OS-9 Editor/Assembler/ Debugger Manual

OS-9 Editor/Assembler/Debugger Manual

Chapter 1. Text Editor

1.1. Introduction to the Macro Text Editor

The Macro Text Editor is a powerful and easy to use text preparation system. It is commonly used to create source programs or other kinds of text files used within the OS-9 system. The editor has many features that make editing faster and more convenient. For example, most commands involve only one or two keystrokes.

The “Macro” part of the editor's name refers to the its macro facility which allows you to create new personalized commands or complex commands for special purposes from the basic built-in command set.

Because the editor has its own variables and loops it can be used as a kind of text-oriented programming language, which is especially useful for large software conversion problems.

The editor can also edit several different files at the same time, and even copy text from one file to another.

1.2. Getting Started

The editor is kept in a file called “edit”, which should be present in your system's CMDS (execution) directory. To run the editor type:

```
OS9: edit RETURN
```

The editor should load and start. When it prints the “E: ” prompt, it is ready to accept a command. The first command to learn is how to quit editing (return to OS-9). To quit, type a “Q” followed by a carriage return as follows:

```
E: q RETURN
```

Now that you can get in and out of the editor, it is time to learn a little about some basic edit commands which are discussed in the sections on Display, Edit Pointer Manipulation, and Insertion/Deletion. After you have mastered the basic commands, you should move on to the more advanced commands. Good Luck!

1.3. The Edit Command Line

When you run EDIT, you can optionally specify initial input and output file(s). Although you may open additional files after the editor has been started, there are several commands which treat the initial files as special cases because it is assumed that these files (if specified) are the main working files.

You may also want to use the OS-9 shell “#” memory size option to give the editor a bigger memory space, which will let you edit bigger sections of a file at once. If you don't, the default size is 4K bytes. For example:

```
OS9: EDIT myfile #24k RETURN
```

Some of the different forms of the edit command line are illustrated below. The name “newfile” refers to the name of a new file you wish to create. The name “oldfile” refers to the name of an existing file you wish the editor to use.

```
EDIT
```

There will be no initial input or output files. Text file operations may be performed by opening files after the editor has started running.

```
EDIT newfile
```

The editor will create a new file called “newfile” which will be the initial output file.V There will be no initial input file. However, read operations may be performed by opening files after the editor has started running.

```
EDIT oldfile
```

The initial input file is “oldfile”. The editor will create a new temporary file called “SCRATCH” that will be the initial output file. when the edit session is done, “oldfile” will be deleted, and then “SCRATCH” will be renamed to the name “oldfile” to give the appearance of “oldfile” simply being updated. Note: The two OS-9 utilities “**DEL**” and “**RENAME**” must be present on your system if you wish to use this method to start the editor.

```
EDIT oldfile newfile
```

The initial input file is “oldfile”. The editor will create a new file called “newfile”, which will be the initial output file.

1.4. Edit Buffers

The text being edited is stored in a memory space called an “edit buffer”. Edit buffers may be thought of as scratch pads used for saving the text to be manipulated by the editor.

There is always at least one edit buffer, but you can create several at the same time if you wish. The buffer currently being used is called the “primary buffer”, and the last most recently used buffer is called the “secondary buffer”. Normally, the primary buffer will simply be referred to as the “buffer” or “edit buffer” for short, except where it is explicitly called the primary buffer. The secondary buffer is important only when you wish to use a command that involves moving text from one buffer to another.

1.5. Edit Pointers

The macro text editor uses what is called an “edit pointer” as a place holder to remember your current position in the text buffer. Many commands operate on the text at the current edit pointer position. This is similar to a person using his finger as a place holder when reading a newspaper. Certain commands may be used to reposition the edit pointer, or show the text that it points to, etc. Each buffer has its own edit pointer to allow you to move from buffer to buffer without losing your place in any of them.

1.6. Entering Commands

Whenever the editor prints the “E:” prompt, it is ready for you to enter a command line. You can type one or more edit commands on a single line followed by RETURN. Some edit commands have “parameters” which are values used by the command.

Multiple commands on a single line can be optionally separated by spaces, but you should not start a command line with a space because the space character is itself a command (insert line).

Many of the commands are single characters designed for rapid entry, for example “L” and “D”. Some other command are function keys such as SPACE and RETURN which only apply if they are the first character of the line. Yet other commands are “built-in macros”, which are descriptive names starting with a “.”, such as “.size” and “.shell”. In general, all three kinds work the same way despite the differences in form.

If you make a mistake while typing, it may be corrected before the mistake reaches the editor by using the backspace key, or by deleting the entire line using the CONTROL+X key (of course editor commands can also be used to fix mistakes!). The usual OS-9 control characters work in the editor, see the *OS-9 Operating System User's Guide*. A summary is given below:

CONTROL+A	Repeat previous input line.
CONTROL+C	Interrupt whatever the editor is doing and return to command entry mode.
CONTROL+D	Redisplay present input line.
CONTROL+H	Backspace.
CONTROL+Q	Same as CONTROL+C.
CONTROL+W	This control key will temporarily halt output to your terminal so that you can read the screen before the data scrolls off. Output is resumed when you type any other key.
CONTROL+X	Line delete.
ESCAPE	Quit editor - same as “Q” command.

1.7. Parameters

Many of the editor's commands allow you to specify some value which may represent such things as the number of times to repeat a command or a phrase you wish to find, etc. There are two types of edit parameters, “numeric” and “string”.

1.7.1. Numeric Parameters

Numeric parameters are used when you wish to specify an amount, such as the number of times to repeat a command or the number of text lines that a command is be applied. The four methods in which a numeric parameter may be specified are:

1. The parameter can be omitted. In this case, a value of 1 is assumed. The example below lists one line:

   ```
   L
   ```

2. The parameter can be a constant number from 0 to 65535. The example below shows the “L” (list lines) command with a parameter of “10”.

   ```
   L10
   ```

3. The parameter can be “*” which means repeat the command as many times as possible. The example below means “list all lines”.

   ```
   L*
   ```

4. The parameter can be a numeric variable, which is a “#” followed by the letters A-Z, to be used in conjunction with macros and is explained in detail in the “Edit Macro” section.

1.7.2. String Parameters

Strings are used to specify a single character, word, phrase, or any other arbitrary group of characters. There are two methods which You may use to specify strings.

The first method is to enclose the text between a pair of “delimiter” characters. The delimiters are not part of the text. Any characters can be used for delimiters, but punctuation characters chosen so they are not included in the actual text are most commonly used. For example, to tell the editor to search for the phrase “fast code” the following command is used:

```
S/fast code/
```

Unlike commands, a distinction is always made between upper case and lower case strings. The example on the previous page would *not* find “FAST CODE”.

1.7.3. Multiple Parameters

When a command requires two to more parameters, they are given in the correct order immediately following the command name. You do not have to type spaces or other characters between the parameters. For example, the command to search for the next two occurrences of the phrase “compiler” is:

```
S2/compiler/
```

If two string parameters are required by a command, three delimiters are used: one at the beginning, one between the first string and the second, and one at the end of the second string. For example, the command to change the phrase “my cat” to “my dog” is:

```
C,my cat,my dog,
```

The second method uses “string variables”, which consist of a “$” followed by a variable name of A-Z. These are most commonly used in macros and are discussed in detail in the “Edit Macro” section.

1.8. Syntax Notation

This manual uses syntax descriptions to help you understand how to enter each command. They describe what you should enter and the order that you should do it in. The first thing in a syntax description is the command name. The name should be typed in exactly as given (except lower case and upper case command names are interchangeable).

The command name is followed by the type of parameters that the command expects. Each of them should be entered as described in the section on parameters. The syntax descriptions for each command use the following definitions:

n	numeric parameter
str	string parameter
SPACE	space character
RETURN	carriage return or “enter” key
text	one or more characters terminated by a RETURN

1.9. Basic Commands

This section describes the basic commands used to manipulate text and control the editor. If you plan to only occasionally use Edit with simple single-buffer editing, these command are all you need to know.

For basic editing, you need only learn the commands listed below, This editor is actually a proper superset of the Basic09 built-in editor, so if you know Basic09; you already know how to use Edit effectively!

+	Move Forward Line(s)
-	Move Backward Line(s)
SPACE	Insert Line
RETURN	Move Forward One Line And Display
C	Change String
L	List Line(s)
S	Search For String
Q	Quit Editing

You will notice that some commands work on entire lines (such as "+"), and others work on characters within lines (such as ">"). We also recommend that beginners master line-oriented commands before moving on to character-oriented commands.

1.9.1. Change String

SYNTAX:	C *n* `str1` `str2`
	.CHANGE *n* `str1` `str2`
FUNCTION:	Change the next *n* occurrence of `str1` to `str2`
MODE:	Line-Oriented

This command is used to change the next *n* occurrences of "`str1`" to "`str2`". Starting at the current edit pointer position; wherever "`str1`" is found, it is changed to "`str2`", and then the updated line is displayed. Changes continue until no more occurrences of "`str1`" are found or until the count "*n*" is reached, whichever occurs first.

The "C" command leaves the edit pointer positioned at pointing to the beginning of the last line changed. The ".CHANGE" built-in macro will leave the edit pointer positioned just past the modified string. If "`str1`" is not found, the edit pointer will not be affected. Some typical examples of its use are:

```
C/this/that/                          .CHANGE/this/that/
C2/in/out/                            .CHANGE 2 /in/out/
C*! seek and find ! seek and found !
```

The first example changes the next occurrence of "this" to "that". The second example changes the next two occurrence of "in" to "out". The last example changes all occurrences of " seek and find " to " seek and found " that are between the edit pointer and end of text.

1.9.2. Delete Characters

SYNTAX:	K *n*
FUNCTION:	Delete (Kill) *n* characters.
MODE:	Character-Oriented

This command is used to erase *n* characters starting at the current edit position; all deleted characters are displayed. Some examples are:

```
K               K4
```

The first command deletes the character at the current edit position. The second command deletes the character at the current edit position and the next three characters.

1.9.3. Delete Lines

SYNTAX:	D *n*
FUNCTION:	Delete *n* lines.

MODE: Line-Oriented

This command is used to delete (erase) *n* entire lines of text starting with the current line, regardless of the edit pointer position in the line. The deleted lines are displayed. Some examples are:

D D4 D*

The first example deletes the current line and displays it. The second example deletes the current line and the next three lines. The last example deletes all the lines from the current line to the end of_the edit buffer.

1.9.4. Extend Lines

SYNTAX:	E `n str`
FUNCTION:	Extend *n* lines with string.
MODE:	Line-Oriented

This command is used to extend (add to the end of) *n* lines with the string given. After each line is extended, the line is displayed and the edit pointer is moved past it. Below are some examples of how it may be used:

E/this is a comment/ E3/XX/

The first example would add the string “this is a comment” to the end of the current line and move the edit pointer to the next line. The second example would add the string “XX” to the end of the current line and the next two lines; the edit pointer would be moved past these lines.

1.9.5. Unextend Line

SYNTAX:	U
FUNCTION:	Unextend (truncate) line
MODE:	Character-Oriented

This command is used to unextend (truncate) a line at the current edit position. The characters in the line from the current edit pointer position to the end of the line are chopped off. The “>” or “<” commands (see Section 1.9.12, “Move Characters Backward”) are usually used to position the pointer prior to use of this command. For example:

U

1.9.6. Insert Line

SYNTAX:	SPACE `text`
FUNCTION:	Insert the line of text before edit position
MODE:	Line-Oriented

This command is used to insert complete text lines. The lines are inserted before the current edit position, and the edit pointer will be positioned following the newly entered line. Therefore, many lines can be automatically entered in order. The new lines will be inserted *before* the line pointed to by the edit pointer prior to entry of the new lines. For example:

```
E: SPACEthis is line one. RETURN
E: SPACEthis is line two. RETURN
E: SPACEthis is line three. RETURN
E: ^ RETURN
E: L* RETURN
This is line one.
This is line two.
This is line three.
E:
```

1.9.7. Insert String

SYNTAX:	I *n str*
FUNCTION:	Insert a line containing *n* copies of `str` string
MODE:	Line-Oriented

This command is used to insert a line made up of *n* copies of the string, which is inserted before the edit pointer. Then the edit pointer is not changed. For example, if you wanted to insert a line containing eighty asterisks you would enter the following command:

```
I80/*/
```

Note that this command always inserts a RETURN after the text. While this command is very similar to the SPACE (insert line) command, it is provided because the SPACE command cannot be used within macros.

1.9.8. List Following Lines

SYNTAX:	L *n*
FUNCTION:	List the next *n* lines of text
MODE:	Line-Oriented

This command displays *n* lines of text starting at the current edit position. The edit position is not changed. For example, the following command will cause the editor to display the current line of text:

```
L
```

If you wish to display the three lines, enter the command line given below:

```
L3
```

If the edit pointer is not at the beginning of the first line, only that part of the line from the edit pointer to end of line will be displayed. To see all the text from the current edit position to the end of the buffer, use an asterisk for the value as in the following command line:

```
L*
```

The “L” command is not affected by “verify” mode;

1.9.9. List Previous Lines

SYNTAX:	X *n*
FUNCTION:	Display previous lines of text
MODE:	Line-Oriented

This command is identical to the L command except it list lines *before* the current edit pointer position.

1.9.10. Memory Size Adjust

SYNTAX:	M *n*
FUNCTION:	Adjust the workspace size to *n* bytes.
MODE:	Directive to Editor

This command is used to adjust the workspace size (total amount of memory available for buffers and macros). If the workspace becomes full and the editor prevents you from entering more text, you may overcome the problem by increasing the workspace size. If you will not be using a large portion of the available workspace, you may wish to decrease the workspace size so that other OS-9 programs may use the memory that you free. Below are some examples of how the “M” command is used:

M5000 M10000

The first example sets the workspace size to 5000 bytes. The second example sets the workspace size to 10000 bytes.

Before using the “Q” command to quit editing, you may want to make the workspace size larger to decrease the amount of time needed to copy the input file to the output file allowing the editor to read and write a larger portion of the file at one time. Note that memory is allocated in 256 byte pages. For the “M” command to have any effect, the desired workspace size must differ from the current size by at least 256 bytes. The “M” command will not allow you to return any part of the workspace which is being used for buffers or macros.

1.9.11. Memory Size Display

SYNTAX: .SIZE
FUNCTION: Display workspace size.
MODE: Directive to Editor

This command is used to display the size of the workspace and the amount that has been used, An example of how this command would be used is:

```
E: .SIZE RETURN
  521 15328
```

in the example above, the numbers that are printed below the “.SIZE” command are “521” which is the amount of the workspace that has been used for buffers and macros, “15328” is the total amount of memory available in the workspace.

1.9.12. Move Characters Backward

SYNTAX: < *n*
FUNCTION: Move backwards *n* characters.
MODE: Character-Oriented

This command is used to move the edit pointer backwards (toward the beginning of the text) *n* characters. It is typically used when moving the edit pointer to some position in a line other than the first character. Here are examples:

< <10

The first command line moves the edit pointer back one character. The second command line moves the edit pointer back ten characters.

1.9.13. Move Characters Forward

SYNTAX: > *n*
FUNCTION: Move forward *n* characters.
MODE: Character-Oriented

This command is used to move the edit pointer forward (toward the end of the text) *n* characters. It is typically used to move the edit pointer to some position in the line other than the first character. Here are examples:

> >25 >*

The first command line moves the edit pointer forward (to the right) one character. The second command line moves the edit pointer forward twenty five characters. The last command line move the edit pointer to the end of the buffer.

1.9.14. Move To End Of Text

SYNTAX:	/
FUNCTION:	Move to end of text.
MODE:	Line- or Character-Oriented

This command moves the edit pointer past the last character of the last line in the buffer. Note that -* is identical in function to ^.

1.9.15. Move To Next Line And Display

SYNTAX:	RETURN
FUNCTION:	Move to next line and show it.
MODE:	Line-Oriented

This command moves the edit pointer to the beginning of the next line and displays it. Note that the only character you should type to enter this command is the RETURN-key. This command is after used to step through the text one line at a time. For example:

```
E:L3 RETURN        list next three lines
  This is line 1
  This is line 2
  This is line 3
E:RETURN
  This is line 2
E:RETURN
  This is line 3
```

1.9.16. Move To Start Of Text

SYNTAX:	^
FUNCTION:	Move to beginning of text.
MODE:	Line and Character-Oriented

This command moves the edit pointer to the beginning (top) character of the first line in the edit buffer.

1.9.17. Move Lines Backwards

SYNTAX:	-*n*
FUNCTION:	If *n* => 1, go backward *n* lines and display the line. If *n* = 0, go to the beginning of the line.
MODE:	Line-Oriented

The “-” command has two uses. If a number of one or more is given, the edit pointer is moved backward that number of lines, and the new line pointed to will be displayed. For example, the following command will move the edit pointer backward 5 lines:

```
-5
```

If the “-” command is given with a number of zero, it moves the edit pointer past to the first character of the current line. This can be useful when you wish to repeat a command within the current line. For example:

```
-0
```

1.9.18. Move Lines Forward

SYNTAX:	+ *n*
FUNCTION:	If *n* => 1, go forward *n* lines and display the line.
	If *n* = 0, go to the end of the line.
MODE:	Line-Oriented

The "+" command has two uses. If a number of one or more is given, the edit pointer is moved forward that number of lines, and the new line pointed to will be displayed. For example, the following command will move the edit pointer forward 5 lines:

```
+5
```

If the "+" command is given with a number of zero, it moves the edit pointer past to the last character of the current line. This can be useful when you wish to append text to the current line. For example:

```
+0
```

Also note that "+*" is identical in function to "/".

