr
clrb
stb V.Wake,u clear proc waiting flag
tfr d,x get process descriptor ptr
lda P$State,x get state flag
anda # Suspend clear suspend state
sta P$State,x save it
```

```
              clrb clear carry bit
              rts
IRQSVCER comb set carry bit
              rts
```

Device Descriptor Modules

Device descriptor modules are small, non-executable modules. Each one provides information that associates a specific I/O device with its logical name, hardware controller address(es), device driver, file manager name, and initialization parameters.

Unlike the device drivers and file managers, which operate on classes of devices, each device descriptor tailors its functions to a specific device. Each device must have a device descriptor.

Device descriptor modules use a standard module header, in which the module type is specified as code $Fx (device descriptor). The name of the module is the name by which the system and user know the device (the device name given in pathlists).

The rest of the device descriptor header consists of the information in the following chart:

Relative Address(es)	Use
$09,$0A	The relative address of the file manager name string address
$0B,$0C	The relative address of the device driver name string
$0D	Mode/Capabilities: D S PE PW PR E W R (directory, single user, public execute, public write, public read, execute, write, read)
$0E,$0F,$10	The absolute physical (24-bit) address of the device controller
$11	The number of bytes (*n* bytes) in the initialization table
$12,$12 + *n*	Initialization table

When OS-9 opens a path to the device, the system copies the initialization table into the option section (PD.OPT) of the path descriptor. (See "Path Descriptors" in this chapter.)

The values in this table can be used to define the operating parameters that are alterable by the Get Status and Set Status system calls (I$GetStt and I$SetStt). For example, parameters that are used when initializing terminals define which control characters are to be used for functions such as backspace and delete.

The initialization table can be a maximum of 32 bytes long. If the table is fewer than 32 bytes long, OS-9 sets the remaining values in the path descriptor to 0.

You might wish to add devices to your system. If a similar device driver already exists, all you need to do is add the new hardware and load another device descriptor. Device descriptors can be in the boot module or they can be loaded into RAM from mass-storage files while the system is running.

The following diagram illustrates the device descriptor format:

Device Descriptor Format

Relative Address	Use		Check Range
$00, $01	Sync Bytes ($87,$CD)		header parity / module CRC
$02, $03	Module Size (bytes)		header parity / module CRC
$04, $05	Offset to Module Name		header parity / module CRC
$06	F$ (Type)	$1 (Lang)	header parity / module CRC
$07	Attributes	Revision	header parity / module CRC
$08	Header Parity Check		module CRC
$09, $0A	Offset to File Manager Name String		module CRC
$0B	Offset to Device Driver Name String		module CRC
$0D	Mode Byte		module CRC
$0E, $0F, $10	Device Controller Absolute Physical Addr. (24 bit)		module CRC
$11	Initialization Table Size		module CRC
$12,$12+n	(Initialization Table)		module CRC
	(Name Strings, and so on)		module CRC
	CRC Check Value		module CRC

Path Descriptors

Every open path is represented by a data structure called a *path descriptor* (PD). The PD contains the information the file managers and device drivers require to perform I/O functions.

PDs are 64 bytes long and are dynamically allocated and deallocated by the I/O manager as paths are opened and closed.

They are internal data structures, that are not normally referenced from user or applications programs. The description of PDs is presented here mainly for those programmers who need to write custom file managers, device drivers, or other extensions to OS-9.

PDs have three sections. The first section, which is ten bytes long, is the same for all file managers and device drivers. The information in the first section is shown in the following chart.

Path Descriptor: Standard Information

Name	Relative Address	Size (Bytes)	Use
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode: 1 = read, 2 = write, 3 = update
PD.CNT	$02	1	Number of open paths using this PD
PD.DEV	$03	2	Address of the associated device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of the caller's register stack
PD.BUF	$08	2	Address of the 256-byte data buffer (if used)
PD.FST	$0A	22	Defined by the file manager
PD.OPT	$20	32	Reserved for the Getstat/Setstat options

PD.FST is 22-byte storage reserved for and defined by each type of file manager for file pointers, permanent variables, and so on.

PD.OPT is a 32-byte option area used for file or device operating parameters that are dynamically alterable. When the path is opened, the I/O manager initializes these variables by copying the initialization table that is in the device descriptor module. User programs can change the values later, using the Get Status and Set Status system calls.

PD.FST and **PD.OPT** are defined for the file manager in the assembly-language equate file (SCFDefs for the SCF manager or RBFDefs for the RBF manager).

Chapter 5

Random Block File Manager

The random block file manager (RBF manager) supports *disk storage*. It is a re-entrant subroutine package called by the I/O manager for I/O system calls to random-access devices. It maintains the logical and physical file structures.

During normal operation, the RBF manager requests allocation and deallocation of 256-byte data buffers. Usually, one buffer is required for each open file. When physical I/O functions are necessary, the RBF manager directly calls the subroutines in the associated device drivers. All data transfers are performed using 256-byte data blocks (pages).

The RBF manager does not deal directly with physical addresses such as tracks and cylinders. Instead, it passes to the device drivers address parameters, using a standard address called a *logical sector number*, or *LSN*. LSNs are integers from 0 to *n-1*, where *n* is the maximum number of sectors on the media. The driver translates the logical sector number to actual cylinder/track/sector values.

Because the RBF manager supports many devices that have different performance and storage capacities, it is highly parameter-driven. The physical parameters it uses are stored on the media itself.

On disk systems, the parameters are written on the first few sectors of Track 0. The device drivers also use the information, particularly the physical parameters stored on Sector 0. These parameters are written by the FORMAT program that initializes and tests the disk.

Logical and Physical Disk Organization

All disks used by OS-9 store basic identification, file structure, and storage allocation information on these first few sectors.

LSN 0 is the *identification sector*. LSN 1 is the *disk allocation map sector*. LSN 2 marks the beginning of the disk's ROOT directory. The following section tells more about LSN 0 and LSN 1.

Identification Sector (LSN 0)

LSN 0 contains a description of the physical and logical characteristics of the disk. These characteristics are set by the FORMAT command program when the disk is initialized.

The following table gives the OS-9 mnemonic name, byte address, size, and description of each value stored in this LSN 0.

Name	Relative Address	Size (Bytes)	Use
DD.TOT	$00	3	Number of sectors on disk
DD.TKS	$03	1	Track size (in sectors)
DD.MAP	$04	2	Number of bytes in the allocation bit map
DD.BIT	$06	2	Number of sectors per cluster
DD.DIR	$08	3	Starting sector of the ROOT directory
DD.OWN	$0B	2	Owner's user number
DD.ATT	$0D	1	Disk attributes
DD.DSK	$0E	2	Disk identification (for internal use)
DD.FMT	$10	1	Disk format, density, number of sides
DD.SPT	$11	2	Number of sectors per track
DD.RES	$13	2	Reserved for future use
DD.BT	$15	3	Starting sector of the bootstrap file
DD.BSZ	$18	2	Size of the bootstrap file (in bytes)
DD.DAT	$1A	5	Time of creation (Y:M:D:H:M)
DD.NAM	$1F	32	Volume name in which the last character has the most significant bit set
DD.OPT	$3F		Path descriptor options

Disk Allocation Map Sector (LSN 1)

LSN 1 contains the *disk allocation map*, which is created by FORMAT. This map shows which sectors are allocated to the files and which are free for future use.

Each bit in the allocation map represents a sector or cluster of sectors on the disk. If the bit is set, the sector is considered to be in use, defective, or non-existent. If the bit is cleared, the corresponding cluster is available. The allocation map usually starts at LSN1. The number of sectors it requires varies according to how many bits are needed for the map. DD.MAP specifies the actual number of bytes used in the map.

Multiple sector allocation maps allow the number of sectors/cluster to be as small as possible for high volume media.

The FORMAT utility bases the size of the allocation map on the size and number of sectors per cluster.

The DD.MAP value in LSN 0 specifies the number of bytes (in LSN 1) that are used in the map.

Each bit on the disk allocation map corresponds to one sector cluster on the disk. The DD.BIT value in LSN 0 specifies the number of sectors per cluster. The number is an integral power of 2 (1, 2, 4, 8, 16, and so on).

If a cluster is available, the corresponding bit is cleared. If it is allocated, non-existent, or physically defective, the corresponding bit is set.

ROOT Directory

This file is the parent directory of all other files and directories on the disk. It is the directory accessed using the physical device name (such as /D1). Usually, it immediately follows the Allocation Map. The location of the ROOT directory file descriptor is specified in DD.DIR. The ROOT directory contains an entry for each file that resides in the directory, including other directories.

File Descriptor Sector

The first sector of every file is the *file descriptor*. It contains the logical and physical description of the file.

The following table describes the contents of the file descriptor.

Name	Relative Address	Size (Bytes)	Use
FD.ATT	$00	1	File attributes: D S PE PW PR E W R (see next chart)
FD.OWN	$01	2	Owner's user ID
FD.DAT	$03	5	Date last modified: (Y M D H M)
FD.LNK	$08	1	Link count
FD.SIZ	$09	4	File size (number of bytes)
FD.CREAT	$0D	3	Date created (Y M D)
FD.SEG	$10	240	Segment list (see next chart)

FD.ATT. (The attribute byte) contains the file permission bits. When set the bits indicate the following:

Bit 7 Directory
Bit 6 Single user
Bit 5 Public execute
Bit 4 Public write
Bit 3 Public read
Bit 2 Execute
Bit 1 Write
Bit 0 Read

FD.SEG (the segment list) consists of a maximum of 48 5-byte entries that have the size and address of each file block in logical order. Each entry has the block's 3-byte LSN and 2-byte size (in sectors). The entry following the last segment is zero.

After creation, a file has no data segments allocated to it until the first write. (Write operations past the current end-of-file cause sectors to be added to the file. The first write is always past the end-of-file.)

If the file has no segments, it is given an initial segment. Usually, this segment has the number of sectors specified by the minimum allocation entry in the device descriptor. If, however, the number of sectors requested is more than the minimum, the initial segment has the requested number.

Later expansions of the file usually are also made in minimum allocation increments. Whenever possible, OS-9 expands the last segment, instead of adding a segment. When the file is closed, OS-9 truncates unused sectors in the last segment.

OS-9 tries to minimize the number of storage segments used in a file. In fact, many files have only one segment. In such cases, no extra Read operations are needed to randomly access any byte in the file.

If a file is repeatedly closed, opened, and expanded, it can become fragmented so that it has many segments. You can avoid this fragmentation by writing a byte at the highest address you want to be used on a file. Do this before writing any other data.

Directories

Disk directories are files that have the D attribute set. A directory contains an integral number of entries, each of which can hold the name and LSN of a file or another directory.

Each directory entry contains 29 bytes for the filename, followed by the three bytes for the LSN of the file's descriptor sector. The filename is left-justified in the field, with the most significant bit of the last character set. Unused entries have a zero byte in the first filename character position.

Every disk has a master directory called the ROOT directory. The DD.DIR value in LSN 0 (identification sector) specifies the starting sector of the ROOT directory.

The RBF Manager Definitions of the Path Descriptor

As stated earlier in this chapter, the PD.FST section of the path descriptor is reserved for and defined by the file manager. The following table describes the use of this section by the RBF manager. For your convenience, it also includes the other sections of the PD.

Name	Relative Address	Size (Bytes)	Use
Universal Section (Same for all file managers and device drivers)			
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode 1 = read, 2 = write, 3 = update
PD.CNT	$02	1	Number of open images (paths using this PD)
PD.DEV	$03	2	Address of the associated device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of the caller's 6809 register stack
PD.BUF	$08	2	Address of the 256-byte data buffer (if used)

Name	Relative Address	Size (Bytes)	Use
The RBF manager Path Descriptor Definitions (PD.FST Section)			
PD.SMF	$0A	1	State flag: Bit 0 = current buffer is altered Bit 1 = current sector is in the buffer Bit 2 = descriptor sector is in the buffer
PD.CP	$0B	4	Current logical file position (byte address)
PD.SIZ	$0F	4	File size
PD.SBL	$13	3	Segment beginning logical sector number (LSN)
PD.SBP	$16	3	Segment beginning physical sector number (PSN)

Name	Relative Address	Size (Bytes)	Use
PD.SSZ	$19	3	Segment size
PD.DSK	$1C	2	Disk ID (for internal use only)
PD.DTB	$1E	2	Address of drive table

Name	Relative Address	Size (Bytes)	Use
The RBF manager Option Section Definitions (PD.OPT Section)			
(Copied from the device descriptor)			
PD.DTP	$20	1	Device class: 0 = SCF 1 = RBF 2 = PIPE 3 = SBF
PD.DRV	$21	1	Drive number (0..*n*)
PD.STP	$22	1	Step rate
PD.TYP	$23	1	Device type
PD.DNS	$24	1	Density capability
PD.CYL	$25	2	Number of cylinders (tracks)
PD.SID	$27	1	Number of sides (surfaces)
PD.VFY	$28	1	0 = verify disk writes
PD.SCT	$29	2	Default number of sectors per track
PD.T0S	$2B	2	Default number of sectors per track (Track 0)
PD.ILV	$2D	1	Sector interleave factor
PD.SAS	$2E	1	Segment allocation size
PD.TFM	$2F	1	DMA transfer mode
PD.EXTEN	$30	2	Path extension for record locking
PD.STOFF	$32	1	Sector/track offsets

Name	Relative Address	Size (Bytes)	Use
(Not copied from the device descriptor):			
PD.ATT	$33	1	File attributes (D S PE PW PR E W R)
PD.FD	$34	3	File descriptor PSN
PD.DFD	$37	3	Directory file descriptor PSN
PD.DCP	$3A	4	File's directory entry pointer
PS.DVT	$3E	2	Address of the device table entry

Any values not determined by this table default to zero.

RBF-Type Device Descriptor Modules

This section describes the use of the initialization table contained in the device descriptor modules for RBF-type devices. The following values are those the I/O manager copies from the device descriptor to the path descriptor.

Name	Relative Address	Size (Bytes)	Use
	$0-$11		Standard device descriptor module header
IT.DTP	$12	1	Device type: 0 = SCF 1 = RBF 2 = PIPE 3 = SBF
IT.DRV	$13	1	Drive number
IT.STP	$14	1	Step rate
IT.TYP	$15	1	Device type (see RBF path descriptor)
IT.DNS	$16	1	Media density: Always 1 (double) (see following information)
IT.CYL	$17	2	Number of cylinders (tracks)
IT.SID	$19	1	Number of sides
IT.VFY	$1A	1	0 = Verify disk writes 1 = no verify
IT.SCT	$1B	2	Default number of sectors per track
IT.T0S	$1D	2	Default number of sectors per track (Track 0)
IT.ILV	$1F	1	Sector interleave factor
IT.SAS	$20	1	Minimum size of segment allocation (number of sectors to be allocated at one time)

IT.DRV is used to associate a 1-byte integer with each drive that a controller handles. Number the drives for each controller as 0 to *n-1*, where *n* is the maximum number of drives the controller can handle.

IT.TYP specifies the device type (all types).

Bit 0 — 0 = 5-inch floppy diskette

Bit 5 — 0 = Non-Color Computer format
1 = Color Computer format

Bit 6 — 0 = Standard OS-9 format
1 = Non-standard format

Bit 7 — 0 = Floppy diskette
1 = Hard disk

IT.DNS specifies the density capabilities (floppy diskette only).

Bit 0 — 0 = Single-bit density (FM)
1 = Double-bit density (MFM)

Bit 1 — 0 = Single-track density (5-inch, 48 tracks per inch)
1 = Double-track density (5-inch, 96 tracks per inch)

IT.SAS specifies the minimum number of sectors allowed at one time.

RBF Record Locking

Record locking is a general term that refers to methods designed to preserve the integrity of files that can be accessed by more than one user or process. The OS-9 implementation of record locking is designed to be as invisible as possible. This means that existing programs do not have to be rewritten to take advantage of record locking facilities. You can usually write new programs without special concern for multi-user activity.

Record locking involves detecting and preventing conflicts during record access. Whenever a process modifies a record, the system locks out other procedures from accessing the file. It defers access to other procedures until it is safe for them to write to the record. The system does not lock records during reads; so, multiple processes can read the record at the same time.

Record Locking and Unlocking

To detect conflicts, OS-9 must recognize when a record is being updated. The RBF manager provides true record locking on a byte basis. A typical record update sequence is:

```
OS9 I$Read      program reads record
                RECORD IS LOCKED
    .
    .           program updates record
    .
OS9 I$Seek      reposition to record
OS9 I$Write     record is rewritten
                RECORD IS RELEASED
```

When a file is opened in update mode, any read causes locking of the record being accessed. This happens because the RBF manager cannot determine in advance if the record is to be updated. The record stays locked out until the next read, write, or close.

However, when a file is opened in the read or execute modes, the system does not lock accessed records because the records cannot be updated in these two modes.

A subtle but important problem exists for programs that interrogate a data base and occasionally update its data. If you neglect to release a record after accessing it, the record might be locked up indefinitely. This problem is characteristic of record locking systems and you can avoid it with careful programming.

Only one portion of a file can be locked out at a time. If an application requires more than one record to be locked out, open multiple paths to the same file and lock the record accessed by each path. RBF notices that the same process owns both paths and keeps them from locking each other out.

Non-Shareable Files

Sometimes (although rarely), you must create a file that can never be accessed by more than one user at a time. To lock the file, you set the single-user (s) bit in the file's attribute byte. You can do this by using the proper option when the file is created, or later using the OS-9 ATTR command. Once the single-user bit is set, only one user can open the file at a time. If other users attempt to open the file, Error 253 is returned. Note however, that non-shareable means only one path can be opened to a file at one time. Do not allow two processes to concurrently access a non-shareable file through the same path.

More commonly, you need to declare a file as single-user only during the execution of a specific program. You can do this by opening the file with the single-user bit set. For example, suppose a process is sorting a file. With the file's single-user bit set, OS-9 treats the file exactly as though it had a single-user attribute. If another process attempts to open the file, OS-9 returns Error 253.

You can duplicate non-shareable paths by using the I$Dup system call. This means that it can be inherited, and therefore accessible to more than one process at a time. Single-user means that the file can be opened only once.

End-of-File Lock

A special case of record locking occurs when a user reads or writes data at the end of a file, creating an *EOF Lock*. An EOF Lock keeps the end of the file locked out until a process performs a READ or WRITE that is not at the end of the file. It prevents problems that might otherwise occur when two users want to simultaneously extend a file. The EOF Lock is the only case in which a WRITE call automatically causes portions of a file to be locked out. An interesting and useful side effect of the EOF Lock function occurs if a program creates a file for sequential output. As soon as the program creates the file, EOF Lock is set and no other process can *pass* the writer in processing the file. For example, if an assembler redirects a listing to a disk file, and a spooler utility tries to print a line from the file before it is written, record locking makes the spooler wait and stay at least one step behind the assembler.

Deadlock Detection

A deadly embrace, or deadlock, typically occurs when two processes attempt to gain control of two or more disk areas at the same time. If each process gets one area (locking out the other process), both processes become permanently stuck. Each waits for a segment that can never become free. This situation is not restricted to any particular record locking scheme or operating system.

When a deadly embrace occurs, RBF returns a deadlock error (Error 254) to the process that caused OS-9 to detect the deadlock. To avoid deadlocks, make sure that processes always access records of shared files in the same sequence.

When a deadlock error occurs, it is not sufficient for a program to retry the operation that caused the error. If all processes use this strategy, none can ever succeed. For any process to proceed, at least one must cancel operation to release its control over a requesting segment.

RBF-Type Device Driver Modules

An RBF-type device driver module contains a package of subroutines that perform sector-oriented I/O to or from a specific hardware controller. Such a module is usually re-entrant. Because of this, one copy of one device driver module can simultaneously run several devices that use identical I/O controllers.

The I/O manager allocates a permanent memory area for each device driver. The size of the memory area is given in the device driver module header. The I/O manager and the RBF manager use some of this area. The device driver can use the rest in any manner. This area is used as follows:

The RBF Device Memory Area Definitions

Name	Relative Address	Size (Bytes)	Use
V.PAGE	$00	1	Port extended address bits A20-A16
V.PORT	$01	2	Device base address (defined by the I/O manager)

Name	Relative Address	Size (Bytes)	Use
V.LPRC	$03	1	ID of the last active process (not used by RBF device drivers)
V.BUSY	$04	1	ID of the current process using driver (defined by RBF) 0 = no current process
V.WAKE	$05	1	ID of the process waiting for I/O completion (defined by the device driver)
V.USER	$06	0	Beginning of file manager specific storage
V.NDRV	$06	1	Maximum number of drives the controller can use (defined by the device driver)
	$07	8	Reserved
DRVBEG	$0F	0	Beginning of the drive tables
TABLES	$0F	DRVMEN*N	Space for number of tables reserved (*n*)
FREE		0	Beginning of space available for driver

These values are defined in files in the DEFS directory on the Development Package disk.

TABLES. This area contains one table for each drive that the controller handles. (The RBF manager assumes that there are as many tables as indicated by V.NDRV.) Some time after the driver Init routine is called, the RBF manager issues a request for the driver to read LSN 0 from a drive table by copying the first part of LSN 0 (up to DD.SIZ) into the table. Following is the format of each drive table:

Name	Relative Address	Size (Bytes)	Use
DD.TOT	$00	3	Number of sectors.
DD.TKS	$03	1	Track size (in sectors).
DD.MAP	$04	2	Number of bytes in the allocation bit map.
DD.BIT	$06	2	Number of sectors per bit (cluster size).
DD.DIR	$08	3	Address (LSN) of the ROOT directory.
DD.OWN	$0B	2	Owner's user number.
DD.ATT	$0D	1	Disk access attributes (D S PE PW PR E W R).
DD.DSK	$0E	2	Disk ID (a pseudo-random number used to detect diskette swaps).
DD.FMT	$10	1	Media format.
DD.SPT	$11	2	Number of sectors per track. (Track 0 can use a different value specified by IT.TOS in the device descriptor.)
DD.RES	$13	2	Reserved for future use.
DD.SIZ	$15	0	Minimum size of device descriptor.
V.TRAK	$15	2	Number of the current track (the track that the head is on, and the track updated by the driver).
V.BMB	$17	1	Bit-map use flag: 0 = Bit map is not in use. (Disk driver routines must not alter V.BMB.)
V.FILEHD	$18	2	Open file list for this drive.

Name	Relative Address	Size (Bytes)	Use
V.DISKID	$1A	2	Disk ID.
V.BMAPSZ	$1C	1	Size of bitmap.
V.MAPSCT	$1D	1	Lowest reasonable bitmap sector.
V.RESBIT	$1E	1	Reserved bitmap sector.
V.SCTKOF	$1F	1	Sector/track byte.
V.SCOFST	$20	1	Sector offset split from V.SCTKOF.
V.TKOFST	$21	1	Track offset split from V.SCTKOF.
RESERVED	$22	4	Reserved for future use.
DRVMEN	$26		Size of each drive table.

The format attributes (DD.FMT) are these:

