Burroughs

Programmer's Manual

ET 2000 series

Utilities

(Relative to Release Level 1.0)

Priced Item
Printed in U.S.A.
June 1984

1171501

Burroughs

Programmer's Manual

ET 2000 Series Utilities

(Relative to Release Level 1.0)

Priced Item
Printed in U.S.A.
June 1984

1171501

Burroughs cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued from time to time to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded, using the Documentation Evaluation Form at the back of the manual, or remarks may be addressed directly to Burroughs Corporation, Corporate Documentation Planning, East, 209 W. Lancaster Ave., Paoli, PA 19301, U.S.A.

LIST OF EFFECTIVE PAGES

Page	Issue
iii	Original
iv	Blank
v thru ix	Original
x	Blank
1-1 thru 1-4	Original
2-1 thru 2-36	Original
3-1 thru 3-13	Original
3-14	Blank
4-1 thru 4-38	Original
5-1 thru 5-10	Original
6-1 thru 6-15	Original
6-16	Blank
7-1 thru 7-21	Original
7-22	Blank

TABLE OF CONTENTS

TABLE OF CONTENTS (CONT.)

TABLE OF CONTENTS (CONT.)

LIST OF TABLES

INTRODUCTION

The ET 2000 Programmer's Utilities Manual is a reference source for experienced programmers; a general understanding of assembly and compiled languages is assumed.

To make information easy to find, the manual includes a detailed Table of Contents and seven sections.

* Section 1 summarizes each of the programmer's utilities, provides an overview of how to use the utilities in program development, and explains Command and Statement syntax.

* Section 2 annotates EDLIN; the line editor utility.

* Section 3 annotates CREF; the cross-reference utility.

* Section 4 annotates DEBUG; the file debugging utility.

* Section 5 annotates FC; the file comparison utility.

* Section 6 annotates LIB; the library manager utility.

* Section 7 annotates LINK; the file linking utility.

SECTION 1

OVERVIEW

UTILITIES

The ET 2000 programmer utilities consist of six software packages for developing assembly and compiled language programs. Each utility is briefly defined and described in the following paragraphs.

EDLIN

EDLIN is a **LIN**e **ED**itor program used to:

1. Create and save source files.
2. Update files and save both original and updated versions.
3. Change, delete, insert, and display program lines.
4. Locate a specified section of text within a line or file and, optionally, replace or delete that text.

CREF

Cross **REF**erence (CREF) is a program that produces an alphabetical listing of all the symbols in a special file produced by the assembler. This list allows you to:

1. Locate, by line number, all occurrences of any symbol in the source program.
2. Determine the value, type, and length of each symbol.

DEBUG

DEBUG is an editing program for use with binary and executable object files. DEBUG allows you to:

1. Make minor changes to a program and run the program without reassembly.
2. Alter the contents of a CPU register and run the program without reassembly.

FC

File Comparison (FC) a program used to compare the contents of two files and report any discrepancies. The types of files that may be compared are:

1. Source files from a programming language.

2. Binary files output by MACRO-86 assembler, LINK utility, or a high level language compiler.

LIB

LIBrary (LIB) a program that manages a set of frequently used program modules that can be linked to customize programs or create new programs. LIB allows you to:

1. Add modules.

2. Delete modules.

3. Modify modules and save both the original and modified versions.

LINK

LINK is a program designed to join modules of 8086 object code. This program:

1. Produces relocatable executable object code.

2. Handles overlays defined by you.

3. Performs multiple library searches using a dictionary search method.

4. Prompts you for input and output modules and other parameters or runs with an automatic response file to answer prompts.

USE IN PROGRAM DEVELOPMENT

The following steps explain how the six programmer utilities work together in developing programs. By following these steps, you can improve your programming efficiency and avoid constant rewriting of identical or similar routines for use in many programs.

1. Design a flowchart for your program, dividing the program into modules for major routines and subroutines.

2. Use EDLIN to create each module.

3. Assemble or compile each module specifying that a cross-reference file be produced. The cross-reference file has numbered lines to make debugging easier.