1.9.19. Quit Editor

SYNTAX:	Q
FUNCTION:	Quit Edit Program
MODE:	Directive to Editor

This command is used to quit editing and return to the OS-9 Shell (or the program that called Edit). For example:

```
Q
```

If input and/or output file(s) were specified on the OS-9 command line when you started the editor, the text in buffer number one would be written to the initial output file, then the remainder of the initial input file will be copied to the output file. After the text has been copied, the editor will be terminated and control returned to the Shell.

1.9.20. Search For String

SYNTAX:	S *n str*
MODE:	Line-Oriented
SYNTAX:	.SEARCH *n str*
MODE:	Character-Oriented
FUNCTION:	Search for the next *n* occurrence of *str*

This command is used to search for the next *n* occurrences of the string specified, starting at the current edit pointer position. When a line containing the string is found, the line is displayed.

If the string is found, the edit pointer will be positioned at the beginning of the last line in which the string was found. If no occurrence of the string was found, the edit pointer position will be unchanged.

The ".SEARCH" built-in macro is similar to "S" except that it leaves the edit pointer just past the occurrence of the string. Some typical examples of its use are:

S/my string/	.SEARCH/my string/

S3"strung out"	.SEARCH 3"strung out"
S*/seek and find/	.SEARCH*/seek and find/

The first example searches for the next occurrence of “my string”. The second example searches for the next three occurrence of “strung out”. The last example searches for all occurrence of “seek and find” that are between the edit pointer and the end of text.

1.9.21. Set Anchor Column

SYNTAX:	A *n*
FUNCTION:	Set the SEARCH/CHANGE anchor to column number *n*.
MODE:	Character-Oriented

This command is used to set the SEARCH/CHANGE anchor to column number *n*. After the anchor has been set, the “S” and “C” commands will find a string only if it begins in column number *n*. For example, if you want to find a string that you know begins in column number one (such as an assembly language label), but don't want to find it if it begins in any other column, you should set the anchor to column one before using the search command to find it to allow you to skip any occurrence of the string that do not start in column one. Some typical examples of its use are:

A A50

The first example would cause SEARCH/CHANGE to find a string only if it began in column number one. The second example would cause SEARCH/CHANGE to find a string only if it began in column number fifty.

To return to the normal mode of searching so that a string may be found regardless of the column that it begins in, the anchor should be set to zero. For example:

```
A0
```

If you use the “A” command to set the anchor, this remains in effect only for the Current command line. After the command line is executed the anchor will automatically return to its normal value of zero.

1.9.22. Shell Command

SYNTAX:	.SHELL *text*
FUNCTION:	Call the OS-9 SHELL to execute the text line.
MODE:	Directive to Editor

This command allows you to use any of the OS-9 commands from within the editor. The remainder of the command line following the “.SHELL” command is passed to the OS-9 shell for execution. Some examples of how this command may be used are:

```
E: .SHELL dir /D1 RETURN

E: .SHELL asm prog.src l o=prog >/p& RETURN
```

Notice that the second example starts the assembler as a background task.

1.9.23. Tab

SYNTAX:	T *n*
FUNCTION:	Tab to character position *n*.
MODE:	Character-Oriented

This command is used to tab (move the edit pointer) to the character position “*n*” of the current line. If “*n*” exceeds the line length, the line will be extended with spaces. Some examples of usage are:

T T5

The first example would move the edit pointer to the first column of the current line. The second example would move the edit pointer to the fifth column of the current line.

1.9.24. Verify On/Off

SYNTAX:	V *n*
FUNCTION:	Turn verify ON / OFF
MODE:	Directive to Editor

This command is used to turn the verify mode on or off. If the verify mode is turned on (the default), many edit commands will display their results which can be annoying and/or time consuming when certain commands are used repetitively (such as “c”, and “d”). If you turn verify off, it may be turned on again by specifying a non-zero value for *n*. For example:

V V20

Either of the two examples above would turn the verify mode on. To turn the verify mode off use the following command:

```
V0
```

Macros inherit the current verify mode, but if a macro changes the mode the change will only apply within the macro call.

1.10. Advanced Commands

This section discusses the editor commands that are available for more sophisticated editing tasks. The advanced commands can be divided into four general categories:

File Manipulation Commands

These commands allow files to be opened, created; closed, read from, and written to. The editor can work with several files at the same time.

Buffer Manipulation Commands

These commands let you create multiple edit buffers, switch between them, and copy lines or blocks of text from buffer to buffer.

Loops and Conditional commands

These commands allow sequences of edit commands to be executed repetitively, and to apply various conditional tests. Using these functions you can create editor “programs”.

Macros

These commands allow sequences of edit commands (including loops and conditionals) to be stored, edited, saved, loaded and executed.

1.11. File Manipulation Commands

1.11.1. New

SYNTAX: .NEW

This command is used when the file being edited is too large to fit into the editor's workspace at one time.

All lines of text before the current edit pointer position are written to the output file. The editor will then try to read new text lines from the input file, which are appended to the end of the edit buffer. The editor attempts to read as many new lines as were written out.; The “NEW” command always uses the initial input and output files (i.e., the files specified in the command line used to run the editor).

If you have finished editing the text currently in the edit buffer, you may “flush” it out and refill the buffer with new text by moving the edit pointer to the bottom of the edit buffer and then use the “.NEW” command as follows:

```
E:/.NEW RETURN
```

If you wish to retain part of the text that is already in the edit buffer,- move the edit pointer to the first line that you wish to retain, before using the “.NEW” command.

1.11.2. Open Input File

SYNTAX: .READ `str`

This command is used to open an OS-9 text file for reading, or to close the file after it is no longer needed. The file opened replaces the original input file specified on the command line used to call the editor.

To open a file, “`str`” is used to specify the OS-9 file name, for example:

```
E:.READ "myfile" RETURN
```

To close an input file, the .READ command is used with an empty string as shown below. Closing a file previously opened using the .READ command also restores the original input file.

```
E:.READ "" RETURN
```

When a file is opened it remains attached to the current primary buffer to allow each buffer to have its own independent input file. These files may be read by switching to the proper buffer, then using the “R” command to read from that buffer's input file. When closing a file, you must select the same primary that the file was opened with.

1.11.3. Create Output File

SYNTAX: .WRITE `str`

This command is used to create a new output file or to close the file after it is no longer needed. The file opened replaces the original output file specified on the command line used to call the editor.

To create an output file, “`str`” gives the OS-9 file name, for example;

```
E:.WRITE "myfile" RETURN
```

To close an output file, the .WRITE command is used with an empty string as shown below. Closing a file previously opened using the .WRITE command also restores the original output file.

```
E:.READ "" RETURN
```

When a file is created it remains attached to the current primary buffer so buffer can have its own independent output file. These files may be written to by switching to the proper buffer, then using the “W” command to write to the buffer's output file. When closing a file, you must select the same primary buffer that the file was opened with.

1.11.4. Read From Input File

SYNTAX: R *n*

This command is used to read "*n*" lines of text from the buffer's input file. The lines read in are displayed and inserted before the current edit position. For example:

R10 R*

The first example reads ten lines of text from the input file, and the second example reads all remaining lines. If there is no more text in a file, the "*END OF FILE*" warning message will be displayed.

1.11.5. Write To Output File

SYNTAX: W *n*

This command is used to write up to "*n*" lines of text from the buffer-to its output file. The lines are written starting at the current edit position. For example:

W10 W*

The first example writes ten lines and the second writes all remaining lines of text.

1.12. Buffer Manipulation Commands

1.12.1. Display Buffer And Macro Directory

SYNTAX: .DIR

This command is used to display the directory of the editors buffers and macros. For example:

```
E:.DIR RETURN

BUFFERS:
$       0
*       1
       50

MACROS:
  MYMACRO
  INDENT
```

Under the heading BUFFERS is a list of all current edit buffers. The current primary buffer is marked with an asterisk (Buffer #1 in the example above). The current secondary buffer is marked with a dollar sign (Buffer #0 in the example above). All other buffers are listed but otherwise unmarked (such as Buffer #50 above).

Under the heading MACROS is a the list all current macros. The example above shows the macros MYMACRO and INDENT.

1.12.2. Change and/or Create Primary Edit Buffer

SYNTAX: B *n*

This command is used to make buffer number "*n*" the primary edit buffer. The previous primary buffer becomes the new secondary buffer. If you specify the number of a buffer that does not already exist, a new buffer will be created and assigned the buffer number. For example, assume the current primary buffer is 1, then the following command is given.

B5
The example above makes buffer 5 the new primary edit buffer, and buffer 1 would become the new secondary buffer.

1.12.3. Move Lines From Primary Buffer

SYNTAX: P *n*

This command is used to move "*n*" lines of text from the primary buffer to the secondary buffer. The lines are removed from the primary edit buffer starting at its edit position and inserted into-the secondary buffer (before its edit position. The text moved is displayed. Some examples of the use of this command are:

```
P                P5               P*
```

The first example moves one line of text, the second example moves five lines of text, and the last example moves all lines that are between the current edit position and end of text.

This command is often used in combination with the "G" command below to perform block transfers of text within a buffer. The example below illustrates the block move operation.

1. Move edit pointer to first line to be moved.

```
E:S/First line/ RETURN
  First line to be moved
```

2. Put lines in secondary buffer (in this case, 2 lines).

```
E:P2 RETURN
```

3. Move edit pointer to where text is to be reinserted.

```
E:+20 RETURN
```

4. Get lines back from secondary buffer

```
E:G2 RETURN
  First line to be moved
  Second line to be moved
```

1.12.4. Move Lines To Primary Buffer

SYNTAX: G *n*

This command is the reverse of the P command. Text lines are taken from the top (beginning) of the secondary buffer and inserted into the primary buffer before its current edit position.

1.13. Looping And Conditionals

An important advanced feature of the editor is its ability_for looping and conditional tests allow sequences of commands to be repeated a number of times until a certain condition is met. For example, you can repeat a sequence of commands until a certain string is encountered in the text.

The looping and conditional features, especially when combined with the macro functions discussed in the next section, give the editor many characteristics of a programming language.

1.13.1. Looping

A loop is constructed by enclosing one or more commands in square brackets. A loop count value (which can be a number or “*”) specifies how many times the command within the loop are to be repeated. If any of the commands fails (such as an unsuccessful string search), the loop will be exited prematurely. For example, the commands in the loop shown below will be executed 12 times or until the string in the search can't be found.

```
[ S/This line should be removed/ -1 D ] 12
```

If a loop is typed in as a command for immediate execution, it must fit in one complete line. If a loop is contained within a macro, it can extend across multiple lines. Loops can be nested within each other.

1.13.2. The Fail Flag

When an edit command is not able to complete its operation, the editor will set an internal flag called the fail flag. For instance if you tried to read from a file that had no more text in it, the editor would set the fail flag. The fail flag is used to control loops and conditional statements.

After the fail flag has been set, the editor skips commands until it reaches: a) the end of a keyboard command line, or b) the end of the current loop or c) an IF (“:”) command.

The testing commands described in this section set or clear the fail flag. The other commands listed below can also set the fail flag upon condition noted:

<	Attempt to move the edit pointer before the beginning of the buffer.
>	Attempt to move the edit pointer past the end of the buffer.
S	Not finding the search string.
C	Not finding the search string.
G	No text left in the secondary buffer.
R	No text left in the read file.
P,W	No text left in the primary buffer.

1.13.3. Conditional Statements

Conditional statements can be formed inside loops using the “:” operator. When the “:” is encountered, all statements that follow until the end of the current loop or macro are skipped if the fail flag is not set (e.g., fail flag is cleared).

Below is an example of a command line which deletes all lines of text that do not begin with “A”:

```
^ [ .NEOB [ .STR"A" + : D ] ]*
```

The “^” moves the edit pointer to the beginning of the buffer. The outer loop tests for when the end of the buffer is reached and terminates the loop. The inner loop tests for an “A” at the beginning of the line. If there is one, the “+” command is executed, otherwise the “D” command is executed.

Below is another example which searches the current line for “find it”. If found, the line will be displayed, otherwise the command line will fail and “* FAIL *” will be printed:

```
[ .EOL V0 -0 V .F : .STR"find it" -0 .S    : [>] ]*
```

In the command line above, the first part is “.EOL V0 -0 V .F”, which tests if the edit pointer is at the end of the line. If it is, verify mode is turned off to prevent the “-0” from displaying the line, and then it is turned back on and the ".F" causes the loop to be terminated. If the edit pointer is not at the end of the line, the ".STR" command will see if "find it" is at the current edit position. If it is, the "-0 .S" commands will be executed to cause the edit pointer to be moved back to the beginning of the line, the line displayed, and the loop terminated. Otherwise the ">" command is executed which moves the edit pointer to the next position in the line. Note that it is enclosed in brackets to prevent it from failing and terminating the main loop if the end of the buffer is reached.

1.14. Testing Commands

The commands that follow test for certain conditions, and either set or clear the fail flag depending on the result of the test.

1.14.1. Test For End of File

SYNTAX: .EOF

This command clears the fail flag if the input file is at end-of file, otherwise it will set the fail flag.

1.14.2. Test For Not End of File

SYNTAX: .NEOF

This command clears the fail flag if the input file is not at end-of file, otherwise it will set the fail flag.

1.14.3. Test For End of Buffer

SYNTAX: .EOB

This command will clear the fail flag if the edit pointer is at the end of the buffer, otherwise it will set the fail flag.

1.14.4. Test For Not End of Buffer

SYNTAX: .NEOB

This command will clear the fail flag if the edit pointer is not at the end of the buffer, otherwise it will set the fail flag.

1.14.5. Test For End of Line

SYNTAX: .EOL

This command will clear the fail flag if the edit pointer is at the end of a text line, otherwise it will set the fail flag.

1.14.6. Test For Not End of Line

SYNTAX: .NEOL

This command will clear the fail flag if the edit-pointer is not at the end of the line, otherwise it will set the fail flag.

1.14.7. Test For Zero

SYNTAX: .ZERO *n*

This command will clear the fail flag if "*n*" is equal to zero, otherwise it will set the fail flag.

1.14.8. Test For Star

SYNTAX: .STAR *n*

This command will clear the fail flag if "*n*" is equal to 65535 (the value of "*"), otherwise it will set the fail flag.

1.14.9. Test For String Match

SYNTAX: .STR `str`

This command will clear the fail flag if the characters at the current edit position match "`str`", otherwise it will set the fail flag.

1.14.10. Test For String Mismatch

SYNTAX: .NSTR `str`

This command will clear the fail flag if the characters at the current edit position do not match "`str`", otherwise the fail flag will be set.

1.14.11. Exit And Clear

SYNTAX: .S

This command is an unconditional exit from the innermost loop or macro; The fail flag is cleared after the exit.

1.14.12. Exit And Fail

SYNTAX: .F

This command is an unconditional exit from the innermost loop or macro. The fail flag is set after the exit.

1.15. Macros

Macros are new commands that you may create to perform a specialized or complex task. For example, there may be a frequently used sequence of commands that you wish to replace with a single macro. You store the sequence of commands in a macro, after which they may be executed by simply typing a period followed by the macro name and optional parameters.

Macros can be saved and loaded from disk files so you can create a personalized macro library.

Macros are made up of two main parts, the macro header and the macro body. The macro header is used to give the macro a name and describe the type and order of its parameters. The macro body is made up of any number of ordinary command lines (any edit command may be used in a macro except the "SPACE" and the "RETURN" commands). Also, macros can not create new macros.

To create a macro, you must first open its definition with the ".MAC" command. After doing so, you may enter the macro's header and body just as you would enter text into an edit buffer (you may use any of the edit commands to do so). When you are satisfied with the macro, you may close its definition with the "Q" command to return you to the normal edit mode.

1.15.1. Macro Headers

A macro header must be the first line in each macro. It is made up of a macro name which may be followed by a "variable list" that describes the macro's parameters if there are any. The macro name consists of any number of consecutive letters and underline characters. Below are some example macro names:

MACRO
trim_spaces
LIST
EXTRA_LONG_MACRO_NAME

Although a macro name may be of any length, it is advisable to limit them to a reasonable length since the name must be spelled exactly the same way each time that you use it. Upper case and lower case letters are taken to be equivalent and may be used interchangeably.

1.15.2. Parameters and Variables

Like other edit commands, macros may also be given parameters so that they are able to work with different values. Parameters are available to the commands that make up the macro. To pass the macro's parameters to these commands, we need a way to tell each command which of the macro's parameters it should use. This is what the variable list in the macro header is for. Each variable in the variable list is used to represent the value of the macro parameter in its corresponding position. Then wherever the parameter's value is needed, the corresponding variable should be used.

There are two types of variables: numeric and string. A numeric variable is a variable name is preceded by a “#” character. A string variable is a variable name preceded by a “$” character. Variable names are just like macro names which are composed of any number of consecutive letters and underline characters. Some example numeric variables are:

#N
#ABC
#LONG_NUMBER_VARIABLE

Some example string variables are:

$A
$B
$STR
$STR_A
$lower_case_variable_name

An example of an entire edit macro is given below. It will do the same thing as the “S” command: search for the next “*n*” occurrences of a string. The first line of the macro is the macro header; it declares the macro's name to be “FIND_LN” and also specifies that the macro needs one numeric parameter “#N” and one string parameter “$STR”. The entire body of the macro is the second line. Here, both of the macro's parameters are passed to the “S” command to do the actual searching:

```
FIND_LN #N  STR
S #N $STR
```

Here is how this macro would be called:

```
E:.FIND_LN 15 "string" RETURN
```

To illustrate the importance of the parameter position in the macro header, we will reverse their order in the next example to make it necessary to use the reverse order when executing the macro. Here is the macro definition:

```
FIND_LN $STR #N
```

```
S #N $STR
```

We must still specify the parameters for the “S” command in the proper order since it is only the “FIND_LN” macro that was changed. Below is an example of how this macro is executed, notice that the order of the parameters directly correspond to the order of the variables in the variable list:

```
.FIND_LN "string" 15
```

1.15.3. Creating and Editing Macros

Macros are created and edited using the normal editor command set. The “.MAC” command is used to create a new macro or open the definition of an existing one so that it may be edited. To create a *new* macro, you use “.MAC” with an empty string, for example:

```
E:.MAC // RETURN
```

This creates a new macro and puts you into macro definition mode. The editor responds with the “M:” prompt instead of the normal “E:” edit prompt when in macro definition mode. If you wish to edit a macro that already exists, “*str*” is used to specify the macro's name. For example:

```
E:.MAC "MYMACRO" RETURN
```

This opens the existing macro “MYMACRO” for editing. When a macro is open, you may edit it or enter its definition by using the edit commands as you would with an ordinary text buffer.