```
Bit B0 = Number of sides
     0 = Single-sided
     1 = Double-sided

Bit B1 = Density
     0 = Single-density
     1 = Double-density

Bit B2 = Track density
     0 = Single (48 tracks per inch)
     1 = Double (96 tracks per inch)
```

RBF Device Driver Subroutines

Like all device driver modules, RBF device drivers use a standard executable memory module format.

The execution offset address in the module header points to a branch table that has six 3-byte entries. Each entry is typically a long branch (LBRA) to the corresponding subroutine. The branch table is defined as follows:

```
ENTRY    LBRA    INIT      Initialize drive
         LBRA    READ      Read sector
         LBRA    WRITE     Write sector
         LBRA    GETSTA    Get status
         LBRA    SETSTA    Set status
         LBRA    TERM      Terminate device
```

Ensure that each subroutine exists with the C bit of the condition code register cleared if no error occurred. If an error occurs, set the C bit and return an appropriate error code Register B.

The rest of this chapter describes the RBF device driver subroutines and their entry and exit conditions.

Init **Initializes a device and the device's memory area.**

Entry Conditions:

Y = *address of the device descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If you want OS-9 to verify disk writes, use the Request Memory system call (F$SRqMem) to allocate a 256-byte buffer area in which a sector can be read back and verified after a write.
- You must initialize the device memory area. For floppy diskette controllers, initialization typically consists of:
  1. Initializing V.NDRV to the number of drives with which the controller works
  2. Initializing DD.TOT (in the drive table) to a non-zero value so that Sector 0 can be read or written
  3. Initializing V.TRAK to $FF so that the first seek finds Track 0
  4. Placing the IRQ service routine on the IRQ polling list, using the Set IRQ system call (F$IRQ)
  5. Initializing the device control registers (enabling interrupts if necessary)
- Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (These areas contain the 24- bit device address.) Ensure the driver initializes each drive table appropriately for the type of diskette that the driver expects to be used on the corresponding drive.

Read **Reads a 256-byte sector from a disk and places it in a 256-byte sector buffer.**

Entry Conditions:

B = *MSB of the disk's LSN*
X = *LSB of the disk's LSN*
Y = *address of the path descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The following is a typical routine for using Read:
  1. Get the sector buffer address from PD.BUF in the path descriptor.
  2. Get the drive number from PD.DRV in the path descriptor.
  3. Compute the physical disk address from the logical sector number.
  4. Initiate the Read operation.
  5. Copy V.BUSY to V.WAKE. The driver goes to sleep and waits for the I/O to complete. (The IRQ service routine is responsible for sending a wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it isn't clear, the driver goes back to sleep.
- Whenever you read LSN 0, you must copy the first part of this sector into the proper drive table. (Get the drive number from PD.DRV in the path descriptor.) The number of bytes to copy is in DD.SIZ. Use the drive number (PD.DRV) to compute the offset for the corresponding drive table as follows:

```
LDA PD.DRV,Y      Get the drive number
LDB #DRVMEN       Get the size of a
                  drive table
MUL
LEAX DRVBEG,U     Get the address of
                  the first table
LEAX D,X          Compute the address
                  of the table
```

Write Writes a 256-byte sector buffer to a disk.

Entry Conditions:

B = *MSB of the disk LSN*
X = *LSB of the disk LSN*
Y = *address of the path descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code*

Additional Information:

- Following is a typical routine for using Write:

1. Get the sector buffer address from PD.BUF in the path descriptor.
2. Get the drive number from PD.DRV in the path descriptor.
3. Compute the physical disk address from the logical sector number.
4. Initiate the Write operation.
5. Copy V.BUSY to V.WAKE. The driver then goes to sleep and waits for the I/O to complete. (The IRQ service routine sends the wakeup signal.) After awakening, the driver tests V.WAKE to see if it is clear. If it is not, the driver goes back to sleep. If the disk controller cannot be interrupt-driven, it is necessary to perform a programmed I/O transfer.
6. If PF.VFY in the path descriptor is equal to zero, read the sector back in and verify that it is written correctly. Verification usually does not involve a comparison of all of the data bytes.

- If disk writes are to be verified, the Init routine must request the buffer in which to place the sector when it is read back. Do not copy LSN 0 into the drive table when reading it back for verification.

- Use the drive number (PD.DRV) to compute the offset to the corresponding drive table as shown for the Read routine.

Getstats and Setstats

Reads or changes device's operating parameters.

Entry Conditions:

U = *address of the device memory area*
Y = *address of the path descriptor*
A = *status code*

Exit Conditions:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- Get/set the device's operating parameters (status) as specified for the Get Status and Set Status system calls. Getsta and Setsta are wild card calls.
- It might be necessary to examine or change the register stack that contains the values of the 6809 register at the time of the call. The address of the register stack is in PD.RGS, which is located in the path descriptor. You can use the following offsets to access any value in the register stack:

Reg.	Relative Addr.	Size	6809 Reg.
R$CC	$00	1	Condition Code Reg.
R$D	$01	2	Register D
R$A	$01	1	Register A
R$B	$02	1	Register B
R$DP	$03	1	Register DP
R$X	$04	2	Register X
R$Y	$06	2	Register Y
R$U	$08	2	Register U
R$PC	$0A	2	Program Counter

- Register D overlays Registers A and B.

Term Terminate a device.

Entry Conditions:

U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- This routine is called when a device is no longer in use in the system (when the link count of its device descriptor module becomes zero).
- Following is a typical routine for using Term:

  1. Wait until any pending I/O is completed.
  2. Disable the device interrupts.
  3. Remove the device from the IRQ polling list.
  4. If the Init routine reserved a 256-byte buffer for verifying disk writes, return the memory with the Return Sysmem system call (F$SRtMem).

IRQ Service Routine

Services device interrupts.

Additional Information:

- The IRQ Service routine sends a wakeup signal to the process indicated by the process ID in V.WAKE when the I/O is complete. It then clears V.WAKE as a flag to indicate to the main program that the IRQ has indeed occurred.
- When the IRQ service routine finishes servicing an interrupt it must clear the carry and exit with an RTS instruction.
- Although this routine is not included in the device driver module branch table and is not called directly by the RBF manager, it is a key routine in interrupt-driven drivers. Its function is to:

  1. Service the device interrupts (receive data from device or send data to it). The IRQ service routine puts its data into and get its data from buffers that are defined in the device memory area.
  2. Wake up a process that is waiting for I/O to be completed. To do this, the routine checks to see if there is a process ID in V.WAKE (if the bit is non-zero); if so, it sends a wakeup signal to that process.
  3. If the device is ready to send more data, and the output buffer is empty, disable the device's *ready to transmit* interrupts.

Boot (Bootstrap Module)

Loads the boot file into RAM.

Entry Conditions:

None

Exit Conditions:

D = *size of the boot file* (in bytes)
X = *address at which the boot file was loaded into memory*
CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Boot module is not part of the disk driver. It is a separate module that is stored on the boot track of the system disk with OS9P1 and REL.
- The bootstrap module contains one subroutine that loads the bootstrap file and related information into memory. It uses the standard executable module format with a module type of $C. The execution offset in the module header contains the offset to the entry point of this subroutine.
- The module gets the starting sector number and size of the OS9Boot file from LSN 0. OS-9 allocates a memory area large enough for the Boot file. Then, it loads the Boot file into this memory area.
- Following is a typical routine for using Boot:
  1. Read LSN 0 from the disk into a buffer area. The Boot module must pick its own buffer area. LSN 0 contains the values for DD.BT (the 24-bit LSN of the bootstrap file), and DD.BSZ (the size of the bootstrap file in bytes).
  2. Get the 24-bit LSN of the bootstrap file from DD.BT.
  3. Get the size of the bootstrap file from DD.BSZ. The Boot module is contained in one logically contiguous block beginning at the logical sector specified in DD.BT and extending for DD.BSZ/256 + 1 sectors.

4. Use the OS-9 Request Sysmem system call (F$SRqMem) to request the memory area in which the Boot file is loaded.
5. Read the Boot file into this memory area.
6. Return the size of the Boot file and its location. Boot file is loaded.

Chapter 6

Sequential Character File Manager

The Sequential Character File Manager (SCF) supports devices that operate on a character-by-character basis. These include terminals, printers, and modems.

SCF is a re-entrant subroutine package. The I/O manager calls the SCF manager for I/O system handling of sequential, character-oriented devices. The SCF manager includes the extensive I/O editing functions typical of line-oriented operation, such as:

- backspace
- line delete
- line repeat
- auto line feed
- screen pause
- return delay padding

The SCF-type device driver modules are CC3IO, PRINTER, and RS-232. They run the video display, printer, and serial ports respectively. See the *OS-9 Commands* manual for additional Color Computer I/O devices.

SCF Line Editing Functions

The SCF manager supports two sets of read and write functions. I$Read and I$Write pass data with no modification. I$ReadLn and I$WritLn provide full line editing of device functions.

Read and Write

The Read and Write system calls to SCF-type devices correspond to the BASIC09 GET and PUT statements. While they perform little modification to the data they pass, they do filter out keyboard interrupt, keyboard terminate, and pause character. (Editing is disabled if the corresponding character in the path descriptor contains a zero.)

Carriage returns are not followed by line feeds or nulls automatically, and the high order bits are passed as sent/received.

Read Line and Write Line

The Read Line and Write Line system calls to SCF-type devices correspond to the BASIC09 INPUT, PRINT, READ, and WRITE statements. They provide full line editing of all functions enabled for a particular device.

The system initializes I$ReadLn and I$WritLn functions when you first use a particular device. (OS-9 copies the option table from the device descriptor table associated with the specific device.)

Later, you can alter the calls—either from assembly-language programs (using the Get Status system call), or from the keyboard (using the TMODE command). All bytes transferred by Readln and Writln have the high order bit cleared.

SCF Definitions of the Path Descriptor

The PD.FST and PD.OPT sections of the path descriptor are reserved for and used by the SCF file manager.

The following table describes the SCF manager's use of PD.FST and PD.OPT. For your convenience, the table also includes the other sections of the PD.

The PD.OPT section contains the values that determine the line editing functions. It contains many device operating parameters that can be read or written by the Set Status or Get Status system call. Any values not set by this table default to zero.

> **Note:** You can disable most of the editing functions by setting the corresponding control character in the path descriptor to zero. You can use the Set Status system call or the TMODE command to do this. Or, you can go a step further by setting the corresponding control character value in the device descriptor module to zero.

To determine the default settings for a specific device, you can inspect the device descriptor.

Name	Relative Address	Size (Bytes)	Use
Universal Section (Same for all file managers)			
PD.PD	$00	1	Path number
PD.MOD	$01	1	Access mode: 1 = read 2 = write 3 = update
PD.CNT	$02	1	Number of open images (paths using this PD)
PD.DEV	$03	2	Address of the associated device table entry
PD.CPR	$05	1	Current process ID
PD.RGS	$06	2	Address of the caller's 6809 register stack
PD.BUF	$08	2	Address of the 256-byte data buffer (if used)

Name	Relative Address	Size (Bytes)	Use
SCF Path Descriptor Definitions (PD.FST Section)			
PD.DV2	$0A	2	Device table address of the second (echo) device
PD.RAW	$0C	1	Edit flag: 0 = raw mode 1 = edit mode
PD.MAX	$0D	2	Read Line maximum character count
PD.MIN	$0F	1	Devices are *mine* if cleared
PD.STS	$10	2	Status routine module address
PD.STM	$12	2	Reserved for status routine

Name	Relative Address	Size (Bytes)	Use
SCF Option Section Definition (PD.OPT Section)			
(Copied from the device descriptor)			
PD.DTP	$20	1	Device class: 0 = SCF 1 = RBF 2 = PIPE 3 = SBF
PD.UPC	$21	1	Case: 0 = upper and lower 1 = upper only
PD.BSO	$22	1	Backspace: 0 = backspace 1 = backspace, space and backspace
PD.DLO	$23	1	Delete: 0 = backspace over line 1 = carriage return, line feed
PD.EKO	$24	1	Echo: 0 = no echo
PD.ALF	$25	1	Auto line feed: 0 = no auto line feed
PD.NUL	$26	1	End-of-line null count: *n* = number of nulls ($00) sent after each carriage return or carriage return and line feed (*n* = $00-$FF)
PD.PAU	$27	1	End of page pause: 0 = no pause
PD.PAG	$28	1	Number of lines per page
PD.BSP	$29	1	Backspace character
PD.DEL	$2A	1	Delete-line character

Name	Relative Address	Size (Bytes)	Use
SCF Option Section Definition continued (PD.OPT Section)			
PD.EOR	$2B	1	End-of-record character (End-of-line character) Read only. Normally set to $0D: 0 = Terminate read-line only at the end of the file
PD.EOF	$2C	1	End-of-file character (read only)
PD.RPR	$2D	1	Reprint-line character
PD.DUP	$2E	1	Duplicate-last-line character
PD.PSC	$2F	1	Pause character
PD.INT	$30	1	Keyboard-interrupt character
PD.QUT	$31	1	Keyboard-terminate character
PD.BSE	$32	1	Backspace-echo character
PD.OVF	$33	1	Line-overflow character (bell CTRL G)
PD.PAR	$34	1	Device-initialization value (parity)
PD.BAU	$35	1	Software setable baud rate
PD.D2P	$36	2	Offset to second device name string
PP.XON	$38	1	ACIA XON char
PD.XOFF	$39	1	ACIA XOFF char
PD.ERR	$3A	1	Most recent I/O error status
PD.TBL	$3B	2	Copy of device table address
PD.PLP	$3D	2	Path descriptor list pointer
PD.PST	$3F	1	Current path status

PD.EOF specifies the end-of-file character. If this is the first and only character that is input to the SCF device, SCF returns an end-of-file error on Read or Readln.

PD.PSC specifies the pause character, which suspends output to the device before the next end-of-record character. The pause character also deletes any type-ahead input for Readln.

PD.INT specifies the keyboard-interrupt character. When the character is received, the system sends a keyboard terminate signal to the last user of a path. The character also terminates the current I/O request (if any) with an error identical to the keyboard interrupt signal code.

PD.QUT specifies the keyboard-terminate character. When this character is received, the system sends a keyboard terminate signal to the last user of a path. The system also cancels the current I/O request (if any) by sending error code identical to the keyboard interrupt signal code.

PD.PAR specifies the parity information for external serial devices.

PD.BAU specifies baud rate, word length and stop bit information for serial devices.

PD.XON contains either the character used to enable transmission of characters or a null character that disables the use of XON.

PD.XOFF contains either the character used to disable transmission of characters or a null character that disables the use of XOFF.

SCF-Type Device Descriptor Modules

The following chart shows how the initialization table in the device descriptors is used for SCF-type devices. The values are those the I/O manager copies from the device descriptor to the path descriptor.

An SCF editing function is turned off if its corresponding value is set to zero. For example, if IT.EOF is set to zero, there is no end-of-file character.

Name	Relative Address	Size (Bytes)	Use
(header)	$00-11		Standard device descriptor module header
IT.DVC	$12	1	Device class: 0 = SCF 1 = RBF 2 = PIPE 3 = SBF
IT.UPC	$13	1	Case: 0 = upper- and lowercase 1 = uppercase only
IT.BSO	$14	1	Backspace: 0 = backspace 1 = backspace, then space and backspace
IT.DLO	$15	1	Delete: 0 = backspace over line 1 = carriage return
IT.EKO	$16	1	Echo: 0 = echo off
IT.ALF	$17	1	Auto line feed: 0 = auto line feed disabled
IT.NUL	$18	1	End-of-line null count
IT.PAU	$19	1	Pause: 0 = end-of-page pause disabled
IT.PAG	$1A	1	Number of lines per page
IT.BSP	$1B	1	Backspace character
IT.DEL	$1C	1	Delete-line character
IT.EOR	$1D	1	End-of-record character
IT.EOF	$1E	1	End-of-file character
IT.RPR	$1F	1	Reprint-line character

Name	Relative Address	Size (Bytes)	Use
IT.DUP	$20	1	Duplicate-last-line character
IT.PSC	$21	1	Pause character
IT.INT	$22	1	Interrupt character
IT.QUT	$23	1	Quit character
IT.BSE	$24	1	Backspace echo character
IT.OVF	$25	1	Line-overflow character (bell)
IT.PAR	$26	1	Initialization value—used to initialize a device control register when a path is opened to it (parity)
IT.BAU	$27	1	Baud rate
IT.D2P	$28	2	Attached device name string offset
IT.XON	$2A	1	X-ON character
IT.XOFF	$2B	1	X-OFF character
IT.COL	$2C	1	Number of columns for display
IT.ROW	$2D	1	Number of rows for display
IT.WND	$2E	1	Window number
IT.VAL	$2F	1	Data in rest of descriptor is valid
IT.STY	$30	1	Window type
IT.CPX	$31	1	X cursor position
IT.CPY	$32	1	Y cursor position
IT.FGC	$33	1	Foreground color
IT.BGC	$34	1	Background color
IT.BDC	$35	1	Border color

SCF-Type Device Driver Modules

An SCF-type device driver module contains a package of subroutines that perform raw (unformatted) data I/O transfers to or from a specific hardware controller. Such a module is usually reentrant so that one copy of the module can simultaneously run several devices that use identical I/O controllers. The I/O manager allocates a permanent memory area for each controller sharing the driver.

The size of the memory area is defined in the device driver module header. The I/O manager and SCF use some of the memory area. The device driver can use the rest in any way (typically as variables and buffers). Typically, the driver uses the area as follows:

Name	**Relative Address**	**Size (Bytes)**	**Use**
V.PAGE	$00	1	Port extended 24 bit address
V.PORT	$01	2	Device base address (defined by the I/O manager)
V.LPRC	$03	1	ID of the last active process
V.BUSY	$04	1	ID of the active process (defined by RBF): 0 = no active process
V.WAKE	$05	1	ID of the process to reawaken after the device completes I/O (defined by the device driver): 0 = no waiting process
V.USER	$06	0	Beginning of file manager specific storage
V.TYPE	$06	1	Device type or parity
V.LINE	$07	1	Lines left until the end of the page
V.PAUS	$08	1	Pause request: 0 = no pause requested
V.DEV2	$09	2	Attached device memory area
V.INTR	$0B	1	Interrupt character

Name	Relative Address	Size (Bytes)	Use
V.QUIT	$0C	1	Quit character
V.PCHR	$0D	1	Pause character
V.ERR	$0E	1	Error accumulator
V.XON	$0F	1	XON character
V.XOFF	$10	1	XOFF character
V.KANJI	$11	1	Reserved
V.KBUF	$12	2	Reserved
V.MODADR	$14	2	Reserved
V.PDLHD	$16	2	Path descriptor list header
V.RSV	$18	5	Reserved
V.SCF	$1D	0	End of SCF memory requirements
FREE	$1D	0	Free for the device driver to use

V.LPRC contains the process ID of the last process to use the device. The IRQ service routine sends this process the proper signal if it receives a quit character or an interrupt character. V.LPRC is defined by SCF.

V.BUSY contains the process ID of the process that is using the device. (If the device is not being used, V.BUSY contains a zero.) The process ID is used by SCF to prevent more than one process from using the device at the same time. V.BUSY is defined by SCF.

SCF Device Driver Subroutines

Like all device drivers, SCF device drivers use a standard executable memory module format.

The execution offset address in the module header points to a branch table that has six 3-byte entries. Each entry is typically an LBRA to the corresponding subroutine. The branch table is defined as follows:

```
ENTRY    LBRA     INIT      Initialize driver
         LBRA     READ      Read character
         LBRA     WRITE     Write character
         LBRA     GETSTA    Get status
         LBRA     SETSTA    Set status
         LBRA     TERM      Terminate device
```

If no error occurs, each subroutine exits with the C bit in the Condition Code Register cleared. If an error occurred, each subroutine sets the C bit and returns an appropriate error code in Register B.

The rest of this chapter describes these subroutines and their entry and exit conditions.

Init Initializes device control registers, and enables interrupts if necessary.

Entry Conditions:

Y = *address of the device descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Prior to being called, the device memory area is cleared (set to zero), except for V.PAGE and V.PORT. (V.PAGE and V.PORT contain the device address.) There is no need to initialize the part of the memory area used by the I/O manager and SCF.
- Follow these steps to use Init:
  1. Initialize the device memory area.
  2. Place the IRQ service routine on the IRQ polling list, using the Set IRQ system call (F$IRQ).
  3. Initialize the device control registers.

Read **Reads the next character from the input buffer.**

Entry Conditions:

Y = *address of the path descriptor*
U = *address of the device memory area*

Exit Conditions:

A = *character read*
CC = carry set on error
B = *error code* (if any)

Additional Information:

- This is a step by step description of a Read operation:
  1. Read gets the next character from the input buffer.
  2. If no data is ready, Read copies its process ID from V.BUSY into V.WAKE. It then uses the Sleep system call to put itself to sleep.
  3. Later, when Read receives data, the IRQ service routine leaves the data in a buffer. Then, the routine checks V.WAKE to see if any process is waiting for the device to complete I/O. If so, the IRQ service routine sends a wakeup signal to the waiting process.
- Data buffers are not automatically allocated. If a buffer is used, it defines it in the device memory area.

Write **Sends a character (places a data byte in an output buffer) and enables the device output interrupts.**

Entry Conditions:

A = *character to write*
Y = *address of the path descriptor*
U = *address of the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If the data buffer is full, Write copies its process ID from V.BUSY into V.WAKE. Write then puts itself to sleep.

  Later, when the IRQ service routine transmits a character and makes room for more data, it checks V.WAKE to see if there is a process waiting for the device to complete I/O. If there is, the routine sends a wakeup signal to that process.

- Write must ensure that the IRQ service routine that starts it begins to place data in the buffer. After an interrupt is generated, the IRQ service routine continues to transmit data until the data buffer is empty. Then, it disables the device's ready-to-transmit interrupts.

- Data buffers are not allocated automatically. If a buffer is used, define it in the device memory area.

Getsta and Setsta

Gets/sets device operating parameters (status) as specified for the Get Status and Set Status system calls. Getsta and Setsta are wildcard calls.

Entry Conditions:

A = depends on the function code
Y = *address of the path descriptor*
U = *address of the device memory area*
Other registers depend on the function code.

Exit Conditions:

B = *error code* (if any)
CC = carry set on error
Other registers depend on the function code

Additional Information:

- Any codes not defined by the I/O manager or SCF are passed to the device driver.
- You might need to examine or change the register stack that contains the values of the 6809 registers at the time of the call. The address of the register stack can be found in PD.RGS, which is located in the path descriptor.
- You can use the following offsets to access any value in the register packet:

Name	Relative Address	Size (Bytes)	6809 Register
R$CC	$00	1	Condition Codes Register
R$D	$01	0	Register D
R$A	$01	1	Register A
R$B	$02	1	Register B
R$DP	$03	1	Register DP
R$X	$04	2	Register X
R$Y	$06	2	Register Y
R$U	$08	2	Register U
R$PC	$0A	2	Program Counter

The function code is retrieved from the R$B on the user stack.

Term

Terminates a device. Term is called when a device is no longer in use (when the link count of the device descriptor module becomes zero).

Entry Conditions:

U = *pointer to the device memory area*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- To use Term:
  1. Wait until the IRQ service routine empties the output buffer.
  2. Disable the device interrupts.
  3. Remove the device from the IRQ polling list.
- When Term closes the last path to a device, OS-9 returns to the memory pool the memory that the device used. If the device has been attached to the system using the I$Attach system call, OS-9 does not return the static storage for the driver until an I$Detach call is made to the device. Modules contained in the Boot file are never terminated, even if their link counts reach 0.

IRQ Service Routine

Receives device interrupts. When I/O is complete, the routine sends a wakeup signal to the process identified by the process ID in V.WAKE. The routine also clears V.WAKE as a flag to indicate to the main program that the IRQ has occurred.

Additional Information:

- The IRQ Service Routine is not included in device driver branch tables, and is not called directly by SCF. However, it is a key routine in device drivers.
- When the IRQ Service routine finishes servicing an interrupt, the routine must clear the carry and exit with an RTS instruction.
- Here is a typical sequence of events that the IRQ Service Routine performs:

  1. Service the device interrupts (receive data from the device or send data to it). Ensure this routine puts its data into and get its data from buffers that are defined in the device memory area.
  2. Wake up any process that is waiting for I/O to complete. To do this, the routine checks to see if there is a process ID in V.WAKE (a value other than zero); if so, it sends a wakeup signal to that process.
  3. If the device is ready to send more data, and the output buffer is empty, disable the device's ready-to-transmit interrupts.
  4. If a pause character is received, set V.PAUS in the attached device storage area to a value other than zero. The address of the attached device memory area is in V.DEV2.

     V.PAUS in the attached device storage area to non-zero value. The address of the attached device memory area is in V.DEV2.
  5. If a keyboard terminate or interrupt character is received, signal the process in V.LPRC (last known process) if any.

Chapter 7

The Pipe File Manager (PIPEMAN)

The Pipe file manager handles control of processes that use paths to pipes. Pipes allow concurrently executing processes to send each other data by using the output of one process (the writer) as input to a second process (the reader). The reader gets input from the standard input. The exclamation point (!) operator specifies that the input or output is from or to a pipe. The Pipe file manager allocates a 256-byte block and a path descriptor for data transfer. The Pipe file manager also determines which process has control of the pipe. The Pipe file manager has the standard file manager branch table at its entry point:

```
PipeEnt   lbra Create
          lbra Open
          lbra MakDir
          lbra ChgDir
          lbra Delete
          lbra Seek
          lbra PRead
          lbra PWrite
          lbra PRdLn
          lbra PWrLn
          lbra Getstat
          lbra Putstat
          lbra Close
```

You cannot use MakDir, ChgDir, Delete, and Seek with pipes. If you try to do so, the system returns E$UNKSVC (unknown service request). Getstat and Putstat are also no-action service routines. They return without error.

Create and Open perform the same functions. They set up the 256-byte data exchange buffer, and save several addresses in the path descriptor.

The Close request checks to see if any process is reading or writing through the pipe. If not, OS-9 returns the buffer.

PRead, PWrite, PRdLn, and PWrLn read data from the buffer and write data to it.

The ! operator tells the Shell that processes wish to communicate through a pipe. For example:

```
proc1 ! proc2 [ENTER]
```

In this example, shell forks Proc1 with the standard output path to a pipe, and forks Proc2 with the Standard input path from a pipe.

Shell can also handle a series of processes using pipes. Example:

```
proc1 ! proc2 ! proc3 ! proc4 [ENTER]
```

The following outline shows how to set up pipes between processes:

```
Open /pipe       save path in variable x
Dup path #1      save stdout in variable y
Dup x            make path available
Fork proc1       put pipe in stdout
                 (Dup uses lowest available)
Close #1         make path available
Dup y            restore stdout
Close y          make path available
Dup path #0      save stdin in Y
Close #0         make path available
Dup x            put pipe in stdin
Fork 2           fork process 2
Close #0         make path available
Dup y            restore stdin
Close x          no longer needed
Close y          no longer needed
```

Example: The following example shows how an application can initiate another process with the stdin and stdout routed through a pipe.

```
Open /pipe1      save path in variable a
Open /pipe2      save path in variable b
Dup 0            save stdin in variable x
Dup 1            save stdout in variable y
Close 0          make path available
Close 1          make path available
Dup a            make pipe1 stdin
Dup b            make pipe2 stdout
Fork new process
Close 0          make path available
Close 1          make path available
Dup x            restore stdin
Dup y            restore stdout
Return a&b       return pipe path numbers to caller
```

Chapter 8

System Calls

System calls are used to communicate between the OS-9 operating system and assembly-language programs. There are two major types of calls—I/O calls and function calls.

Function calls include user mode calls and system mode calls.

Each system call has a mnemonic name. Names of I/O calls start with I$. For example, the Change Directory call is I$ChgDir. Names of function calls start with F$. For example, the Allocate Bits call is F$AllBit. The names are defined in the assembler-input conditions equate file called OS9Defs.

System mode calls are privileged. You can execute them only while OS-9 is in the system state (when it is processing another system call, executing a file manager or device driver, and so on).

System mode calls are included in this manual primarily for programmers writing device drivers and other system-level applications.

Calling Procedure

To execute any system calls, you need to use an SWI2 instruction:

1. Load the 6809 registers with any appropriate parameters.
2. Execute an SWI2 instruction, followed immediately by a constant byte, which is the request code. In the references in this chapter, the first line is the system call name (for example Close Path) and the second line contains the call's mnemonic name (for example I$Close), the software interrupt Code 2 (103F), and the call's request code (for example, 8F) in hexadecimal.
3. After OS-9 processes the call, it returns any parameters in the 6809 registers. If an error occurs, the C bit of the condition code register is set, and Register B contains the appropriate error code. This feature permits a BCS or BCC instruction immediately following the system call to branch either if there is an error or if no error occurs.

As an example, here is the Close system call:

```
LDA PATHNUM
SWI2
FCB  $8F
BCS  ERROR
```

You can use the assembler's *OS9 directive* to simplify the call, as follows.

```
LDA   PATHNUM
OS9   I$Close
BCS   ERROR
```

The ASM assembler allows any combination of upper- or lower-case letters. The RMA assembler, included in the OS-9 Level Two Development Pak, is case sensitive. The names in this manual have been spelled with upper and lower case letters, matching the defs for RMA.

I/O System Calls

OS-9's I/O calls are easier to use than many other systems' I/O calls. This is because the calling program does not have to allocate and set up *file control blocks, sector buffers,* and so on.

Instead, OS-9 returns a 1-byte path number whenever a process opens a path to a file or device. Until the path is closed, you can use this path number in later I/O requests to identify the file or device.

In addition, OS-9 allocates and maintains its own data structures; so, you need not deal with them.

System Call Descriptions

The rest of this chapter consists of the system call descriptions. At the top of each description is the system call name, followed by its mnemonic name, the SWI2 code and the request code. Next are the call's entry and exit conditions, followed by additional information about the code where appropriate.

In the system call descriptions, registers not specified as entry or exit conditions are not altered. Strings passed as parameters are normally terminated with a space character and end-of-line character, or with Bit 7 of the last character set.

If an error occurs on a system call, the C bit of Register CC is set, and Register B contains the *error code*. If no error occurs, the C bit is clear, and Register B contains a value of zero.

User Mode System Calls Quick Reference

Following is a summary of the User Mode System Calls referenced in this chapter:

F$AllBit	Sets bits in an allocation bit map
F$Chain	Chains a process to a new module
F$CmpNam	Compares two names
F$CpyMem	Copies external memory
F$CRC	Generates a cyclic redundancy check
F$DelBit	Deallocates bits in an allocation bit map
F$Exit	Terminates a process
F$Fork	Starts a new process
F$GBlkMp	Gets a copy of a system block map
F$GModDr	Gets a copy of a module directory
F$GPrDsc	Gets a copy of a process descriptor
F$Icpt	Sets a signal intercept trap
F$ID	Returns a process ID
F$Link	Links to a memory module
F$Load	Loads a module from mass storage
F$Mem	Changes a process's data area size
F$NMLink	Links to a module; does not map the module into the user's address space
F$NMLoad	Loads a module but does not map it into the user's address space
F$Perr	Prints an error message
F$PrsNam	Parses a pathlist name
F$SchBit	Searches a bit map

F$Send	Sends a signal to a process
F$Sleep	Suspends a process
F$SPrior	Sets a process's priority
F$SSWI	Sets a software interrupt vector
F$STime	Sets the system time
F$SUser	Sets the user ID number
F$Time	Returns the current time
F$UnLink	Unlinks a module
F$UnLoad	Unlinks a module by name
F$Wait	Waits for a signal
I$Attach	Attaches an I/O device
I$Chgdir	Changes a working directory
I$Close	Closes a path
I$Create	Creates a new file
I$Delete	Deletes a file
I$DeletX	Deletes a file from the execution directory
I$Detach	Detaches an I/O device
I$Dup	Duplicates a path
I$GetStt	Gets a device's status
I$MakDir	Creates a directory file
I$Open	Opens a path to an existing file
I$Read	Reads data from a device
I$ReadLn	Reads a line of data from a device
I$Seek	Positions a file pointer
I$SetStt	Sets a device's status
I$Write	Writes data to a device
I$WritLn	Writes a data line to a device

System Mode Calls Quick Reference

Following is a summary of the System Mode Calls referenced in this chapter:

F$Alarm	Sets up an alarm
F$All64	Allocates a 64-byte memory block
F$AllHRAM	Allocates high RAM
F$AllImg	Allocates image RAM blocks
F$AllPrc	Allocates a process descriptor
F$AllRAM	Allocates RAM blocks
F$AllTsk	Allocates a process task number
F$AProc	Enters active process queue
F$Boot	Performs a system bootstrap
F$BtMem	Performs a memory request bootstrap
F$ClrBlk	Clears the specified block of memory
F$DATLog	Converts a DAT block offset to a logical address
F$DelImg	Deallocates image RAM blocks
F$DelPrc	Deallocates a process descriptor
F$DelRAM	Deallocates RAM blocks
F$DelTsk	Deallocates a process task number
F$ELink	Links modules using a module directory entry
F$FModul	Finds a module directory entry
F$Find64	Finds a 64-byte memory block
F$FreeHB	Gets a free high block
F$FreeLB	Gets a free low block
F$GCMDir	Compacts module directory entries
F$GProcP	Gets a process's pointer

F$IODel	Deletes an I/O module
F$IOQu	Puts an entry into an I/O queue
F$IRQ	Makes an entry into IRQ polling table
F$LDABX	Loads Register A from 0,X in Task B
F$LDAXY	Loads A[X,[Y]]
F$LDDDXY	Loads D[D+X,[Y]]
F$MapBlk	Maps the specified block
F$Move	Moves data to a different address space
F$NProc	Starts the next process
F$RelTsk	Releases a task number
F$ResTsk	Reserves a task number
F$Ret64	Returns a 64-byte memory block
F$SetImg	Sets a process DAT image
F$SetTsk	Sets a process's task DAT registers
F$SLink	Performs a system link
F$SRqMem	Performs a system memory request
F$SRtMem	Performs a system memory return
F$SSvc	Installs a function request
F$STABX	Stores Register A at 0,x in Task B
F$VIRQ	Makes an entry in a virtual IRQ polling table
F$VModul	Validates a module

User System Calls

Allocate Bits

OS9 F$AllBit 103F 13

Sets bits in an allocation bit map

Entry Conditions:

D = *number of the first bit to set*
X = *starting address of the allocation bit map*
Y = *number of bits to set*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Bit numbers range from 0 to *n-1*, where *n* is the number of bits in the allocation bit map.
- **Warning:** Do not issue the Allocate Bits call with Register Y set to 0 (a bit count of 0).

Chain

OS9 F$Chain 103F 05

Loads and executes a new primary module without creating a new process

Entry Conditions:

A = *language/type code*
B = *size of the data area* (in pages); must be at least one page
X = *address of the module name or filename*
Y = *parameter area size* (in bytes); defaults to zero if not specified
U = *starting address of the parameter area*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Chain loads and executes a new primary module, but does not create a new process. A Chain system call is similar to a Fork followed by an Exit, but it has less processing overhead. Chain resets the calling process program and data memory areas and begins executing a new primary module. It does not affect open paths. This is a user mode system call.

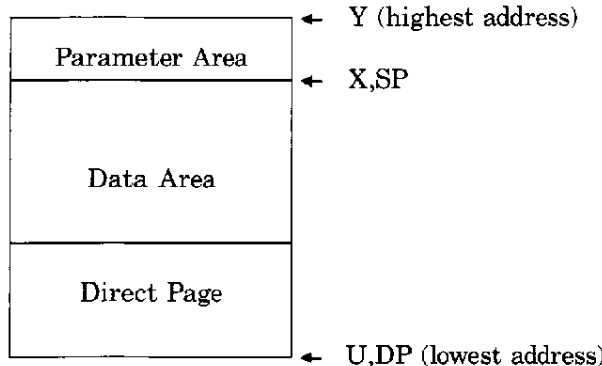
- **Warning:** Make sure that the hardware stack pointer (Register SP) is located in the direct page before Chain executes. Otherwise the system might crash or return a suicide attempt error. This precaution also prevents a suicide in the event that the new module requires a smaller data area than that in use. Allow approximately 200 bytes of stack space for execution of the Chain system call.

- Chain performs the following steps:

  1. It causes OS-9 to unlink the process's old primary module.

2. OS-9 parses the name string of the new process's primary module (the program that is to be executed first). Then, it causes OS-9 to search the system module directory to see if a module with the same name, type, and language is already in memory.
3. If the module is in memory, it links to it. If the module is not in memory, it uses the name string as the pathlist of a file to load into memory. Then, it links to the first module in this file. (Several modules can be loaded from a single file.)
4. It reconfigures the data memory area to the size specified in the new primary module's header.
5. It intercepts and erases any pending signals.

   The following diagram shows how Chain sets up the data memory area and registers for the new module.



D = *parameter area size*
PC = *module entry point absolute address*
CC = F=0, I=0; others are undefined

Registers Y and U (the top-of-memory and bottom-of-memory pointers, respectively) always have values at page boundaries. If the parent process does not specify a size for the parameter area, the size (Register D) defaults to zero. The data area must be at least one page long.

(For more information, see the Fork system call.)

Compare Names

OS9 F$CmpNam 103F 11

Compares two strings for a match

Entry Conditions:

B = *length of string1*
X = *address of string1*
Y = *address of string2*

Exit Conditions:

CC = carry clear if the strings match

Additional Information:

- The Compare Names call compares two strings and indicates whether they match. Use this call with the Parse Name system call. The second string must have the most significant bit (Bit 7) of the last character set.

Copy External Memory

Reads external memory into the user's buffer for inspection

OS9 F$CpyMem
103F 1B

Entry Conditions:

D = *DAT image pointer*
X = *offset in block to begin copy*
Y = *byte count*
U = *caller's destination buffer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- You can view any system memory through the use of the Copy External Memory call. The call assumes X is the address of the 64K space described by the DAT image given.
- If you pass the entire DAT image of a process, place a value in the X Register that equals the address in the process space. If you pass a partial DAT image (the upper half), then place a value in Register X that equals the offset from the beginning of the DAT image ($8000).
- The support module for this call is OS9p2.

CRC

OS9 F$CRC 103F 17

Calculates the CRC of a module

Entry Conditions:

X = *starting byte address*
Y = *number of bytes*
U = *address of the 3-byte CRC accumulator*

Exit Conditions:

Updates the CRC accumulator.

Additional Information:

- The CRC call calculates the CRC (cyclic redundancy count) for use by compilers, assemblers, or other module generators.
- The calculation begins at the *starting byte address* and continues over the specified *number of bytes*.
- You need not cover an entire module in one call, since the CRC can be accumulated over several calls. The CRC accumulator can be any 3-byte memory area. You must initialize it to $FFFFFF before the first CRC call.
- When checking an existing module CRC, the calculation should be performed on the entire module (including the module CRC). The CRC accumulator will contain the CRC constant bytes if the module CRC is correct.
- If the CRC of a new module is to be generated, the CRC is accumulated over the module (excluding CRC). The accumulated CRC is complemented then stored in the correct position in the module.
- Be sure to initialize the CRC accumulator only once for each module checked by CRC.

Deallocate Bits

OS9 F$DelBit 103F 14

Clears allocation map bits

Entry Conditions:

D = *number of the first bit to set*
X = *starting address of the allocation bit map*
Y = *number of bits to set*

Exit Conditions: None

Additional Information:

- The Deallocate Bits call clears bits in the allocation bit map pointed to by Register X. Bit numbers are in the range 0 to *n-1*, where *n* is the number of bits in the allocation bit map.
- **Warning:** Do not call Deallocate Bits with Register Y set to 0 (a bit count of 0).

Exit

Terminates the calling process

OS9 F$Exit 103F 06

Entry Conditions:

B = *status code to return to the parent*

Exit Conditions:

The process is terminated.

Additional Information:

- The Exit system call is the only way a process can terminate itself. Exit deallocates the process's data memory area, and unlinks the process's primary module. It also closes all open paths automatically.
- The Wait system call always returns to the parent the status code passed by the child in its Exit call. Therefore, if the parent executes a Wait and receives the status code, it knows the child has died. This is a user mode system call.
- Exit unlinks only the primary module. Unlink any module that is loaded or linked to by the process before calling Exit.

Fork

Creates a child process

OS9 F$Fork 103F 03

Entry Conditions:

A = *language/type code*
B = *size of the optional data area* (in pages)
X = *address of the module name or filename* (See the following example.)
Y = *size of the parameter area* (in pages); defaults to zero if not specified
U = *starting address of the parameter area*; must be at least one page

Exit Conditions:

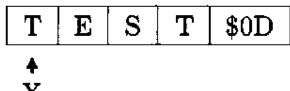
X = *address of the last byte of the name* + 1 (See the following example.)
A = new process IO number

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- Fork creates a new process, a child of the calling process. Fork also sets up the child process's memory and 6809 registers and standard I/O paths.
- Before the Fork call:



- After the Fork call:

T	E	S	T	$0D

(X points to the last T)

- This is the sequence of Fork's operations:

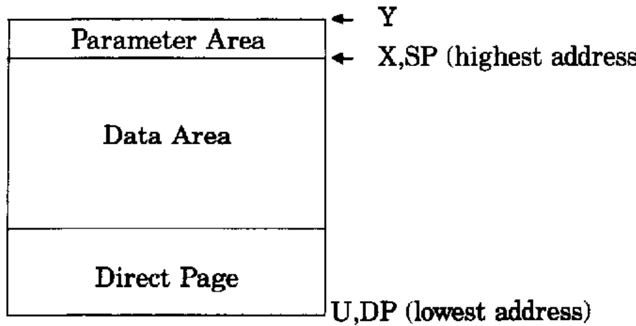
1. OS-9 parses the name string of the new process's primary module (the program that OS-9 executes first). Then, it searches the system module directory to see if the program already is in memory.

2a. The next step depends on whether or not the program is already in memory. If the program is in memory, OS-9 links the module to the process and executes it.

b. If the program is not in memory, OS-9 uses the name as the pathlist of the file that is to be loaded into memory. Then, the first module in this file is linked to and executed. (Several modules can be loaded from one file.)

3. OS-9 uses the primary module's header to determine the initial size of the process's data area. It then tries to allocate a contiguous RAM area of that size. (This area includes the parameter passing area, which is copied from the parent process's data area.)

4. The new process's data memory area and registers are set up as shown in the following diagram. OS-9 uses the execution offset given in the module header to set the program counter to the module's entry point.



D = *size of the parameter area*
PC = *module entry point absolute address*
CC = F = 0, I = 0, other condition code flags are undefined

Registers Y and U (the top-of-memory pointer and bottom-of-memory pointer, respectively) always have values at page boundaries.

As stated earlier, if the parent does not specify the size of the parameter area, the size defaults to zero. The minimum overall data area size is one page.

When the shell processes a command line, it passes a string in the parameter area. This string is a copy of the parameter part of the command line. To simplify string-oriented processing, the shell also inserts an end-of-line character at the end of the parameter string.

Register X points to the start byte of the parameter string. If the command line includes the optional memory size specification (#*n* or #*n*K), the shell passes that size as the requested memory size when executing the Fork.

- If any of the preceding operations is unsuccessful, the Fork is terminated and OS-9 returns an error to the caller.
- The child and parent processes execute at the same time unless the parent executes a Wait system call immediately after the Fork. In this case, the parent waits until the child dies before it resumes execution.
- Be careful when recursively calling a program that uses the Fork system call. Another child can be created with each new execution. This continues until the process table becomes full.
- Do not fork a process with a memory size of 0.

Get System Block Map

Gets a copy of the system block map

OS9 F$GBlkMp 103F 19

Entry Conditions:

X = *pointer to the 1024-byte buffer*

Exit Conditions:

D = *number of bytes per block* ($2000) (MMU block size dependent)
Y = *system memory block map size*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Get System Block Map call copies the system's memory block map into the user's buffer for inspection. The OS-9 MFREE command uses this call to find out how much free memory exists.
- The support module for this call is OS9p2.

Get Module Directory

Gets a copy of the system module directory

F$GModDr 103F 1A

Entry Conditions:

X = *pointer to the 2048-byte buffer*
Y = *end of copied module directory*
U = *start address of system module directory*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Get Module Directory call copies the system's module directory into the user's buffer for inspection. The OS-9 MDIR command uses this call to read the module directory.
- The support module for this call is OS9p2.

Get Process Descriptor

Gets a copy of the process's process descriptor

F$GPrDsc 103F 18

Entry Conditions:

A = *requested process ID*
X = *pointer to a 512-byte buffer*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Get Process Descriptor call copies a process descriptor into the calling process's buffer for inspection. The data in the process descriptor cannot be changed. The OS-9 PROCS command uses this call to get information about each existing process.
- The support module for this call is OS9p2.

Intercept

Sets a signal intercept trap

OS9 F$Icpt 103F 09

Entry Conditions:

X = *address of the intercept routine*
U = *starting address of the routine's memory area*

Exit Conditions:

Signals sent to the process cause the intercept routine to be called instead of the process being killed.

Additional Information:

- Intercept tells OS-9 to set a signal intercept trap. Then, whenever the process receives a signal, OS-9 executes the process's intercept routine.
- Store the address of the signal handler routine in Register X and the base address of the routine's storage area in Register U.
- Once the signal trap is set, OS-9 can execute the intercept routine at any time because a signal can occur at any time.
- Terminate the intercept routine with an RTI instruction.
- If a process has not used the Intercept system call to set a signal trap, the process terminates if it receives a signal.
- This is the order in which F$Icpt operates:

  1. When the process receives a signal, OS-9 sets Registers U and B as follows:

     U = *starting address of the intercept routine's memory area*
     B = *signal code* (process's termination status)

     Note: The value of Register DP cannot be the same as it was when the Intercept call was made.

  2. After setting the registers, OS-9 transfers execution to the intercept routine.

Get ID

Return's a caller's process ID and user ID

OS9 F$ID 103F 0C

Entry Conditions:

None

Exit Conditions:

A = *process ID*
Y = *user ID*

Additional Information:

- The *process ID* is a byte value in the range 1 to 255. OS-9 assigns each process a unique process ID.
- The *user ID* is an integer from 0 to 65535. It is defined in the system password file, and is used by the file security system and a few other functions. Several processes can have the same user ID.
- On a single user system (such as the Color Computer 3), the user ID is inherited from CC3go, which forks the initial shell.

Link

OS9 F$Link 103F 00

Links to a memory module that has the specified name, language, and type

Entry Conditions:

A = *type/language byte*
X = *address of the module name* (See the following example.)

Exit Conditions:

A = *type/language code*
B = *attributes / revision level* (if no error)
X = *address of the last byte of the module name* + 1 (See the following example.)
Y = *module entry point absolute address*
U = *module header absolute address*

Error Output:

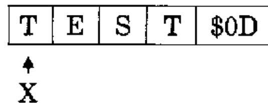
CC = C bit set if error encountered

Additional Information:

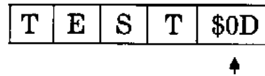
- The module's link count increases by one whenever Link references its name. Incrementing in this manner keeps track of how many processes are using the module.

- If the module requested is not shareable (not re-entrant), only one process can link to it at a time.

- Before the Link call:



- After the Link call:



X

- This is the order in which the Link call operates:

  1. OS-9 searches the module directory for a module that has the specified name, language, and type.
  2. If OS-9 finds the module, the address of the module's header is returned in Register U, and the absolute address of the module's execution entry point is returned in Register Y. (This, and other information is contained in the module header.)

- If OS-9 doesn't find the module—or if the type/language codes in the entry and exit conditions don't match—OS-9 returns one of the following errors:

  - Module not found
  - Module busy (not shareable and in use)
  - Incorrect or defective module header

Load

Loads a module or modules from a file

OS9 F$Load 103F 01

Entry Conditions:

A = *language/type code;* 0 = any language/type
X = *address of the pathlist (filename)* (See the following example.)

Exit Conditions:

A = *language/type code*
B = *attributes / revision level* (if no error)
X = *address of the last byte of the pathlist (filename)* + 1 (See the following example.)
Y = *primary module entry point address*
U = *address of the module header*

Error Output:

CC = carry set if error encountered

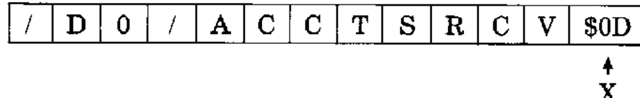
Additional Information:

- The Load call loads one or more modules from the file specified by a complete pathlist or from the working execution directory (if an incomplete pathlist is given).
- The file must have the execute access mode bit set. It also must contain one or more with proper module headers.
- OS-9 adds all modules loaded to the system module directory. It links the first module read. The exit conditions apply only to the first module loaded.
- Before the Load call:

/	D	0	/	A	C	C	T	S	R	C	V	$0D

↑
X

After the Load call:



- Possible errors:
  - Module directory full
  - Memory full
  - Errors that occur on the Open, Read, Close, and Link system calls

Memory

Changes process's data area size

OS9 F$Mem
103F 07

Entry Conditions:

D = *size of the new memory area* (in bytes);
0 = return current size and upper bound

Exit Conditions:

Y = *address of the new memory area upper bound*
D = *actual size of the new memory* (in bytes)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The memory call expands or contracts the process's data memory area to the specified size. Or, if you specify zero as the new size, the call returns the current size and upper boundaries of data memory.
- OS-9 rounds off the size to the next page boundary. In allocating additional memory, OS-9 continues upward from the previous highest address. In deallocating unneeded memory, it continues downward from that address.

Link to a module

Links to a module; does not map the module into the user's address space

OS9 F$NMLink
103F 21

Entry Conditions:

A = *type/language byte*
X = *address of the module name*

Exit Conditions:

A = *type/language code*
B = *module revision*
X = *address of the last byte of the module name* + 1; any trailing blanks are skipped
Y = *storage requirement for the module*

Error Output:

CC = carry set on error
B = *error code* if any

Additional Information:

- Although this call is similar to F$Link, it does not map the specified module into the user's address space but does return the memory requirement for the module. A calling process can use this memory requirement information to fork a program with a maximum amount of space. F$NMLink can therefore fork larger programs than can be forked by F$Link.

Load a module

OS9 F$NMLoad
103F 22

Loads one or more modules from a file but does not map the module into the user's address space

Entry Conditions:

A = *type/language byte*
X = *address of the pathlist*

Exit Conditions:

A = *type/language code*
B = *module revision*
X = *address of the last byte of the pathlist* + 1
Y = *storage requirement for the module*

Error Output:

CC = carry set on error
B = *error code* if any

Additional Information:

- If you do not provide a full pathlist for this call, it attempts to load from a file in the current execution directory.
- Although this call is similar to F$Load, it does not map the specified module into the user's address space but does return the memory requirement for the module. A calling process can use this memory requirement information to fork a program with a maximum amount of space. F$NMLoad can therefore fork larger programs than can be forked by F$Load.

Print Error

OS9 F$Perr 103F 0F

Writes an error message to a specified path

Entry Conditions:

B = *error code*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Print Error writes an error message to the standard error path for the specified process. By default, OS-9 shows:

```
ERROR #decimal number
```

- The error reporting routine is vectored. Using the Set SVC system call, you can replace it with a more elaborate reporting module.

Parse Name

OS9 F$PrsNam 103F 10

Scans an input string for a valid OS-9 name

Entry Conditions:

X = *address of the pathlist* (See the following example.)

Exit Conditions:

X = *address of the optional slash* + 1
Y = *address of the last character of the name* +1
A = *trailing byte* (delimiter character)
B = *length of the name*

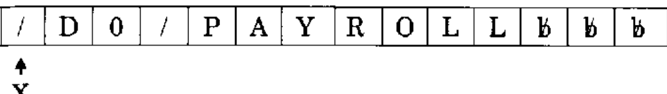
Error Output:

CC = carry set
B = *error code*
Y = *address of the first non-delimiter character in the string*

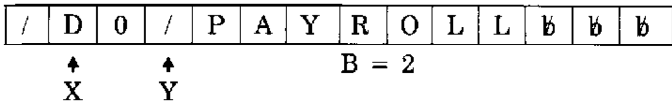
Additional Information:

- Parses, or scans, the input text string for a legal OS-9 name. It terminates the name with any character that is not a legal name character.
- Parse Name is useful for processing pathlist arguments passed to new processes.
- Because Parse Name processes only one name, you might need several calls to process a pathlist that has more than one name. As you can see from the following example, Parse Name finishes with Register Y in position for the next parse.
- If Register Y was at the end of a pathlist, Parse Name returns a bad name error. It then moves the pointer in Register Y past any space characters so that it can parse the next pathlist in a command line.

- Before the Parse Name call:



After the Parse Name call:



Search Bits

OS9 F$SchBit 103F 12

Searches a specified memory allocation bit map for a free memory block of a specified size

Entry Conditions:

D = *starting bit number*
X = *starting address of the map*
Y = *bit count (free bit block size)*
U = *ending address of the map*

Error Output:

CC = C bit set

Exit Conditions:

D = *starting bit number*
Y = *bit count*

Additional Information:

- The Search Bit call searches the specified allocation bit map for a free block (cleared bits) of the required length. The search starts at the *starting bit number*. If no block of the specified size exists, the call returns with the carry set, starting bit number, and size of the largest block.

Send

OS9 F$Send 103F 08

Sends a signal to a specified process

Entry Conditions:

A = *destination's process ID*
B = *signal code*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The *signal code* is a single byte value in the range 0 through 255.
- If the destination process is sleeping or waiting, OS-9 activates the process so that the process can process the signal.
- If a signal trap is set up, F$Send executes the signal processing routine (Intercept). If none was set up, the signal terminates the destination process, and the signal code becomes the exit status. (See the Wait system call.) An exception is the wakeup signal; that signal does not cause the signal intercept routine to be executed.
- Signal codes are defined as follows:

  0 = System terminate (cannot be intercepted)
  1 = Wake up the process
  2 = Keyboard terminate
  3 = Keyboard interrupt
  128-255 = User defined

- If you try to send a signal to a process that has a signal pending, OS-9 cancels the current Send call, and returns an error. Issue a Sleep call for a few ticks; then, try again.
- The Sleep call saves CPU time. See the Intercept, Wait, and Sleep system calls for more information.

Sleep

Temporarily turns off the calling process

OS9 F$Sleep 103F 0A

Entry Conditions:

X = One of the following:
sleep time (in ticks)
0 (sleep indefinitely)
1 (sleep for the remainder of the current time slice)

Exit Conditions:

X = *sleep time minus the number of ticks that the process was asleep*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If Register X contains 0, OS-9 turns the process off until it receives a signal. Putting a process to sleep is a good way to wait for a signal or interrupt without wasting CPU time.
- If Register X contains 1, OS-9 turns the process off for the remainder of the process's current time slice. It inserts the process into the active process queue immediately. The process resumes execution when it reaches the front of the queue.
- If Register X contains an integer in the range 2-255, OS-9 turns off the process for the specified number of ticks, *n*. It inserts the process into the active process queue after *n-1* ticks. The process resumes execution when it reaches the front of the queue. If the process receives a signal, it awakens before the time has elapsed.
- When you select processes among multiple windows, you might need to set sleep for two ticks.

Set Priority

Changes the priority of a process

OS9 F$SPrior 103F 0D

Entry Conditions:

A = *process ID*
B = *priority*
0 = lowest
255 = highest

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Set Priority changes the process's priority to the priority specified. A process can change another process's priority only if it has the same user ID.

Set SWI

OS9 F$SSWI 103F 0E

Sets the SWI2 and SWI3 vectors

Entry Conditions:

A = *SWI type code*
X = *address of the user software interrupt routine*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Sets the interrupt vectors for SWI, SWI2 and SWI3 instructions.
- Each process has its own local vectors. Each Set SWI call sets one type of vector according to the code number passed in Register A:

  1 = SWI
  2 = SWI2
  3 = SWI3

- When OS-9 creates a process, it initializes all three vectors with the address of the OS-9 service call processor.
- **Warning:** Microware-supplied software uses SWI2 to call OS-9. If you reset this vector, these programs cannot work. If you change all three vectors, you cannot call OS-9 at all.

Set Time

Sets the system time and date

OS9 F$STime 103F 16

Entry Conditions:

X = *relative address of the time packet*

Error Output:

CC = C bit set
B = *error code*

Additional Information:

- Set Time sets the current system date and time and starts the system real-time clock. The date and time are passed in a time packet as follows.

Relative Address	Value
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

Then, the call makes a link system call to find the clock. If the link is successful, OS-9 calls the clock initialization. The clock initialization:

1. Sets up hardware dependent functions
2. Sets up the F$Time system call via F$SSvc

Set User ID Number

F$SUser 103F 1C

Changes the current user ID without checking for errors or checking the ID number of the caller

Entry Conditions:

Y = *desired user ID number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The support module for this call is OS9p1.

Time

Gets the system date and time

OS9 F$Time 103F 15

Entry Conditions:

X = *address of the area in which to store the date and time packet*

Exit Conditions:

X = *date and time*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Time call returns the current system date and time in the form of a 6-byte packet (in binary). OS-9 copies the packet to the address passed in Register X.
- The packet looks like this:

Relative Address	Value
0	year
1	month
2	day
3	hours
4	minutes
5	seconds

- Time is a part of the clock module and it does not exist if no previous call to F$Time has been made.

Unlink

Unlinks (removes from memory) a module that is not in use and that has a link count of 0

OS9 F$UnLink 103F 02

Entry Conditions:

U = *address of the module header*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Unlink unlinks a module from the current process's address space, decreases its link count by one and, if the link count becomes zero, returns the memory where the module was located to the system for use by other processes.
- You cannot unlink system modules or device drivers that are in use.
- Unlink operates in the following order:
  1. Unlink tells OS-9 that the calling process no longer needs the module.
  2. OS-9 decreases the module's link count by one.
  3. When the resulting link count is zero, OS-9 destroys the module.

     If any other process is using the module, the module's link count cannot fall to zero. Therefore, OS-9 does not destroy the module.
- If you pass a bad address, Unlink cannot find a module in the module directory and does not return an error.

Unlink A Module By Name

F$UnLoad 103F 1D

Decrements a specified module's link count, and removes the module from memory if the resulting link count is zero

Entry Conditions:

A = *module type*
X = *pointer to module name*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- This system call differs from Unlink in that it uses a pointer to the module name, instead of the address of the module header.
- The support module for this call is OS9p2.

Wait

Temporarily turns off a calling process

OS9 F$Wait 103F 04

Entry Conditions: None

Exit Conditions:

A = *deceased child process's ID*
B = *deceased child process's exit status code* (if no error)

Error Output:

CC = carry set on error
B = *error code* if any

Additional Information:

- The Wait call turns off the calling process until a child process *dies*, either by executing an Exit system call, or by receiving a signal. The Wait call helps you save system time.
- OS-9 returns the child's process's ID and exit status to the parent. If the child died because of a signal, the exit status byte (Register B) contains the signal code.
- If the caller has several children, OS-9 activates the caller when the first one dies. Therefore, you need to use one Wait system call to detect the termination of each child.
- OS-9 immediately reactivates the caller if a child dies before the Wait call. If the caller has no children, Wait returns an error. (See the Exit system call for more information.)
- If the Wait call returns with the carry bit set, the Wait function was not successful. If the carry bit is cleared, Wait functioned normally and any error that occurred in the child process is returned in Register B.

I/O User System Calls

Attach

OS9 I$Attach 103F 80

Attaches a device to the system or verifies device attachment

Entry Conditions:

A = *access mode*
X = *address of the device name string*

Exit Conditions:

X = *updated past device name*
U = *address of the device table entry*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- Attach does not reserve the device. It only prepares the device for later use by any process.
- OS-9 installs most devices automatically on startup. Therefore, you need to use Attach only when installing a device dynamically or when verifying the existence of a device. You need not use the Attach system call to perform routine I/O.
- The access mode parameter specifies the read and/or write operations to be allowed. These are:

  0 = Use any special device capabilities
  1 = Read only
  2 = Write only
  3 = Update (read and write)

- Attach operates in this sequence:

  1. OS-9 searches the system module to see if memory contains a device descriptor that has the same name as the device.

  2a. OS-9's next operation depends on whether or not the device is already attached. If OS-9 finds the descriptor and if the device is not already attached, OS-9 links the device's file manager and device driver. It then places the address of the manager and the driver in a new device table entry. OS-9 then allocates any memory needed by the device driver, and calls the driver's initialization routine which initializes the hardware.

  b. If OS-9 finds the descriptor, and if the device is already attached, OS-9 verifies the attachment.

Change Directory

Changes the working directory of a process to a directory specified by a pathlist

OS9 I$Chgdir 103F 86

Entry Conditions:

A = *access mode*
X = *address of the pathlist*

Exit Conditions:

X = *updated past pathlist*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If the access mode is read, write, or update, OS-9 changes the current data directory. If the access mode is execute, OS-9 changes the current execution directory.
- The calling process must have read access to the directory specified (public read if the directory is not owned by the calling process).
- The access modes are:

  1 = Read
  2 = Write
  3 = Update (read and write)
  4 = Execute

Close Path

Terminates an I/O path

OS9 I$Close 103F 8F

Entry Conditions:

A = *path number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Close Path terminates the I/O path to the file or device specified by *path number*. Until you use another Open, Dup, or Create system call for that path, you can no longer perform I/O to the file or device.
- If you close a path to a single-user device, the device becomes available to other requesting processes. OS-9 deallocates internally managed buffers and descriptors.
- The Exit system call automatically closes all open paths. Therefore, you might not need to use the Close Path system call to close some paths.
- Do not close a standard I/O path unless you want to change the file or device to which it corresponds.
- Close Path performs an implied I$Detach call. If it causes the device link count to become 0, the device termination routine is executed. See I$Detach for additional information.

Create File

Creates and opens a disk file

OS9 I$Create 103F 83

Entry Conditions:

A = *access mode* (write or update)
B = *file attributes*
X = *address of the pathlist;* (See the following example.)

Exit Conditions:

A = *path number*
X = *address of the last byte of the pathlist* + 1; skips any trailing blanks (See the following example.)

Error Output:

CC = carry set on error
B = *error code* (if any)

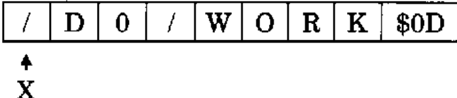
Additional Information:

- OS-9 parses the pathlist and enters the new filename in the specified directory. If you do not specify a directory, OS-9 enters the new filename in the the working directory.
- OS-9 gives the file the attributes passed in Register B, which has bits defined as follows:

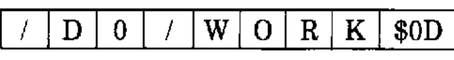
Bit	Definition
0	Read
1	Write
2	Execute
3	Public read
4	Public write
5	Public execute
6	Shareable file

- The access mode parameter passed in Register A must have the write bit set if any data is to be written. These access codes are defined as follows: 2 = write; 3 = update. The mode affects the file only until the file is closed.

- You can reopen the file in any access mode allowed by the file attributes. (See the Open system call.)
- Files opened for write can allow faster data transfer than those opened for update because update sometimes needs to pre-read sectors.
- If the execute bit (Bit 2) is set, the file is created in the working execution directory instead of the working data directory.
- Create File causes an implicit I$Attach call. If the device has not previously been attached, the device's initialization routine is called.
- Later I/O calls use the path number to identify the file, until the file is closed.
- OS-9 does not allocate data storage for a file at creation. Instead, it allocates the storage either automatically when you first issue a write or explicitly by the Setstat subroutine.
- If the filename already exists in the directory, an error occurs. If the call specifies a non-multiple file device (such as a printer or terminal), Create behaves the same as Open.
- You cannot use Create to make directories. (See the Make Directory system call for instructions on how to do make directories.)
- Before the Create File call:



After the Create File call:



Delete File

Deletes a specified disk file

OS9 I$Delete 103F 87

Entry Conditions:

X = *address of the pathlist* (See the following example.)

Exit Conditions:

X = *address of the last byte of the pathlist* – 1; any trailing blanks are skipped (See the following example.)

Error Output:

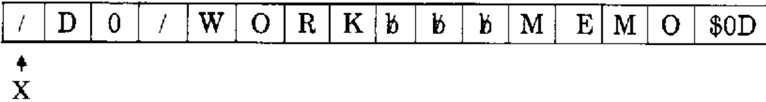
B = *error code* (if any)
CC = carry set on error

Additional Information:

- The Delete File call deletes the disk file specified by the pathlist. The file must have write permission attributes (public write, if the calling process is not the owner). An attempt to delete a device results in an error. The caller must have non-shareable write access to the file or an error results.

Example:

Before the Delete File call:



/	D	0	/	W	O	R	K	ƀ	ƀ	ƀ	M	E	M	O	$0D
↑ X															

After the Delete File call:

/	D	0	/	W	O	R	K	ƀ	ƀ	ƀ	M	E	M	O	$0D
															↑ X

Delete A File

OS9 I$DeletX 103F 90

Deletes a file from the current data or current execution directory

Entry Conditions:

A = *access mode*
X = *address of the pathlist*

Exit Conditions:

X = *address of the last byte of the pathlist + 1*; any trailing blanks are skipped

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Delete A File call removes the disk file specified by the selected pathlist. This function is similar to I$Delete except that it accepts an access mode byte. If the access mode is execute, the call selects the current execution directory. Otherwise, it selects the current data directory.
- If a complete pathlist is provided (the pathlist begins with a slash (/), the access mode the call ignores the access mode.
- Only use this call to delete a file. If you attempt to use I$DeletX to delete a device, the system returns an error.

Detach Device

OS9 I$Detach 103F 81

Removes a device from the system device table

Entry Conditions:

U = *address of the device table entry*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Detach Device call removes a device from both the system and the system device table, assuming the device is not being used by another process. You must use this call to detach devices attached using the Attach system call. Attach and Detach are both used mainly by the IO manager. SCF also uses Attach and Detach to set up its second device (echo device).

- This is the sequence of the operation of Detach Device:

  1. Detach Device calls the device driver's termination routine. Then, OS-9 deallocates any memory assigned to the driver.

  2. OS-9 unlinks the associated device driver and file manager modules.

  3. OS-9 then removes the driver, as long as no other module is using that driver.

Duplicate Path

Returns a synonymous path number

OS9 I$Dup 103F 82

Entry Conditions:

A = *old path number* (number of path to duplicate)

Exit Conditions:

A = *new path number* (if no error)

Error Output:

B = *error code* (if error encountered)
CC = carry set on error

Additional Information:

- The Duplicate Path returns another, synonymous path number for the file or device specified by the *old path number*.
- The shell uses the Duplicate Path call when it redirects I/O.
- System calls can use either path number (old or new) to operate on the same file or device.
- Make sure that no more than one process is performing I/O on any one path at the same time. Concurrent I/O on the same path can cause unpredictable results with RBF files.
- The I$Dup call always uses the lowest available path number. This lets you manipulate standard I/O paths to contain any desired paths.

Get Status

Returns the status of a file or device

OS9 I$GetStt 103F 8D

Entry Conditions:

A = *path number*
B = *function code*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Status is a *wildcard* call. Use it to handle device parameters that:
  - Are not the same for all devices
  - Are highly hardware-dependent
  - Must be user-changeable
- The exact operation of the Get Status system call depends on the device driver and file manager associated with the path. A typical use is to determine a terminal's parameters for such functions as backspace character and echo on/off. The Get Status call is commonly used with the Set Status call.
- The Get Status function codes that are currently defined are listed in the "Get Status System Calls" section.

Make Directory

Creates and initializes a directory

OS9 I$MakDir 103F 85

Entry Conditions:

B = *directory attributes*
X = *address of the pathlist*

Exit Conditions:

X = *address of the last byte of the pathlist* + 1; Make Directory skips trailing blanks.

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- The Make Directory call creates and initializes a directory as specified by the pathlist. The directory contains only two entries, one for itself (.) and one for its parent directory (..)
- OS-9 makes the calling process the owner of the directory.
- Because the Make Directory call does not open the directory, it does not return a path number.
- The new directory automatically has its directory bit set in the access permission attributes. The remaining attributes are specified by the byte passed in Register B. The bits are defined as follows:

Bit	Definition
0	Read
1	Write
2	Execute
3	Public read
4	Public write
5	Public execute
6	Single-user
7	Don't care

- Before the Make Directory call:

/	D	0	/	N	E	W	D	I	R	$0D
↑ X										

After the Make Directory call:

/	D	0	/	N	E	W	D	I	R	$0D
										↑ X

Open Path

OS9 I$Open 103F 84

Opens a path to an existing file or device as specified by the pathlist

Entry Conditions:

A = *access mode* (D S PE PW PR E W R)
X = *address of the pathlist* (See the following example.)

Exit Conditions:

A = *path number*
X = *address of the last byte of the pathlist* + 1

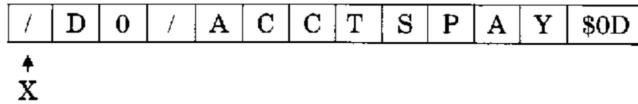
Error Output:

B = *error code* (if any)
CC = carry set on error

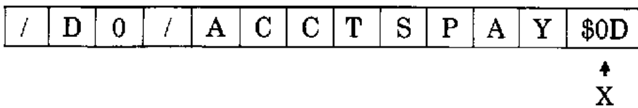
Additional Information:

- OS-9 searches for the file in one of the following:
  - The directory specified by the pathlist if the pathlist begins with a slash.
  - The working data directory, if the pathlist does not begin with a slash.
  - The working execution directory, if the pathlist does not begin with a slash and if the execution bit is set in the access mode.
- OS-9 returns a path number for later system calls to use to identify the file.
- The access mode parameter lets you specify which read and/or write operations are to be permitted. When set, each access mode bit enables one of the following: Write, Read, Read and Write, Update, Directory I/O.
- The access mode must conform to the access permission attributes associated with the file or device. (See the Create system call.) Only the owner can access a file unless the appropriate public permission bits are set.

- The update mode might be slightly slower than the others because it might require pre-reading of sectors for random access of bytes within sectors.
- Several processes (users) can open files at the same time. Each device has an attribute that specifies whether or not it is shareable.
- Before the Open Path call:



After the Open Path call:



- If the single-user bit is set, the file is opened for single-user access regardless of the settings of the file's permission bits.
- You must open directory files for read or write if the directory bit (Bit 7) is set in the access mode.
- Open Path always uses the lowest path number available for the process.

Read

Reads *n* bytes from a specified path

OS9 I$Read 103F 89

Entry Conditions:

A = *path number*
Y = *number of bytes to read*
X = *address in which to store the data*

Exit Conditions:

Y = *number of bytes read*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- The Read call reads the specified number of bytes from the specified path. It returns the data exactly as read from the file/device, without additional processing or editing. The path must be opened in the read or update mode.
- If there is not enough data in the specified file to satisfy the read request, the call reads fewer bytes than requested but an end-of-file error is **not** returned. After all data in a file is read, the next I$Read call returns an end-of-file error.
- If the specified file is open for update, the record read is locked out on RBF-type devices.
- The keyboard terminate, keyboard interrupt, and end-of-file characters are filtered out of the Entry Conditions data on SCF-type devices unless the corresponding entries in the path descriptor have been set to zero. You might want to modify the device descriptor so that these values are initialized to zero when the path is opened.

- The call reads the number of bytes requested unless Read encounters any of the following:
  - An end-of-file character
  - An end-of-record character (SCF only)
  - An error

Read Line With Editing

Reads a text line with editing

OS9 I$ReadLn 103F 8B

Entry Conditions:

A = *path number*
X = *address at which to store data*
Y = *maximum number of bytes to read*

Exit Conditions:

Y = *number of bytes read*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- Read Line is similar to Read. The difference is that Read Line reads the input file or device until it encounters a carriage return character or until it reaches the maximum byte count specified, whichever comes first. The Read Line also automatically activates line editing on character oriented devices, such as terminals and printers. The line editing refers to auto line feed, null padding at the end of the line, backspacing, line deleting, and so on.
- SCF requires that the last byte entered be an end-of-record character (usually a carriage return). If more data is entered than the maximum specified, Read Line does not accept it and a PD.OVF character (usually a bell) is echoed.
- After one Read Line call reads all data in a file, the next Read Line call generates an end-of-file error.
- (For more information about line editing, see "SCF Line Editing Functions" in Chapter 6.)

Seek

OS9 I$Seek 103F 88

Repositions a file pointer

Entry Conditions:

A = *path number*
X = *MS 16 bits of the desired file position*
U = *LS 16 bits of the desired file position*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Seek Call repositions the path's logical file pointer, the 32-bit address of the next byte in the file to be read from or written to.
- You can perform a seek to any value, regardless of the file's size. Later writes automatically expand the file to the required size (if possible). Later reads, however, return an end-of-file condition. Note that a seek to Address 0 is the same as a rewind operation.
- OS-9 usually ignores seeks to non-random access devices, and returns without error.
- On RBF devices, seeking to a new disk sector causes the internal disk buffer to be rewritten to disk if it has been modified. Seek does not change the state of record locking.

Set Status

Sets the status of a file or device

OS9 I$SetStt 103F 8E

Entry Conditions:

A = *path number*
B = *function code*
Other registers depend on the function code.

Error Output:

B = *error code* (if any)
CC = carry set on error
Other registers depend on the function code.

Additional Information:

- Set Status is a wildcard call. Use it to handle device parameters that:
  - Are not the same for all devices
  - Are highly hardware-dependent
  - Must be user-changeable
- The exact operation of the Set Status system call depends on the device driver and file manager associated with the path. A typical use is to set a terminal's parameters for such functions as backspace character and echo on/off. The Set Status call is commonly used with the Get Status call.
- The Set Status function codes that are currently defined are listed in the "Set Status System Calls" section.

Write

Writes to a file or device

OS9 I$Write 103F 8A

Entry Conditions:

A = *path number*
X = *starting address of data to write*
Y = *number of bytes to write*

Exit Conditions:

Y = *number of bytes written*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- The Write system call writes to the file or device associated with the path number specified.
- Before using Write, be sure the path is opened or created in the Write or Update access mode. OS-9 writes data to the file or device without processing or editing the data. OS-9 automatically expands the file if you write data past the present end-of-file.

Write Line

OS9 I$WritLn 103F 8C

Writes to a file or device until it encounters a carriage return

Entry Conditions:

A = *path number*
X = *address of the data to write*
Y = *maximum number of bytes to write*

Exit Conditions:

Y = *number of bytes written*

Error Output:

B = *error code* (if any)
CC = *carry set on error*

Additional Information:

- Writes to the file or device that is associated with the path number specified.
- Write Line is similar to Write. The difference is that Write Line writes data until it encounters a carriage return character. It also activates line editing for character-oriented devices, such as terminals and printers. The line editing refers to auto line feed, null padding at the end of the line, backspacing, line deleting, and so on.
- Before using Write Line, be sure the path is opened or created in the write or update access mode.
- (For more information about line editing, see "SCF Line Editing Functions" in Chapter 6.)

Privileged System Mode Calls

Set an alarm

OS9 F$Alarm 103F 1E

Sets an alarm to ring the bell at a specified time

Entry Conditions:

X = *relative address of time packet*

Error Output:

CC = carry set on error
B = *appropriate error code*

Additional Information:

- When the system reaches the specified alarm time, it rings the bell for 15 seconds.
- The time packet is identical to the packet used in the F$STime call. See F$STime for additional information on the format of the packet.
- All alarms begin at the start of a minute and any seconds in the packet are ignored.
- The system is limited to one alarm at a time.

Allocate 64

OS9 F$All64 103F 30

Dynamically allocates 64-byte blocks of memory

Entry Conditions:

X = *base address of the page table*; 0 = the page table has not been allocated

Exit Conditions:

A = *block number*
X = *base address of the page table*