4. Use CREF to create a listing of line numbers for each symbol definition, and references to that symbol, used in the module. Any assembler or compiler error messages referring to that symbol are now easy to cross-reference.

5. Use DEBUG to correct minor errors without reassembling, or use EDLIN to correct more significant errors.

6. Use FC to compare original and modified files to ensure that you made all intended corrections.

7. Use LIB to add completed modules to your library of routines and to extract modules for linking.

8. Use LINK to join the program modules.

COMMAND AND STATEMENT SYNTAX

The following notation describes command and statement syntax:

[ ]	Square brackets enclose optional entries.
<lowercase>	Angle brackets enclosing lowercase text indicate that you must supply information. For example, <filename> asks for the name of a file.
<UPPERCASE>	Angle brackets enclosing uppercase text indicate that you must press a key. For example, <ESC> asks you to press the ESCape key.
{ }	Braces indicate that you must choose at least one of the enclosed items. For example, {A:, B:} indicates that you must choose between Drive A and Drive B.
[{ }]	Square brackets enclosing braces indicate that you may choose at least one of the enclosed items. For example, [{A:, B:}] indicates that you may choose between Drive A and Drive B.
...	Ellipses indicate that you may repeat an entry as many times as needed.
CAPS	Capital letters indicate portions of commands or statements that must be entered exactly as shown.

NOTE

All punctuation not listed in the previous chart must be entered exactly as it appears in this manual.

SECTION 2
EDLIN - LINE EDITOR UTILITY

WHAT EDLIN CAN DO

The Line Editor utility is a tool for creating and editing files that are organized as lines of text. EDLIN may be used to:

1. Create and save source files.

2. Update files and save both original and updated versions.

3. Change, insert, delete and display program lines.

4. Search for, delete, or replace text within one or more lines.

The text of a file is divided into lines of up to 253 characters. Consecutive line numbers are automatically generated and displayed by EDLIN but are not present in the saved file. These line numbers are automatically incremented or decremented to reflect insertions or deletions of lines.

USING EDLIN

Creating or Loading Files

To invoke this utility from the MS[TM]-DOS command level, use the following syntax:

```
EDLIN  <filespec>
```

filespec is the name of a file to be either created or edited. If EDLIN does not find the file on the specified (or default) drive, it creates a file with the specified name.

If a new file is created, EDLIN uses an asterisk as a prompt:

```
New file
*[]
```

EDLIN is then ready to accept text lines from the user. Refer to the Insert command in the alphabetic listing at the end of this section.

TM MS is a trademark of Microsoft Inc.

If the specified file exists on the designated (or default) drive, the file is loaded until the memory is 75% full.

If the entire file is loaded, EDLIN prompts:

```
End of input file
*[]
```

If the entire file cannot be loaded, the following prompt appears when the memory is 75% full:

```
*[]
```

EDLIN is then ready to accept editing commands.

To edit the remainder of the file, write edited lines to disk in order to free memory for unedited lines. Refer to "Append Command" and "Write Command" later in this section.

ıving Files

en a file has been edited, (see alphabetic listing of commands ter in this section) it can be saved by using the End command. wly created files are stored under the filespec used when EDLIN s invoked. Updated files are stored under the filespec of the iginal version, which is saved as a backup. This backup has the iginal filename, with the extension .BAK.

cause EDLIN assumes that files with .BAK extensions are backups, ese files cannot be edited. You may use the MS-DOS Rename mmand to change the .BAK extension; then restart EDLIN and ecify the new name.

GENERAL INFORMATION ON THE COMMANDS

Command Structure

The user should understand the following information before using any EDLIN commands:

1. Pathnames are acceptable as options to commands. For example, typing: EDLIN /BIN/USER/BOB/TEXT.TXT allows you to edit the TEXT.TXT file in the subdirectory BOB.

2. You can reference line numbers relative to the current line (the line with the asterisk). Use a minus sign with a number to indicate lines before the current line. Use a plus sign with a number to indicate lines after the current line. For example:

   ```
   -10,+10L
   ```

   This command lists 10 lines before the current line, the current line, and 10 lines after the current line.