The “Q” command is used to close the definition of a macro and return to the normal edit mode. For example:

```
M:Q RETURN
```

This would close the definition of the macro currently open and return the editor to its normal edit mode. Before the editor will allow you to close the definition of a macro, the first line of the macro must begin with a legal macro name that has not already been used for another macro.

1.16. Macro Commands

1.16.1. Comment

SYNTAX: `!` *text*

The “!” command may be used to place comments inside of a macro for documentation. The remainder of the line following the “!” command is retained but never processed as a command.

1.16.2. Load Macros From File

SYNTAX: .LOAD *str*

This command is used to load macros from an OS-9 file. The file name pathlist is specified by “*str*”. As each macro is loaded, EDIT will make sure that no other macro already exists with the same name. If one does, the macro will not be loaded and EDIT will skip to the next macro on the file. EDIT will display the names of all the macros that it loads. Some examples of this command are:

```
E:.LOAD "macrofile"RETURN
E:.LOAD "MYFILE" RETURN
```

1.16.3. Save Macros

SYNTAX: .SAVE *str1 str2*

This command is used to save macros on an OS-9 file. The first string is used to specify a list of macros that are to be saved; the macro names are separated by spaces. The second string specifies the pathlist for the file on which the macros are to be saved. Some typical examples of the use of this command are:

```
E:.SAVE "MYMACRO"MYFILE" RETURN
E:.SAVE "MACA MACB MACC"MFILE" RETURN
```

The first example saves the macro “MYMACRO” on the file “MYFILE”, the second saves the macros “MACA”, “MACB”, and “MACC” on the file “MFILE”. When more than one macro is to be saved on a single file, their names should be separated by spaces.

1.16.4. Delete Macro

SYNTAX: .DEL *str*

This command is used to delete (erase) the macro specified by “*str*”. For example to delete a macro called “MYMACRO” use:

```
E:.DEL "MYMACRO" RETURN
```

Chapter 2. Assembler

2.1. Introduction

The actual machine instructions executed by a computer are sequences of binary numbers that are difficult and inconvenient for people to deal with directly. Creating a machine language program of any length by hand is tedious, error prone, and time consuming, making it an almost impossible task. *Assembly language* bridges the gap between computers and people who must write machine-language programs. In assembly language, descriptive mnemonics (abbreviations) for each machine instruction are used which are much easier to learn, read, and remember are used instead of numerical codes. The assembler also lets the programmer assign *symbolic names* to memory addresses and constant values. The Assembler also has many other features to make assembly language programming easier.

This assembler was designed expressly for the modular, multi-tasking environment of the OS-9 Operating System, and incorporates built-in functions for calling OS-9, generating memory modules, encouraging the creation of position-independent-code, and maintaining separate program and data sections. It has also been optimized for use by OS-9 high-level language compilers such as Pascal and C, and can be used on either OS-9 Level One or OS-9 Level Two systems.

Another noteworthy characteristic of this assembler is its extremely fast assembly speed which is attributable to its tree-structured symbol table organization. The tree structure dramatically reduces symbol table searching, which is the most time-consuming operation performed by an assembler.

This manual describes how to use the OS-9 Assembler and basic programming techniques for the OS-9 environment. It is not intended to be a comprehensive course on assembly language programming or the 6809 instruction set. If you are not familiar with these topics, you should consult the Motorola 6809 programming manuals and one of the many excellent assembly-language programming books available at libraries or bookstores.

2.2. Installation

The OS-9 Assembler uses the following files:

asm	the assembler program
DEFS	a directory containing OS-9 common system-wide definition files. These files are: OS9Defs SysType SCFDefs RBFDefs

The file “ASM” should be located in the “CMDS” directory of your system disk. “DEFS” should be present in the root directory.

2.3. Assembly Language Program Development

Writing and testing of assembly language programs involves an edit/assemble/test cycle. In detail, the individual steps are:

1. Create a source program file using the text editor.

2. Run the assembler to translate the source file to a machine object (machine language) file.

3. If the assembler reported errors, use the text editor to correct the source file, then go to step 2.

4. Run and test the program. The OS-9 Interactive Debugger is frequently used for testing.

5. If the program has bugs, use the text editor to correct the source file, then go to step 2.
6. Document the program and you are done!

2.4. Operational Modes

The OS-9 Assembler has a number of features specifically designed to conveniently develop machine language programs for the OS-9 environment. These features include: special assembler directive statements for generating OS-9 memory modules, identification of 6809 addressing modes that are not usually permitted in OS-9 programs, and separate data and program address counters.

The assembler has two operating modes: “normal”, and “Motorola-compatible”.

In normal mode, the features mentioned above are active. In the Motorola-compatible mode, the assembler works the same way as a standard 6809 “absolute” assembler (without separate program and data counters). This mode exists so that the assembler can be used to generate programs for 6809 computers that are not equipped with OS-9.

The assembler. will be in the normal mode unless the “m” option is used in the command line or in an OPT statement. Similarly, the “-m” option will return the assembler to the normal mode (modes can be freely switched to achieve special effects).

The assembler performs two “passes” (complete scans) over the source file. During each pass, input lines are read and processed one at a time. During the first pass, the symbol table is created. Most error messages, the program listing, and the object code are generated during the second pass.

2.5. Running the Assembler

The assembler is a command program that can be run from the OS-9 **Shell**, from a Shell procedure file, or from another program. The disk file and memory module names are “asm”. The basic format of a command line to run the assembler is:

```
asm filename [option(s)] [#memsize] [ >listing ]
```

Brackets enclose optional things, thus the only items absolutely required are the “asm” command name, and “filename” which is the source text file name (or more correctly, pathlist). A typical command line looks like this:

```
OS9: asm prog5 l s -c #12k >/p RETURN
```

In. this example, the source program is read from the file “prog5”. The source file name can be followed by an *option list*, which allows you to control various factors such as whether or not a listing or object file is to be generated, control the listing format, etc. The option list consists of one or more option abbreviations separated by spaces or commas. An option is turned on by its presence in the list, or a minus followed by an option abbreviation acts to turn the function off. If an option is not expressly given, the assembler will assume a *default* condition for. it. Also, command line options can be overridden by OPT statements within the source program (see the OPT statement description for more information). In the example above, the options “l” and “s” are turned on, and “c” is turned off.

The optional “#memsize” item is actually processed by the **Shell** to specify how much data area memory the assembler is assigned. If memory is not specified, the assembler will be assigned 4K bytes of memory in its data area. Most of this space is used to store the symbol table. Any additional memory requested by this option allows the symbol table to be larger. Large programs generally use more symbols, so their memory requirements are correspondingly larger. If the assembler generates a “Symbol Table Full” error message, this option should be used to increase the assembler's memory size. In the previous example, 12K bytes of memory is specified.

The final item, “>listing”, allows the program listing generated by the assembler (on the standard output path) to be optionally redirected to another pathlist, which may be an output device such as a printer, a disk file, or a pipe to another program. Like the memory size option, output redirection is handled by the **Shell** and not the assembler itself. If this item is omitted from the command line, the output will appear on your terminal display. In the above example, the listing output was directed to a device called “p”, which is the name of the printer on most OS-9 systems.

Below are examples of various forms of command lines and detailed explanations of their output. There are also other options available so many variations of the command line are possible. The examples listed here illustrate the most commonly used forms.

```
asm disk_crash
```

This command line will assemble the file `disk_crash`.

There will be:	no listing created.
	no object file created.
	errors reported to standard error path.
	4k memory for symbols (default).

```
asm work.rec o #16k
```

This command line will assemble the file `work.rec`.

There will be:	no listing created.
	an object file created with the name “work.rec” in the current commands dir.
	errors reported to standard output path.
	16k of memory for symbols.

```
asm tyco o=/d0/cmds/tyco.obj l #16k
```

This command line will assemble the file `tyco`.

There will be:	a listing directed at standard output.
	an object file created with the name “tyco” in the /d0/cmds directory.
	errors reported to the listing path.
	16k of memory for symbols.

```
asm it_works o,l #16k >/p
```

This command line will assemble the file `it_works`.

There will be:	a listing directed at /p.
	an object file created with the name “it_works” in the current commands dir.
	errors reported to the listing path.
	16k of memory for symbols.

```
asm rent_a_duck 1,5,w72,d25 #10k
```

This command line will assemble the file `rent_a_duck`.

There will be:	a listing directed to standard output.
	no object file created.
	errors reported to the listing path.
	10k of memory for symbols.

a symbol table created.
listing will have 25 line pages.
listing will have 72 column lines.

```
asm /term i l o=/d0/progs/woof
```

This command line will assemble input from the terminal.

There will be:	a listing directed at standard output.
	an object file created with the name woof in the /d0/progs directory.
	errors reported to the listing path.
	4k of memory for symbols (asm default).

2.6. Source Program Format and Syntax

2.6.1. Assembler Source Files

The Assembler reads its input from an input file (path) which contains variable-length lines of text. Input files may be created and edited by the OS-9 Text Editor described in Section 1, or any other standard text editor.

The maximum length of the input line is 120 characters. Each line contains assembler statements as explained in this manual. Every line is terminated by a [RETURN] character.

2.6.2. Source Statement Fields

Each input line is a text string terminated by a [RETURN]. The line can have from one to four “fields”:

- an optional label field
- an operation field
- an operand field (for some operations)
- an optional comment field

There are also special cases: if the first character of a line is an asterisk, the entire line is treated as a comment which is printed in the listing but not otherwise processed. Blank lines are ignored but are included in the listing.

2.6.2.1. Label Field

The label field begins in the first character position of the line. Labels are usually optional (instructions), but there are exceptions. They are required by some statements (i.e. EQU and SET), or not allowed on others (assembler directives such as SPC, TTL, etc.). The first character .of the line must be a space if the line does not contain a label.

The label must be a legal symbolic name consisting of from one to eight uppercase or lowercase characters, decimal digits, or the characters “$”, “_”, or “.”, however the first character must be a letter (see Sect. 3.3). Labels (and names in general) must be unique, i.e., they cannot be defined more than once in a program (except when used with the “SET” directive).

Label names are stored in the symbol table with an associated 16-bit value, which is normally the program counter address before code is generated for the line. In other words, instructions and most constant-definition statements associate the label name with the value of.the program address of the first object code byte generated for the line.

An exception to this rule is that labels on SET and EQU statements are given the value of the result of evaluation of the operand field. In other words, these statements allow any value to be associated with a symbolic name. Likewise, labels on RMB statements are given the value of the data address counter when in normal assembler mode, or the value of the program address counter when in Motorola-compatible mode.

2.6.2.2. Operation Field

This field specifies the machine language instruction or assembler directive statement mnemonic name. It immediately follows and is separated from the label field by one or more spaces.

Some instructions must include a register name which is part of the operation field (i.e., LDA, LDD, LDU). In these instructions the register name must be part of the name and *cannot* be separated by spaces as in older 6800-type assemblers. The assembler accepts instruction mnemonic names in either uppercase or lowercase characters.

Instructions cause one to five bytes of object code to be generated depending on the specific instruction and addressing mode. Some assembler directive statements (such as FCB, FCC) also cause object code to be generated.

2.6.2.3. Operand Field

The operand field follows, and must be separated by, at least one space from the instruction field. Some instructions don't use an operand field; other instructions and assembler directives require an operand field to specify an addressing mode, operand address, parameters, etc. The sections describing the instructions and assembler directives explain the format for operand(s), if any.

2.6.2.4. Comment Field

The last field of the source statement is the optional comment field which can be used to include a descriptive comment in the source statement. This field is not processed other than being copied to the program listing.

2.7. Expressions

Operands of many instructions and assembler directives can include arithmetic expressions in various 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.

Expressions consists of *operands*, which are symbolic names or constants, and *operators*, which specify an arithmetic or logical function. All assembler arithmetic uses two-byte (internally, 16 bit binary) signed or unsigned integers in the range of 0 to 65535 for unsigned numbers, or -32768 to +32767 for signed numbers.

In some cases, expressions are expected to evaluate to a value which must fit in one byte (such as 8-bit register instructions), and therefore must be in the range of 0 to 255 for unsigned values and -128 to 127 for signed values. In these cases, if the result of an expression is outside of this range an error message will be given.

Expressions are evaluated from left-to-right using the algebraic order of operations (i.e. multiplications and divisions are performed before additions and subtractions). Parentheses can be used to alter the natural order of evaluation.

2.7.1. Operands

The following items may be used as operands within an expression:

2.7.1.1. Decimal Numbers

Can have an optional minus sign and one to five digits, for example:

100	-32761	12	5	-1

2.7.1.2. Hexadecimal Numbers

Consist of a dollar sign (“$”) followed by one to four hexadecimal characters (0-9, A-F or a-f), for example:

$EC00	$100	$3

2.7.1.3. Binary Numbers

Consist of a percent sign (“%”) followed by one to sixteen binary digits (0 or 1), for example:

%0101	%1111000011110000	%10101010

2.7.1.4. Character Constants

Consist of single quote (“'”) followed by any printable ASCII character. For example:

'X	'c	'5	'c

2.7.1.5. Symbolic Names

Names are defined by EQU or SET statements, or by use as a label. They consist of one to eight characters: upper and lower case alpha (A-Z, a-z), digits (0-9), and special characters _, ., or $ (underscore, period or dollar sign), the first character of which cannot be a digit. See Page 2-12 for more information.

2.7.1.6. Program Instruction Counter

The asterisk (“*”) represents the program instruction counter value as of the beginning of the line.

2.7.1.7. Program Data Counter

The period (“.”) represents the data storage counter value as of the beginning of the line. It is not used in Motorola-compatible mode.

2.7.2. Arithmetic and Logical Operators

Operators used in expressions operate on one operand (negative and not) or on two operands (all others). The table below shows the available operators, listed in the order they are evaluated relative.to each other, e.g, logical OR operations are performed before multiplications. Operators listed on the same line have identical precedence and are processed from left to right when they occur in the same expression.

Table 2.1. Operators By Order of Evaluation

- negative	^ logical NOT	(highest)
& logical AND	! logical OR	
* multiplication	/ division	
+ addition	- subtraction	(lowest)

Logical operations are performed bitwise, i.e., the logical function is performed bit-by-bit on each bit of the operands.

Division and multiplication functions assume *unsigned* operands, but subtraction and addition work on signed (2's complement) or unsigned numbers. Division by zero or multiplication resulting in a product larger than 65536 have undefined results and are reported as errors.

2.7.3. Symbolic Names

A symbolic name consists of from one to eight uppercase or lowercase characters, decimal digits, or the characters “$”, “_”, or “.”. However, the first character must be a letter. The following are examples of *legal* symbol names:

HERE	there	SPL030	VX_GH
abc.def	Q1020.1	L.123.x	t$integer

These are examples of illegal symbol names with reasons why they are illegal:

2move - does not start with a letter
main.backup - more than 8 characters
lb1#123 - # is not a legal name character

Names are defined when first used as a label on an instruction or directive statement. They must be defined exactly one time in the program (except SET labels: see SET statement description). If a name is redefined (used as a label more than once) an error message is printed on subsequent definition(s). Multiple forward references (i.e. a definition using currently undefined names) are not allowed.

Symbolic names are stored with their associated type and value in an assembler data structure called the “symbol table”, which uses most of the assembler's data memory space. Using the default memory size of 4K there is room in the symbol table for approximately 200 names. The **shell** optional memory size modifier can be used to give the assembler a larger memory space. Each entry in the table requires 15 bytes, so each additional 4K of memory adds space for about 273 additional names. For example, the command line:

```
asm sourcefile #16K
```

gives the symbol table enough space for a little over a thousand names. If the “S” option is selected, the assembler will generate an alphabetical listing of all symbol names, types, and values which is printed at the end of the assembly.

2.7.4. Instruction Addressing Modes

One of the 6809's features is that its instruction set has a large variety of addressing modes. Each group of similar instructions can be used with specific addressing modes, which are usually specified in the assembler source statement operand field. The assembler will generate an error message if an addressing mode is specified which cannot be legally used with the specific instruction.

2.7.4.1. Inherent Addressing

Certain instructions don't need operands (SYNC, SWI, etc.), or implicitly specify operands (MUL, ABX, etc.), therefore no operand field is needed.

2.7.4.2. Accumulator Addressing

Some instructions have the A or B accumulators as operands. Examples:

```
CLRA
ASLB
INCA
```

2.7.4.3. Immediate Addressing

In immediate addressing, the operand bytes are the actual value used by the instruction. Instructions that use 8-bit registers must have operand expressions that evaluate to 0 to 255 (unsigned) or -128 to 127 (signed), or an error will be reported. The syntax is:

```
instr #expression
```

Examples:

```
LDD #$1F00
ldb #bufsiz+2
ORCC #$FF-CBIT
```

2.7.4.4. Relative Addressing

This addressing mode is used by branch-type instructions such as BCC, BEQ, LBNE, BSR, LBSR, etc. The operand field is an expression which is the “destination” of the instruction, which is almost always a name used as a statement label somewhere in the program. The assembler computes an 8 or 16-bit program counter offset to the destination which is made part of the instruction. The destination of short branch-type instructions must be in the range of -126 to +129 bytes of the instruction address or an error message will be generated. Long branch-type instructions can reference any destination. If a long branch instruction references a destination that would be within the range of a smaller and faster short branch instruction a “W” warning symbol will be placed in the listing line's information field. All instructions using relative addressing are inherently position-independent code.