Y = *address of the block*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Allocate 64 system call allocates the 64-byte blocks of memory by splitting pages (256-byte sections) into four sections.
- OS-9 uses the first 64 bytes of the base page as a page table. This table contains the page number (most significant byte of the address) of all pages in the memory structure. If Register X passes a value of zero, the call allocates a new base page and the first 64-byte memory block.
- Whenever a new page is needed, a Request System Memory system call (F$SRqMem) executes automatically.
- The first byte of each block contains the block number. Routines that use the Allocate 64 call should not alter this byte.

- The following diagram shows how seven blocks might be allocated:

Base Page →

Any Memory Page	Any Memory Page
Page Table (64 bytes)	X Block 4 (64 bytes)
X Block 1 (64 bytes)	X Block 5 (64 bytes)
X Block 2 (64 bytes)	X Block 6 (64 bytes)
X Block 3 (64 bytes)	X Block 7 (64 bytes)

Allocate High RAM

Allocate system memory from high physical memory

OS9 F$AlHRam 103F 53

Entry Conditions:

B = *number of blocks*

Error Output:

CC = carry set on error
B = *appropriate error code*

Additional Information:

- This call searches for the desired number of contiguous free RAM blocks, starting its search at the top of memory. F$AllHRam is similar to F$AllRAM except F$AllRAM begins its search at the bottom of memory.
- Screen allocation routines use this call to provide a better chance of finding the necessary memory for a screen.

Allocate Image

OS9 F$AllImg 103F 3A

Allocates RAM blocks for process DAT image

Entry Conditions:

A = *starting block number*
B = *number of blocks*
X = *process descriptor pointer*

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Use the Allocate Image system call to allocate a data area for a process. The blocks that Allocate Image defines might not be contiguous.
- The support module for this system call is OS9p1.

Allocate Process Descriptor

Allocates and initializes a 512-byte process descriptor

OS9 F$AllPrc 103F 4B

Entry Conditions: None

Exit Conditions:

U = *process descriptor pointer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The process descriptor table houses the address of the descriptor. Initialization of the process descriptor consists of clearing the first 256 bytes of the descriptor, setting up the state as a system state, and marking as unallocated as much of the DAT image as the system allows—typically, 60-64 kilobytes.
- The support module for this system call is OS9p2. The call also branches to the F$SRqMem call.

Allocate RAM

OS9 F$AllRAM 103F 39

Searches the memory block map for the desired number of contiguous free RAM blocks

Entry Conditions:

B = *number of blocks*

Exit Conditions:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The support module for this system call is OS9p1.

Allocate Process Task Number

OS9 F$AllTsk 103F 3F

Determines whether OS-9 has assigned a task number to the specified process

Entry Conditions:

X = *process descriptor pointer*

Error Output:

CC = C bit set
B = *appropriate error code*

Additional Information:

- If the process does not have a task number, OS-9 allocates a task number and copies the DAT image into the DAT hardware.
- The support module for this call is OS9p1. Allocate Process Task number also branches to F$ResTsk and F$SetTsk.

Insert Process

OS9 F$AProc 103F 2C

Inserts a process into the queue for execution

Entry Conditions:

X = *address of the process descriptor*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Insert Process system call inserts a process into the active process queue so that OS-9 can schedule the process for execution.
- OS-9 sorts all processes in the queue by process age (the count of how many process switches have occurred since the process's last time slice). When a process is moved to the active process queue, OS-9 sets its age according to its priority—the higher the priority, the higher the age.

  An exception is a newly active process that was deactivated while in the system state. OS-9 gives such a process higher priority because the process usually is executing critical routines that affect shared system resources.

Bootstrap System

OS9 F$Boot 103F 35

Links either the module named Boot or the module specified in the INIT module

Entry Conditions: None

Error Output:

CC = *C bit set on error*
B = *appropriate error code*

Additional Information:

- When it calls the linked module, Boot expects to receive a pointer giving it the location and size of an area in which to search for the new module.
- The support module for this call is OS9p1. Bootstrap System also branches to the F$Link and F$VModul system calls.

Bootstrap Memory Request

OS9 F$BtMem 103F 36

Allocates the requested memory (rounded to the nearest block) as contiguous memory in the system's address space

Entry Conditions:

D = *byte count requested*

Exit Conditions:

D = *byte count granted*
U = *pointer to memory allocated*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- This call is identical to F$SRqMem.
- The Bootstrap Memory Request support module is OS9p1.

Clear Specified Block

Marks blocks in the process DAT image as unallocated

OS9 F$ClrBlk 103F 50

Entry Conditions:

B = *number of blocks*
U = *address of first block*

Exit Conditions: None

Additional Information:

- After Clear Specified Block deallocates blocks, the blocks are free for the process to use for other data or program areas. If the block address passed to Clear Specified Block is invalid or if the call attempts to clear the stack area, it returns E$IBA.
- The support module for the call is OS9p2.

DAT to Logical Address

OS9 F$DATLog 103F 44

Converts a DAT image clock number and block offset to its equivalent logical address

Entry Conditions:

B = *DAT image offset*
X = *block offset*

Exit Conditions:

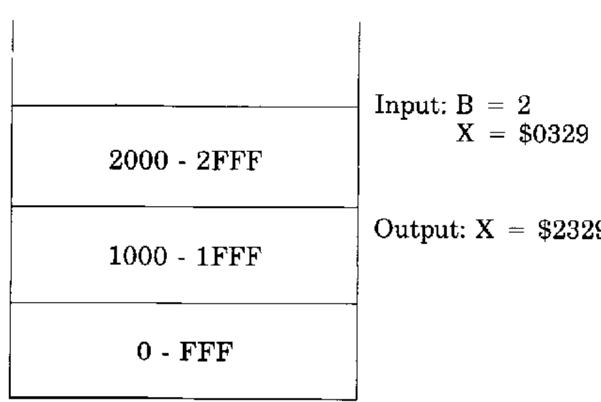
X = *logical address*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- Following is a sample conversion:



Input: B = 2
X = $0329

Output: X = $2329

- The support module for this call is OS9p1.

Deallocate Image RAM Blocks

Deallocates image RAM blocks

OS9 F$DelImg 103F 3B

Entry Conditions:

A = *number of starting block*
B = *block count*
X = *process descriptor pointer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- This system call deallocates memory from a process's address space. It frees the RAM for system use and frees the DAT image for the process. Its main use is to let the system clean up after a process death.
- The support module for this call is OS9p2.

Deallocate Process Descriptor

Returns a process descriptor's memory to a free memory pool

OS9 F$DelPrc 103F 4C

Entry Conditions:

A = *process ID*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- Use this call to clear the system scratch memory and stack area associated with the process.
- The support module for this call is OS9p2.

Deallocate RAM blocks

Clears a block's RAM In Use flag in the system memory block map

OS9 F$DelRAM 103F 51

Entry Conditions:

B = *number of blocks*
X = *starting block number*

Exit Conditions: None

Additional Information:

- The Deallocate RAM Blocks call assumes the blocks being deallocated are not associated with any DAT image.
- The support module for this call is OS9p2.

Deallocate Task Number

OS9 F$DelTsk 103F 40

Releases the task number that the process specified by the passed descriptor pointer

Entry Conditions:

X = *process descriptor pointer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The support module for this call is OS9p1.

Link Using Module Directory Entry

Performs a link using a pointer to a module directory entry

OS9 F$ELink 103F 4D

Entry Conditions:

B = *module type*
X = *pointer to module directory entry*

Exit Conditions:

U = *module header address*
Y = *module entry point*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- This call differs from Link in that you supply a pointer to the module directory entry rather than a pointer to a module name.
- The support module for this call is OS9p1.

Find Module Directory Entry

Returns a pointer to the module directory entry

OS9 F$FModul 103F 4E

Entry Conditions:

A = *module type*
X = *pointer to the name string*
Y = *DAT image pointer* (for name)

Exit Conditions:

A = *module type*
B = *module revision number*
X = *updated name string;* (if Register A contains 0 on entry)
U = *module directory entry pointer*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The Find Module Directory Entry call returns a pointer to the module directory entry for the first module that has a name and type matching the specified name and type. If you pass a module type of zero, the system call finds any module.
- The support module for this call is OS9p1.

Find 64

OS9 F$Find64 103F 2F

Returns the address of a 64-byte memory block

Entry Conditions:

A = *block number*
X = *address of the block*

Exit Conditions:

Y = *address of the block*
CC = carry set if block not allowed or not in use

Additional Information:

- OS-9 uses Find 64 to find path descriptors when given their block number. The block number can be any positive integer.

Get Free High Block

OS9 F$FreeHB 103F 3E

Searches the DAT image for the highest set of contiguous free blocks of the specified size

Entry Conditions:

B = *block count*
Y = *DAT image pointer*

Exit Conditions:

A = *starting block number*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The Get Free High Block call returns the block number of the beginning memory address of the free blocks.
- The support module for this system call is OS9p1.

Get Free Low Block

OS9 F$FreeLB 103F 3D

Searches the DAT image for the lowest set of contiguous free blocks of the specified size

Entry Conditions:

B = *block count*
Y = *DAT image pointer*

Exit Conditions:

A = *starting block number*

Error Output:

CC = C bit set on error
B = *appropriate error code*

Additional Information:

- The Get Free Low Block call returns the block number of the beginning memory address of the free blocks.
- The support module for this system call is OS9p1.

Compact Module Directory

Compacts the entries in the module directory

OS9 F$GCMDir 103F 52

Entry Conditions: None

Exit Conditions: None

Additional Information:

- This function is only for internal OS-9 system use. You should never call it from a program.

Get Process Pointer

Gets a pointer to a process

F$GProcP 103F 37

Entry Conditions:

A = *process ID*

Exit Conditions:

B = *pointer to process descriptor* (if no error)

Error Output:

CC = carry set on error
B = *error code* (If an error occurs (E$BPrcID))

Additional Information:

- The Get Process Pointer call translates a process ID number to the address of its process descriptor in the system address space. Process descriptors exist only in the system task address space. Because of this, the address returned only refers to system address space.
- The support module for this call is OS9p2.

I/O Delete

Deletes an I/O module that is not being used

OS9 F$IODel 103F 33

Entry Conditions:

X = *address of an I/O module*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The I/O Delete deletes the specified I/O module from the system, if the module is not in use. This system call is used mainly by the I/O MANAGER, and can be of limited or no use for other applications.
- This is the order in which I/O Delete operates:

  1. Register X passes the address of a device descriptor module, device driver module, or file manager module.
  2. OS-9 searches the device table for the address.
  3. If OS-9 finds the address, it checks the module's use count. If the count is zero, the module is not being used; OS-9 deletes it. If the count is not zero, the module is being used; OS-9 returns an error.

- I/O Delete returns information to the Unlink system call after determining whether a device is busy.

I/O Queue

OS9 F$IOQu 103F 2B

Inserts the calling process into another process's I/O queue, and puts the calling process to sleep

Entry Conditions:

A = *process number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The I/O Queue call links the calling process into the I/O queue of the specified process and performs an untimed sleep. The IO Manager and the file managers are primary and extensive users of I/O Queue.
- Routines associated with the specified process send a wake-up signal to the calling process.

Set IRQ

OS9 F$IRQ 103F 2A

Adds a device to or removes it from the polling table

Entry Conditions:

D = *address of the device status register*
X = 0 (to remove a device) or *the address of a packet* (to add a device)
- the address at X is the flip byte
- the address at X+1 is the mask byte
- the address at X+2 is the priority byte

Y = *address of the device IRQ service routine*
U = *address of the service routine's memory area*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Set IRQ is used mainly by device driver routines. (See "Interrupt Processing" in Chapter 2 for a complete discussion of the interrupt polling system.)
- Packet Definitions:

  The Flip Byte. Determines whether the bits in the device status register indicate active when set or active when cleared. If a bit in the flip byte is set, it indicates that the task is active whenever the corresponding bit in the status register is clear (and vice versa).

  The Mask Byte. Selects one or more bits within the device status register that are interrupt request flag(s). One or more set bits identify which task or device is active.

  The Priority Byte. Contains the device priority number (0 = lowest priority, 255 = highest priority).

Load A From Task B

Loads A from 0,X in task B

F$LDABX 103F 49

Entry Conditions:

B = *task number*
X = *pointer to data*

Exit Conditions:

A = *byte at 0,X in task address space*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The value in Register X is an offset value from the beginning address of the Task module. The Load A From Task B call returns one byte from this logical address. Use this system call to get one byte from the current process's memory in a system state routine.

Get One Byte

Loads A from [X, [Y]]

F$LDAXY 103F 46

Entry Conditions:

X = *block offset*
Y = *DAT image pointer*

Exit Conditions:

A = *contents of byte at DAT image (Y) offset by X*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Get One Byte system call gets the contents of one byte in the specified memory block. The block is specified by the DAT image in (Y), offset by (X). The call assumes that the DAT image pointer is to the actual block desired, and that X is only an offset within the DAT block. The value in Register X must be less than the size of the DAT block. OS-9 does not check to see if X is out of range.

Get Two Bytes

Loads D from [D+X],[Y]

F$LDDDXY 103F 48

Entry Conditions:

D = *Offset to the offset within the DAT image*
X = *Offset within the DAT image*
Y = *DAT image pointer*

Exit Conditions:

D = *contents of two bytes at [D+X,Y]*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Get Two Bytes loads two bytes from the address space described by the DAT image pointer. If the DAT image pointer is to the entire DAT, then make D+X equal to the process address for data. If the DAT image is not the entire image (64K), then you must adjust D+X relative to the beginning of the DAT image. Using D+X lets you keep a local pointer within a block, and also lets you point to an offset within the DAT image that points to a specified block number.

Map Specific Block

F$MapBlk 103F 4F

Maps the specified block(s) into unallocated blocks of process space

Entry Conditions:

X = *starting block number*
B = *number of blocks*

Exit Conditions:

U = *address of first block*

Error Output:

B = *error code* (if any)
CC = carry set on error

Additional Information:

- The system maps blocks from the top down. It maps new blocks into the highest available addresses in the address space. See Clear Specified Block for information on unmapping.

Move Data

F$Move 103F 38

Moves data bytes from one address space to another

Entry Conditions:

A = *source task number*
B = *destination task number*
X = *source pointer*
Y = *byte count*
U = *destination pointer*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- You can use the Move Data system call to move data bytes from one address space to another, usually from system to user, or vice versa.
- The support module for this call is OS9p1.

Next Process

OS9 F$NProc 103F 2D

Executes the next process in the active process queue

Entry Conditions: None

Exit Conditions:

Control does not return to caller.

Additional Information:

- The Next Process system call takes the next process out of the active process queue and initiates its execution. If the queue contains no process, OS-9 waits for an interrupt, and then checks the queue again.
- The process calling Next Process must already be in one of the three process queues. If it is not, it becomes unknown to the system even though the process descriptor still exists and can be displayed by a PROCS command.

Release A Task

F$RelTsk 103F 43

Releases a specified DAT task number from use by a process, making the task's DAT hardware available for use by another task

Entry Conditions:

B = *task number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The support module for this call OS9p1.

Reserve Task Number

Reserves a DAT task number

F$ResTsk 103F 42

Entry Conditions: none

Exit Conditions:

B = *task number* (if no error)

Error Output:

CC = carry set on error
B = *error code* if an error occurs

Additional Information:

- The Reserve Task Number call finds a free DAT task number, reserves it, and returns the task number to the caller. The caller often then assigns the task number to a process.
- The support module for this call is OS9p1.

Return 64

Deallocates a 64-byte block of memory

OS9 F$Ret64 103F 31

Entry Conditions:

A = *block number*
X = *address of the base page*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- See the Allocate 64 system call for more information.

Set Process DAT Image

Copies all or part of the DAT image into a process descriptor

F$SetImg 103F 3C

Entry Condition:

A = *starting image block number*
B = *block count*
X = *process descriptor pointer*
U = *new image pointer*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- While copying part or all of the DAT image, this system call also sets an image change flag in the process descriptor. This flag guarantees that as a process returns from the system call. The call updates the hardware to match the new process DAT image.
- The support module for this call is OS9p1.

Set Process Task DAT Registers

Writes to the hardware DAT registers

F$SetTsk 103F 41

Entry Conditions:

X = *pointer to process descriptor*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- This system call sets the process task hardware DAT registers, and clears the image change flag in the process descriptor. It also writes to DAT RAM the process's segment address information.
- The support module for this call is OS9p1.

System Link

F$SLink 103F 34

Adds a module from outside the current address space into the current address space

Entry Conditions:

A = *module type*
X = *module name string pointer*
Y = *name string DAT image pointer*

Exit Conditions:

A = *module type*
B = *module revision (if no error)*
X = *updated name string pointer*
Y = *module entry point*
U = *module pointer*

Error Output:

CC = carry set on error
B = *error code* (If an error occurs)

Additional Information:

- The I/O System uses the System Link call to link into the current process's address space those modules specified by a device name in a user call. User calls such as Create File and Open Path use this System Link.
- The support module for this call is OS9p1.

Request System Memory

OS9 F$SRqMem 103F 28

Allocates a block of memory of the specified size from the top of available RAM

Entry Conditions:

D = *byte count*

Exit Conditions:

U = *starting address of the memory area*
D = *new memory size*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The Request System Memory call rounds the size request to the next page boundary.
- This call allocates memory only for system address space.

Return System Memory

Deallocates a block of contiguous pages

OS9 F$SRtMem 103F 29

Entry Conditions:

U = *starting address of memory to return;* must point to an even page boundary
D = *number of bytes to return*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Register U must point to an even page boundary.
- This call deallocates memory for system address space only.

Set SVC

Adds or replaces a system call

OS9 F$SSvc 103F 32

Entry Conditions:

Y = *address of the system call initialization table*

Error Output:

CC = C bit set
B = *error code*

Additional Information:

- Set SVC adds or replaces a system call, which you have written, to OS-9's user and system mode system call tables.
- Register Y passes the address of a table, which contains the function codes and offsets, to the corresponding system call handler routines. This table has the following format:

Relative Address	Use	
$00	Function Code	← First entry
$01	Offset From Byte 3	
$02	To Function Handler	
$03	Function Code	← Second entry
$04	Offset From Byte 6	
$05	To Function Handler	
	More Entries	← More entries
	$80	← End-of-table mark

- If the most significant bit of the function code is set, OS-9 updates the system table.

  If the most significant bit of the function code is not set, OS-9 updates the system and user tables.

- The function request codes are in the range $29-$34. IO calls are in the range $80-$90

- To use a privileged system call, you must be executing a program that resides in the system map and that executes in the system state.

- The system call handler routine must process the system call and return from the subroutine with an RTS instruction.

- The handler routine might alter all CPU registers (except Register SP).

- Register U passes the address of the register stack to the system call handler as shown in the following diagram:

	Register	Relative Address	Name
U →	CC	$00	R$CC
		$01	R$D
	A	$01	R$A
	B	$02	R$B
	DP	$03	R$DP
	X	$04	R$X
	Y	$06	R$Y
	U	$08	R$U
	PC	$0A	R$PC

Codes $70-$7F are reserved for user definition.

Store A Byte In A Task

Stores A at 0,X in Task B

F$STABX 103F 4A

Entry Conditions:

A = *byte to store*
B = *destination task number*
X = *logical destination address*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- This system call is similar to the assembly language instruction "STA 0,X". The difference is that in the system call, X refers to an address in the given task's address space, instead of the current address space.
- The support module for this system call is OS9p1.

Install virtual interrupt

Installs a virtual interrupt handler routine

OS9 F$VIRQ 103F 27

Entry Conditions:

D = *initial count value*
X = 0 to delete entry
    1 to install entry
Y = *address of 5-byte packet*

Error Output:

CC = carry set on error
B = *appropriate error code*

Additional Information:

- Install VIRQ for use with devices in the Multi-Pak Expansion Interface. This call is explained in detail in Chapter 2.

Validate Module

OS9 F$VModul 103F 2E

Checks the module header parity and CRC bytes of a module

Entry Conditions:

D = *DAT image pointer*
X = *new module block offset*

Exit Conditions:

U = *address of the module directory entry*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- If the values of the specified module are valid, OS-9 searches the module directory for a module with the same name. If one exists, OS-9 keeps in memory the module that has the higher revision level. If both modules have the save revision level, OS-9 retains the module in memory.

Get Status System Calls

You use the Get Status system calls with the RBF manager subroutine GETSTA. The OS-9 Level Two system reserves function Codes 7-127 for use by Microware. You can define Codes 128-255 and their parameter-passing conventions for your own use. (See the sections on device drivers in Chapters 4, 5, and 6.)

The Get Status routine passes the register stack and the specified function code to the device driver.

Following are the Get Status functions and their codes.

SS.OPT

(Function code $00). Reads the option section of the path descriptor, and copies it into the 32-byte area pointed to by Register X

Entry Conditions:

A = *path number*
B = $00
X = *address to receive status packet*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Use SS.OPT to determine the current settings for editing functions, such as echo and auto line feed.

SS.RDY

(Function code $01). Tests for data available on SCF-supported devices

Entry Conditions:

A = *path number*
B = $01

Exit Conditions:

If the device is ready:
CC = carry clear
B = $00

If the device is not ready:
CC = carry set
B = $F6 (E$SRNDY)

Error Output:

CC = carry set
B = *error code*

SS.SIZ

(Function code $02). Gets the current file size on a RBF-supported devices only

Entry Conditions:

A = *path number*
B = $02

Exit Conditions:

X = *most significant 16 bits of the current file size*
U = *least significant 16 bits of the current file size*

Error Output:

CC = carry set on error
B = *error code* (if any)

SS.POS

(Function code $05). Gets the current file position (RBF-supported devices only)

Entry Conditions:

A = *path number*
B = $05

Exit Conditions:

X = *most significant 16 bits of the current file position*
U = *least significant 16 bits of the current file position*

Error Output:

CC = carry set on error
B = *error code* (if any)

SS.EOF

(Function code $06). Tests for the end of the file (EOF)

Entry Conditions:

A = *path number*
B = $06

Exit Conditions:

If there is no EOF:
CC = carry clear
B = $00

If there is an EOF:
CC = carry set
B = $D3 (E$EOF)

Error Output:

CC = carry set
B = *error code*

SS.DevNm

(Function Code $0E). Returns a device name

Entry Conditions:

A = *path number*
B = $0E
X = *address of 32-byte buffer for name*

Exit Conditions:

X = *address of buffer, name moved to buffer*

SS.DSTAT

(Function code $12). Returns the display status

Entry Conditions:

A = *path number*
B = $12

Exit Conditions:

A = *color code of the pixel at the cursor address*
X = *address of the graphics display memory*
Y = *graphics cursor address;* X = MSB, Y = LSB

Additional Information:

- This function is supported only with the VDGINT module and deals with VDG-compatible graphics screens. See SS.AAGBf for information regarding Level Two operation.

SS.JOY

(Function code $13). Returns the joystick values

Entry Conditions:

A = *path number*
B = $13
X = *joystick number*
  0 = (right joystick)
  1 = (left joystick)

Exit Conditions:

A = *fire button down*
  0 = none
  1 = Button 1
  2 = Button 2
  3 = Button 1 and Button 2
X = *selected joystick x value* (0-63)
Y = *selected joystick y value* (0-63)

Note: Under Level 1, the following values are returned by this call:

A = fire button status
  $FF = fire button is on
  $00 = fire button is off

SS.AlfaS

(Function code $1C). Returns VDG alpha screen memory information

Entry Conditions:

A = *path number*
B = $1C

Exit Conditions:

A = *caps lock status*
$00 = lower case
$FF = upper case
X = *memory address of the buffer*
Y = *memory address of the cursor*

Additional Information:

- SS.AlfaS maps the screen into the user address space. The call requires a full block of memory for screen mapping. This call is only for use with VDG text screens handled by VDGINT.
- The support module for this call is VDGINT.
- **Warning:** Use extreme care when poking the screen, since other system variables reside in screen memory. Do not change any addresses outside of the screen.

SS.Cursr

(Function code $25). Returns VDG alpha screen cursor information

Entry Conditions:

A = *path number*
B = $25

Exit Conditions:

A = *character code of the character at the current cursor address*
X = *cursor X position (column)*
Y = *cursor Y position (row)*

Additional Information:

- SS.Cursr returns the character at the current cursor position. It also returns the X-Y address of the cursor relative to the current device's window or screen. SS.Cursr works only with text screens.
- The support module for this call is VDGINT.

SS.ScSiz

(Function code $26). Returns the window or screen size

Entry Conditions:

A = *path number*
B = $26

Exit Conditions:

X = *number of columns on screen/window*
Y = *number of rows on screen/window*

Additional Information:

- Use this call to determine the size of an output screen. The values returned depend on the device in use:

  For non-CCIO devices, the call returns the values following the XON/XOFF bytes in the device descriptor.

  For CCIO devices, the call returns the size of the window or screen in use by the specified device.

  For window devices. the call returns the size of the current working area of the window.

- The support modules for this call are VDGINT, GrfInt, and WindInt.

SS.KySns

(Function code $27). Returns key down status

Entry Conditions:

A = *path number*
B = $27

Exit Conditions:

A = *keyboard scan information*

Additional Information:

- Accumulator A returns with a bit pattern representing eight keys. With each keyboard scan, OS9 updates this bit pattern. A set bit (1) indicates that a key was pressed since the last scan. A clear bit (0) indicates that a key was not pressed. Definitions for the bits are as follows:

Bit	Key
0	[SHIFT]
1	[CTRL] or [CLEAR]
2	[ALT] or [@]
3	[↑] (up arrow)
4	[↓] (down arrow)
5	[←] (left arrow)
6	[→] (right arrow)
7	Space Bar

  The bits can be masked with the following equates:

```
SHIFTBIT   equ   %00000001
CNTRLBIT   equ   %00000010
ALTERBIT   equ   %00000100
UPBIT      equ   %00001000
DOWNBIT    equ   %00010000
LEFTBIT    equ   %00100000
RIGHTBIT   equ   %01000000
SPACEBIT   equ   %10000000
```

- The support module for this call is CC3IO.

SS.ComSt

(Function code $28). Return serial port configuration information

Entry Conditions:

A = *path number*
B = $28

Exit Conditions:

Y = *high byte: parity*
low byte: baud rate
(See the Setstat call SS.ComSt for values)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The SCF manager uses this call when performing an SS.Opt Getstat on an SCF-type device. User calls to SS.ComSt do not update the path descriptor. Use the SS.OPT Getstat call for most applications, because it automatically updates the path descriptor.

SS.MnSel

(Function code $87). Requests that the high-level menu handler take control of menu selection

Entry Conditions:

A = *path number*
B = $87

Exit Conditions:

A = *menu ID* (if valid selection)
0 (if invalid selection)
B = *item number of menu* (if valid selection)

Error Output:

CC = carry set on error
B = *error code* (if invalid selection)

Additional Information:

- After detecting a valid mouse click (when the mouse is pointing to a control area on a window), a process needs to call SS.MnSel. This call tells the enhanced window interface to handle any menu selection being made. If accumulator A returns with 0, then no selection has been made. The calling process needs to test and handle other returned values.
- A condition where Register A returns a valid menu ID number and Register B returns 0 signals the selection of a menu with no items. The application can now take over and do a special graphics pull down of its own. The menu title remains highlighted until the application calls the SS.UMBar SetStat to update the menu bar.
- The support module for this call is WindInt.

SS.Mouse

(Function code $89). Gets mouse status

Entry Conditions:

A = *path number*
B = $89
X = *data storage area address*
Y = *mouse port select:*
  0 = automatic selection
  1 = right side
  2 = left side

Exit Conditions:

X = *data storage area address*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.Mouse returns information on the current mouse and its fire button. The following list defines the 32-byte data packet that SS.Mouse creates:

Pt.Valid	rmb 1	Is returned info valid? (0 = no, 1 = yes)
Pt.Actv	rmb 1	Active side (0 = off, 1 = right, 2 = left)
Pt.ToTm	rmb 1	Timeout initial value
Pt.TTTo	rmb 1	Time until timeout
	rmb 2	RESERVED
Pt.TSSt	rmb 2	Time since counter start
Pt.CBSA	rmb 1	Current button state (Button A)
Pt.CBSB	rmb 1	Current button state (Button B)
Pt.CCtA	rmb 1	Click count (Button A)
Pt.CCtB	rmb 1	Click count (Button B)
Pt.TTSA	rmb 1	Time this state counter (Button A)
Pt.TTSB	rmb 1	Time this state counter (Button B)
Pt.TLSA	rmb 1	Time last state counter (Button A)
Pt.TLSB	rmb 1	Time last state counter (Button B)
	rmb 2	RESERVED
Pt.BDX	rmb 2	Button down frozen Actual X
Pt.BDY	rmb 2	Button down frozen Actual Y
Pt.Stat	rmb 1	Window pointer type location
Pt.Res	rmb 1	Resolution (0-640 by 0 = 10/1 = 1)
Pt.AcX	rmb 2	Actual X value
Pt.AcY	rmb 2	Actual Y value
Pt.WRX	rmb 2	Window relative X
Pt.WRY	rmb 2	Window relative Y
Pt.Siz	equ .	Packet size 32 bytes
SPt.SRX	rmb 2	System use, screen relative X
SPt.SRY	rmb 2	System use, screen relative Y
SPt.Siz	equ .	Size of packet for system use

- Button Information:

  Pt.Valid. The valid byte gives the caller an indication of whether the information contained in the returned packet is accurate. The information returned by this call is only valid if the process is running on the current interactive window. If the process is on a non-interactive window, the byte is zero and the process can ignore the information returned.

Pt.Actv. This byte shows which port is selected for use by all mouse functions. The default value is Right (1). You can change this value with the SS.GIP Setstat call.

Pt.ToTm. This is the initial value used by Pt.TTTo.

Pt.TTTo. This is the count down value (as of the instant the Getstat call is made). This value starts at the value contained in the Pt.ToTm and counts down to zero at a rate of 60 counts per second. The system maintains all counters until this value reaches 0, at which point it sets all counters and states to 0. The mouse scan routine changes into a quiet mode which requires less overhead than when the mouse is active. The timeout begins when both buttons are in the up (open) state. The timer is reinitialized to the value in Pt.ToTm when either button goes down (closed).

Pt.TSSt. This counter is constantly increasing, beginning when either button is pressed while the mouse is in the quiet state. All counts are a number of ticks (60 per second). The timer counts to $FFFF, then stays at that value if additional ticks occur.

Pt.CBSA. These flag bytes indicate the state of the button
Pt.CBSB. at the last system clock tick. A value of 0 indicates that the button is up (open). A value other than zero indicates that the button is down (closed). Button A is available on all Tandy joysticks and mice. Button B is only available for products that have two buttons.

The system scans the mouse buttons each time it scans the keyboard.

Pt.CCtA. This is the number of clicks that have occurred
Pt.CCtB. since the mouse went into an active state. A click is defined as pressing (closing) the button, then releasing (opening) the button. The system counts the clicks as the button is released.

Pt.TTSA. This counter is the number of ticks that have
Pt.TTSB. occurred during the current state, as defined by Pt.CBSx. This counter starts at one (counts the tick when the state changes) and increases by one for each tick that occurs while the button remains in the same state (open or closed).

Pt.TLSA. This counter is the number of ticks that have
Pt.TLSB. occurred during the time that a button is in a state opposite of the current state. Using this count and the TTSA/TTSB count, you can determine how a button was in the previous state, even if the system returns the packet after the state has changed. Use these counters, along with the state and click count values, to define any type of click, drag, or hold convention you want.

Reserved. Two packet bytes are reserved for future expansion of the returned data.

- Position Information:

Pt.BDX. These values are copies of the Pt.AcX and Pt.AcY
Pt.BDY. values when either of the buttons change from an open state to a closed state.

Pt.Stat. This byte contains information about the area of the screen on which the mouse is positioned. Pt.Valid must be a value other than 0 for this information to be accurate. If Pt.Valid is 0, this value is also 0 and not accurate. Three types of areas are currently defined:

0 = content region or current working area of the window
1 = control region (for use by Multi-View)
2 = off window, or on an area of the screen that is not part of the window

Pt.Res. This value is the current resolution for the mouse. The mouse must always return coordinates in the range of 0-639 for the X axis and 0-191 for the Y axis. If the system is so configured, you can use the high-resolution mouse adapter which provides a 1 to 1 ratio with these values plus 1. If the adapter is not in use, the resolution is a ratio of 1 to 10 on the X axis and 1 to 3 on the Y axis. The keyboard mouse provides a resolution of 1 to 1. The values in Pt.Res are:

0 = low res (x:10, y:3)
1 = high res (x,y:1)

Pt.AcX. The values read from the mouse returned in the
Pt.AcY. resolution as described under Pt.Res.

Pt.WRX. The values read from the mouse minus the
Pt.WRY. starting coordinates of the current window's working area. These values return the coordinates in numbers relative to the type of screen. For example, the X axis is in the range 0-639 for high-resolution screens and 0-319 for low resolution screens. You can divide the window relative values by 8 to obtain absolute character positions. These values are most helpful when working in non-scaled modes.

- The support modules for this call are CC3IO, GrfInt, and WindInt.

SS.Palet

(Function code $91). Gets palette information

Entry Conditions:

A = *path number*
B = $91
X = *pointer to the 16-byte palette information buffer*

Exit Conditions:

X = *pointer to the 16-byte palette information buffer*

Additional Information:

- SS.Palet reads the contents of the 16 screen palette registers, and stores them in a 16-byte buffer. When you make the call, be sure the X register points to the desired buffer location. The pointer is retained on exit. The palette values returned are specific to the screen on which the call is made.
- The support modules for this call are VDGINT, GrfInt, and WindInt.

SS.ScTyp

(Function code $93). Returns the type of a screen to a calling program.

Entry Conditions:

A = *path*
B = $93

Exit Conditions:

A = *screen type code*
1 = 40 x 24 text screen
2 = 80 x 24 text screen
3 = not used
4 = not used
5 = 640 x 192, 2-color graphics screen
6 = 320 x 192, 4-color graphics screen
7 = 640 x 192, 4-color graphics screen
8 = 320 x 192, 16-color graphics screen

Additional Information:

- Support modules for this system call are GrfInt and WindInt.

SS.FBRgs

(Function code $96). Returns the foreground, background and border palette registers for a window.

Entry Conditions:

A = *path number*
B = $96

Exit Conditions:

A = *foreground palette register number*
B = *background palette register number* (if carry clear)
X = *least significant byte of border palette register number*

Error Output:

B = *error code if any*
CC = *carry set on error*

Additional Information:

- Support modules for SS.FBRgs are GrfInt and WindInt.

SS.DFPal

(Function code $97). Returns the default palette register settings.

Entry Conditions:

A = *path number*
B = $97
X = *pointer to 16-byte data space*

Exit Conditions:

X = *default palette data moved to user space*

Error Output:

B = *error code,* if any
CC = carry set on error

Additional Information:

- You can use SS.DFPal to find the values of the default palette registers that are used when a new screen is allocated by GrfInt or WindInt. The corresponding SetStat can alter the default registers. This GetStat/SetStat pair is for system configuration utilities and should not be used by general applications.

Set Status System Calls

Use the Set Status system calls with the RBF manager subroutine SETSTA. The OS-9 Level Two system reserves function Codes 7-127 for use by Microware. You can define Codes 200-255 and their parameter-passing conventions for your own use. (See the sections on device drivers in Chapters 4, 5, and 6.)

Following are the Set Status functions and their codes.

SS.OPT

(Function code $00). Writes the option section of the path descriptor

Entry Conditions:

A = *path number*
B = $00
X = *address of the status packet*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.OPT writes the option section of the path descriptor from the 32-byte status packet pointed to by Register X. Use this system call to set the device operating parameters, such as echo and line feed.

SS.SIZ

(Function code $02). Changes the size of a file for RBF-type devices

Entry Conditions:

A = *path number*
B = $02
X = *most significant 16 bits of the desired file size*
U = *least significant 16 bits of the desired file size*

Error Output:

CC = carry set on error
B = *error code* (if any)

SS.RESET

(Function code $03). Restores the disk drive head to Track 0 in preparation for formatting and error recovery (use only with RBF-type devices)

Entry Conditions:

A = *path number*
B = $03

Error Output:

CC = carry set on error
B = *error code* (if any)

SS.WTRK

(Function code $04). Formats (writes) a track on a disk (RBF-type devices only)

Entry Conditions:

A = *path number*
B = $04
U = *track number* (least significant 8 bits)
X = *address of the track buffer*
Y = *side/density*
  Bit B0 = side
    0 = Side 0
    1 = Side 1
  Bit B1 = density
    0 = single
    1 = double

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- For hard disks or floppy disks that have a "format entire diskette command," SS.WTRK formats the entire disk only when the *track number* is zero.

SS.SQD

(Function code $0C). Starts the shutdown procedure for a hard disk that has sequence-down requirements prior to removal of power. (Use only with RBF-type devices.)

Entry Conditions:

A = *path number*
B = $0C

Exit Conditions: None

SS.KySns

(Function code $27). Turns the key sense function on and off

Entry Conditions:

A = *path number*
B = $27
X = *key sense switch value*
  0 = normal key operation
  1 = key sense operation

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- When SS.KySns switches the keyboard to key sense mode, the CC3IO module suspends transmission of keyboard characters to the SCF manager and the user. While the computer is in key sense mode, the only way to detect key press is with SS.KySns.
- The support module for this call is CC3IO.

SS.ComSt

(Function code $28). Used by the SCF manager to configure a serial port

Entry Conditions:

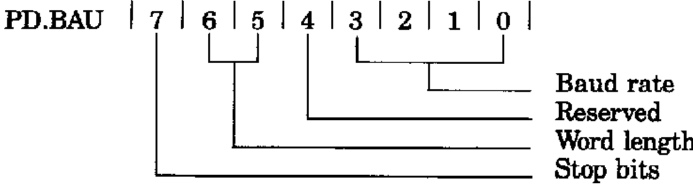
A = *path number*
B = $28
Y = *high byte: parity*
low byte: baud rate

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

Baud Configuration. The high order byte of Y determines the baud rate, the word length, and number of stop bits. The byte is configured as follows:

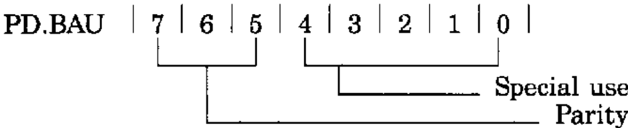


Stop bits:
0 = 1
1 = 2

Word length:
00 = 8 bit
01 = 7 bit

Baud rate:
0000 = 110
0001 = 300
0010 = 600
0011 = 1200
0100 = 2400
0101 = 4800
0110 = 9600
0111 = 19200
1xxx = undefined

- **Parity Configuration.** The low order byte of Y determines parity. The byte is configured as follows:



Parity:

xx0 = none
001 = odd (ACIAPAK and MODPAK only)
011 = even (ACIAPAK and MODPAK only)
101 = transmit: mark
receive: ignore
111 = transmit: space
receive: ignore

- The SCF manager uses SS.ComSt to inform a driver that serial port configuration information has been changed in the path descriptor. After calling SS.ComSt, a user routine must call the SS.OPT SetStat to correctly update the path descriptor.
- This call is for the use of the SCF manager. Use SS.OPT for changing baud, stop bit, and parity values.

SS.Close

(Function code $2A). Informs a device driver that a path is closed.

Additional Information:

This call is used internally by OS-9's SCF file manager and is not available for user programs. It can be used only by device drivers and file managers.

SS.AAGBf

(Function code $80). Reserves an additional graphics buffer

Entry Conditions:

A = *path number*
B = $80

Exit Conditions:

X = *buffer address*
Y = *buffer number (1-2)*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.AAGBf allocates an additional 8K graphics buffer. The first buffer (Buffer 0) must be allocated by using the DISPLAY GRAPHICS command. To use the DISPLAY GRAPHICS command, send control code $0F to the standard terminal driver. SS.AAGBf can allocate up to two additional buffers (Buffers 1 and 2), one at a time.
- After calling SS.AAGBf, Register X contains the address of the new buffer. Register Y contains the buffer number.
- To deallocate all graphics buffers, use the END GRAPHICS control code.
- When SS.AAGBf allocates a buffer, it also maps the buffer into the application's address space. Each buffer uses 8K of the available memory in the application's address space. Also, if SS.DStat is called, Buffer 0 is also mapped into the application's address space. Allocation of all three buffers reduces the application's free memory by 24K.
- The support module for this call is VDGINT.

SS.SLGBf

(Function code $81). Selects a graphics buffer

Entry Conditions:

A	= *path number*	
B	= $81	
X	= $00	*select buffer for use*
	$01-$FF	*select buffer for use and display*
Y	= *buffer number* (0-2)	

Exit Conditions:

X	= *unchanged from entry*
Y	= *unchanged from entry*

Error Output:

CC	= carry set on error
B	= *error code* (if any)

Additional Information:

- Use DISPLAY GRAPHICS to allocate the first graphics buffer. Use SS.AAGBf to allocate the second and third graphics buffers.
- Save each return address when writing directly to a screen. It is not necessary to save return addresses when using operating system graphics commands.
- SS.SLGBf does not update hardware information until the next vertical retrace (60Hz rate). Programs that use SS.AAGBf to change current draw buffers need to wait long enough to ensure that OS-9 has moved the current buffer to the screen.
