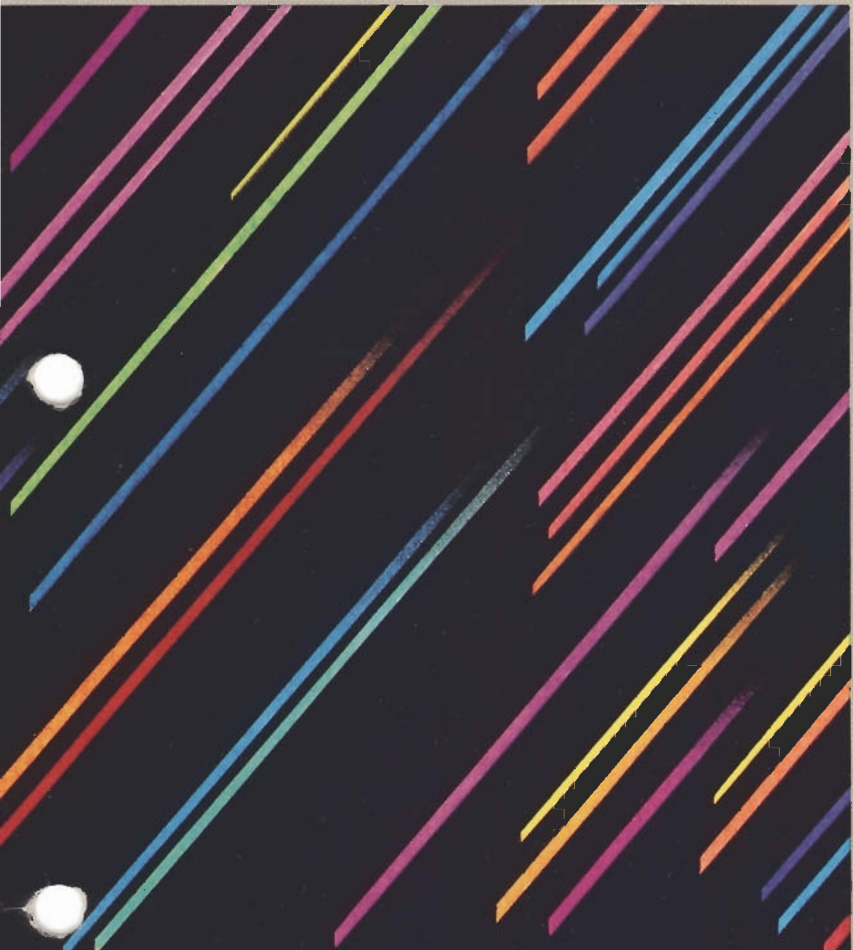
OS-9 Level Two Development System



TANDY®

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

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.

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.

Interactive Debugger

Contents

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

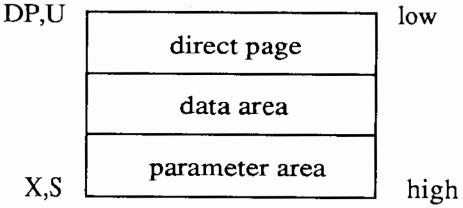
The ESTABLISH (E) command prepares Debug for testing a specific program module:

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 equ $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

Screen Editor

Contents

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)
- VDG (for VDG screen)
- ABM85
- KT7
- ANSI
- 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 Comm'and 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