3. In most cases, multiple commands may be issued on one command line without delimiters between commands. To edit a single line using a line number, insert a semicolon between commands on the line. For a Search or Replace command, the <string> may end with <CTRL-Z> instead of a <cursor-return>.

   Examples:

   The following command line edits line 15, then displays lines 10 through 20 on the screen.

   ```
   15;-5,+5L
   ```

   The command line in the next example searches for "This string" and then displays five lines before and after the line containing the matched string. If the search is unsuccessful, the lines displayed are those with line numbers relative to the current line.

   ```
   SThis string<CTRL-Z>-5,+L
   ```

You can type EDLIN commands with or without a space between the line number and command. For example, to delete line 6, the command 6D is the same as 6 D.

CTRL-V allows you to insert a control character (such as CTRL-C) into text; it allows MS-DOS to recognize the next capital letter typed as a control character. It is possible to use a control character in any of the string arguments of Search or Replace. For example:

S<CTRL-V>Z
will find the first occurrence
of CTRL-Z in a file

R<CTRL-V>Z<CTRL-Z>framis
will replace all occurrences
of CTRL-Z in a file by framis

S<CTRL-V>C<CTRL-Z>bar
will replace all occurrences
of CTRL-C by bar

It is possible to insert CTRL-V into the text by typing CTRL-V-V.

The CTRL-Z character ordinarily notifies EDLIN, "This is the end of the file." If you have CTRL-Z characters elsewhere in your file, you must tell EDLIN that these other control characters do not mean "End of File." Use the /B switch to tell EDLIN to ignore any CTRL-Z characters in the file and to show you the entire file.

Command Summary

The EDLIN commands appears in table 2-1 with a detailed description of each command presented alphabetically at the end of this section.

Table 2-1. EDLIN Commands

Command	Purpose
<line>	Edits line no.
A	Appends lines
C	Copies lines
D	Deletes lines
E	Ends editing
I	Inserts lines
L	Lists text
M	Moves lines
P	Pages text
Q	Quits editing
R	Replaces lines
S	Searches text
T	Transfers text
W	Writes lines

mmand Options

eral EDLIN commands accept one or more of the following options. effect of an option depends on the associated command.

ne> <line> indicates a line number that you type. Line numbers must be separated from other line numbers, other options, and the command, by a comma or space. <line> may be specified in one of three ways:

Number Any number less than 65534. If a number larger than the largest existing line number is specified, then <line> means the line after the last line number.

Period (.) If a period is specified for <line>, then <line> means the current line number. The current line is the last line edited and is not necessarily the last line displayed. The current line is marked on your screen by an asterisk (*) between the line number and the first character.

Pound (#) The pound sign indicates the line after the last line number. Specifying # for <line> has the same effect as specifying a number larger than the last line number.

<cursor return> A cursor return entered without any of the <line> specifiers listed above directs EDLIN to use a default value appropriate to the command.

The question mark option directs EDLIN to ask you if the correct string has been found. The question mark is used only with the Replace and Search commands. Before continuing, EDLIN waits for either a "Y" or <cursor return> as a yes response or for any other key as a no response.

ring> <string> represents text to find, or to replace or be replaced by other text. The <string> option is used only with the Search and Replace commands. Each <string> must end with <CTRL-Z> or <cursor-return>. No spaces should remain between strings or between a string and its command letter, except when spaces are part of the string.

DETAILED DESCRIPTIONS OF THE COMMANDS

The remainder of this section consists of an alphabetical presentation of all of the EDLIN commands.

Append Command

FUNCTION:

Loads a specified number of file lines from disk into memory when the file being edited is too large to fit into memory.

SYNTAX:

[<n>]A

REMARKS:

As many lines as possible are read into memory for editing when starting EDLIN. To edit the remainder of the file, lines already edited must be written to disk. You can then load the unedited lines from disk into memory with the Append command. Refer to the Write Command in this alphabetic list of commands for informatiion on how to write edited lines to disk.

NOTES

1. If you do not specify the number of lines to append, lines are appended to memory until the available memory is 3/4 full. No action is taken if the memory is already 3/4 full.