Examples:

```
BCS  LOOP
LBNE LABEL5
LBSR START+3
BLT  COUNT
```

2.7.4.5. Extended and Extended Indirect Addressing

Extended addressing uses the second and third bytes of the instruction as the absolute address of- the operand. Data section addresses of OS-9 programs are assigned when the program is actually executed, so absolute memory addresses are not known before the program is run. Therefore this addressing mode is not normally used in OS-9 programs. The assembler will print an informational warning flag, “W”, if this addressing mode is specified.

Extended Indirect addressing is similar to extended addressing except that the address part of the machine instruction is used as the address of a memory location containing the address of the operand. Because this mode also uses absolute addresses, it is not frequently used in OS-9 for the reasons given above, and is also flagged with a warning by the assembler. This addressing mode is selected by enclosing the address expression in brackets.

Examples:

```
ADDA $1C48     extended addressing
ADDA [$D58A]   extended indirect addressing
LDB START      extended addressing
stb [end]      extended indirect addressing
```

2.7.4.6. Direct Addressing

Direct addressing uses the second byte of the instruction as the least significant byte of the operand's address. The most significant byte is obtained from the MPU's direct page register. This addressing mode is preferred for accessing most variables in OS-9 programs because OS-9 automatically assigns unique direct pages to each task at run-time, and also because this mode produces short, fast instructions. The syntax for extended and direct addressing has the same form:

```
instr <addr expr>
```

The assembler automatically selects direct addressing mode if the high-order byte of the address matches it's internal “direct page”. This “direct page” is not the same as the run-time direct page register: it is strictly an assembly-time value. It is ordinarily set to zero and can be changed with the SETDP directive.

You can force the assembler to use direct addressing by using the “<” symbol just before the address expression, or extended addressing by using the “>” symbol in the same manner.

Examples:

```
lda temp          (assembler selects mode)
LDD >PIA+1        (forces extended addressing)
ldx <count        (forces direct addressing)
STD [pointer]     (extended indirect)
```

2.7.4.7. Register Addressing

Some instructions operate on various MPU registers, which are referred to by a one or two letter name. In these instructions, the operand field specifies one or more register names. The names can he uppercase or lowercase. The register names are:

A	accumulator A (8 bits)
B	accumulator B (8 bits)
D	accumulator A:B concatenated (16 bits)
DP	direct page register (8 bits)
CC	condition codes register (8 bits)
X	index register X (16 bits)
Y	index register Y (16 bits)
S	stack pointer register (16 bits)
U	user stack pointer register (16 bits)
PC	program counter register (16 bits)

The EXG and TFR instructions have the form:

```
instr reg,reg
```

The registers given must be of the same size (either 8 or 16 bits) or the assembler will report an error.

The PSHS, PSHU, PULS, and PULU instructions accept a list of one or more register names. Even though the assembler will accept register names in any order, the MPU stacks and unstacks them in a specific order.

The syntax for these instructions is:

```
instr reg {,reg}
```

Examples:

```
TFR X,Y
```

```
EXG A,DP
pshs a,b,x,dp
PULU d,x,pc
```

2.7.5. Overview of Indexed Addressing Modes

One of the highlights of the 6809's architecture is the wide variety of indexed addressing modes (23 varieties). Indexed addressing is analogous to “register indirect”, meaning that an indexable register (X, Y, U, S, or PC) is used as the basic address of the instruction's operand. The different varieties of indexed addressing use the specified register v contents which may be unchanged, temporary modified, or permanently modified, depending on the exact mode used.

All indexed modes must specify an index register, either X, Y, U, or SP. The PC register is used with the program-counter relative mode only. Any of the indexed addressing modes can be made “indirect” by enclosing the operand field in brackets to cause the effective address generated by the addressing mode to be used as the address of a pointer to the operand, rather than as the address of the operand.

2.7.5.1. Constant Offset Indexed

This mode uses an optional signed (two's complement) offset which is temporarily added to the register's value to form the operand's effective address. The offset can be any number, or zero in which case the register's unaltered contents is used as the effective address. The assembler automatically picks the shortest of four possible varieties that can represent the offset. Therefore it is important to make sure that any symbolic name used in the offset expression has been previously defined, or the assembler will generate longer code than necessary or produce phasing errors. The syntax for constant offset indexed instructions is:

```
instr ,reg              zero offset
instr offset,reg        constant offset
instr [,reg]            zero offset indirect
instr [offset,reg]      constant-offset indirect
```

Examples:

```
lda ,x               no offset
lda 0,x              no offset
ldx 100,x            offset of 100
Ldb COUNT,S          offset of COUNT
ldd temp+2,y         offset of temp+2
leax -2,y            offset of -2
clr [PIA,X]          indirect mode
```

2.7.5.2. Program Counter Relative Indexed

This addressing mode is similar to constant-offset indexed except that the program counter register (PC or PCR) is used as an index register, and the assembler computes the offset differently. Instead of using the offset expression directly, the expression is assumed to refer to the address of the operand. The assembler calculates the required offset from the current program counter location to the operand's address and uses the resulting value as the offset. There are two forms.of this instruction: one -using an 8-bit offset and the other using a 16-bit offset; The assembler will use the 16-bit form unless you force the short form by preceding the operand field with a “<” character.

The syntax for program-counter relative indexed is:

```
instr addr,PC         program counter relative
instr addr,PCR        program counter relative
instr [addr,PCR]      program counter relative indirect
instr [addr,PC]       program counter relative indirect
```

This addressing mode is important in OS-9 programs because it permits addresses of constants and constant tables to be accessed using position-independent-code as required by OS-9.

Examples:

```
ldd     temp,pcr
LDD     temp,pc          same as instruction above
leax    table,pcr
jsr     addr,pcr         same as “lbsr addr”
CLR     [control+4,PCR] dangerous; uses absolute address
                         at “control+4,PCR” as effective
                         address for clear
```

2.7.5.3. Accumulator Offset Indexed

In this mode the contents of the A, B, or D accumulators is temporarily added to the specified index register to form the address of the operand. This addition is signed two's complement. If the A or B accumulators are specified, the sign bit is “extended” to form the 16 bit value which is added to the index register. Meaning that if the most significant bit of the accumulator is set, the high order byte of the offset will be $FF. *BEWARE:* this is a commonly overlooked characteristic that can produce unexpected results! Using the D register avoids this because it gives all 16 bits. The syntax for accumulator-offset indexed is:

```
instr A,reg
instr B,reg
instr D,reg
```

Examples:

```
LDX B,Y
LEAY D,X
ROL [B,U]
```

2.7.5.4. Auto-Increment and Auto-Decrement Indexed

These addressing modes use the specified index register as the effective address of the operand, while permanently adding or subtracting one or two from the register. In auto-increment mode, the increment is performed AFTER the register is used. In auto-decrement mode, the decrement is performed BEFORE the register is used. This is consistent with the way 6809 stack pointers operate in PSH and PUL instructions. If indirect addressing is used, the decrement and increment are performed before the effective address is used as a pointer to the operand.

SINGLE AUTO-INCREMENT AND SINGLE AUTO-DECREMENT ARE NOT PERMITTED WHEN INDIRECT ADDRESSING IS SELECTED.

Syntax for auto-increment and auto-decrement indexed addressing is:

```
inst: ,-reg       single auto-decrement
instr ,--reg      double auto-decrement
inst: ,reg+       single auto-increment
instr ,reg++      double auto-increment
instr [,reg--]    double auto-decrement indirect
instr [,reg++]    double auto-increment indirect
```

Examples:

```
clr ,x++
LDX ,--Y
lda ,s+   is the same as puls a (except CCR is affected)
sta ,-s   is the same as pshs a (except CCR is affected)
ldd [,s++]
```

2.8. Assembler Directive Statements

In addition to the 6809 instruction mnemonic statements, the assembler includes a number of directive statements which perform a variety of functions which can be loosely categorized as follows:

Assembly Control

Statements such as IF, OPT, END, USE, etc., are used to control the operation of the assembler itself but do not directly affect object code generation.

Storage Declarations

The ORG and RMB statements are used to assign variable (data area) storage for the assembly language program,

Symbolic Name Declarations

The EQU and SET statements are used to assign value to assembler symbolic names.

Constant Declarations

The FCB, FDB, FCC, and FCS statements are used to insert constant values in the assembler program.

Operating System Functions

The MOD/EMOD and OS9 statements are used to create system-required object code for memory modules and system call, respectively.

Each assembler directive statement is described in detail in the following pages.

2.8.1. END Statement

Indicates the end of a program. Its use is optional since END will be assumed upon an end-of-file condition on the source file. END statements may not have labels.

2.8.2. EQU and SET Statements

SYNTAX: EQU <*expression*>
SET <*expression*>

These statements are used to assign a value to a symbolic name (the label) and thus require labels. The value assigned to the symbol is the value of the operand, which may be an expression, a name, or a constant.

The difference between the EQU and SET statements is that:

- Symbols defined by EQU statements can be defined only once in the program.
- Symbols defined by SET statements can be redefined again by subsequent SET statements.

In EQU statements the label name must not have been used previously, and the operand cannot include a name that has not yet been defined (i.e., it cannot contain as-yet undefined names whose definitions also use undefined names). Good programming practice, however, dictates that all equates should be at the beginning of the program to allow the assembler to generate the most Compact code by selecting direct addressing wherever possible.

EQU is normally used to define program symbolic constants, especially those used in conjunction with instructions. SET is usually used for symbols used to control the assembler operations, especially conditional assembly and listing control. Example:

```
TRUE    equ $FF
FALSE   equ 0
SUBSET  set TRUE
        ifne SUBSET
        use subset.defs
        else
        use full.defs
        endc
SUBSET  set FALSE
```

2.8.3. FCB and FDB Statements

SYNTAX: FCB <*expression*> {, <*expression*>}
FDB <*expression*> {, <*expression*>}

The Form Constant Byte and Form Double Byte directives generate sequences of single (FCB) and double (FDB) constants within the program. The operand is a list of one or more expressions which are-evaluated and output as constants. If more than one constant is to be generated, the expressions are separated by commas.

FCB will report an error if an expression has a value of more than 255 or less that -128 (the largest number representable by a byte). If FDB evaluates an expression with an absolute value of less than 256 the high order-byte will be zero.

Examples:

```
FCB 1,20,'A
fcb index/2+1,0,0,1
FDB 1,10,100,1000,10000
fdb $F900,$FA00,$FB00,$FC00
```

2.8.4. FCC and FCS Statements

SYNTAX: FCC <delim> string <delim>
FCS <delim> string <delim>

These directives generate a series of bytes corresponding to a string of one or more characters operand. The output bytes are the literal numeric value of each ASCII character in the string. FCS is the same as FCC except the most significant bit (the sign bit) of the last character in the string is set, which is a common OS-9 programming technique to indicate the end of a text string without using additional storage.

The character string must be enclosed by delimiters before the first character and after the last character. The characters that can be used as delimiters are:

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

Both delimiters must be the same character and cannot be included in the string itself. Examples:

```
FCC /most programmers are strange people/
FCS ,0123456789,
fcc $z$
```

2.8.5. IF, ELSE, and ENDC Statements

```
SYNTAX:  IFxx <expression>
           <statements>
         [ ELSE ]
           <statements>
         ENDC
```

An important feature of the Assembler is its *conditional assembly* capability to selectively assemble or not assemble one or more parts of a program depending on a variable or computed value. Thus, a single source file can be used to selectively generate multiple versions of a program.

Conditional compilation uses statements similar to the branching statements found in high level languages such as Pascal and Basic. The generic IF statement is the basis of this capability. It has as an operand a symbolic name or an expression. A comparison is made with the result: if the result of the comparison is true, statement following the IF statement will be processed. If the result of the comparison is false, the following statements will not be processed until an ENDC (or ELSE) statement is encountered. Hence, the ENDC statement is used to mark the end of a conditionally assembled program section. Here is an example that uses the IFEQ statement which tests for equality of its operand with zero:

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

The ELSE statement allows the IF statement to explicitly select one of two program sections to assemble depending on the truth of the IF statement. Statements following the ELSE statement are processed only if the result of the comparison was false. For example:

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

Multiple IF statements may be used, and “nested” within other IF statements if desired. They cannot, however, have labels.

There are several kinds of IF statements, each performing a different comparison. They are:

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 first assembler pass (no operand)

The IF statements that test for less than or greater than can be used to test the relative value of two symbols if they are subtracted in the operand expression, for example,

IFLE MAX-MIN

will be true if MIN is *greater* than MAX. Note the reversal of logic due to the fact that this statement literally means

IF MAX-MIN <= 0

The IFP1 statement causes subsequent statements to be processed during pass 1, but skipped during pass 2. It is useful because it allows program sections which contain only symbolic definitions to be processed only once during the assembly. The first pass is the only pass during which they are actually processed because they do not generate actual object code output. The `OS9Defs` file is an example of a rather large section of such definitions. For example, the following statement is used at the beginning of many source files.

```
IFP1
use /d0/defs/OS9Defs
ENDC
```

2.8.6. MOD and EMOD Statements

SYNTAX: MOD *`size,nameoff,typelang,attrrev {,execoff,memsize}`*
EMOD

These instructions provide a convenient means of creating OS-9 memory modules and their associated module headers and CRC check values. A detailed discussion of the module and module header format can be found in the *System Programmer's Manual*. Programs can be loaded into memory by OS-9 only if they are in module header format.

The MOD statement is used at the beginning of an OS-9 module. Its function is to create a standard OS-9 module header and to initialize a CRC (cyclical redundancy check) value which will be automatically computed by the assembler as the program is processed by the assembler.

The MOD statement must have an operand list of exactly four or exactly six expressions separated by commas. Each operand corresponds, in order, to the elements of a module header. The exact operation of the MOD statement is as follows:

1. The assembler's program address counter and data address counters are reset to zero (same as ORG 0), and the internal CRC and vertical parity generators are initialized.
2. The sync codes $87 and $CD are generated as object code.
3. The first four expressions in the operand list are evaluated and output as object code. They are:
   a. module size (two bytes)
   b. module name offset (two bytes)
   c. type/language byte (one byte)
   d. attribute/revision byte (one byte)
4. The “header parity” byte is automatically computed by the assembler from the previous bytes and generated as object code.

5. If the two optional additional operands are present, they are evaluated and generated as object code. They are:

   e. execution offset

   f. permanent storage size

Note that some of the expressions in the operand list are one byte long, and others are two bytes.

Because the origin of the object program is zero, all labels used in the program are inherently relative to the beginning of the module; this is perfect for the module name and execution address offsets. The code in the body of the module follows. As subsequent lines are assembled, the internal CRC generator continuously updates the module's CRC value. The EMOD statement (which has no operand) is used to terminate the module. It outputs the correct 3-byte CRC generated over the entire module.

IMPORTANT NOTE: The MOD and EMOD statements will not work correctly if the assembler is in “Motorola-compatible” mode unless you do not use RMB or ORG statements after the MOD and before the EMOD.

The following example illustrates the basic techniques of creating a module using MOD and EMOD statements.

Example 2.1. Sample Program Illustrating Use Of MOD and EMOD Directives

```
* Repeat Utility - Copy one line of standard input to standard output
* Module Header Declaration

type   set PRGRM+OBJCT  (these are defined in OS9DEFS)
revs   set REENT+1      (this is defined in OS9DEFS)
       MOD pgmlen,name,type,revs,start,memsiz
       fcb 1            edition number (optional)
name   FCS /repeat/     module name string

* symbolic definitions

STDIN  equ 0            standard input path
STDOUT equ 1            standard output path
LINLEN equ 80           maximum line length

* data storage declarations

buffer RMB 80
stack  RMB 250
memsiz EQU .            data storage size is final “.” value

* Program instructions

start  equ *

       lda #STDIN       load input path number
       ldy #LINLEN      load max input count
       leax buffer,u    get buffer address
       os9 i$readln     call OS-9 to read line
       bcs exit         abort if error

       lda #STDOUT      load output path number
       leax buffer,u    get buffer address
       os9 i$writln     call OS-9 to write line
       bcs exit         abort if error
       clrb             return not error code
exit   os9 F$EXIT       return to OS-9

       emod             end module

pgmlen EQU *  program size is addr of last byte +1
```

2.8.7. NAM and TTL Statements

SYNTAX: NAM *string*
TTL *string*

These statements allow the user to define or redefine a program name and listing title line which will be printed on the first line of each listing page's header. These statements cannot have label or comment fields.

The program name is printed on the left side of the second line-of each listing page, followed by a dash, then by the title line. The name and title may be changed as often as desired.

Examples:

```
nam Datac
ttl Data acquisition System
```

2.8.8. OPT Statement

SYNTAX: OPT <*option*>

OPT allows any of several assembler control options to be set or reset. The operand of the OPT statement is one of the characters that represent the various options. If the option name is preceded by a minus sign, the option is turned off, otherwise it is turned on. Two exceptions are the “D” and “W” options which must be followed by a number. This statement must not have label or comment fields.