Relocating Macro Assembler

Contents

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

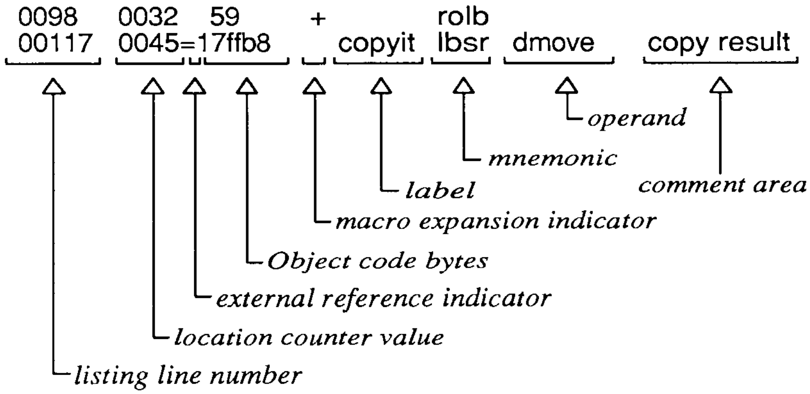
The operand field follows the operation field. You must separate the two fields by at least one space. Some instructions do not use the operand field; other instructions and assembler directives require an operand to specify an addressing mode, operand address, parameters, and so on.

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:



```
0098   0032  59      +        rolb
00117  0045=17ffb8     copyit lbsr dmove     copy result
```

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.l 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 equu    I               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     I               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     I               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 characters:

```
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

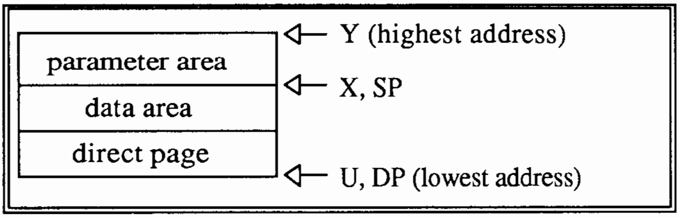
In general, the RMA assumes that the program will access data using indexed or direct page addressing modes. By convention, one index register contains the starting address of the data area, and the direct page register contains the page number of the lowest-address page of the data area. The RMA/RLINK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

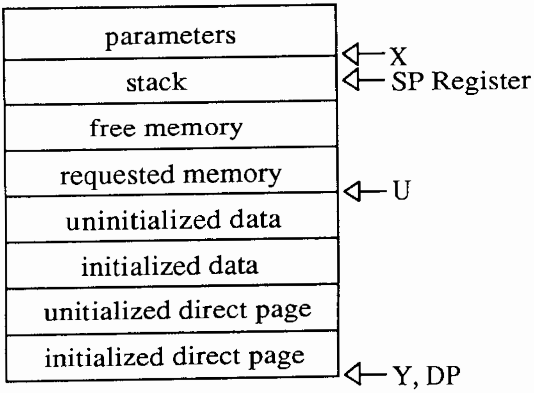
> **Note:** Because the memory addresses assigned to the program section and the address section are not a fixed distance apart, you cannot use program-counter relative addressing to obtain the address of objects in the data section.

Using Initialized Data

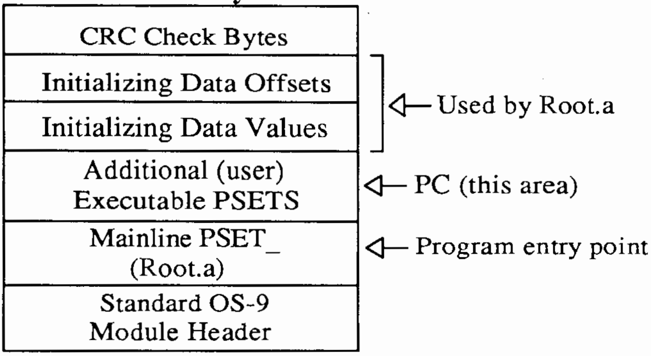
If you plan to use initialized data, you need to copy the data from the initialized data section in the object module to the data storage area pointed to by the Register U. Do so by using the Root.a mainline module (object code that is directly executable by using the OS-9 F$FORK). The function of the 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:

Process Data Area Layout



Process Object Code Module Layout



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

Utilities

Contents

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.

Chapter 4

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

Commands

Contents

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,/DO/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

RADIO SHACK
A Division of Tandy Corporation
Fort Worth, Texas 76102

6/87-SWCG 874-8016 Printed in U.S.A.