2. The message "End of input file" is displayed when the Append command has read the last line of the file into memory.

UNCTION:

Copies a range of lines to a specified line number. The lines can be copied as many times as needed by using the <count> option.

YNTAX:

[<line>],[<line>],<line>,[<count>]C

EMARKS:

If you do not specify a number in <count>, EDLIN copies the lines once. If the first or second <line> is omitted, the default is the current line. The file is renumbered automatically after the Copy Command has been executed.

The line numbers must not overlap or an "Entry error" message appears. For example, 3,20,15C would result in an error message.

XAMPLES:

Assume that the following file is ready to edit:

```
1: This is a sample file
2: to demonstrate line copying.
3: See what happens when you use
4: the Copy command
5: (the C command)
6: to copy text in your file.
```

You can copy this entire block of text by issuing the following command:

```
1,6,7C
```

The result is:

```
1: This is a sample file
2: to demonstrate line copying.
3: See what happens when you use
4: the Copy command
5: (the C command)
6: to copy text in your file.
7: This is a sample file
8: to demonstrate line copying.
9: See what happens when you use
10: the Copy command
11: (the C command)
12: to copy text in your file.
```

If you want to place the text within other text, the third <line> option should specify the line before the one in which the copied text is to appear. For example, assume that you want to copy lines and insert them within the following file:

```
1: This is a sample file
2: to demonstrate line copying.
3: See what happens when you use
4: the Copy command
5: (the C command)
6: to copy text in your file.
7: You can also use COPY
8: to copy lines of text
9: to the middle of your file.
10: End of sample file.
```

The command 3,6,9C results in the following file:

```
1: This is a sample file
2: to demonstrate line copying.
3: See what happens when you use
4: the Copy command
5: (the C command)
6: to copy text in your file.
7: You can also use COPY
8: to copy lines of text
9: to the middle of your file.
10: See what happens when you use
11: the Copy command
12: (the C command)
13: to copy text in your file.
14: End of sample file.
```

Delete Command

FUNCTION:

Deletes a specified range of lines in a file.

SYNTAX:

[<line>][,<line>]D

REMARKS:

If the first <line> is omitted, that option defaults to the current line (the line with the asterisk after the line number). If the second <line> is omitted, then only the first <line> is deleted. When lines have been deleted, the first line following the deleted section becomes the current line.

EXAMPLES:

Assume that the following file is ready to edit:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
 .
 .
 .
25: (the D and I commands)
26: to edit the text
27:*in your file.
```

To delete multiple lines, type <line>,<line>D:

```
5,24D
```

The result is:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
6: to edit text
7:*in your file.
```

To delete a single line, type:

```
6D
```

The result is:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
6:*in your file.
```

Next, delete a range of lines from the following file:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3:*See what happens when you use
4: Delete and Insert
5: (the D and I commands)
6: to edit text
7: in your file.
```

To delete a range of lines beginning with the current line, type:

```
,6D
```

The result is:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3:*in your file.
```

Notice that the lines are automatically renumbered.

FUNCTION:

Edits line of text.

SYNTAX:

[<line>]

REMARKS:

When a line number is typed, EDLIN displays the line number and its text. Then, on the line below, EDLIN reprints the line number. The line is then ready for editing. You may use any of the EDLIN editing commands to edit the line. The existing text in the line serves as the template until the cursor-return key is pressed.

If no line number is typed (that is, if only the cursor-return key is pressed), the line after the current line (marked with an asterisk) is edited. If no changes are needed in the current line, and the cursor is at the beginning or end of the line, press the cursor-return key to accept the line as it is.

CAUTION

If the cursor-return key is pressed while the cursor is in the middle of the line, the remainder of the line is deleted.

:AMPLES:

Assume that the following file is ready to edit:

```
1: This is a sample file
2: used to show
3: the editing of line
4:*four.
```

To edit line 4, type:

```
4
```

The contents of the line are displayed with a cursor below the line:

```
4:* four.
4:*[]
```

End Command

FUNCTION:

Ends the editing session.

SYNTAX:

E

REMARKS:

This command saves the edited file on disk, renames the original input file <filename>.BAK, and then exits EDLIN. If a new file has been created during the editing session, no .BAK file is generated.

The E command uses no options. Therefore, you cannot use EDLIN to indicate on which drive to save the file. The drive on which the file is to be saved must be selected at the start of the editing session. If the drive is not selected at that time, the file will be saved on the disk in the default drive. The user can still COPY the file to a different drive using the MS-DOS COPY command.

Be sure that the disk contains enough free space for the entire file. If not, the write is aborted, and the edited file lost (although part of the file might be written to the disk).

EXAMPLES:

```
E<cursor-return>
```

After execution of the E command, the MS-DOS default drive prompt (A>, for example) is displayed.

NCTION:

Inserts text immediately before the specified <line>.

NTAX:

[<line>]I

MARKS:

If you are creating a new file, the I command must be used before any text can be typed (inserted). Text begins with line number 1. Successive line numbers appear automatically each time the user presses the cursor-return key.

EDLIN remains in insert mode until <CTRL-C> is typed. When the insertion is complete and the insert mode has been exited, the line immediately following the inserted text becomes the current line. All line numbers following the inserted section are incremented by the number of lines inserted.

If <line> is not specified, the default will be the current line number, and the insertion will be made immediately before the current line. If <line> is any number larger than the last line number, or if a pound sign (#) is specified as <line>, the inserted text is appended to the end of the file. In such a case, the last line inserted will become the current line.

MPLES:

Assume that the following file is ready to edit:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
6: to edit text
7:*in your file.
```

To insert text before a specific line that is not the current line, type <line>I:

```
7I
```

The result is:

```
7:[]
```

Now, type the new text for line 7:

```
7: and renumber lines
```

Then to end the insertion, press <CTRL-Z> on the next line:

```
8: <CTRL-Z>
```

Now type L to list the file. The result is:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
6: to edit text
7. and renumber lines
8:*in your file.
```

To insert text immediately before the current line, type:

```
I
```

The result is:

```
8: []
```

Now, insert the following text and terminate with a <CTRL-Z> on the next line:

```
8: so they are consecutive
9: <CTRL-Z>
```

To list the file and see the result, type:

```
L
```

The result is:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
6: to edit text
7: and renumber lines
8: so they are consecutive
9:*in your file.
```

To append new lines to the end of the file, type:

```
10I
```

The result is:

```
10: []
```

Now, type the following new lines:

```
10: The insert command can place new lines
11: in the file; there's no problem
12: because the line numbers are dynamic;
13: they'll go all the way to 65533.
```

End the insertion by pressing <CTRL-Z> on line 14. The new lines appears at the end of all previous lines in the file. Now, type the list command, L:

The result is:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
6: to edit text
7: and renumber lines
8: so they are consecutive
9: in your file.
10: The insert command can place new lines
11: in the file; there's no problem
12: because the line numbers are dynamic;
13: they'll go all the way to 65533.
```

FUNCTION:

Displays a range of lines (including the specified lines).

SYNTAX:

[<line>][,<line>]L

REMARKS:

Default values are provided if one or both of the options are omitted. If the first option is omitted, as in:

,<line>L

the display starts 11 lines before the current line and end with the specified <line>. The beginning comma is required to indicate the omitted first option.

NOTE

If the specified <line> is more than 11 lines before the current line, the display is the same as if you had omitted both options.

If the second option is omitted, as in:

<line>L

23 lines will be displayed, starting with the specified <line>. If you omit both parameters, as in:

L

23 lines will be displayed (11 lines before the current line, the current line, and 11 lines following the current line). If there are less than 11 lines before the current line, more than 11 lines after the current line will be displayed, to make a total of 23 lines.

EXAMPLES:

Assume that the following file is ready to edit:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
 .
 .
 .
15:*The current line contains an asterisk.
 .
 .
 .
26: to edit text
27: in your file.
```

To list a range of lines without reference to the current line, type:

```
<line>,<line>L  (actually 2,5L)
```

The result is:

```
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
```

To list a range of lines beginning with the current line, type:

```
,<line> L  (actually ,26L)
```

The result is:

```
15:*The current line contains an asterisk.
26: to edit text
```

To list a range of 23 lines centered around the current line, type:

```
L
```

The result is:

```
 4: Delete and Insert
 5: (the D and I commands)
  .
  .
  .
13: The current line is listed in the middle of the
    range.
14: The current line remains unchanged by the L
    command.
15:*The current line contains an asterisk.
  .
  .
  .
26: to edit text.
```

FUNCTION:

Moves a block of text to a specified location.