The options and default (initial) states are:

C	Conditionals On: print conditional assembly statements in listing. (C)
D*num*	Page Depth: set the number of lines per listing page including headings and blank lines. (D66)
E	Error Messages On: print error messages in listing. Errors can be detected when this option is off by presence of an “E” in a statement's informational field. (E)
F	Use Form Feed: Use a form feed for page eject instead of line feeds. (-F)
G	Generate All Constant Lines: Prints all lines of code generated by directives. Otherwise only the first line is printed. (-G)
L	Listing On: Causes formatted assembly listing to be generated. If off, only error messages printed. (-L)
M	Turn on Motorola-compatible mode. For further information refer to Section 2.4, “Operational Modes” and Section 2.8.12, “SETDP Statement”. (-M)
N	Narrow listing: generate listing in a non-columnized, compressed format for better presentation on narrow video display devices. (-N)
O[=*filename*]	Generate object code file: (-O) If no filename is given, an object file having the same name as the input file will be created in the current *execution* directory. If a single name is given, the object file having that name will be created, but still in the current execution directory.

	If a full *pathlist* is given, it will be used as the name specification of the device, directory(s), and file to create.

S	Generate Symbol Table: Prints the entire contents of the symbol table at the end of the assembly listing. Prints each name, its value and a type code character:
	
	D = data variable (RMB definitions)
	E = equate label (EQU)
	L = program label
	S = set label
	U = undefined name
	
	The table is printed across the page in alphabetical order. (-S)

W*num*	Set Page Width: defines the maximum length of each listing line. Lines are truncated if they exceed this number. The comment field starts at column 50 so a number less than this may cause important parts of the listing to be lost. (W80)

Examples:

```
opt l
opt w72
opt s
```

2.8.9. OS9 Statement

SYNTAX: OS9 *<expression>*

This statement is a convenient way to generate OS-9 system calls. It has an operand which is a byte value to be used as the request code. The output is equivalent to the instruction sequence:

```
SWI2
FCB operand
```

A file called “`OS9Defs`”, which is distributed with each copy of OS-9, contains standard definitions of the symbolic names of all the OS-9 service requests. These names are commonly used in conjunction with the OS9 statement to improve the readability, portability, and maintainability of assembly language software.

Examples:

```
OS9 I$READ            (call OS-9 “READ” service request)
os9 F$EXIT            (call OS-9 “EXIT” service request)
```

2.8.10. ORG Statement

SYNTAX: ORG *<expression>*

Changes the value of the assembler's data location counter (normal mode) or the instruction location counter (Motorola- compatible mode). The expression is evaluated and the appropriate counter is set to the value of the result. ORG statements cannot have labels.

Note: OS-9 does NOT use “load records” that specify absolute addresses of the generated object code: the object code is assumed to be a contiguous memory module. Therefore, programs assembled using the Motorola-compatible mode that alter the instruction address counter will not load correctly.

Examples:

```
ORG  DATAMEM
ORG  .+200
```

2.8.11. PAG and SPC Statements

SYNTAX: PAG[E]
SPC <*expression*>

These statements are used to improve the readability of program listings. They are not themselves printed, and cannot have labels.

The PAG statement causes the assembler to begin a new page of the listing. The alternate form of PAG is PAGE for Motorola compatibility.

The SPC directive puts blank lines in the listing. The number of blank lines to be generated is determined by the value of the operand, which can be an expression, constant, name. If no operand is used a single blank line is generated.

2.8.12. SETDP Statement

SYNTAX: SETDP <*expression*>

Assigns a value to the assembler's internal direct page counter, which is used to automatically select direct versus extended addressing. The direct page counter does not necessarily correspond to the program's actual direct page register during execution.

The default value of the counter is zero, and should NOT be changed in OS-9 programs: this statement is intended for use with the “Motorola-compatible” mode only. SETDP statements cannot have labels.

2.8.13. USE Statement

SYNTAX: USE *pathlist*

Causes the assembler to temporarily stop reading the current input file. It then requests OS-9 to open another file/device specified by the pathlist, from which input lines are read until an end-of-file occurs. At that point, the latest file is closed, and the assembler resumes reading the previous file from the statement following the USE statement.

USE statements can be nested (e.g., a file being read due to a USE statement can also perform USE statements) up to the number of simultaneously open files the operating system will allow (usually 13, not including the standard I/O paths). Some useful applications of the USE statement are to accept read input from the keyboard during assembly of a disk file (as in USE /TERM); and including library definitions or subroutines into other programs. USE statements cannot have labels.

2.9. Assembly Language Programming Techniques

For programs to run correctly in the OS-9 environment, they must be written following these two key rules:

1. Programs *must* be position-independent-code (PIC).
2. All memory locations modified by the program (variables and data structures) must be located in a data memory area whose location is assigned by OS-9 at run-time.

The reason for these rules is simple: OS-9 dynamically assigns memory space *at the time the program is run*. You have no control over which specific addresses are assigned at a load area for the program, or where the program's variables are assigned. Because of the powerful 6809 instruction set and addressing modes, these rules do not force you into writing tricky or complex programs; rather, they require that programs be written a specific way. When you do this, you enjoy the advantages of having reentrant and highly portable programs.

Your programs will usually fall into one of three categories:

1. A subroutine or subroutine package: You must write your subroutines in position-independent code. Data sections are usually a matter of coordination with the calling program, and OS-9 usually plays no direct role in this.

2. A program to be executed as an individual process (commands are of this type). You must use position independent code, and receive data area parameters that delineate the memory space assigned when run.

3. Programs to be run on another, non-OS-9 computer. Have fun! Anything goes!

2.9.1. Program Sections and Data Sections

If your program is to be run as a process (by means of the OS-9 **shell**, fork system call, or execute system call), OS-9 will assign two separate and distinct memory areas. The program object code is loaded into one memory space in the form of a memory module. The other space is for variables and data structures. The program's module header specifies the minimum permissible size for each of these area. The distinction between these two spaces is extremely important. It is also why the assembler has two memory address counters: the data address counter is for the data area and the instruction address counter is for the program area. THE VALUES OF THE COUNTERS ARE NEVER ABSOLUTE ADDRESSES. They are addresses relative to the beginning of an OS-9-assigned address.

2.9.2. Program Area

This area is a single, continuously-allocated memory space, where the program is loaded by OS-9. In order for OS-9 to be able to load the program, it must be in memory module format. The *OS-9 System Programmer's Manual* contains a detailed description of what memory modules are and how they work. This manual assumes you are familiar with them. Programs generated by this assembler can consist of one or more memory modules. They are all written to the same file and will be loaded together by OS-9. In assembly language source programs, modules usually begin with a MOD directive and end with a EMOD directive to take care of the header and module CRC generation for you.

The program area should never be modified by the program itself; especially if the program is to be reentrant and/or placed in ROM. It can (and should) contain constants and constant tables, as long as they are not altered by the program.

2.9.3. Writing Position Independent Code

You do not know the actual absolute address of anything in the program until it is actually run. The 6809 position-independent addressing modes are based on “program counter relative addressing”. This addressing mode is used by all branch and long branch instructions.

- Use BRA and LBRA instead of JMP; use BSR and LBSR instead of JSR extended or JSR direct mode instructions (JSR indexed is OK).

- Use program-counter-relative (PCR) indexed addressing mode (usable by all load, store, arithmetic and logical instructions) to access constants declared in FCB, FDB, FCC, and FCS statements.

- Don't use immediate addressing to load a register with an absolute address (instruction label name): use PCR indexed addressing instead.

Many well-written programs use constant tables of addresses (often called dispatch tables or pointer tables). For the program to be PIC, these tables cannot contain absolute addresses. The correct technique is to create tables of addresses relative to some arbitrary location. The routines that use the tables read the table entries, then add them to the absolute address of the arbitrary location. The sum is the run-time absolute address. The absolute address of the “arbitrary location” is determined Using PCR instructions (typically LEA). The choice of the common address is arbitrary, but two places may have specific advantages: the beginning address of the table (an index register probably will contain this address anyway); and the first byte of the module.

Making table entries relative to the start of the module is especially handy because the value of the assembler's instruction address counter is also relative to the beginning address of the module. Here's an example of a routine that jumps to one of several subroutines whose relative addresses are contained in a table. The routine is passed a number in the B accumulator to be used as an index to select the routine.

```
begin  mod a,b,c,d,e,f start of module

   (various instructions)

dispat leax table,pcr  get the absolute address of the table
       aslb            multiply index by 2 (two bytes/entry)
       ldd  b,x        get contents of table entry
       leax begin,pcr  get beginning address of module
       jmp  d,x        add relative address and go...

table  fdb  routine1
       fdb  routine2
       fdb  routine3
       fdb  routine4
```

The example below does the exact same thing, but the entries are relative to the beginning of the table instead of the beginning of the module:

```
dispat leax table,pcr  get the absolute address of the table
       aslb            multiply index by two
       ldd b,x         get routine offset
       jmp d,x         add and go ...

table  fdb routinel-table
       fdb routine2-table
       fdb routine3-table
       fdb routine4-table
```

Note that this technique has fewer instructions (and is faster) because we already had the reference address in a register, thereby eliminating a LEAX instruction.

The same technique is useful for accessing character strings, constants, complex data types, etc.

2.9.4. Accessing The Data Area

The size of the data area is specified by the “minimum.permanent storage size” entry of the module header. Of course it is possible for a program to optionally receive more than this minimum. Remember that OS-9 allocates memory in exact multiples of 256-byte pages, and all processes get at least one page. The data area must have enough room for all the program's variables and data structures, plus a stack (at least 250 bytes) and space to receive parameters in, if any are passed.

When the process is invoked by OS-9, the bounds of the data area are passed to the process in the MPU registers: U will contain the beginning address and Y the ending address. The SP register is set to the ending address+1, unless parameters were passed. The direct page register will be set to the page number of the beginning page.

In the assembly language source program, storage in the data area is assigned using the RMB directive which uses the separate data address counter. It is good practice (but not mandatory) to declare all variables and structures at the beginning of the program. Smaller, frequently used variables should be declared first. They will usually all fit in the first page, meaning they can be accessed using short, fast direct page addressing instructions. Larger items should follow. These can be addressed in one of two ways:

1. If the U register is maintained throughout the program, constant-offset-indexed addressing can be used.
2. Part of the program's initialization routine can compute the actual addresses of the data structures and store these addresses in pointer locations in the direct page. The addresses can be obtained later using direct-page addressing mode instructions.

Important note: you cannot use program-counter relative addressing to obtain addresses of objects in the data section due to the fact that the memory addresses assigned to the program section and the address section are not a fixed distance apart. Of course, immediate and extended addressing are also not generally usable.

An example that illustrates the U-relative technique is shown on the following page.

Example 2.2. Example of Data Area Access

```
* declare variables
temp    rmb 1
buf1    rmb 400
buf2    rmb 400
buf3    rmb 400

* clear each 400-byte buffer
        leax buf1,u     get address of buf1
        bsr clrbuf
        leax buf2,u     get address of buf2
        bsr clrbuf
        leax buf3,u     get address of buf3
        bsr clrbuf

* clear buffer subroutine
* X = address of buffer
clrbuf  ldd  #400       D = byte count
cloop   clr  ,x+        clear byte and advance pointer
        subd #1         decrement count
        bne  cloop      loop if no done yet
        rts
```

2.9.5. Additional Comments

This information is given as a minimal reference only. The 6809 has many powerful instructions that can do the same things other ways while still retaining PIC and reentrant characteristics. The LEA and TFR instructions can be quite useful, and all the indexed addressing modes are helpful. For more information refer to the *OS-9 System Programmer's Manual*.

2.10. Using the DEFS Files

There are many common symbolic names that occur in almost every OS-9 assembly language program. For example, each system call has a name such as “I$READ”. Although you can include definitions using EQU statements in each program for only those system names it uses, it is generally more convenient to utilize the system definitions files supplied with OS-9 which are generally referred to as the “DEFS files”. They are so named because they are included in a directory called “DEFS”.

Using the DEFS files also minimizes chances of error and improves the maintainability of your programs. In the event a future release of OS-9 redefines a system data structure, for example, you need only to reassemble your original program with the DEFS files for the new release.

In addition to definitions of names of system calls, error codes, memory module formats, etc., the DEFS files also have convenience definitions for ASCII characters, condition code register bits, register names, etc.

The DEFS files are not sacred - you should feel free to add your own definitions when and where. you want. You should note your additions, however, so they can be carried easily to future release editions.

To include the main DEFS file in your program, include the following assembler statement at the beginning of your program:

```
USE /D0/DEFS/OS9DEFS
```

This tells the assembler to include this file with your source code when assembling the file. It is recommended to use conditionals (IFP1) with the USE statement to speed up assembly and to prevent the `OS9Defs` file from being printed out in your listing every time.

There are individual DEFS files that generally correspond to the part of the system you are dealing with. For example, definitions used by the SCF file manager-related functions are contained in a file called “`SCFDefs`”. You would include this file in your program if you were writing an SCF-type device driver. In the following pages, each of the major DEFS files is discussed in detail to assist you in selecting appropriate files to include with each of your programs.

NOTE: The actual names of DEFS files may vary from the generic names. given in this manual according to the level and release number of your system. For example, your “os9defs” file may actually be named “os9defs.lii” on a Level II system, etc.

2.10.1. The OS9Defs File

The most commonly used DEFS is the `OS9Defs` file which contains general system wide definitions. This file contains:

System Service Request Code Definitions
Signal Codes
Status Codes For Getstat/Setstat
Direct Page Variables
Table Sizes
Module Format & Offsets
Module Field Definitions
Module Type/Language Masks & Definitions
Module Attributes/Revision Masks & Definitions
Process Descriptor Format & Offsets
Process Status Flags
OS-9 System Entry Vectors
Path Descriptor Offsets
File Access Modes
Pathlist Special Symbols
File Manager Entry Offsets
Device Driver Entry Offsets
Device Table Format & Offsets
Device Static Storage Offsets
Interrupt Polling Table Format & Offsets
Register Offsets on Stack
Condition Codes
System Error Codes

I/O Error Codes

System Service Request Code Definitions

The main purpose of the DEFS files is to define all the OS-9 system calls with their associated values. This is the main purpose of the `OS9Defs` file. It enables you to use the system request name in an OS9 call. All OS-9 service request codes are listed (i.e., user, system and I/O function requests);

Signal Codes

This group of labels define the four signals defined and used by OS-9 with their associated values. User defined signals should be added here.

Status Codes For Getstat/Setstat

When using the OS-9 system calls I$SETSTT and I$GETSTT there exist predefined status call functions. Those supported by OS-9 file managers and device drivers are listed in this area with their values. These labels are then available to be used for loading the 'B' register before the call is made.

Direct Page Variables

These labels define the offsets into page 0 of OS-9 system variables. Zero page variables are used by OS-9 for interrupt vectors, table addresses, process queues and internal memory information. It is *strongly* recommended that you don't use the page 0 variables in your programs. These variables are not physically accessible to user programs in Level II. They are given in `OS9Defs` for those who might need to write special drivers and interrupt handlers, or need to do system debugging. Improper use of these variables can cause unexpected and perhaps fatal system operation.

Table Sizes

These equates define the size of the table used under the OS-9 operating system. This information is used internally to OS-9 and its value is in defining the size of this table.

Module Format & Offsets

These labels define the offsets into a module header of all OS-9 compatible modules. Module offsets can be used by programmers to find information in a module (i.e. module size, name, type, language etc.). This area has the Universal Module offsets and the offsets for specific module types. This is due to the fact that descriptors, drivers, programs and file managers have a different module format.

Module Field Definitions

Module Type/Language Masks & Offsets

Module Attributes/Revision Masks and Offsets

This is the area where the different bits of information that go into a module header are defined to be of use to those who need to decode. a module header and possibly modify one. Since the Assembler generates a module header for you using the 'mod' and 'emod' statements, the value of this area will be when you do need to read a header. Masks are defined in this area for the type, language, attribute, and revision bytes for accurate and understandable masking. Also, the different values for each field are listed here for making comparisons.

Process Descriptor Format

Process Status Flags

In this area are the definitions and offsets for the process descriptor and the status flag values. Again, for most programmers, this information will be of little programming use. This information will be primarily of educational value for those who are interested in the internal workings of OS-9. The process descriptor is the table of information describing a process. The status flags are the definitions for the flags used by OS-9 to mark a process for different states (i.e. dead, sleeping etc.).

OS-9 System Entry Vectors

OS-9 system entry points are defined in this area of `OS9Defs`. These are the vector addresses for the different types of interrupts. These are pseudo vectors, not the actual hardware vector points. For more information, see the *System Programmer's Manual*.

Path Descriptor Offsets

The offsets for OS-9 path descriptors.are defined in this area. Path descriptors are created by OS-9 for every path opened in the system. This is again information used mainly by the system and will be of little value to most users except to garner an understanding of OS-9.

File Access Modes

These are the definitions for the file access modes under OS-9. The definitions will mainly be used for I$CREATE and I$OPEN system calls which require the file attributes to be set at the time of the call. See the *System Programmer's Manual* and the *User's Guide* for more information.

Pathlist Special Symbols

This area lists the definitions of special pathlist characters. These will be used by those programmers who need to parse.out a pathlist. This would be a good area to add special characters that you frequently use in your programs.

File Manager Entry Offsets

All file managers on an OS-9 system have the entry offsets defined in this area. Programmers who plan on writing their own file manager will need to provide these entry offsets. Those who have no ambition to write a file manager will most likely have no use for these offsets.

Device Driver Entry Offsets

Like file managers, all drivers have their own set of entry offsets. Programmers will need to provide these offsets at the beginning of their drivers. Once again, refer to the *System Programmer's Manual*.

Device Table Format

OS-9 keeps an entry in a table for every active device in the system. The form of the table entry is defined in this area. This information is used internally to the operating system. The format is provided here for those who are curious as to the operation of OS-9. The *System Programmer's Manual* contains a discussion on how OS-9 handles I/O.

