OS-9 Interactive Debugger

1 / Introduction

DEBUG is an interactive debugger that aids in diagnosing systems and testing 6809 machine-language programs. 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

DEBUG is supplied on your OS-9 system disk. When the screen shows the OS-9 prompt, call DEBUG by typing:

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 typing (ENTER). Correct a typing error by using the backspace (left arrow) key, or delete the entire line by typing (X) while pressing (CLEAR).

Each command starts with a single character, which may be followed by *text* or by one or two arithmetic *expressions*, depending on the command. You may use upper- or lower-case 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. Example:

In the calculator mode, obtain hexadecimal and decimal notation for the hexadecimal expression A+2:

You Type: (SPACEBAR) (A) (+) (2)
Screen Shows: DB: A+2
$000C #00012

Note: In the examples in this manual, general instructions are followed by specific typing instructions and then by what the screen shows. In some cases, examples will follow the explanation of more than one command. Be sure to execute these examples in the exact order in which they are given so that you will obtain the specified display on your screen.

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, which are the 6809's "native" arithmetic representation. The allowable range of numbers is 0 to 65535. Two's complement addition and subtraction is performed correctly, but will print out as large positive numbers in decimal form.

Some commands require byte values, and 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 they are automatically converted to 16-bit "words" without sign extension.

Spaces, other than a space at the beginning of a command, do not affect evaluation; 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 #. All other numbers are assumed to be hexadecimal and may have the prefix S. Examples:

Decimal	Hexadecimal	Binary
#100	64	%1100110
#255	FF	%11111111
#6000	1770	%1011101110000
#65535	FFFF	%1111111111111111

You may also use character constants. Use a single quote (') for 1-character constants and a double quote (") for 2- character constants. These produce the numerical value of the ASCII codes for the character(s) that follow. Examples:

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

Special Names

Dot (.) is 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 Registers MPU with a colon (:) followed by the mnemonic name of the register. Examples:

: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 activve. One-byte registers are promoted to a word when used in expressions.

Note: When a breakpoint interrupts a program, the Register SP points at the bottom of the Register MPU stack.

Operators

"Operators" specify arithmetic or logical operations to be performed within an expression. DEBUG executes operators in the following order: (1) −, negative numbers; (2) & and !, logical AND and OR; (3) * and /, multiplication and division; (4) + and −, addition and subtraction. Operators in a single expression having 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.

Examples:

<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 value of the word at Address Dot plus 10.

3 / Debug Commands

Calculator Commands

(SPACEBAR)<*expression*> (ENTER)

evaluates the expression and displays the results in both hexadecimal and decimal. Examples:

You Type: (SPACEBAR)5000+200 (ENTER)
Screen Shows: DB: 5000+200
$5200 #20992

You Type: (SPACEBAR)8800/2 (ENTER)
Screen Shows: DB: 8800/2
$4400 #17408

You Type: (SPACEBAR)#100+#12 (ENTER)
Screen Shows: DB: #100+#12
$0070 #00112

These commands also convert values from one representation to another. Examples:

Convert a binary expression to hexadecimal and decimal:
You Type: (SPACEBAR)%11110000 (ENTER)
Screen Shows: DB: %11110000
$00F0 #00240

Convert a 1-character constant to hexadecimal ASCII and decimal ASCII:
You Type: (SPACEBAR)'A (ENTER)
Screen Shows: DB: 'A
$0041 #00065

Convert a decimal expression to hexadecimal and decimal:
You Type: (SPACEBAR)#100 (ENTER)
Screen Shows: DB: #100
$00C4 #00100

You can also use indirect addressing to look at memory without changing Dot. Example:
You Type: (SPACEBAR). (ENTER)
Screen Shows: DB: .
$004F #00079

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: (SPACEBAR)`[:D+:Y]` (ENTER)
Screen Shows: `DB: [:D+:Y]`
`$0110 *00272`

Dot and Memory Examine and Change Commands

displays the current value of Dot (the current working memory address) and its contents. Example:

You Type: `.` (ENTER)
Screen Shows: `DB: .`
`2201 B0`

The present value of Dot is 2201, and B0 is the contents of memory location 2201.