SYNTAX:

[<line>],[<line>],<line>M

REMARKS:

Use the Move command to move a block of text (from the first <line> to the second <line>) to another location in the file. The lines are renumbered according to the direction of the move. For example,

```
,+25,100M
```

moves the text from the current line plus 25 lines to line 100. If the line numbers overlap, EDLIN will display an "Entry error" message. To move lines 20 to 30 to line 100, type:

```
20,30,100M
```

Page Command

FUNCTION:

Displays a file 23 lines at a time.

SYNTAX:

[<line>][,<line>]P

REMARKS:

If the first <line> is omitted, that number will default to the current line plus one. If the second <line> is omitted, 23 lines will be listed. The last line displayed becomes the current line, which is displayed with an asterisk.

NCTION:

rminates the editing session, without saving any editing
anges, and exits to the MS-DOS operating system.

NTAX:

Q

MARKS:

LIN issues a prompt to make certain you do not want to save the
anges.

pe Y if you want to quit the editing session. No editing changes
e saved and no .BAK file is created. Refer to the End command
scribed earlier for information about the .BAK file.

pe N, or any other character, if you wish to continue the editing
ssion.

NOTE

When started, EDLIN erases any previous copy of the file with an extension of .BAK to make room to save the new copy. If you reply Y to the "Abort edit (Y/N)?" message, your previous backup copy will no longer exist.

AMPLES:

```
Q

Abort edit (Y/N)?Y<cursor-return>

A>[]
```

Replace Command

FUNCTION:

Replaces all occurrences of a given text string (within the specified range) with a new text string, a null (empty) string, or spaces.

SYNTAX:

[<line>][,<line>][?]R<string1><CTRL-Z><string2>

REMARKS:

As each occurrence of <string1> is found, it is replaced by <string2>. Each line with a replacement is displayed. If a line contains two or more replacements of <string1> by <string2>, then the line is displayed once for each occurrence. When all occurrences of <string1> in the specified range have been replaced by <string2>, the R command terminates and the asterisk prompt reappears.

If a second string is provided as a replacement, then <string1> must be separated from <string2> by a <CTRL-Z>. <String2> must also be followed by either a <CTRL-Z><cursor return> combination or a simple cursor-return.

If <string1> is omitted, then Replace will use the old <string1> as its value. If there is no old <string1>, i.e., this is the first Replace done, then the replacement process terminates immediately. If <string2> is omitted, then <string1> may be ended with a cursor-return.

If the first <line> is omitted in the range argument (as in ,<line>) then the first <line> will default to the line after the current line. If the second <line> is omitted (as in <line> or <line>,), the second <line> will default to #. Therefore, this entry is the same as <line>,#. Remember that # indicates the line after the last line of the file.

If <string1> is followed by a <CTRL-Z> and there is no <string2>, <string2> will be taken as an empty string and will become the new Replace string. For example:

R<string1><CTRL-Z><cursor return>

deletes occurrences of <string1>, but

R<string1><return> and
R<cursor return>

replaces <string1> with the old <string2>, and the old <string1> with the old <string2>, respectively. Note that "old" here refers to a previous string specified in either a Search or Replace command.

If the question mark (?) option is given, the Replace command stops at each line with a string that matches <string1>, displays the line with <string2> in place, and then displays the prompt "O.K.?". If you press Y or the cursor-return key, then <string2> replaces <string1>, and the next occurrence of <string1> is found. The "O.K.?" prompt is again displayed.

This process continues until the end of the range, or until the end of the file. After the last occurrence of <string1> is found, EDLIN displays the asterisk prompt.