Device Static Storage Offsets

Every active device also has associated with it a static storage area that contains information about the device and is filled in when the device is activated. The actual filling in of the parameters is done by three sources; IOMAN, the file manager, and the device driver. The offsets listed in this area are filled in by IOMAN. For more information on the rest of these offsets, look in `SCFDefs`, `RBFDefs`, your driver sources, and the *System Programmer's Manual*.

Interrupt Polling Table Format

The polling table is formed from entries having the structure defined here. It contains all the information the interrupt service routine needs to handle interrupts generated by active devices. This information is used internally by OS-9 and is, in general, of no programming value. The *System Programmer's Manual* has more information on interrupt servicing.

Register Offsets on Stack

Any time the 6809 CPU gets an interrupt of the form NMI, IRQ, SWI, SWI2 (an OS-9 system call), or SWI3, the registers are pushed on the stack. The offset to those registers are defined in this area of

`OS9Defs`. This information will be of particular value to those who write drivers and need to get the I$GETSTT or I$SETSTT codes. These could also be used to pass parameters on the stack to different procedures in a program.

Condition Code Bits

Every program written will at some point need to twiddle with the condition code bits. In order to make the programs more legible there are defined in this area the values for each condition code. By using these masks one can set or reset the bits as needed. It is good programming practice to use these labels in your code.

System Error Codes

I/O Error Codes

The final entry in `OS9Defs` is the error code definitions. These labels define all the errors returned by OS-9 and the I/O handlers. If your programs have any form of error trapping you will need to compare the error to a-known error definition in order to determine what should occur. It is in this area where they are defined. For information on what a specific error code means when it is returned, refer to the *User's Guide* and the *System Programmer's Manual*.

2.10.2. The SCFDefs File

Another DEFS file included is `SCFDefs`. This file contains the definitions pertaining to the sequential file manager and sequential file devices. Specifically it contains:

Static Storage Requirements
Character Definitions
File Descriptor Offsets

`SCFDefs` will be used when writing drivers for sequential devices and managers. It is also the area to add your own SCF-type definitions.

2.10.2.1. Static Storage Requirements

This area defines the offsets to the static storage required by SCF devices. This area continues from V.USER defined in `OS9Defs`. SCF devices must reserve this space for the SCF manager; The storage reserved after this group is determined by the driver. For information on SCF static storage refer to the *System Programmer's Manual*.

2.10.2.2. Character Definitions

Certain SCF devices will at some time need to filter special characters. These could be X-ON or X-OFF characters for example. This is the area where these symbols are defined. This would be a good area to add your own SCF special characters.

2.10.2.3. File Descriptor Offsets

The last entry in `SCFDefs` is the file descriptor offsets for SCF devices. The actual total storage is declared in `OS9Defs` under the entry called Path Descriptor Offsets. Both SCF and RBF have their own definitions of the PD.FST and PD.OPT fields. This area is where SCF's definitions are located. Refer to the *System Programmer's Manual* for more information on descriptors.

2.10.3. The RBFDefs File

`RBFDefs` is the parallel to `SCFDefs` but for random block file managers and devices. This file includes:

Random Block Path Descriptor Format

State Flags
Device Descriptor Format
File Descriptor Format
Segment List Entry Format
Directory Entry Format
Static Storage

This DEFS file will be used when writing random block device drivers and managers. This is also the area to add your own RBF type definitions.

2.10.3.1. Random Block Path Descriptor Format

This entry defines the file descriptor offsets for RBF devices. The actual total storage is declared in `OS9Defs` under the entry called Path Descriptor Offsets. Both SCF and RBF have their own definitions of the PD.FST and PD.OPT fields. This is where RBF's definitions are located. Refer to the *System Programmer's Manual* for more information on descriptors.

2.10.3.2. State Flags

The flags defined here are used internally by OS-9 to mark the state of the disk buffer. These are of little programming value but are supplied for those who are interested in the workings of the operating system. For more information on file handling read the *System Programmer's Manual*.

2.10.3.3. Device Descriptor Format

This is the format of what goes into sector zero of an RBF device and is what RBF uses to find the actual physical information on the device. This information is used to fill in the drive table by OS-9. RBF devices differ from SCF type devices in that the actual device information is kept on the media. The device descriptor in memory is then mainly used by the format program. Refer to the *System Programmer's Manual* for more information on the subject.

2.10.3.4. File Descriptor Format

The format defined here is kept on disk and contains information about the file (i.e. size, segment list, owner etc.). The information is read in by RBF and is used when accessing a file. Any time the file is modified this sector is modified. Again this information is provided for those who are curious as to the structure of RBF device files. For more information read the *System Programmer's Manual*.

2.10.3.5. Segment List Entry Format

The segment list is the area that tells the sector extensions of a file. The actual list is composed of the beginning sector and the size (in number of sectors) of each segment of the file. Files that are extended have another segment added to the segment list in the file descriptor sector above. This information is of little programming value to most users. Refer to the *System Programmer's Manual*.

2.10.3.6. Directory Entry Format

Every file in a directory has an entry of the format described here. Only two pieces of information exist here, the file name and the file descriptor sector address. For information on directories, again refer to the *System Programmer's Manual*.

2.10.3.7. Static Storage

The final entry in `RBFDefs` is the static storage requirement for the drive tables. The drive tables are allocated by the driver and have a size and format of that shown here. They begin at DRVBEG and continue to DRVMEM. Also V.NDRV is allocated before the tables and defines the number of drives and therefore the number of tables used by a driver. The rest of the static storage is defined in `OS9Defs` and in the driver. This information will be used by those programmer's who need to write their own RBF device driver. The *System Programmer's Manual* has more information on the subject.

2.10.4. The SysType File

The `SysType` file is a file that describes the physical (hardware-dependent) parameters of the various types of OS-9-based systems. Some of those parameters are:

CPU Type Definitions
Memory Management Unit Definitions
CPU Speed Definitions
Disk Controller Definitions
Clock Module Definitions
PIA Type Definitions
System Type Definitions
Disk Port Address
Disk Definition
Disk Parameters
Clock Port
I/O Port

This DEFS file is relatively straightforward. The main use of this file will be with those who are writing or modifying their drivers. All the necessary physical information is supplied here, and no discussion is needed.

Chapter 3. Interactive Debugger

3.1. Introduction

The Interactive Debugger is a powerful tool for system diagnostics or testing 6809 machine language programs. It is also useful when you need to directly access the computer's memory for any of a number of reasons: testing I/O interfaces, verifying data, etc. The calculator mode can simplify computation of addresses, radix conversion, and other mathematical problems often encountered by machine language programmers.

3.1.1. Installation

The Interactive Debugger program is supplied on a file called “DEBUG” which should be located in the system's execution directory (usually “/D0/CMDS”) .

3.1.2. Calling the Interactive Debugger

After the installation procedure, given above, has been performed, the Interactive Debugger may be executed from OS-9 by typing:

```
OS9: debug
```

3.1.3. Basic Concepts

The debugger operates in response to single line commands typed in from the keyboard. You can tell when you are communicating with_the debugger because it always displays a “DB:” prompt when it expects a command line.

Each line is terminated by a RETURN. If you make a mistake while typing, you can use the backspace key, or you can delete the entire line using the CONTROL+X key.

Each command line starts with a single character command which may be followed by text or one or two arithmetic expressions depending on the specific command. Uppercase and lowercase characters can be used interchangeably. Here's an example of the “space” command which displays the result of an expression in hexadecimal and decimal notation:

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

Numbers entered into or displayed by the debugger are in hexadecimal notation unless special commands are used, such as the decimal conversion command shown above.

Important note

The exact format of displays generated by some of the commands may be different than the examples in this book. This is due to customization for the screen size of specific computers.

3.2. Expressions

A powerful feature of the Interactive Debugger is its integral expression interpreter, which permits you to type in simple or complex expressions wherever an input value is called for in a command. Expressions used by the Interactive Debugger are similar to those used with high-level languages such as BASIC, except there are some extra operators and operands that are unique to the debugger.

Numbers used in expressions are 16 bit unsigned integers, which is the 6809's “native” arithmetic representation. The allowable range of numbers is therefore zero 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 an error message will be given if the result of the expression is too large to be stored in a byte (when the result > 255). Also, some operands are only a byte long (such as individual memory locations and some registers). These are automatically converted to 16-bit “words” without sign extension. Spaces may be used between operators and operands as desired to improve readability but do not affect evaluation.

3.2.1. Constants

Constants can be in base 2 (“binary”), base 10 (“decimal”), or base 16 (hexadecimal or “hex”). Binary and decimal constants require a prefix character: % (binary) or # (decimal). All other numbers are assumed to be hex. Hex numbers may also have an optional $ prefix. Here are some examples:

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

Character constants may also be used. A single quote ' for one character constants and a double quote " for two character constants. These produce the numerical value of the ASCII codes for the character(s) which follow. For example:

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

3.2.2. Special Names

There are other legal operands in expressions; “Dot”, “Dot-Dot”, and register names.

“Dot” is simply the debugger's current working address in memory. It can be examined, changed, updated, used in expressions, or recalled. It has the main effect of eliminating a tremendous amount of memory address typing. The following command prints the current working address:

```
DB: . RETURN
    2204 82
```

“Dot-Dot” is the value of “Dot” *before* the last time it was changed. It is convenient to use if you change dot to an incorrect value, or as a second address “memory”. For example, the following command prints the value of “Dot-Dot”:

```
DB: . 400 RETURN                (set Dot to 400)
DB: . 800 RETURN                (change Dot to 800)
DB: .. RETURN                   (display Dot-Dot)
    0400 12
DB: . .. RETURN                 (change Dot to Dot-Dot)
```

3.2.3. Register Names

MPU Registers may be specified by a colon character “:” followed by the mnemonic name of the register; for example:

: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 register
:CC	Condition Codes Register

The values returned are of the program under test's registers, which are “stacked” when the debugger is active. Those registers which are a single byte long are promoted to a word when used in expressions.

NOTE: When a program is interrupted by a breakpoint, the SP register will be pointing at the bottom of the MPU register stack.

3.2.4. Operators

Operators specify arithmetic or logical operations to be performed within an expression. The operators having a higher precedence (priority) are executed before those having lower precedence. For example, all multiplications are performed before additions. Operators in a single expression having equal precedence are evaluated left to right. Parentheses may be used to override precedence as desired. Here are the operators, in precedence order:

+ addition	- subtraction	<- Lowest
* multiplication	/ division	
& logical AND	! logical OR	
- negative		<- Highest

3.2.5. Indirect Addressing

The Indirect Addressing function returns the data at the memory address using a value (expression, constant, special name, etc.) as the memory address. The Interactive debugger has two indirect addressing modes:

< `expr` >	returns the value of a memory *byte* using the value of “`expr`” as an address
[`expr`]	returns the value of a 16-bit *word* using the value of “`expr`” as an address

Here are some examples:

<200>	returns the value of the byte at address 200
[:X]	returns the value of the word pointed to by the X register
[.+10]	returns the value of the word at address dot plus 10.

3.2.6. Forming Expressions

An expression can be composed of any combination of constants, register names, special names, and operators. Here are some examples:

#1024+#128	:X-:B-2	.+20
:Y*(:X+:A)	:U & FFFE	#125

3.3. Debug Commands

3.3.1. Calculator Command

To use the calculator command, enter a line starting with one or more spaces followed by any legal expression, then “return”. The expression will be evaluated and the result displayed on the following. line in both hexadecimal and decimal representations.

Here are some examples:

```
DB:SPACE 5000+200 RETURN
    $5200 #20992

DB:SPACE 8800/2 RETURN
    $4400 #17408

DB:SPACE #100+#12 RETURN
    $0070 #00112
```

These commands are also handy for converting values from one representation to another:

```
DB:SPACE %11110000 RETURN
    $00F0 #00240

DB:SPACE 'A RETURN
    $0041 #00065

DB:SPACE #100 RETURN
    $00C4 #00100
```

You can also use indirect addressing to look at memory without changing Dot:

```
DB:SPACE <.> RETURN
    $004F #00079
```

Another trick is simulating 6809 indexed or indexed indirect instructions. For example:

```
DB:SPACE [:D+:Y] RETURN
```

is the same as the assembly language syntax [D,Y].

3.3.2. “Dot” and Memory Examine/Change Commands

3.3.2.1. Display Dot Command

Typing “.” causes the current value of Dot and the contents of that memory address to be displayed, for example:

```
DB:. RETURN  display Dot
2201 B0
```

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

3.3.2.2. Change Dot Command

To change the value of Dot, type a period followed by an expression which is to be the new value, for example:

```
DB:. 500 RETURN set Dot to 500
    500 12
```

One additional feature: whenever Dot is changed; its last value is saved, and can be restored by typing two periods:

```
DB:. RETURN display Dot
    1000 23 .
DB:. 2000 RETURN set Dot to 2000
    2000 9C
DB:.. RETURN restore old value of Dot
    1000 23
```

3.3.2.3. Advance Dot Command

Typing a line with just RETURN, advances Dot to the next memory address and prints its new value and contents. The example below shows how to “step through” sequential memory locations.

```
DB:RETURN
    2202 05
DB:RETURN
    2203 C2
DB:RETURN
    2204 82
```

3.3.2.4. Backstep Dot Command

The “-” command backs up Dot one address and prints its value and contents:

```
DB:. RETURN display Dot
    2204 82
DB:- RETURN backstep one address
    2203 C2
DB:- RETURN backstep one address
    2202 05
```

3.3.2.5. Change Memory Command

To change the contents of the memory-location pointed to by Dot, type an equal sign followed by an expression. The expression is evaluated, and the result stored at the current Dot address. Dot is then advanced to the location whose address and contents are displayed.

The memory location is checked after the new value is stored to make sure it actually changed to the correct value. If it didn't, an error message is displayed. Error messages are displayed when an attempt is made to alter non-RAM memory. In particular, many 6800-family interface devices (such as PIAs, ACIAs, etc.) have registers that will not read the same as when written to.

```
DB:. RETURN display Dot
    2203 C2
DB:=FF RETURN change memory to FF
    2204 01
```

```
DB:- RETURN backspace and display
    2203 FF
```

Warning

This command permits any memory location to be changed. You can accidentally crash the debugger, the program under test, or OS-9 if you incorrectly change their memory areas.

3.3.3. Register Examine/Change Commands

Several forms of the “:” register command can be used to examine one or all registers, or to change a specific register's contents.

The “registers” affected by these commands, are actually stored copies in memory (“images”) of the register values of_the.program under test. They are stored on the stack when the program is stopped by a breakpoint or other interrupt. The value shown for the stack pointer register is the address ,of the stack where the register values are stored. Before the program under test is first executed, the register images must be made valid using the “E” command.

When a running program is interrupted by a CONTROL+C key or by a breakpoint, the registers are once again stacked for inspection and/or change. Note that if you manually alter the value of the SP register during program testing, the address of the stack and the register values will also change. If you change the condition codes (CC) register, the E bit must be kept set to ensure all registers are reloaded properly when program execution is resumed.

3.3.3.1. Display All Registers Command.

To display all registers, type “:” followed by RETURN. For example:

```
DB:: RETURN
     SP  CC  A  B DP   X    Y    U   PC
    C499 C4 20 1C 01 DD3E 239A 0000 240C
```

3.3.3.2. Display Specific Register Command

To display the contents of a specific register, enter a colon ": followed by the register name. The Debugger will respond by displaying the current register contents in hex. Examples:

```
DB::PC RETURN display PC register
    C499
DB::B RETURN display B register
    007E
DB::SP RETURN display SP register
    42FD
```

3.3.3.3. Change Register Command

To assign a new value to a register, type the register name followed by an expression. The expression is evaluated and stored in the register specified. When 8-bit registers are named, the expression given must have a value that fits in a single byte, or an error message is displayed and the register is not changed.

Here are some examples:

```
DB::X #4096 RETURN Set x register to 4096

DB::DP 0        Set DP register to 0
```

```
DB::D 24CF+:Y  Set D register to 24CF plus contents of Y register
```

3.3.4. Breakpoint Commands

Breakpoints allow you to specify address(es) where execution of the program under test is to be suspended and the debugger re-entered. When a breakpoint is encountered, the values of the MPU registers and the “DB:” prompt will be displayed. After a breakpoint is reached,-registers can be examined or changed, memory can be altered, or any other debugger command can be used. Program execution can be resumed from the breakpoint location using the “G” command. Breakpoints may be inserted at up to 12 different addresses.

The debugger uses the 6809 SWI instruction for breakpoints. They are inserted in memory in place of the machine language instruction opcodes which are saved for restoration later. The SWI instructions are automatically inserted and removed by the debugger at the right time so you will not see them in memory.

When a SWI is executed it interrupts.the program and saves the register contents .on the stack so they can be examined or changed using the “:” command. Because SWIs operate by temporarily replacing an instruction opcode, there are three restrictions on their use:

1. Breakpoints cannot be used with programs in ROM.
2. Breakpoints must be located in the first (opcode) byte of the instruction.
3. User programs cannot utilize the SWI instruction for other purposes (but SWI2 and SWI3 can be used).

When the breakpoint is encountered during execution of the program under test, the debugger is reentered and the program's register contents is displayed using the same format as the “display register” command.