(ENTER)

increments Dot and displays its new value and contents. Example:

"Step through" sequential memory locations:

You Type: (ENTER)
Screen Shows: `DB:`
`2202 05`
You Type: (ENTER)
Screen Shows: `DB:`
`2203 C2`
You Type: (ENTER)
Screen Shows: `DB:`
`2204 82`

-

backs up Dot one address and displays its value and contents. Example:

Display the current value of Dot:
You Type: . (ENTER)
Screen Shows: DB: .
2204 82

Back up one address and display its value and contents:
You Type: - (ENTER)
Screen Shows: DB: -
2203 C2

Back up another address and display its value and contents:
You Type: - (ENTER)
Screen Shows: DB: -
2202 05

. *expression*

changes the value of Dot. This command evaluates the specified *expression*, which becomes the new value for Dot. Example:

You Type: . 500 (ENTER)
Screen Shows: DB: . 500
0500 12

..

restores the last value of Dot. Example:

Display the current value of Dot and its contents:
You Type: . (ENTER)
Screen Shows: DB: .
1000 23

Change the value of Dot:
You Type: . 2000 (ENTER)
Screen Shows: DB: . 2000
2000 9C

Restore the last value of Dot:

You Type: . . ENTER

Screen Shows: DB: . . ENTER
1000 23

= *expression*

changes the contents of Dot. This command evalu[illegible] the ex-*pression* and stores the results at Dot. It then increme[illegible] and displays the next address and contents.

This command also checks Dot after the new value [illegible] to make sure it changed to the correct value. If it did not [illegible] shows an error message. This happens when you atte[illegible] non-RAM memory. In particular, the registers of [illegible] family interface devices (such as PIAs and ACIAs) [illegible] the same as when written to.

Example:

Display the current value of Dot and its contents:

You Type: . ENTER

Screen Shows: DB: .
2203 C2

Change the contents of Dot:

You Type: =FF ENTER

Screen Shows: DB: =FF
2204.01

Show that the contents of Dot have changed:

You Type: - ENTER

Screen Shows: DB: -
2203 FF

Warning: This command can change any memory [illegible]. You can destroy DEBUG, the program under test, [illegible] if you incorrectly change any of their memory areas.

Register Examine and Change Commands

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, which 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.

Note:

1. If you change the Register SP, you move your stack and the other register contents change.
2. 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.

: *register*

displays the contents of a specific *register*. The contents are in hexadecimal. Examples:

You Type: `:PC` (ENTER)
Screen Shows:
```
DB:  :PC
     C499
```

You Type: `:B` (ENTER)
Screen Shows:
```
DB:  :B
     007E
```

You Type: `:SP` (ENTER)
Screen Shows:
```
DB:  :SP
     42FD
```

:

displays all *registers* and their contents. Example:

You Type: : (ENTER)
Screen Shows: DB : :

```
PC=B265 A=01 B=0B CC=B0
  DP=0C
SP=0CF4 X=FF0D Y=000B
  U=00AE
```

:*<register>* *<expression>*

assigns a new value to a *register*. DEBUG evaluates the *expression* and stores it in the specified *register*. When you name 8-bit registers, the value of the *expression* must fit in a single byte. If it does not, the screen shows an error message, and the register does not change. Examples:

You Type: :X #4096 (ENTER)
Screen Shows: DB: :X #4096

Breakpoint Commands

The breakpoint capabilities of DEBUG let you specify addresses where you wish to suspend execution of the program under test and reenter DEBUG. When you encounter a breakpoint, the screen shows the values of the Registers MPU and the DB: prompt.

After a breakpoint is reached, you can examine or change registers, alter memory, and resume program execution. You may insert breakpoints at up to 12 addresses.

You can insert breakpoints by using 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 the SWIs operate by temporarily replacing an instruction OP code, there are three restrictions on their use:

1. You cannot use breakpoints in programs in ROM.
2. You must locate breakpoints in the first byte (OP code) of the instruction.
3. You cannot utilize 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 name*>. The screen shows the program's register contents.

B

displays all present breakpoint addresses.

B <*expression*>

inserts a breakpoint at a specified expression.

Examples:

Insert a breakpoint at the specified expression:
You Type: B 1C00 (ENTER)
Screen Shows: DB: B 1C00

Insert another breakpoint at the specified expression:
You Type: B 4FD3 (ENTER)
Screen Shows: DB: B 4FD3

Display the current value of Dot and its contents:
You Type: . (ENTER)
Screen Shows: DB: .
1277 39

Insert the breakpoints at Dot:
You Type: B . (ENTER)
Screen Shows: DB: B .