If you press any key besides Y or cursor return after the "O.K.?" prompt, the <string1> is left as it was in the line, and Replace goes to the next occurrence of <string1>. If <string1> occurs more than once in a line, each occurrence of <string1> is replaced individually, and the "O.K.?" prompt is displayed after each replacement. In this way, only the desired <string1> substitutions are made.

EXAMPLES:

Assume that the following file is ready to edit:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
6: to edit text
7: in your file.
8: The insert command can place new lines
9: in the file; there's no problem
10: because the line numbers are dynamic;
11: they'll go all the way to 65533.
```

To replace all occurrences of <stringl> with <string2> in a specified range, type:

```
2,12 Rand<CTRL-Z>or<cursor return>
```

The result is:

```
4: Delete or Insert
5: (the D or I commors)
8: The insert commor can place new lines
```

Note that in the preceding replacement, some unwanted substitutions have occurred. To avoid these, and to confirm each replacement, the same original file can be used with a slightly different command.

In the next example, to replace only certain occurrences of the first <string> with the second <string>, type:

```
2? Rand<CTRL-Z>or<cursor return>
```

The result is:

```
4: Delete or Insert
O.K.? Y
5: (The D or I commands)
O.K.? Y
5: (The D or I commands)
O.K.? N
8: The insert command can place new lines
O.K.? N
*[]
```

Now, type the List command (L) to see the result of all these changes:

```
 .
 .
4: Delete or Insert
5: (The D or I commands)
 .
8: The insert command can place new lines
 .
 .
```

Search Command

FUNCTION:

Searches the designated range of lines for a specified string of text

SYNTAX:

[<line>][,<line>][?]S<string><cursor return>

REMARKS:

The <string> must be followed by a cursor return. The first line that matches <string> is displayed and becomes the current line. If the question mark option is not specified, the Search command terminates when a match is found. If no line contains a match for <string>, the message "Not found" is displayed.

If the question mark option is included in the command, EDLIN displays the first line with a matching string. It then prompts you with the message "O.K.?". If you press either the Y or cursor-return key, the line becomes the current line, and the search terminates. If you press any other key, the search continues until another match is found, or until all lines have been searched and the "Not found" message is displayed.

If the first <line> is omitted (as in ,<line> S<string>), the first <line> defaults to the line after the current line. If the second <line> is omitted (as in <line> S<string> or <line>, S<string>), the second <line> defaults to # (line after last line of file), which is the same as <line>,# S<string>. If <string> is omitted, Search uses the old string if there is one. (Note that "old" here refers to a string specified in a previous Search or Replace command.) If there is not an old string (i.e., no previous search or replace has been done), the command terminates immediately.

AMPLES:

Assume that the following file is ready to edit:

```
1: This is a sample file
2: to demonstrate dynamic line numbers.
3: See what happens when you use
4: Delete and Insert
5: (the D and I commands)
6: to edit text
7: in your file.
8: The insert command can place new lines
9: in the file; there's no problem
10: because the line numbers are dynamic;
11:*they'll go all the way to 65533.
```

To search for the first occurrence of the string "and", type:

```
2,12 Sand<cursor return>
```

and observe that the following line is displayed:

```
4: Delete and Insert
```

To get the "and" in line 5, modify the Search command by typing:

```
<SKIP1><COPYALL>,12 Sand<cursor return>
```

The search then continues from the line after the current line (line 4), since no first line was given. The result is:

```
5: (the D and I commands)
```

To search through several occurrences of a string until the correct string is found, type:

```
1, ? Sand
```

Observe that the result is:

```
4: Delete and Insert
O.K.?[]
```

If you press any key (except Y or cursor-return), the search continues, so type N here:

```
O.K.? N
```

Continue:

```
5: (the D and I commands)
O.K.?[]
```

Now press Y to terminate the search:

```
O.K.? Y
*[]
```

To search for string XYZ without the verification (O.K.?), type:

```
SXYZ
```

EDLIN reports a match and continues to search for the same string when you issue the S command:

```
S
```

EDLIN reports another match.