3.3.4.1. “B” Set or Display Breakpoints Command

The B command-will insert a breakpoint if followed by an expression, or will display all present breakpoint addresses if used alone.

```
DB:B 1C00 RETURN set breakpoint at 1C00
DB:B 4FD3 RETURN set breakpoint at 4FD3
DB:. RETURN display Dot
    1277 39
DB:B . RETURN set breakpoint at Dot (1277)
DB:B RETURN display current breakpoints
    1C00 4FD3 1277
```

3.3.4.2. “K” Remove Breakpoint Command

The K command removes (“kills”) a breakpoint at a specific address if followed by an expression or ALL (caution!) breakpoints if used alone;

```
DB:B RETURN                    display current breakpoints
    1C00 4FD3 1277
DB:K 4FD3 RETURN               kill breakpoint at 4Fd3
DB:B RETURN                    display current breakpoints
    1C00 1277
DB:K RETURN                    kill all remaining breakpoints
DB:B RETURN                    display current breakpoints
                                none left so display is blank
```

```
DB:
```

3.3.5. Program Setup And Run Commands

3.3.5.1. Prepare To Execute Command

The Execute command consists of a “E” by a Shell-style command line. It prepares for testing of a specific program module and must be used before the program is run.

The “E” command performs the rough equivalent of what the OS-9 **Shell** does to start a program except for I/O redirection (<, >, >>), memory size override (“#”), and multitasking (“&”) functions. It sets up a stack, parameters, registers, and data memory area in preparation for execution of the program to be tested. The “G” command is actually used to start running the program.

Note that this command will allocate program and data area memory as appropriate. The new program uses the debugger's current standard I/O paths, but can open other paths as necessary. In effect, the debugger and the program become coroutines.

This command is acknowledged by a register dump showing the program's initial register values. The “G” command is used to begin actual program execution. The “E” command will set up the MPU registers as if you had just performed an F$CHAIN service request as shown below:

```
high    +---------------+ <-- Y
        !               !
        !    parameter  !
        !      area     !
        !               !
        +---------------+ <-- X, SP
        !               !
        !               !
        !   data area   !
        !               !
        !               !
        +---------------+
        !  direct page  !
low     +---------------+ <-- U, DP

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

Example:

```
DB:E myprogram
      SP   CC   A   B DP   X    Y    U    PC
     0CF3 C8 00 01 0C 0CFF 0D00 0C00 9214
DB:
```

3.3.5.2. Go To Program Command

The G (“Go”) command is used to resume execution of a program under test under any of the following circumstances:

1. After the “E” command has been used to prepare a new program for execution.
2. To resume program execution after a breakpoint.
3. To resume execution after a CONTROL+C interrupt.

If the “G” command is used to continue execution after a breakpoint that has not been removed, the breakpoint will not be reinserted so the program will not “hang-up”. This features allows you to execute breakpoints within loops without continually inserting and removing them if two breakpoints are placed at different addresses within the loop.

You can optionally put an expression after the “G”. The result will be assigned to the PC register just before execution is resumed in order to change the re-start address. Here is an example:

```
DB:E myprogram RETURN      prepare for execution
    SP   CC  A  B  DP  X    Y    U    PC
    0CF3 C8 00 01 0C 0CFF 0D00 0C00 9214
DB:B 9466 RETURN           put breakpoint at 9466
DB:G RETURN                start program
    BKPT:
    SP   CC  A  B  DP  X    Y    U    PC
    0CEA C0 FF 01 0C 4000 0D00 0C00 9466
DB:G RETURN                resume after breakpoint
```

3.3.5.3. “L” Link To Module Command

This command (L followed by text) attempts to link to the module whose name is given in the text line. If successful, Dot is set to the address of the first byte of the program and is displayed. This command is commonly used to find the starting address of an OS-9 memory module.

Example:

```
DB:L gotoxy RETURN   link to module “gotoxy”
    EC00 87
DB:
```

3.3.6. Utility Commands

3.3.6.1. Clear and Test Memory Command

The “C” command followed by TWO expressions simultaneously performs a “walking bit” memory test and clears all memory between the two evaluated addresses. The first expression gives the starting address, and the second the ending address (which must be higher). If any byte(s) fail the test, its address is displayed. Of course, only RAM memory can be tested and cleared.

Warning

This command can be dangerous for obvious reasons. Be sure of what memory you are clearing.

Examples:

```
DB:C 1200 15FF RETURN          test from 1200 to 15FF
    12E4                         indicates bad memory at 12E4 and
    12E7                         12E7
DB:C . .+256 RETURN            test from Dot to Dot+256
DB:                           indicates no bad memory
```

3.3.6.2. Dump Memory Command

The M command, which is also followed by two addresses, displays a screen-sized display of memory contents in tabular form in both hexadecimal and ASCII form. The starting address of each line is printed 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. Those locations containing nondisplayable characters have periods in their place. The high order bit is ignored for the display of ASCII characters. For example:

```
DB:m f100 f17f RETURN

F100 6225 306C 6120 ED4F 5FED 62ED 6439 E680 b%01a .O_.b.d9..
F110 C130 2504 C139 2303 1A01 39C0 301C F339 .0%..9#...9.0..9
F120 301F 6D61 2706 EC62 1CFE 2004 1A04 1A01 0.ma'..b.. .....
F130 3264 391C FB20 F7A6 8081 2027 FA30 1F39 2d9.. .... '.0.9
F140 3416 A663 3D34 06A6 62E6 643D 3406 A664 4..c=4..b.d=4..d
F150 E667 8D13 A665 E666 8D0D 1CFE EC62 AEE4 .g...e.f.....b..
F160 2702 1A01 3268 393D E363 ED63 2402 6C62 '...2h9=.c.C$.1b
F170 3934 36EC E426 041A 0120 20CC 0010 E764 946..&...  ....d
```

3.3.6.3. Search Memory Command

The “S” command is used to search an area of memory for a one or two byte pattern. The search begins at the present Dot address. The “S” is followed by two expressions: the first expression is the ending address of the search, and the second expression is the data to be searched for. If this value is less than 256, a byte comparison is used, otherwise two bytes are compared. If a matching pattern is found in memory, Dot is set to the address where it was located (which is displayed). If no match occurred, another “DB:” prompt is displayed.

3.3.6.4. Shell Command

This command calls the OS-9 “shell” to execute one or more system command lines. Its format is a dollar sign optionally followed by a shell command line. If the command line is given, the shell will execute just that line and return back to the debugger. If the dollar sign is immediately followed by an end-of-line, the shell will print prompts for one or more command lines in its usual manner. You can return to the (undisturbed) debugger by typing an end-of-file character (usually ESCAPE).

This command is useful for calling the system utility programs and the Interactive Assembler from within the debugger. For example:

```
DB:$dir RETURN

DB:$unlink mypgm; mdir e; load test5 RETURN

DB:$asm myprogram o=myprogram.bin RETURN
```

3.3.6.5. “Q” Quit Debugger Command

This command (Q) causes the Interactive Debugger to terminate and return to OS-9 or the program that called the debugger.

Example:

```
DB: Q RETURN
OS9:
```

3.3.7. Using The Debugger

The Interactive Debugger is mostly used for one of three purposes:

- To test system memory and I/O devices

- To “patch” the operating system or other programs
- To test hand-written or compiler-generated programs.

The simple assembly language program shown below is used in some of the examples in this chapter to illustrate how debug commands are be used with a real program. The 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
0052                     RMB   150          HARDWARE STACK
00E7            STACK    EQU   .-1
00E8            DATMEM   EQU   .            DATA AREA MEMORY SIZE

                * Program Section
0000 87CD0047            MOD   ENDPGM,NAME,$11,$81,ENTRY,DATMEM
000D 4558414D   NAME     PCS   /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$WRLN       WRITE THE LINE
0021 2512                BCS   ERROR        BRA IF ANY ERRORS
0023 3042                LEAX  INPBUF,U     ADDR OF INPUT BUFFER
0025 108E0050            LDY   #80          MAX OF 80 CHARACTERS
0029 8600                LDA   #0           STANDARD INPUT PATH
002B 103F88              OS9   I$RDLN       READ THE LINE
002E 2505                BCS   ERROR        BRA IF ANY I/O ERRORS
0030 109F00              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
```

3.3.7.1. A Session With The Debugger

Below is an example of how DEBUG might be used with the sample program on the previous page. DEBUG is called from OS-9, the $ command is used to tell SHELL to load “EXAMPLE” into memory, the “L” command is used to link to it, etc.

```
OS9:DEBUG RETURN                        run Debug program

Interactive Debugger
DB:$LOAD /D1/EXAMPLE RETURN             load test program
DB:L example RETURN                     find program start addr.
    9200 87
DB:. RETURN                             display Dot; L set it to
    9200 87                              start address
DB:M . .+44 RETURN                      dump program code
```

```
9200 87CD 0047 000D 1181 9300 1400 E845 5841 ...D.........EXA
9210 4D50 4CC5 308D 0020 1083 000C 8601 103F MPL.0.. ....,..?
9220 8C25 1230 4210 8800 5086 0010 3F8B 2505 .%.0B...P...?.%.
9230 109? 00C6 0010 3F06 4845 4C4C 4F20 574F ......?.HELLO WO
9240 524C 440D A484 7F8D D4A6 A02A F639 3432 RLD........*.942

DB:E EXAMPLE RETURN                    prepare to run program
      SP  CC  A  B  DP  X    Y    U    PC
     0DF3 C8 00 01  0D 0DFF 0E00 0D00 9214

DB:B .+ZE RETURN                       set breakpoint at 9223
DB:G RETURN                            run program
HELLO WORLD
hello computer

    BKPT:                               breakpoint encountered
     SP  CC  A  B  DP  X    Y    U    PC
    0DF3 C0 00 01  0D 0D02 000F 0D00 922E

DB:M :U :U+20 RETURN                   display data area

0D00 FA31 6865 6C6C 6F20 636F 6D70 7574 6572 .1hello computer
0D10 0DDF C005 E9F1 95FA 4C0D 1DFA 0AC4 5900 ........L.....Y.
0D20 0B64 360B CFB1 0091 F820 SAE2 5AF8 5AF8 .d6...... Z.Z.Z.

DB:. :U+2 RETURN                       display relative data
    0D02 68                             area offset 2
DB:RETURN                              step through data area
    0D03 65
DB:RETURN
    0D04 6C
DB:RETURN
    0D05 6C
DB:Q RETURN                            quit debugging
OS9:
```

3.3.7.2. Patching Programs

“Patching” (changing the object code of) a program involves four steps:

1. Loading the program into memory using OS-9's “LOAD” command.
2. Changing the program in memory using the Debugger's “L” and “=” commands.
3. Saving the new, patched version of the program on a disk file using the OS-9 “SAVE” command.
4. Updating the program module's CRC check value using the OS-9 “VERIFY” command.

The fourth step is unique to OS-9 (as compared to other operating systems) and often overlooked. However, it is essential because OS-9 will refuse to load the patched program into memory until its CRC check value is updated and correct.

The example that follows shows how the program listed on page 4-1 is “patched” - this case changing the “LDY #80” instruction to “LDY #32”.

```
OS9:load example RETURN         load program to be patched
OS9:debug RETURN                run debug
```

```
Interactive Debugger
DB:L EXAMPLE RETURN              set dot to start addr of program
    2000 87                     note module starts at 2000
DB:. .+28 RETURN                 add offset of byte to change
    2028 50                     current value is 50
DB:=#32 RETURN                   change to decimal 32
    2028 10                     change confirmed
DB:Q RETURN quit debugger

OS9:SAVE TEMP EXAMPLE RETURN     save patched module on file TEMP
OS9:VERIFY <TEMP >EX2 RETURN     update CRC and copy to file EX2
OS9:DEL TEMP                    TEMP no longer needed
```

After the above procedure has been completed, the file EX2 contains a patched version of the module with a correct module CRC value. It can be run and/or loaded into memory as desired.

3.3.7.3. Patching OS-9 Component Modules

Patching modules that are part of OS-9 (modules contained in the "`OS9Boot`" file) is a bit trickier than patching regular program because the "COBBLER" and "OS9GEN" programs must be used to create a new "`OS9Boot`" file. The example below shows how an OS-9 "device descriptor" module is permanently patched, in this case to change the uppercase lock of the device "/TERM" from "off" to "on". This example assumes a copy of the system disk is loaded in drive one ("/D1").

Caution

Always use a *copy* of your OS-9 system disk when patching it in case something goes wrong!

NOTE: SOME LEVEL TWO SYSTEMS DO NOT PERMIT SYSTEM MODULES TO BE PATCHED - IN ORDER TO BE CHANGED, THEY MUST BE REASSEMBLED AND INCLUDED IN A NEW BOOT DISK.

```
OS9: debug RETURN            run debug

Interactive Debugger
DB: L TERM RETURN            set dot to addr of TERM module
    D300 87                 (actual address will vary)
DB: . .+13 RETURN            add offset of byte to change
    D313 00                 current value is 00
DB: =1 RETURN                change value to 1 for "ON"
    D313 01                 change confirmed
DB: Q RETURN                 exit debugger

OS9:COBBLER /D1 RETURN       write new bootfile on /D1
OS9:VERIFY </D1/OS9BOOT >/D1/TEMP U RETURN   update CRC value
OS9:OS9GEN /D1 RETURN        write new bootfile again
TEMP RETURN                  name of file with good CRCs
ESCAPE                       End-of-file key for OS9GEN
```

Appendix A. Error Messages

A.1. Text Editor Error Messages

BAD MACRO NAME

> This error is caused by trying to close a macro definition, when the first line in the macro does not start with a legal macro name. The editor will allow you to close definition of a macro after you have given it a legal name. See the section on macro names.

BAD NUMBER

> An illegal numeric parameter has been entered. This is usually caused by entering a number that is larger than 65535.

BAD VAR NAME

> This error is caused by specifying a variable name that is illegal. Usually the variable name has been omitted, or you inadvertently included a "$" or "\#" character in the commands parameter list.

BRACKET MISMATCH

> This is caused by either having one too many left or right brackets (they must be used in pairs to repeat a command sequence). This error may also be caused by nesting the brackets too deeply.

BREAK

> This message is printed when you type a (CONTROL+C) or (CONTROL+Q) to interrupt whatever the editor is doing. After printing the error message, the editor will return to command entry mode. It is important to remember that the printout of edit command results may not be synchronized with the actual operation of a command.

DUPL MACRO

> This error is caused by trying to close a macro definition when there is another macro with the same name. The problem may be solved by renaming the macro before trying to close its definition.

END OF FILE

> This means that there is no more text remaining in the input file that is being read.

END OF TEXT

> This means that you have reached the end of the edit buffer. This is used only as a reminder.

FILE CLOSED

> This means that you tried to write to a file that was never opened. You should either specify a write file when starting up the editor from OS-9, or open an output file using the ".WRITE" pseudo macro.

MACRO IS OPEN

> You must first close the macro definition before using the command that caused this error.

MISSING DELIM

> The editor could not find a matching delimiter to complete the string that you specified. A string must be completely specified on a single line.

NOT FOUND

The editor can not find the string or macro that was specified in a command parameter.

UNDEFINED VAR

This error occurs when you try to use a variable that was not specified in the macros definition parameter list. A variable parameter may be used only in the macro in which it is declared. See the section of this manual on macros.

WHAT ??

The editor did not understand a command that you typed. This is usually caused by entering a command that does not exist (misspelling its name).

WORKSPACE FULL

This error is caused by entering a command that tried to insert more text into the buffer than there was room for. The.problem may be solved by increasing the workspace size using the “M” command, or by removing some text from the edit buffers.

A.2. Assembler Error Messages

BAD LABEL

The statement's label has an illegal character or does not start with A-Z or a-z.

BAD INSTR

The assembler did not recognize the instruction given in the source statement.

ADDRESS MODE

The addressing mode specified is not legal for the instruction.

OUT OF RANGE

The destination (label) of the branch is too far to use a short branch instruction (e.g. the a 16-bit offset using a LBRA-type instruction must be used).

REG NAM

The register name required is missing or misspelled.

REG SIZES

The registers specified in a TFR or EXG instruction were of different lengths (e.g., 8 bit vs. 16 bit).

INDEX REG

The name of an index register is required by the instruction but none was found.

] MISSING

A closing bracket was omitted (indirect addressing).

CONST DEF

The instruction requires a constant or an expression which is missing or in error.

LABEL NOT ALLOWED

This type of statement cannot have a label.

NEEDS LABEL

The statement is required to have a label.

IN NUMBER

A constant number (decimal, hex or binary) is too large or had an illegal character.

DIV BY 0

A division with a zero divisor was attempted within an expression.

MULT OVERFL

The result of a multiplication is greater than 65535 (two bytes).

EXPR SYNTAX

The arithmetic instruction is illegally constructed or is missing an operand following an operator.

PARENS

There is an unequal number of left and right parentheses in the expression.

RESULT>255

The result of the expression is too large to be represented in the one-byte value used by the instruction.

REDEFINED NAME

The label was defined previously in the program.

UNDEFINED NAME

The symbolic name was never defined in the program.

PHASING

The statement's label had a different address during the first assembly pass. This usually happens when an instruction changes addressing modes and thus its length after the first pass because its operand becomes defined *after* the source line is processed. Usually the error occurs on all labels following the offending source line.

MEMORY FULL

The symbol table became full - more memory is required to assemble the program.

OPT LIST

An illegal or missing option in the assembler command line or in an OPT statement.

INPUT PATH

A read error occurred on the input path.

OBJECT PATH

A write error occurred on the object file path.

CAN'T OPEN PATH

The file cannot be opened (source file) or created (object file).

A.3. Interactive Debugger Error Codes

0 ILLEGAL CONSTANT

The expression included a constant that had an illegal character or was too large (> 65535).

1 DIVIDE BY ZERO

A division was attempted using a divisor of zero.

2 MULTIPLICATION OVERFLOW

The product of the multiplication was greater then 65535.

3 OPERAND MISSING

An operator was not followed by a legal operand.

4 RIGHT PARENTHESIS

Right paren is expression missing: misnested parentheses.

5 RIGHT BRACKET MISSING

Misnested brackets.

6 RIGHT CARAT MISSING

Misnested byte-indirect (< and >).

7 INCORRECT REGISTER

Misspelled, missing or illegal register name followed the colon.