Display all present breakpoint addresses:

You Type: B (ENTER)

Screen Shows: DB: B
1C00 4FD3 1277

K

kills (removes) all breakpoints.

K *<expression>*

kills a breakpoint at the specified *expression*. Examples:

Display all present breakpoint addresses:

You Type: B (ENTER)

Screen Shows: DB: B
1C00 4FD3 1277

Kill a breakpoint at the address specified by the *expression*:

You Type: K 4FD3 (ENTER)

Screen Shows: DB: K 4FD3

Display all present breakpoint addresses:

You Type: B (ENTER)

Screen Shows: DB: B
1C00 1277

Kill all breakpoints:

You Type: K (ENTER)

Screen Shows: DB: K

Display all present breakpoint addresses:

You Type: B (ENTER)

Screen Shows: DB: B

Program Setup and Run Commands

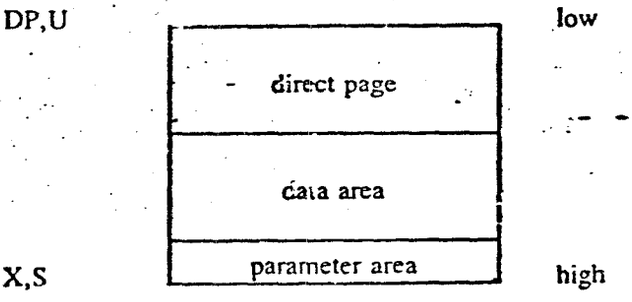
E ***module name***

prepares DEBUG for testing a specific program module.

This command's function is similar to that of the OS-9 Shell in starting a program. It 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: This 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 coroutines.

This 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 Registers MPU as if you had just performed an FSCHAIN service request as shown below:



D = Parameter area size
PC = Module entry point absolute address
CC = (F=0), (I=0) Interrupts disabled

Example:

Display the program's initial register values:

You Type: E *myprogram* (ENTER)

Screen Shows:
```
DB: E myprogram
   SP   CC   A   B DP
   X     Y     PC
 0CF3 C8 00 01 0C
 0CFF 0D00 9214
```

G

goes to (resumes) program execution after a breakpoint. If a breakpoint exists at the present program counter address, that breakpoint is not inserted so that it is not immediately reexecuted. A loop must contain at least two breakpoints if execution is to be suspended each time through the loop.

Note: The E command is usually used before the first G command to set up the program to be tested. DEBUG initially sets up a default stack; so G *expression* can be used to start a program using the results of the *expression* as a starting address. Examples:

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

L ***module name***

links to the module. If successful, it 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.

Example:

Link to the module FPMATH:

You Type: L FPMATH (ENTER)

Screen Shows:
```
DB: LFPMATH
          EC00 87
```

Utility Commands

C *<expression1> <expression2>*

performs a "walking bit" memory test and clears all memory between the two evaluated addresses. *Expression1* gives the starting address, and *expression2* gives the ending address, which must be higher. If any bytes fail the test, this command displays their address. Of course, you can test and clear only RAM memory.

Warning: This command can be dangerous. Be sure which memory address you are clearing.

Examples:

Clear all memory between Addresses 2000 and 15FF:

You Type: `C 15FF 2000 (ENTER)`
Screen Shows:
```
DB: C 15FF 2000
    17E4
    17E7
```

The screen's display of 17E4 and 17E7 indicates bad memory at those addresses.

Clear all memory between the last value of Dot and Address FF.

You Type: `C . .+FF (ENTER)`
Screen Shows:
```
DB: C . .+FF
```

The screen shows a blank line following the command line, which indicates good memory.

M *<expression1> <expression2>*

produces a screen-sized tabular display of memory contents in both hexadecimal and ASCII form.

The starting address of each line is on the left, followed by the contents of the subsequent memory locations. On the far right is the ASCII representation of the same memory locations.

Periods are substituted for nondisplayable characters. The high order bit is ignored for the display of the ASCII character.

S *<expression1> <expression2>*

searches an area of memory for a 1- or 2-byte pattern, beginning at the present Dot address. *Expression1* is the ending address of the search, and *expression2* is the data for which to search. If *expression2* is less than 256, a 1-byte comparison is used; if it is greater than 256, a 2-byte comparison is used. If a matching pattern is found, Dot is set to its address, which is displayed. If a matching pattern is not found, the screen shows the DB: prompt.