- The screen shows the buffer only if the buffer is selected as the interactive device. If the device does not possess the keyboard, OS-9 stores the information until the device is selected as the interactive device. When the device is selected as the interactive device, the display shows the selected device's screen.
- The support module for this call is VDGINT.

SS.MpGPB

(Function code $84). Maps the Get/Put buffer into a user address space

Entry Conditions:

A = *path number*
B = $84
X = *high byte: buffer group number*
low byte: buffer number
Y = *action to take*
1 = map buffer
0 = unmap buffer

Exit Conditions:

X = *address of the mapped buffer*
Y = *number of bytes in buffer*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The support modules for this call are GrfInt and WindInt.
- SS.MpGPB maps a Get/Put buffer into the user address space. You can then save the buffer to disk or directly modify the pixel data contained in the buffer. Use extreme care when modifying the buffer so that you do not write outside of the buffer data area.

SS.WnSet

(Function code $86). Set up a high level window handler

Entry Conditions:

A = *path number*
B = $86
X = *window data pointer* (if Y = WT.FSWin or WT.Win)
Y = *window type code*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- The C language data structures for windowing are defined in the wind.h file in the DEFS directory of the system disk.
- The support module for this call is WindInt.

SS.SBar

(Function code $88). Puts a scroll block at a specified position

Entry Conditions:

A = *path number*
B = $88
X = *horizontal position of scroll block*
Y = *vertical position of scroll block*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- WT.FSWin-type windows have areas at the bottom and right sides to indicate their relative positions within a larger area. These areas are called scroll bars. SS.SBar gives an application the ability to maintain relative position markers within the scroll bars. The markers indicate

the location of the current screen within a larger screen. Calling SS.SBar, updates both scroll markers.

- The support module for this call is WindInt.

SS.Mouse

(Function code $89). Sets the sample rate and button timeout for a mouse

Entry Conditions:

A = *path number*
B = $89
X = *mouse sample rate and timeout*
most significant byte = *mouse sample rate*
least significant byte = *mouse timeout*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS. Mouse allows the application to define the mouse parameters. The sample rate is the number of clock ticks between the actual readings of the mouse status.

- The support module for the call is CC3IO.

SS.MsSig

(Function code $8A). Sends a signal to a process when the mouse button is pressed

Entry Conditions:

A = *path number*
B = $8A
X = *user defined signal code* (low byte only)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.MsSig sends the process a signal the next time a mouse button changes state (from open to closed). Once SS.MsSig sends the signal, the process must repeat the Setstat each time that it needs to set up the signal.
- Processes using SS.MsSig should have an intercept routine to trap the signal. By intercepting the signal, other processes can be notified when the change occurs. Therefore, the other processes do not need to continually poll the mouse.
- The SS.Relea Setstat clears the pending signal request, if desired. It also clears any pending signal from SS.SSig. Because of this, if you want to clear only one signal, you must reset the other signal after calling SS.MsSig.
- The support module for this call is CC3IO.

SS.AScrn

(Function code $8B). Allocates and maps a high-resolution screen into an application address space

Entry Conditions:

A = *path number*
B = $8B
X = *screen type*
  0 = 640 x 192 x 2 colors (16K)
  1 = 320 x 192 x 4 colors (16K)
  2 = 160 x 192 x 16 colors (16K)
  3 = 640 x 192 x 4 colors (32K)
  4 = 320 x 192 x 16 colors (32K)

Exit Conditions:

X = *application address space of screen*
Y = *screen number* (1-3)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.AScrn is particularly useful in systems with minimal memory when you want to allocate a high resolution graphics screen with all screen updating handled by a process.
- This call uses VDGInt (GRFINT is not required).
- All screens are allocated in multiples of 8K blocks. You can allocate a maximum of three buffers at one time. To select between buffers, use the SS.DScrn Setstat call.
- Screen memory is allocated but not cleared. The application using the screen must do this.
- Screens must be allocated from a VDG-type device—a standard 32-column text screen must be available for the device.
- The support module for this call is VDGINT.

SS.DScrn

(Function code $8C). Causes VDGINT to display a screen that was allocated by SS.AScrn

Entry Conditions:

A = *path number*
B = $8C
Y = *screen number* (1-3)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.DScrn shows the requested screen if the requested screen is the current interactive device.
- The support module for this call is VDGINT.

SS.FScrn

(Function code $8D). Frees the memory of a screen allocated by SS.AScrn

Entry Conditions:

A = *path number*
B = $8D
Y = *screen number* (1-3)

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- Do not attempt to free a screen that is currently on the display.
- SS.FScrn returns the screen memory to the system and removes it from an application's address space.
- The support module for this call is VDGINT.

SS.PScrn

(Function code $8E). Converts a screen to a different type

Entry Conditions:

A = *path number*
B = $8E
X = *new screen type*
  0 = 640 x 192 x 2 colors (16K)
  1 = 320 x 192 x 4 colors (16K)
  2 = 160 x 192 x 16 colors (16K)
  3 = 640 x 192 x 4 colors (32K)
  4 = 320 x 192 x 16 colors (32K)
Y = *screen number*

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.PScrn changes a screen allocated by SS.AScrn to a new screen type. You can change a 32K screen to either a 32K screen, or a 16K screen. You can change a 16K screen only to another 16K screen type. SS.PScrn updates the current display screen at the next clock interrupt.
- However, if you change a 32K screen to a 16K screen, OS-9 does not reclaim the extra 16K of memory. This means that you can later change the 16K screen back to a 32K screen.
- The support module for this call is VDGINT.

SS.Montr

(Function code $92). Sets the monitor type

Entry Conditions:

A = *path number*
B = $92
X = *monitor type*
  0 = color composite
  1 = analog RGB
  2 = monochrome composite

Error Output:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- SS.Montr loads the hardware palette registers with the codes for the default color set for three types of monitors. The system default initializes the palette for a composite color monitor.
- The monochrome mode removes color information from the signals sent to a monitor.
- When a composite monitor is in use, a conversion table maps colors from RGB color numbers. In RGB and monochrome modes, the system uses the RGB color numbers directly.
- The support modules for this call are VDGINT and GrfDrv.

SS.GIP

(Function code $94). Sets the system wide mouse and key repeat parameters

Entry Conditions:

A = *path number*
B = $94
X = *mouse resolution;* in the most significant byte
    0 = low resolution mouse
    1 = optional high resolution adapter
  = *mouse port location;* in the least significant byte
    1 = right port
    2 = left port
Y = *key repeat start constant;* in the most significant byte
  = *key repeat delay;* in the least significant byte
    00XX = no repeat
    FFFF = unchanged

Error Output:

CC = carry set if error
B = *error code*, if any

Additional Information:

- Because this function affects system-wide settings, it is best to use it from system configuration utilities and not from general application program.
- The support module for this call is CC3IO.

SS.UMBAR

(Function code $95). Requests the high level menu manager to update the menu bar.

Entry Conditions:

A = *path number*
B = $95

Exit Conditions:

CC = carry set on error
B = *error code* (if any)

Additional Information:

- An application can call SS.UMBar when it needs to redraw menu bar information, such as when it enables or disables menus, or when it completes a window *pull down* and needs to restore the menu.
- The support module for this call is WindInt.

SS.DFPal

(Function code $97). Sets the default palette register values

Entry Conditions:

A = *path number*
B = $97
X = *pointer to 16 bytes of palette data*

Exit Conditions:

X = unchanged, bytes moved to system defaults
CC = carry set on error
B = *error code* (if any)

Additional Information:

- Use SS.DFPal to alter the system-wide palette register defaults. The system uses these defaults when it allocates a new screen using the DWSet command.

- Because this function affects system wide settings, it is best to use it from system configuration utilities, not general application programs.

SS.Tone

(Function code $98). Creates a sound through the terminal output device.

Entry Conditions:

A = *path number*
B = $98
X = *duration and amplitude of the tone*
  LSB = duration in ticks (60-sec) in the range 0-255
  MSB = amplitude of tone in the range 0-63
Y = *relative frequency counter* (0 = low, 4095 = high)

Exit Conditions:

These are the same as the entry conditions. There are no error conditions.

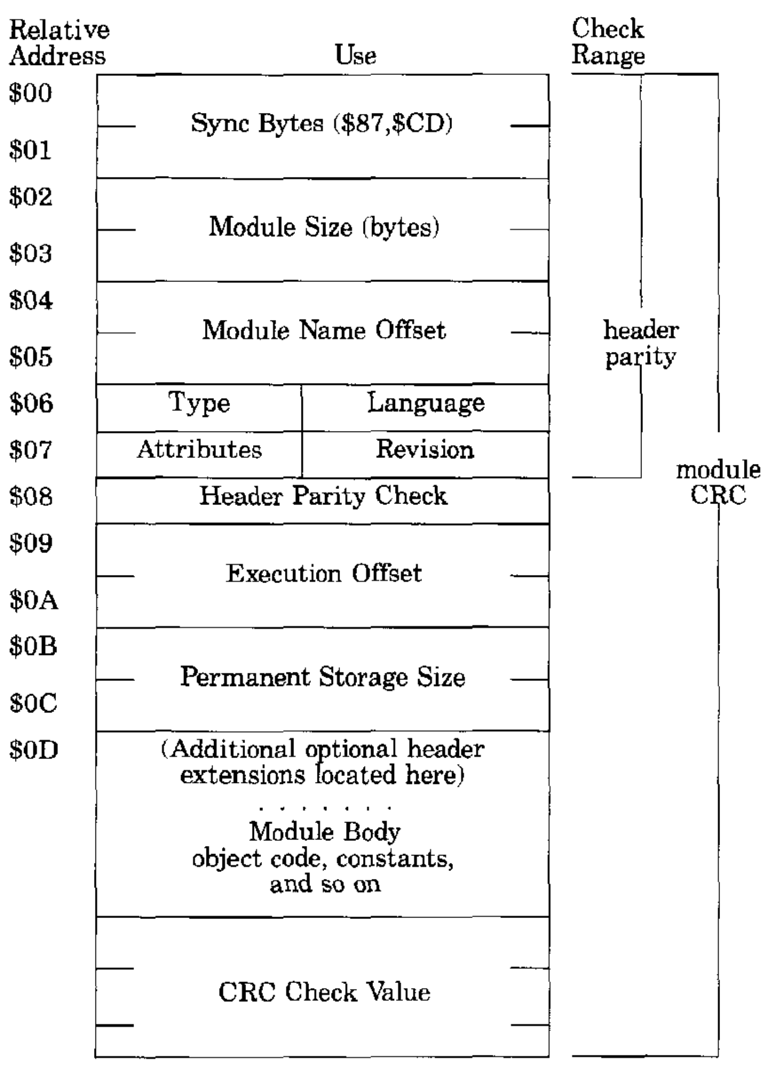
Additional Information:

- This call produces a programmed IO tone through the speaker of the monitor used by the terminal device. You can make the call on any valid path open to term or to a window device.
- The system does not mask interrupts during the time the tone is being produced.
- The frequency of the tone is a relative number ranging from 0 for a low frequency to 4095 for a high frequency. The widest variation of tones occurs at the high range of the scale.

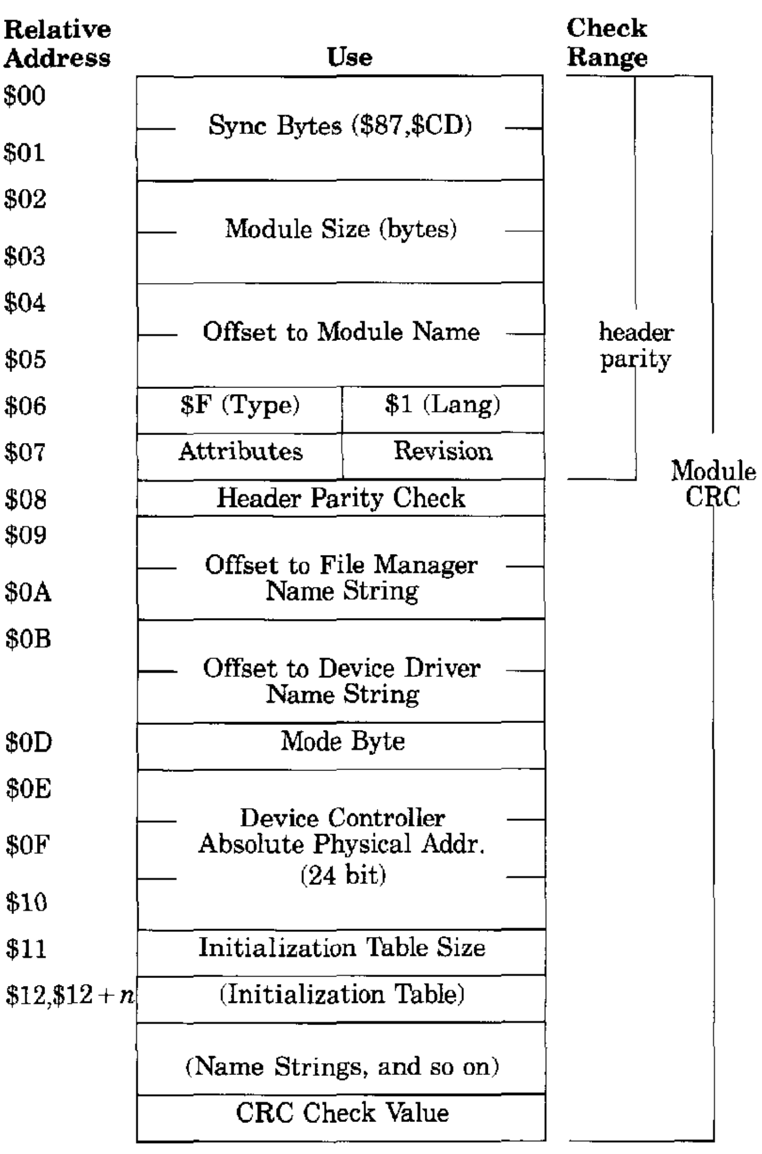
Appendix A

Memory Module Diagrams

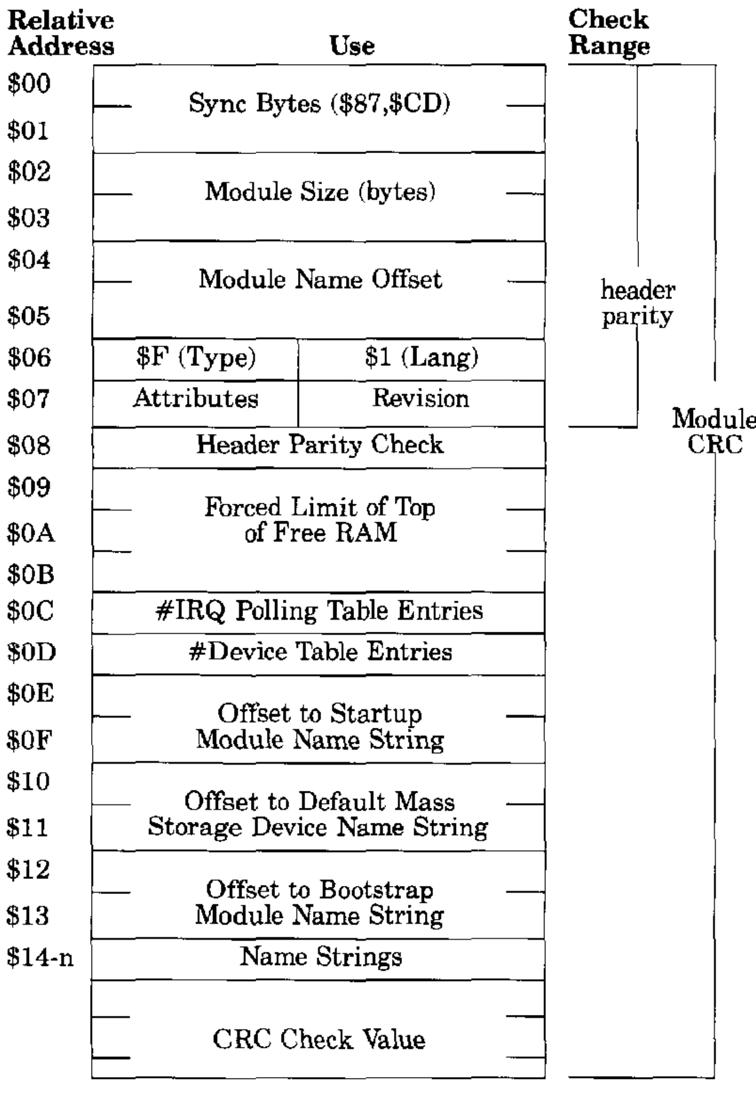
Executable Memory Module Format



Device Descriptor Format



INIT Module Format



Appendix B

Standard Floppy Disk Format

Color Computer 3

Physical Track Format Pattern

Format	Bytes (Dec)	Value (Hex)
Header pattern (once per track)	32	4E
	12	00
	3	F5
	1	FC
	32	4E
Sector pattern (repeated 18 times)	12	00
	3	F5
	1	track number (0-34)
	1	side number (0-1)
	1	sector number (1-18)
	1	sector length code (1)
	2	CRC
	22	4E
	12	00
	3	F5
	1	FB
	256	data area
	2	CRC
	24	4E
Trailer pattern (once per track)	N	4E (fill to index mark)

Appendix C

System Error Codes

The error codes are shown in both hexadecimal and decimal. The error codes listed include OS-9 system error codes, BASIC error codes, and standard windowing system error codes.

Code HEX	DEC	Code Meaning
$01	001	UNCONDITIONAL ABORT — An error occurred from which OS-9 cannot recover. All processes are terminated.
$02	002	KEYBOARD ABORT — You pressed BREAK to terminate the current operation.
$03	003	KEYBOARD INTERRUPT — You pressed SHIFT BREAK either to cause the current operation to function as a background task with no video display or to cause the current task to terminate.
$B7	183	ILLEGAL WINDOW TYPE — You tried to define a text type window for graphics or used illegal parameters.
$B8	184	WINDOW ALREADY DEFINED — You tried to create a window that is already established.
$B9	185	FONT NOT FOUND — You tried to use a window font that does not exist.
$BA	186	STACK OVERFLOW — Your process (or processes) requires more stack space than is available on the system.
$BB	187	ILLEGAL ARGUMENT — You have used an argument with a command that is inappropriate.
$BD	189	ILLEGAL COORDINATES — You have given coordinates to a graphics command which are outside the screen boundaries.
$BE	190	INTERNAL INTEGRITY CHECK — System modules or data are changed and no longer reliable.
$BF	191	BUFFER SIZE IS TOO SMALL — The data you assigned to a buffer is larger than the buffer.

Code HEX	Code DEC	Code Meaning
$C0	192	ILLEGAL COMMAND — You have issued a command in a form unacceptable to OS-9.
$C1	193	SCREEN OR WINDOW TABLE IS FULL — You do not have enough room in the system window table to keep track of any more windows or screens.
$C2	194	BAD/UNDEFINED BUFFER NUMBER — You have specified an illegal or undefined buffer number.
$C3	195	ILLEGAL WINDOW DEFINITION — You have tried to give a window illegal parameters.
$C4	196	WINDOW UNDEFINED — You have tried to access a window that you have not yet defined.
$C8	200	PATH TABLE FULL — OS-9 cannot open the file, because the system path table is full.
$C9	201	ILLEGAL PATH NUMBER — The path number is too large, or you specified a non-existent path.
$CA	202	INTERRUPT POLLING TABLE FULL — Your system cannot handle an interrupt request, because the polling table does not have room for more entries.
$CB	203	ILLEGAL MODE — The specified device cannot perform the indicated input or output function.
$CC	204	DEVICE TABLE FULL — The device table does not have enough room for another device.
$CD	205	ILLEGAL MODULE HEADER — OS-9 cannot load the specified module because its sync code, header parity, or Cyclic Redundancy Code is incorrect.
$CE	206	MODULE DIRECTORY FULL — The module directory does not have enough room for another module entry.

Code HEX	Code DEC	Code Meaning
$CF	207	MEMORY FULL — Process address space is full or your computer does not have sufficient memory to perform the specified task.
$D0	208	ILLEGAL SERVICE REQUEST — The current program has issued a system call containing an illegal code number.
$D1	209	MODULE BUSY — Another process is already using a non-shareable module.
$D2	210	BOUNDARY ERROR — OS-9 has received a memory allocation or deallocation request that is not on a page boundary.
$D3	211	END OF FILE — A read operation has encountered an end-of-file character and has terminated.
$D4	212	RETURNING NON-ALLOCATED MEMORY — The current operation has attempted to deallocate memory not previously assigned.
$D5	213	NON-EXISTING SEGMENT — The file structure of the specified device is damaged.
$D6	214	NO PERMISSION — The attributes of the specified file or device do not permit the requested access.
$D7	215	BAD PATH NAME — The specified pathlist contains a syntax error, for instance an illegal character.
$D8	216	PATH NAME NOT FOUND — The system cannot find the specified pathlist.
$D9	217	SEGMENT LIST FULL — The specified file is too fragmented for further expansion.
$DA	218	FILE ALREADY EXISTS — The specified filename already exists in the specified directory.
$DB	219	ILLEGAL BLOCK ADDRESS — The file structure of the specified device is damaged.

Code HEX	DEC	Code Meaning
$DC	220	PHONE HANGUP-DATA CARRIER DETECT LOST — The data carrier detect is lost on the RS-232 port.
$DD	221	MODULE NOT FOUND — The system received a request to link a module that is not in the specified directory.
$DF	223	SUICIDE ATTEMPT — The current operation has attempted to return to the memory location of the stack.
$E0	224	ILLEGAL PROCESS NUMBER — The specified process does not exist.
$E2	226	NO CHILDREN — The system has issued a *wait service* request but the current process has no dependent process to execute.
$E3	227	ILLEGAL SWI CODE — The system received a software interrupt code that is less than 1 or greater than 3.
$E4	228	PROCESS ABORTED — The system received a signal Code 2 to terminate the current process.
$E5	229	PROCESS TABLE FULL — A fork request cannot execute because the process table has no room for more entries.
$E6	230	ILLEGAL PARAMETER AREA — A fork call has passed incorrect high and low bounds.
$E7	231	KNOWN MODULE — The specified module is for internal use only.
$E8	232	INCORRECT MODULE CRC — The CRC for the module being accessed is bad.
$E9	233	SIGNAL ERROR — The receiving process has a previous, unprocessed signal pending.
$EA	234	NON-EXISTENT MODULE — The system cannot locate the specified module.

Code HEX	DEC	Code Meaning
$EB	235	BAD NAME — The specified device, file, or module name is illegal.
$EC	236	BAD MODULE HEADER — The specified module header parity is incorrect.
$ED	237	RAM FULL — No free system random access memory is available: the system address space is full, or there is no physical memory available when requested by the operating system in the system state.
$EE	238	UNKNOWN PROCESS ID — The specified process ID number is incorrect.
$EF	239	NO TASK NUMBER AVAILABLE — All available task numbers are in use.

Device Driver Errors

I/O device drivers generate the following error codes. In most cases, the codes are hardware-dependent. Consult your device manual for more details.

Code HEX	DEC	Code Meaning
$F0	240	UNIT ERROR — The specified device unit doesn't exist.