8 BYTE OVERFLOW

Attempted to store a value greater than 255 in a byte-sized destination.

9 COMMAND ERROR

Misspelled, missing or illegal command.

10 NO CHANGE

The memory location did not match the value assigned to it.

11 BREAKPOINT TABLE FULL

The maximum number of twelve breakpoints already exist.

12 BREAKPOINT NOT FOUND

No breakpoint exists at the address specified.

13 ILLEGAL SWI

A SWI instruction was encountered in the user program at an address other than a breakpoint.

Appendix B. Quick Reference

B.1. Editor Quick Reference Summary

. *macro_name* *parameters*	The "." command is used to execute a macro.
! *text*	Comment.
SPACE *text*	Insert the text line before the current edit pointer position.
RETURN	Move edit the pointer to the next line and display it.
+*n*	Move the edit pointer forward *n* lines and display.
-*n*	Move the edit pointer backward *n* lines and display.
+0	Move the edit pointer to the last character of the line.
-0	Move the edit pointer to the first character of the line and display it.
>*n*	Move the edit pointer forward *n* characters.
<*n*	Move the edit pointer backward *n* characters.
^	Move the edit pointer to the beginning of the text.
/	Move the edit pointer to the end of the text.
[commands] *n*	Repeat the sequence of commands between the two brackets *n* times.
:	Skip to the end of the innermost loop or macro if the fail flag is off, otherwise turn the fail flag off and resume execution.
A*n*	Set the SEARCH/CHANGE anchor to column *n*, restricting searches and changes to strings starting in the *n*th column.
A 0	Turn off the SEARCH / CHANGE anchor.
B *n*	Make buffer *n* the primary buffer.
C *n* *str1* *str2*	Change the next *n* occurrences of *str1* to *str2*.
D *n*	Delete *n* lines.
E *n* *str*	Extend (add the string to the end of) the next *n* lines.
G *n*	Get *n* lines from the secondary edit buffer starting from the top of the secondary buffer. The lines are inserted before the current position in the primary edit buffer.
I *n* *str*	Insert a line containing *n* copies of the *str* before the current edit pointer position.
K *n*	Kill *n* characters starting at the current edit pointer position.
L *n*	List (display) the next *n* lines from the current edit pointer position.
M *n*	Change workspace (memory) size to *n* bytes.

P `n`	Put (move) `n` lines from the present edit position in the primary buffer to the present edit position in the secondary buffer.
Q	Quit editing and return to OS-9.
Q	Terminate macro definition and return to the “E:” prompt.
R `n`	Read `n` lines from the buffer's input file.
S `n str`	Search for the next `n` occurrences of `str`.
T `n`	Tab to column number `n` of the present line.
U	Unextend (truncate) line at the current edit position.
V `n`	Turn verify mode (display text changes) off if `n` = 0, or on if `n` <> 0.
W `n`	Write `n` lines to the buffer's output file.
X `n`	Display `n` lines of text that precede the edit position.
.CHANGE `n str1 str2`	Similar to “C” command.
.DEL `str`	Delete the macro with the name specified by `str`.
.DIR	Display the editor's directory of buffers and macros.
.EOB	Test for end of buffer, if the edit pointer is at the end of the buffer succeed, otherwise fail.
.EOF	Test for end of file.
.EOL	Test for end of line. If the edit pointer is at the end of the line, this command will succeed, otherwise fail.
.F	Exit innermost loop or macro and set the fail flag.
.LOAD `str`	Load macros from the path specified by “`str`”.
.MAC `str`	Open the macro specified by “`str`” for definition. If an empty string is given, a new macro will be created.
.NEOB	Test for not end of buffer.
.NEOF	Test for not end of file.
.NEOL	Test for not end of line.
.NEW	Write lines to the output file up to the current line, then try to read an equal amount from the input file.
.NSTR `str`	Test if “`str`” does not match characters at the current edit position.
.READ `str`	Open an file for reading, using “`str`” as the pathlist.
.S	Exit the innermost loop or macro and succeed.
.SAVE `str1 str2`	Save the macro(s) specified in “`str1`” on the file specified by the pathlist in `str2`.
.SEARCH `n str`	Similar to the “S” command.

.SHELL `text`	Call OS-9 shell to execute the command line.
.SIZE	Display the size of memory used and the total amount of memory available in the workspace.
.STAR `n`	Test if `n` is equal to asterisk (infinity).
.STR `str`	Test if “`str`” matches the characters at the current edit position.
.WRITE `str`	Open an file for writing using “`str`” as pathlist.
.ZERO `n`	Test `n` to see if it is zero.

B.2. Interactive Debugger Quick Reference

SPACE `expr`	Evaluate expression and display result in hex and decimal
.	Print Dot address and contents
..	Restore last DOT, print address and contents
. `expr`	Set Dot to result, print address and contents
= `expr`	Set memory at Dot to result
-	Backup Dot, print address and contents
RETURN	Move Dot forward, print address and contents
:	Display all register contents
:reg	Display specific register contents
:reg `expr`	Set register to result
E `text`	Prepare for execution
G	Go to program
G `expr`	Go to program at result address
L `text`	Link to module named, print address
B	Display all breakpoints
B `expr`	Set breakpoint at result address
K	Kill all breakpoints
K `expr`	Kill breakpoint at result address
M `expr1` `expr2`	Dump memory
C `expr1` `expr2`	Clear and test memory
S `expr1` `expr2`	Search memory for pattern
$ `text`	Run Shell command line
Q	Quit debugging

Appendix C. Example Assembly Language Programs

C.1. Assembly Language Programming Examples

The following pages contain three assembly language programming examples. They are:

UpDn	-Program to convert input case to upper or lower.
PIA	-Parallel interface driver.
P	-Parallel interface descriptor.

These programs are given only as examples of assembly language programs and should not be considered as current system software.

Example C.1. UpDn - Assembly Language Programming Example

```
       *
       * this is a program to convert characters from
       *    lower to upper case (by using the u option)
       *    upper to lower case (by using no option)
       * the method of passing the parameters through
       *    os9 is used here (system calls)
       * to use type
       *  "updn u(opt for lower to upper) <'input' >'output'"
       *
                           nam UpDn
       * file include in assembly
                           ifp1
                           use /D0/defs/OS9defs
                           endc
       *
       * OS-9 System Definition File Included
       *
                           opt   l
                           ttl   Assembly Language Example
       *
       * module header macro
       *
  0000 87CD005C            mod   UDSIZ,UDNAM,TYPE,REVS,START,SIZE
  000D 757064EE   UDNAM    fcs   /updn/       module name for memory
  0011            TYPE     set   PRGRM+OBJCT mod type
  0081            REVS     set   REENT+1      mod revision
       *
       * storage area for variables
       *
D 0000            TEMP     rmb   1           temp storage for read
D 0001            UPRBND   rmb   1           storage for upperbound
D 0002            LWRBND   rmb   1           storage for lowerbound
D 0003                     rmb   250         storage for stack
D OOFD                     rmb   200         storage for parameters
D 01C5            SIZE     equ   .           end of data area
       *
       * actual code starts here
```

```
          * x register is pointing to Start of parameter area
          * y register is pointing to end of parameter area,
          * this is how to get a parameter that is passed on
          *    the command line and where to look for it
          *
0011             START    equ   *          start of executable
0011 A680        SRCH     lda   ,x+        search parameter area
0013 84DF                 anda  #$df       make upper case
0015 8155                 cmpa  #U         see if a U was input
0017 2703                 beq   UPPER      branch to set uppercase
0019 810D                 cmpa  #$0d       see if a carriage return
001B 26F4                 bne   SRCH       go get another char
          *
          * fall through to set upper to lower bounds
          *
001D 8641                 lda   #'A        get lower bound
001F 9702                 sta   LWRBND     set it in storage area
0021 865A                 lda   #'Z        get upper bound
0023 9701                 sta   UPRBND     set it in storage area
0025 2008                 bra   START1     go to start of code
          *
          * set lower to upper bounds
          *
0027 8661        UPPER    lda   #'a        get lower bound
0029 9702                 sta   LWRBND     set it in storage
002B 867A                 lda   #'z        get upper bound
002D 9701                 sta   UPRBND     set it in storage
          *
          * converting code
          * this part uses the I$READ and
          *    the I$WRIT system calls
          * read the systems programmers manual
          *    for information relating to them
          *
002F 30C4        START1   leax  temp,u     get storage address
0031 8600                 lda   #0         standard input
0033 108E0001             ldy   #$01       number of characters
0037 103F89      LOOP     os9   I$READ     do the read
003A 2515                 bcs   EXIT       exit if error
003C D600                 ldb   TEMP       get character read
003E D102                 cmpb  LWRBND     test char bound
0040 2506                 blo   WRITE      branch if out
0042 D101                 cmpb  UPRBND     test char bound
0044 2202                 bhi   WRITE      branch if out
0046 C820                 eorb  #$20       flip case bit
0048 D700        WRITE    stb   TEMP       put it in storage
004A 4C                   inca             reg 'a' stand output
004B 103F8A               os9   I$WRIT     write the character
004E 4A                   deca             return to stand input
004F 24E6                 bcc   LOOP       get char if no error
0051 C1D3        EXIT     cmpb  #E$EOF     is it an EOF error
0053 2601                 bne   EXIT1      not eof, leave carry
0055 5F                   clrb             clear carry, no error
0056 103F06      EXIT1    os9   F$EXIT     error returned, exit
0059 260409               emod             last command
005C             UDSIZ    equ   *          size of program
```

Example C.2. PIA - OUTPUT Parallel Interface Driver

```
                           NAM   PIA

                           ifpl
                           use   /D0/defs/os9defs
                           use   /D0/defs/scfdefs
                           use   /D0/defs/systype
                           endc

                           ttl   OUTPUT Parallel Interface Driver

       *************************
       * Device Driver for PIA Port

  0000 87CD00CD            mod   PIASIZ,PIAN,DRIVR+OBJCT,
                                          REENT+1,PIAENT,PIAMEM
  000D 07                  fcb   EXEC.+UPDAT.
  000E 5049C1     PIAN     fcs   "PIA"       MODULE NAME

  0011 02                  fcb   2           revision number
                           use   /D0/defs/copyright

    ************************************************************
    *                                                          *
    *         (C) 1981 Microware Systems Corporation           *
    *                                                          *
    ************************************************************

  0012 28432931            FCC   /(C)1981Microware/

D 000F                     ORG   V.SCF       STATIC STORAGE
D 000F            PIADDR   rmb   2           Pia True Port address
D 0011            PIAMEM   equ   .           TOTAL STATIC STORAGE

W 0022 160012     PIAENT   LBRA  PPINIT
  0025 1600A0              LBRA  PPEXIT      read
W 0028 160053              LBRA  PPWRIT
  002B 16009A              LBRA  PPEXIT      get status
  002E 160097              LBRA  PPEXIT      set status
W 0031 16007A              LBRA  PPTERM

  0034 00         PPMASK   fcb   0           FLIP (NONE)
  0035 80                  fcb   $80         IRQ POLLING MASK
  0036 04                  fcb   4           (low) PRIORITY

       **********
       * PPINIT - Initialize PIA

       * Passed: (U)=Static storage
       *         (Y)=Initial Device Descriptor

  0037 A341       PPINIT   LDX   V.PORT,Y    get PIA port addr
  0039 A6A811              LDA   M$OPT,Y     get option byte count
  003C 8114                CMPA  #PD.PAR-PD.OPT pia side given?
  003E 2515                BLO   PPIN15      ..No; default B-side
  0040 E6A826              LDB   PD.PAR-PD.OPT+M$DTYP,Y
```

```
0043 E746               STB   V.TYPE,U   save pia type
0045 C101               CMPB  #a.side    A-side PIA?
0047 2604               BNE   PPIN10     ..No
0049 C63E               LDB   #$3E       A-side non auto-latch
004B 200C               BRA   PPIN20
004D C102      PPIN10   CMPB  #MP.L2     Southwest ACIA?
004F 2604               BNE   PPIN15     ..No
0051 860E               LDA   #$0E       setup SWTPC MP-L2 card
0053 A70E               STA   $0E,X      note: must be b.side
0055 3002      PPIN15   LEAX  2,X        Adjust address B-side
0057 C62F               LDB   #$2F       B-side is auto-latch
0059 6F01      PPIN20   CLR   1,X        reset PIA
005B AF4F               STX   PIADDR,U   save port address
005D 3414               PSHS  B,X        save ctl code, addr
005F 3001               LEAX  1,X
0061 1F10               TFR   X,D
0063 308DFFCD           LEAX  PPMASK,PCR
0067 318C4F             LEAY  <PPIRQ,PCR addr of SERVICE ROUTINE
006A 103F2A             OS9   F$IRQ      ADD to IRQ POLLING TBL
006D 250D               BCS   PPIN90     ..Error; return it
006F 3514               PULS  B,X
0071 86FF               LDA   #$FF
0073 1A10               ORCC  #IRQM      disable interrupts
0075 ED84               STD   0,X        Initialize Pia
0077 A684      PPCLRQ   LDA   0,X        Clear IRQs
0079 1CEE               ANDCC #$FF-IRQM-CARRY enable intrpts
007B 39                 RTS              return

007C 3592 PPIN90        PULS  A,X,PC     Return (B)=error

     **********
     * PPWRIT - write one char to PIA

     * Passed: (U)=Static Storage
     *         (Y)=Path Descriptor
     *         (A)=char to write to PIA
     * Returns: CC,B set if Error

007E AE4F      PPWRIT   LDX   PIADDR,U   port addr
0080 E646               LDB   V.TYPE,U   get pia type
0082 1A10               ORCC  #IRQM      disable interrupts
0084 A784               STA   0,X        Write char to Pia
0086 C101               CMPB  #a.side    A-side port?
0088 2608               BNE   PPWR10     ..No; auto-latching
008A C637               LDB   #$37
008C E701               STB   1,X
008E C63F               LDB   #$3F
0090 E701               STB   1,X        latch A-side output
0092 8D34      PPWR10   BSR   PPEXIT     Delay shortly fast PIA
0094 6D01               TST   1,X        character already gone?
0096 2BDF               BMI   PPCLRQ     ..Yes; remove interrupt
0098 E644               LDB   V.BUSY,U
009A E745               STB   V.WAKE,U
009C 8E0000             LDX   #0
009F 1CEF               ANDCC #$FF-IRQM  enable interrupts
00A1 103F0A             OS9   F$SLEEP     wait for I/O to occur
00A4 5F                 clrb             clear carry
00A5 9E4B               ldx   D.PROC
```

```
00A7 E68836             ldb   P$SIGN,X   Signal waiting?
00AA 2701               beq   PPWR90     ..No; return
00AC 43                 coma
00AD 39        PPWR90   RTS

     **********
     * PPTERM - Remove PIA from system

00AE AE4F      PPTERM   LDX   PIADDR,U
00B0 6F01               CLR   1,X        reset PIA
00B2 8E0000             LDX   #0         remove PIA
00B5 103F2A             OS9   F$IRQ      from polling tbl
00B8 39                 RTS

     ***************
     * PROCESS PIA INTERRUPT

     * Passed: (A)=PIA Status Reg

00B9 AE4F      PPIRQ    LDX   PIADDR,U   PIA port addr
00BB 6D84               TST   0,X        remove interrupt
00BD A645               LDA   V.WAKE,U   User's Process ID
00BF 2707               BEQ   PPEXIT     ..No; return
00C1 C601               LDB   #S$WAKE    (wake up)
00C3 103F08             OS9   F$SEND
00C6 6F45               clr   V.WAKE,U
00C8 5F        PPEXIT   clrb
00C9 39                 RTS

00CA FDA2DE             emod
00CD           PIASIZ   equ   *
```

Example C.3. P - Device Descriptor for “P”

```
                        nam   P

                        ifp1
                        endc

                        ttl   Device Descriptor for "P"
     **************
     *  PRINTER device module
     *
0000 87CD0035           mod   PRTEND,PRTNAM,DEVIC+OBJCT,
                                          REENT+1,PRTMGR,PRTDRV
000D 02                 fcb   WRITE.     mode
000E FF                 fcb   $FF
000F E040               fcb   A.P        port address
0011 18                 fcb   PRTNAM-*-1 option byte count
0012 00                 fcb   DT.SCF     Device Type: SCF

     * Default path options

0013 00                 fcb   0          case=UPPER and lower
0014 00                 fcb   0          backspace=BS char only
0015 01                 fcb   1          delete=CRLF
0016 00                 fcb   0          no auto echo
```

```
0017 01                 fcb   1          auto line feed on
0018 00                 fcb   0          no nulls after CR
0019 00                 fcb   0          no page pause
001A 42                 fcb   66         lines per page
001B 08                 fcb   C$BSP      backspace char
001C 18                 fcb   C$DEL      delete line char
001D 0D                 fcb   C$CR       end of record char
001E 00                 fcb   0          no end of file char
001F 04                 fcb   C$RPRT     reprint line char
0020 01                 fcb   C$RPET     dup last line char
0021 17                 fcb   C$PAUS     pause char
0022 00                 fcb   0          no abort character
0023 00                 fcb   0          no interrupt character
0024 5F                 fcb   '_         backspace echo char
0025 07                 fcb   C$BELL     line overflow char
0026 01                 fcb   PIASID     Printer Type
0027 00                 fcb   0          undefined baud rate
0028 0000               fdb   0          no echo device
002A D0        PRTNAM   fcs   "P"        device name
002B B0                 fcs   "0"        room for name patching
002C 5343C6    PRTMGR   fcs   "SCF"      file manager
002F 5049C1    PRTDRV   fcs   "PIA"      driver

0032 A9B118             emod
0035           PRTEND   EQU   *
```

Colophon

This book is scanned from an OS-9 manual found on the Internet in 2017.