$ (ENTER)

calls the OS-9 Shell, which responds with prompts for one or more command lines.

$ *Shell Command*

executes the command and returns to DEBUG.

Also use $ to call the system utility programs and the Interactive Assembler from within DEBUG. Examples:

You Type: `$DIR` (ENTER)
Screen Shows:
```
DB: $DIR
            DIRECTORY OF . 00:00:21
OS9 BOOT      CMDS        SYS
     DEFS     STARTUP     OLDFILE
NEWFILE       BUSINESS    FILE1
```

Q

quits (leaves) DEBUG and returns to the OS-9 Shell. Example:

You Type: `Q` (ENTER)
Screen Shows:
```
DB: Q
            OS9:
```

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 below illustrates the use of DEBUG commands. This program prints "HELLO WORLD" and then waits for a line of input.

```
                         NAM                EXAMPLE
                         USE   /D0/DEFS/OS9DEFS
                    * Data Section
0000                     ORG   0
0000                LINLEN   RMB   2          LINE LENGTH
0002                INPBUF   RMB   80         LINE INPUT BUFFER
3052                         RMB   50         HARDWARE STACK
00E7                STACK    EQU   .-1
00E8                DATMEM   EQU   .          DATA AREA MEMORY SIZE

                    * Program Section
000  87CD0047               MOD   ENDPGM,NAME,$11,$81,ENTRY,DATMEM
000D 4558414D       NAME     FCS   /EXAMPLE/   MODULE NAME STRING

0014                ENTRY    EQU   *          MODULE ENTRY POINT
0014 308D0020               LEAX  OUTSTR,PCR  OUTPUT STRING ADDRESS
0018 108E000C               LDY   #STRLEN     GET STRING LENGTH
001C 8601                   LDA   #1          STANDARD OUTPUT PATH
001E 103F8C                 OS9   I$WRITLN    WRITE THE LINE
0021 2512                   BCS   ERROR       BRA IF ANY ERRORS
0023 3042                   LEAX  INPBUF,U    ADDR OF INPUT BUFFER
0025 108E0050               LDY   #80         MAX OF 80 CHARACTERS
0029 8600                   LDA   #0          STANDARD INPUT PATH
002B 103F8B                 OS9   I$READLN    READ THE LINE
002E 2505                   BCS   ERROR       BRA IF ANY I/O ERRORS
0030 09F00                  STY   LINLEN      SAVE THE LINE LENGTH
0033 C600                   LDB   #0          RETURN WITH NO ERRORS
0035 103F06         ERROR   OS9   F$EXIT      TERMINATE THE PROCESS
0038 48454C4C       OUTSTR   FCC   /HELLO WORLD/ OUTPUT STRING

0043 0D                      FCB   $0D        END OF LINE CHARACTER
000C                STRLEN   EQU   *-OUTSTR   STRING LENGTH
0044 268A06                  EMOD              END OF MODULE
0047                ENDPGM   EQU   *          END OF PROGRAM
                             END
```

A Session With DEBUG

The following example illustrates how to use DEBUG with the program on the previous page. (The actual RAM address may vary depending on your computer's installation of OS-9.)

```
OS9:DEBUG #2K

Interactive Debugger
DB: $I.OAD /D1/EXAMPLE
DB: L EXAMPLE
A900  87

DB:
        A900 87
DB:     M . .+44            (dump program on display)
A900  87CD0047000D1181 ...G....
A908  6F00140084455841 O....EXA
A910  4D504CC5308D0020 MPL.0..
A918  108E000C8601103F .......?
A920  8C25123042108E00 .%.0B...
A928  508600103F8B2505 P...?.%.
A930  109F00C600103F06 ......?.
A938  48454C4C4F20574F HELLO WO
A940  524C440DDB72DDFF RLD..R..

DB:     E EXAMPLE           (prepare to run program)
         SP   CC A  B  DP   X    Y    U    PC
        0DF3  C8 00 01 0D 0DFF 0E00 0D00 9214
DB:
        A900 87
DB:     B .+2E              (set breakpoint at address A92E)
DB:     G                   (run program)
HELLO WORLD
hello computer
(ENTER)

        BKPT:               (breakpoint encountered)
         SP   CC A  B  DP   X    Y    U    PC
        0DF3  C0 00 01 0D 0D02 0D00 0D00 922E
DB:     M :0  :U+20         (display register area)
0A00  00010D020000000C ........
0A08  0CF400004C000000 ....L...
```