$F1	241	SECTOR ERROR — The specified sector number is out of range.
$F2	242	WRITE PROTECT — The specified device is write-protected.
$F3	243	CRC ERROR — A Cyclic Redundancy Code error occurred on a read or write verify.
$F4	244	READ ERROR — A data transfer error occurred during a disk read operation, or there is a SCF (terminal) input buffer overrun.

Code HEX	Code DEC	Code Meaning
$F5	245	WRITE ERROR — An error occurred during a write operation.
$F6	246	NOT READY — The device specified has a *not ready* status.
$F7	247	SEEK ERROR — The system attempted a seek operation on a non-existent sector.
$F8	248	MEDIA FULL — The specified media has insufficient free space for the operation.
$F9	249	WRONG TYPE — An attempt is made to read incompatible media (for instance an attempt to read double-side disk on single-side drive).
$FA	250	DEVICE BUSY — A non-shareable device is in use.
$FB	251	DISK ID CHANGE — You changed diskettes when one or more files are open.
$FC	252	RECORD IS LOCKED-OUT — Another process is accessing the requested record.
$FD	253	NON-SHAREABLE FILE BUSY — Another process is accessing the requested file.
$FE	254	I/O DEADLOCK ERROR — Two processes have attempted to gain control of the same disk area at the same time.

Index

ACIAPAK 8-135
active process 2-12 - 2-13
  queue 2-14, 8-98
  state 2-13 - 2-14
address
  find 64K block 8-85
  lines 2-7
  polling 2-17
  space, add module 8-104
age, process 2-14
alarm, set 8-66
allocate
  high RAM 8-69
  image 8-70
  memory 8-76
  memory blocks 8-67 - 8-68
  process descriptor 8-71
  process task number 8-73
  RAM 8-72
allocation
  bit map 8-7
  map sector 5-1
  of memory 2-5 - 2-7
  polling 2-17
allocation map
  clear 8-13
  disk 5-3
alpha screen
  cursor 8-118
  memory 8-117
ASM assembler 8-2
assembler, RMA 8-2
attach a device 8-44 - 8-45
attribute
  byte 5-5,
  file 5-12

background color, get 8-129
bell, set alarm 8-66
bit map 2-5
  allocation 8-7
bit map (cont'd.)
  search memory allocation 8-33
block
  allocate system memory 8-105
  deallocate system memory 8-106
  map into process space 8-96
  number 2-7
  scroll 8-139
block map, system 8-18
boot
  file, load 5-26
  module, link 8-75
booting OS-9 1-3
bootstrap
  memory request 8-76
  system 8-75
border color, get 8-129
buffer
  map (Get/Put) 8-138
  reserve graphics 8-136
button
  state, mouse 8-124 - 8-125, 8-126
  timeout, mouse 8-140
byte
  attribute 5-5
  deallocate 64-byte block 8-101
  get from memory block 8-94
  get two bytes 8-95
  read from path 8-59 - 8-60
  store in task 8-109

calling process
  insert in I/O queue 8-91
  terminate 8-14
  turn off 8-35, 8-43
CC3DISK 1-2

CC3GO module 2-19
CC3IO 1-2, 6-1
chain 8-8 - 8-9
change
  device operating parameters 5-23
  directory 8-46
character
  read SCF input 6-13
  write, SCF 6-14
ChgDir 4-4
child process 2-13
  create 8-15 - 8-17
clear specified block 8-77
click 8-126
CLOCK 1-2
clock
  module 1-2, 2-19
  real-time 2-12, 2-17
close
  file 4-7
  path 8-47, 8-135
codes
  signal 2-15
  system error C-1
command interpreter 1-4
communication, interprocess 2-15
compact module directory 8-88
compare strings 8-10
compatibility with Level One 2-1
concurrent execution 7-1 - 7-3
copy external memory 8-11
count, link 2-5
counter start, mouse 8-124
CPU 2-7
  time 4-11
CRC
  calculate 8-12
  validate module 8-111
  value 3-1 - 3-3
create
  directory 8-55 - 8-56
create (cont'd.)
  file 8-48 - 8-49
current
  data directory 8-51
  execution directory 8-51
cursor positioning 4-5
cyclic redundancy check 3-1 - 3-3

DAT
  hardware 8-99
  registers 8-103
  to logical address 8-78
data
  available, SCF test 8-113
  directory 8-51
  stream 4-3
  transfer, pipes 7-1 - 7-3
  move in memory 8-97
DAT image 8-70
  conversion 8-78
  copy into process descriptor 8-102
  deallocate block 8-77
  high block 8-86
  low block 8-87
  pointer 8-95
DAT task number
  release 8-99
  reserve 8-100
date
  get system 8-40
  set 8-38
deadlock 5-13
deadly embrace 5-13
deallocate
  image RAM blocks 8-79
  map bits 8-13
  process descriptor 8-80
  RAM blocks 8-81
  task number 8-82
default palette registers 8-129, 8-149
delete file 8-50 - 8-51

descriptions, system call 8-2
descriptor
- get process 8-20
- path 4-18
- pointer 8-82
- process 2-13

detach device 8-52
device
- add or remove from polling table 8-92
- attach 8-44 - 8-45
- attachment, verify 8-44 - 8-45
- controller 5-15
- control registers, initialize 6-12
- control registers, SCF 6-12
- descriptor 1-4, 4-2, 4-17, A-2
- detach 8-52
- modules 5-15
- modules, RBF 5-8 - 5-10
- modules, SCF 6-6 - 6-8
- name, get 8-115
- open path to 8-57 - 8-58
- operating parameters, RBF 5-23
- operating parameters, SCF 6-15
- status 2-17 - 2-18, 8-63
- status, get 8-54
- table 4-2, 8-52
- terminate, RBF 5-24
- terminate, SCF 6-16
- write to 8-64 - 8-65

device driver 1-3, 4-11
- close path 8-135
- modules 4-8
- name 5-15
- SCF 6-9 - 6-17
- SCF subroutines 6-10 - 6-17
- subroutines, RBF 5-16 - 5-27

device driver modules, RBF 5-13 - 5-17
device interrupt 5-25
- SCF 6-17

directory
- attribute byte 5-5
- change 8-46
- disk 5-5
- entry, module 8-83
- get module 8-19
- make 8-55 - 8-56
- module 2-12, 8-88

disk
- directories 5-5
- sector read 5-19, 5-21

disk allocation map 5-3
- sector 5-1

diskette format B-1
display
- screen 8-143
- status, get 8-115

drag 8-126
drive head, restore 8-131
duplicate path 8-53

editing, line 6-1, 8-61
end-of-file, test for 8-114
equate file 2-4
equivalent logical address 8-78
error
- codes, system C-1 - C-6
- message, write 8-30
- print 8-30

exclamation point, pipes 7-1 - 7-3
execute
- mode 5-11
- system calls 8-1 - 8-2

execution
- directory 8-51
- offset, module 3-7

exit calling process 8-14
external memory, read 8-11

fatal signal 2-13
file
  attribute byte 5-12
  closing 4-7
  create 4-4, 5-12, 8-48 - 8-49
  deadlock 5-13
  delete 4-5, 8-50 - 8-51
  descriptor 5-3 - 5-4
  execute mode 5-11
  get pointer position 8-114
  line reading/writing 4-6
  load module 8-29
  locking 5-12
  non-shareable 5-12
  opening 4-4
  open path 8-57 - 8-58
  permission bits 5-4
  pipe 7-1 - 7-3
  pointer 4-5, 8-62
  position, RBF 8-114
  read 5-1, 4-5
  sharing 5-12
  size, get 8-114
  status, get 8-54, 8-114
  update mode 5-11
  write line to 8-64 - 8-65
  writing 4-6
file manager 1-3
  modules 4-3
  name 5-15
find
  64-byte block 8-85
  module directory entry 8-84
fire button 8-123 - 8-127
FIRQ 4-12
  interrupt 2-17
flag, RAM In Use 8-81
flip byte 2-17
floppy diskette format B-1
foreground color, get 8-129
FORK 2-8
fork, child process 8-15 - 8-17
FORMAT 5-2
format
  device descriptor 4-17, A-2
  INIT module A-3
  memory module 3-6 - 3-7, A-1
  of device driver modules 4-10
  track 8-132
function
  calls 2-4 - 2-5, 8-1
  key sense 8-133

get
  a byte 8-94
  free high block 8-86
  free low block 8-87
  ID 8-22
  process pointer 8-89
  status 8-54
  Status system calls 8-112 - 8-130
  system time 8-40
Get/Put buffer, map 8-138
GETSTA 8-112
  SCF 6-15
GetStat 4-6
Getstats 5-23
graphics buffer
  reserve 8-136
  select 8-137
graphics interface 1-2
GRFINT 1-2

handler routine, virtual interrupt 8-110
hard disk shutdown 8-133
hardware
  controller, SCF 6-9
  DAT registers 8-103
  vector 2-16
header
  module 3-1 - 3-2
  parity 8-111

header (cont'd.)
  pattern, floppy
    diskette B-1
high block, memory search
  8-86
high-level
  menu handler 8-122
  menu manager 8-148
  window handler 8-139
high-resolution
  mouse adapter 8-126
  screen, allocate 8-142
hold, button 8-126
I/O
  calls 2-4 - 2-5, 8-1
  device accessing 2-11
  module, delete 8-90
  path, close 8-47
  queue, insert calling
    process 8-91
I/O system 1-3 - 1-4
  calls 2-1, 8-2
  system modules 1-1 -
    1-4, 4-1
  transfers 4-8
ID
  return caller's
    process 8-22
  set user 8-39
identification sector 5-1
image, allocate 8-70
INIT 1-2, 5-18
INIT module 2-17
  format A-3
  link 8-75
Init, SFC 6-12
initialization table, SCF
  device 6-6 - 6-8
initialize device memory 5-18
input buffer, read SCF
  character 6-13
insert process 8-74
install virtual interrupt
  8-110
intercept, set signal 8-21
interface
  graphics 1-2
  VDG 4-2
  Windint 4-2
interprocess
  communication 2-15
interrupt
  device 5-25
  enable, SCF 6-12
  FIRQ 2-17
  processing 2-1
IOMAN 1-2
IRQ 4-12
  add/remove device from
    polling table 8-92
  interrupt 2-17
  polling 2-17
  polling table 2-18
  service routine 5-25
IRQSVC routine 4-13
IRQSV 4-11

joystick value, get 8-116

kernel 1-2
key
  repeat parameters,
    set 8-147
  sense function 8-133
  status, get 8-120
keyboard scan 2-17

language byte 3-4
line
  editing 6-1, 8-61
  reads 4-6, 8-61
  writes 4-6, 8-65
link
  to memory module 8-23
    - 8-24, 8-28
  using module directory
    entry 8-83
link count 2-5
  decrease 8-42

load
  boot file 5-26
  byte from memory block 8-94
  from task offset 8-93
  module 8-25 - 8-26, 8-29
  two bytes 8-95
lock, end-of-lock 5-12
locking
  files 5-12
  record 5-10 - 5-11
logical
  address space 2-6, 2-8
  sector number 5-1
LSN 5-2, 5-5

macro 2-4
MAKDIR 4-4
make directory 8-55 - 8-56
manager
  file 1-3
  random block 1-3
  sequential file 1-3
map
  block 8-96
  search allocation 8-33
mask byte 2-18
memory
  allocate 8-76
  allocate blocks 8-67 - 8-68
  allocate high RAM 8-69
  change process data size 8-27
  deallocate 2-5
  find low block 8-87
  free screen 8-144
  map 2-6
  module format 3-6 - 3-7, A-1
  module, link 8-23 - 8-24
  move data 8-97
  page 2-5
  pool 8-80
  request, bootstrap 8-76
memory (cont'd.)
  segment 2-8
memory allocation 2-5 - 2-7
memory block 2-7
  find 64K 8-85
  get byte 8-94
  get high 8-86
  map 8-81
  map, search 8-72
memory management 2-1, 2-5 - 2-12
  unit 2-7 - 2-8
menu
  manager, update request 8-148
  selection 8-122
message, write error 8-30
MMU registers 2-8
mnemonic name, LSN 5-2
MODPAK 8-135
module
  add into address space 8-104
  body 3-1 - 3-2
  clock 2-19
  CRC calculate 8-12
  decrease link count 8-42
  delete I/O module 8-90
  device descriptor 5-15
  device driver 4-8
  file manager 4-3
  finding 2-12
  format 3-1 - 3-3
  link 8-28
  link count, decrease 8-42
  linking 1-2
  load 8-25 - 8-26, 8-29
  load and execute primary 8-8 - 8-9
  name 3-3
  RBF-type device drivers 5-13 - 5-17
  SCF device descriptor 6-6 - 6-8

module (cont'd.)
types 3-1, 3-5
unlink 8-41
validate 8-111
module directory 2-5, 2-12
compact 8-88
entry, link using 8-83
find 8-84
get 8-19
pointer 8-84
module header 3-1 - 3-3, 5-15
SCF device driver 6-9
monitor, set type 8-146
mouse
button state 8-125
button timeout 8-140
click 8-122
coordinates 8-127
countdown 8-125
countup 8-125
parameters, set 8-147
port 8-125
resolution 8-126
screen position 8-126
send signal to process 8-141
status, get 8-123
timeout 8-124
window working area 8-127
move data 8-97
multiplexer 2-8
multiprogramming 2-12 - 2-16
management 2-1
multitasking 1-2

name parse 8-31 - 8-32
names, compare 8-10
next process 8-98
NMI interrupt 2-17
non-shareable file 5-12
number, path 8-53

open
file 8-48 - 8-49
path 8-57 - 8-58
operation of memory management 2-8 - 2-12
OS-9
Level One
compatibility 2-1
modules 1-2
scheduler 2-14 - 2-15
OS9P3 2-1
module 2-2

packet size 8-124
palette, get information 8-127
palette register 8-129
set default 8-149
settings 8-129
parameters, mouse and key repeat 8-147
parent
directory 5-3
process 2-13
parity 8-135
parse name 8-31 - 8-32
path
close 8-47, 8-135
duplicate 8-53
open 8-57 - 8-58
read bytes 8-59 - 8-60
table 4-2
path descriptor 4-18, 5-5 - 5-8
read option section 8-112
SCF 6-2 - 6-6
write option section 8-130
permanent storage size, module 3-7
physical address space 2-7
pipe file manager 4-3
PIPEMAN 1-2 - 1-3, 4-3
pipes 4-3, 7-1 - 7-3

process descriptor 2-13 - 2-14, 8-102
deallocate 8-80
descriptor, allocate 8-71
get 8-20
pointer 8-82
processes
active 2-12
data size, change 8-27
process ID 2-13
return caller's 8-22
pseudo vector 2-16
PutStat 4-6

RAM 2-5 - 2-7
allocate 8-69, 8-72
allocate blocks 8-70
allocation 2-13
blocks, deallocate 8-81
blocks, deallocate image 8-79
interrupt vector 2-18
random
access 5-1
block file manager 1-3, 4-3
RBF
change file size 8-131
format track 8-132
get file size 8-114
manager 4-3
tables 5-14 - 5-17
read
bytes 8-59 - 8-60
device operating parameters 5-23
disk sector 5-19
external memory 8-11
input character, SCF 6-13
line 6-2, 8-61
mode 5-11
system call 6-1
real-time clock 2-12, 2-17
record locking 5-10
reference
System Mode calls 8-5 - 8-6
User Mode system calls 8-3 - 8-4
registers
DAT 8-103
MMU 2-8
release a task 8-99
request system memory 8-105
reserved memory 2-5 - 2-7
reserve task number 8-100
return
64 bytes 8-101
system memory 8-106
RMA assembler 8-2
ROOT directory 5-3, 5-5
RTS instruction 2-18

SCF
configure serial port 8-134 - 8-135
data available test 8-113
device control registers 6-12
Getsta 6-15
manager 4-3
path descriptor 6-2 - 6-6
terminate device 6-16
scheduler, OS-9 2-14 - 2-15
screen
allocate high-resolution 8-142
convert type 8-145
display 8-143
free memory 8-144
mouse position 8-126
palette 8-127
size, get 8-119
type 8-128, 8-142, 8-145
scroll block, install 8-139
search bits 8-33

sector 5-3
 pattern, floppy diskette B-1
seek, file pointer 8-62
segment, memory 2-8
select graphics buffer 8-137
send signal 8-34
sequential character
 file manager 1-3, 4-3
 I/O 6-1
serial port configuration 8-121
service
 request processing 2-1
 routine, IRQ 5-25
set
 alarm 8-66
 date 8-38
 IRQ 8-92
 priority 8-36
 process DAT image 8-102
 process task DAT registers 8-103
 status 8-63
 SVC 8-107 - 8-108
 SWI 8-37
 time 8-38
 user ID 8-39
Setstats 5-23
Set Status system calls 8-130 - 8-150
shareable bit 3-5
sharing, file 5-12
shell 1-4
shutdown hard disk 8-133
signal 2-15 - 2-16
 codes 2-15
 fatal 2-13
 from mouse to process 8-141
 intercept trap 2-15 - 2-16
 intercept, set 8-21
 send to process 8-34
single-user
 attribute 5-12
 bit, files 5-12
size
 of screen 8-119
 of window 8-119
sleep
 calling process 8-35
sleeping process 2-14, 2-16
slices, time 2-12
sound, create 8-150
speaker, create sound 8-150
state
 active 2-13
 of button 8-126
 sleeping 2-14
 suspend 4-13
 waiting 2-13
static storage address 2-18
status
 display 8-115
 get, SCF 6-15
 get mouse 8-123 - 8-127
 of key 8-120
 register 2-17
 set, SCF 6-15
status, get 8-54
status, set 8-63
store byte in a task 8-109
string, scan input 8-31 - 8-32
strings, compare 8-10
subroutines
 RBF device driver 5-16 - 5-27
 SCF device drivers 6-10 - 6-17
suspend
 bit 4-13 - 4-14
 state 4-13
SWI, set 8-37
SWI2 instruction 2-4
symbolic names 2-4
sync byte 3-3
synonymous path number, return 8-53

system
- block map, get 8-18
- boot 1-3
- bootstrap 8-75
- date, get 8-40
- device, attach 8-44
- error codes C-1 - C-6
- initialization 2-1
- link 8-104
- mode call reference 8-5 - 8-6
- time, get 8-40

system call
- add 8-107 - 8-108
- descriptions 8-2, 2-4
- execution 8-1 - 8-2
- get status 8-112 - 8-130
- mnemonics names 8-1
- User Mode reference 8-3 - 8-4

system memory
- allocate high RAM 8-69
- block map 8-81
- deallocate 8-106
- module directory, get 8-19
- request 8-105

system modules 1-1 - 1-4

table
- device 8-52
- IRQ polling 2-18
- RBF 5-14 - 5-17
- SCF device descriptor 6-6 - 6-8
- VIRQ 2-20

task
- map 2-12
- offset, load from 8-93
- register 2-8
- release 8-99
- store byte 8-109

task number 8-73
- DAT 8-100
- deallocate 8-82

terminal, create sound 8-150

terminate
- a device 5-24
- calling process 8-14
- SCF device 6-16

ticks 4-11

time
- CPU 4-11
- get system 8-40
- set 8-38
- sharing 2-11
- slice 2-16, 2-12

timeout, mouse 8-124

track
- format 8-132
- restore drive head 8-131

trailer pattern, floppy diskette B-1

trap, signal intercept 2-15 - 2-16

type
- convert screen 8-145
- of screen 8-128
- set monitor 8-146
- window screen 8-142

unlink module 8-41 - 8-42

update mode 5-11

user calls 2-5

user ID 2-13
- set 8-39

User Mode system calls reference 8-3 - 8-4

validate module 8-111

VDG 1-2
- alpha screen cursor 8-118
- alpha screen memory 8-117
- interface 4-2

vector
- pseudo 2-16
- set SWI 8-37

vectoring 2-16

verify device attachment 8-44 - 8-45
video display generator 1-2
VIRQ 2-19 - 2-20
    polling table 2-19 - 2-20
virtual interrupt, install 8-110

wait
    calling process 8-43
    state 2-13 - 2-14
waiting process 2-13
wildcard 4-6
WINDINT 1-2
Windint interface 4-2
window
    descriptors 1-2
    high-level handler 8-139
    pointer location 8-124
    screen, type 8-142
    size, get 8-119
    type 8-145
    working area, mouse 8-127
working directory, change 8-46
write
    character to SCF output 6-14
    disk sector 5-21
    path descriptor 8-130 - 8-131
    to file or device 8-64
write line 8-65
    line system call 6-2

RADIO SHACK
A Division of Tandy Corporation
Fort Worth, Texas 76102

6/87-SWCG 874 8016 Printed in U.S.A.