The example that follows shows how the program on page 00 is "patched." In this case the LDY #80 instruction is changed to LDY #32.

```
OS9: DEBUG                  (call DEBUG)
Interactive Debugger
DB: SLOAD EXAMPLE           (call OS-9 to load program)
DB: L EXAMPLE               (set dot to beg addr of program)
    2000  87                (actual address will vary)
DB: . .+28                  (add offset of byte to change*)
    2028  50                (current value is 00)
DB  =#32                    (change to decimal 12)
    2028  10                (change confirmed)
DB: $SAVE TEMP EXAMPLE      (save on file called "TEMP")
DB: $VERIFY U <TEMP>        (update CRC and copy to "NEWEX")
    NEWEX
DB: $ATTR NEWEX E PE        (set execution status)
DB: $DEL TEMP               (delete temporary file)
DB: q                       (exit DEBUG)
```

Patching OS-9 Component Modules

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

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

```
OS9: DEBUG (ENTER)          (call DEBUG)
Interactive Debugger
DB: L TERM (ENTER)          (set dot to addr of TERM module)
    CA82  87                (actual address will vary)
DB: . . +13 (ENTER)         (add offset of byte to change*)
    CA95 01                 (current value is 01)
DB: =1 (ENTER)              (change value to 01 for "OFF")
    CA96 01
DB: - (ENTER)               (move back one byte)
CA95  00                    (change confirmed)
DB: Q (ENTER)               (exit DEBUG)
OS9: COBBLER /D1 (ENTER)    (write new bootfile on /D1)
```

```
OS9: VERIFY </D1/OS9BOOT >/D01/TEMP U ENTER
                                    (update CRC value)
OS9: DEL /D1/OS9BOOT ENTER  (delete old boot file)
OS9: COPY /D01/TEMP/D1/OS9BOOT
                                    (install updated boot file)
```

Then you can use the Dsave command to build a new systems disk.

Appendix / Debug Command Summary

(SPACEBAR) *expression*	Evaluate; display in hexadecimal and decimal.

Dot Commands

.	Display Dot address and contents.
..	Restore last DOT, display address and contents.
. *expression*	Set Dot to result, display address and contents.
= *expression*	Set memory at Dot to result.
–	Decrement Dot, display address and contents.
(ENTER)	Increment Dot, display address and contents.

Register Commands

:	Display all register contents.
:*register*	Display specific register contents.
:*register* *expression*	Set register to result.

Program Setup and Run Commands

E *module name*	Prepare for execution.
G	Go to program.
G *expression*	Go to program at result address.
L *module name*	Link to module named, display address.

Breakpoint Commands

B	Display all breakpoints.
B *expression*	Set breakpoint at result address.
K	Kill all breakpoints.
K *expression*	Kill breakpoint at result address.

Utility Commands

M *expression1*	Display memory dump in tabular form. *expression2*
C *expression1* *expression2*	Clear and test memory
S *expression1* *expression2*	Search memory for pattern
$ *text*	Call OS-9 Shell
Q	Quit (exit)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 below. Error codes other than those listed are standard OS-9 error codes returned by various system calls.

0 ILLEGAL CONSTANT: The expression included a constant that had an illegal character or that was greater than 65,535.

1 DIVIDE BY ZERO: A division was attempted using a divisor of zero.

2 MULTIPLICATION OVERFLOW: The product of the multiplication was greater than 65,535.

3 OPERAND MISSING: An operator was not followed by a legal operand.

4 RIGHT PARENTHESIS MISSING: Parentheses were misnested.

5 RIGHT BRACKET MISSING: Brackets were misnested.

6 RIGHT ANGLE BRACKET MISSING: A byte-indirect was misnested.

7 INCORRECT REGISTER: A misspelled, missing, or illegal register name followed the colon.

8 BYTE OVERFLOW: An attempt was made to store a value greater than 255 in a byte-sized destination.

9 COMMAND ERROR: A command was misspelled, missing, or illegal.

10 NO CHANGE: The memory location did not match the value assigned to it.

11 BREAKPOINT TABLE FULL: The maximum number of 12 breakpoints already exist.

12 BREAKPOINT NOT FOUND: No breakpoint exists at the address given.

13 ILLEGAL SWI: An SWI instruction was encountered in the user program at an address other than a breakpoint.

TEXT EDITOR INDEX

ASSEMBLER INDEX