```
S
```

EDLIN reports that the string is not found.

NOTE

<string> defaults to any string specified by a previous Replace or Search command.

Transfer Command

FUNCTION:

Inserts (merges) the contents of <filename> into the file currently being edited at <line>. If <line> is omitted, the current line will be used.

SYNTAX:

[<line>]T<filename>

REMARKS:

This command is useful if you want to put the contents of a file into another file or into the text you are typing. The transferred text is inserted at the line number specified by <line>, and the lines are renumbered.

Write Command

FUNCTION:

Writes a specified number of lines to disk from the text that is being edited in memory. Text is written to disk beginning with line number 1.

SYNTAX:

[<n>]W

REMARKS:

This command is meaningful only if the file you are editing is too large to fit into memory. When you start EDLIN, it reads lines into memory until memory is 3/4 full.

To edit the remainder of your file, you must write edited lines from memory to disk. Then you can load additional unedited lines from disk into memory by using the Append command.

NOTE

If you do not specify the number of lines, lines will be written until memory is 3/4 full. No action will be taken if available memory is already more than 3/4 full. All lines are renumbered, so that the first remaining line becomes line number 1.

EDLIN ERROR MESSAGES

When EDLIN encounters an error, one of the following messages is displayed:

Cannot edit .BAK file--rename file

Cause: You attempted to edit a file with a filename extension of .BAK. .BAK files cannot be edited because this extension is reserved for backup copies.

Cure: If you need the .BAK file for editing purposes, you must either RENAME the file with a different extension, or COPY the .BAK file and give the copy a different filename extension.

No room in directory for file

Cause: When you attempted to create a new file, either the file directory was full or you specified an illegal filename or disk drive.

Cure: Check the command line that started EDLIN for illegal filename and illegal disk drive entries. If this command line contains no illegal entries, run the CHKDSK program for the specified disk drive. If the status report shows that the disk directory is full, remove the disk. Insert and format a new disk.

Entry Error

Cause: The last command typed contained a syntax error.

Cure: Retype the command with the correct syntax and press the cursor return key.

Line too long

Cause: During a Replace command, the string used as the replacement caused the line to expand beyond the limit of 253 characters. EDLIN therefore aborted the Replace command.

Cure: Divide the long line into two lines, then try the Replace command again.

Disk Full--file write not completed

Cause: When executing the End command, the disk did not contain enough free space for the whole file. EDLIN aborted the E command and returned to the operating system. Some of the file may have been written to the disk.

Cure: Only a portion (if any) of the file has been saved. You should probably delete that portion of the file and restart the editing session. The file is not be available after this error. Always be sure that the disk has sufficient free space for the file to be written to disk before beginning your editing session.

Incorrect DOS version

Cause: You attempted to run EDLIN under a version of MS-DOS that was not 2.0 or higher.

Cure: You must make certain that the version of MS-DOS that you are using is 2.0 or higher.

Invalid drive name or file

Cause: You did not specify a valid drive or filename when starting EDLIN.

Cure: Specify the correct drive or filename.

Filename must be specified

Cause: You did not specify a filename when you started EDLIN.

Cure: Specify a filename.

Invalid Parameter

Cause: You specified a switch other than /B when starting EDLIN.

Cure: Specify the /B switch when you start EDLIN.

Insufficient memory

Cause: There is not enough memory available to run EDLIN.

Cure: Before starting EDLIN, you must free some memory by writing files to disk or by deleting files.

File not found

Cause: The filename specified during a Transfer command was not found.

Cure: Specify a valid filename when issuing a Transfer command.

Must specify destination number

Cause: A destination line number was not specified for a Copy or Move command.

Cure: Reissue the command with a destination line number.

Not enough room to merge the entire file

Cause: There was not enough room in memory to hold the file during a Transfer command.

Cure: You must free some memory by writing some files to disk or by deleting some files before you can transfer this file.

SECTION 3

CREF - CROSS-REFERENCE UTILITY

WHAT CREF CAN DO

The Cross-Reference Utility (CREF) is a debugging aid for use by assembly language programmers. CREF produces a special file containing an alphabetic listing of all the symbols produced by your assembler. With this listing, you can quickly locate, by line number, all occurrences of any symbol in the source program.

The list produced by CREF is used with the symbol table produced by the assembler. The symbol table shows the value, type, and length of each symbol. With this information the user can correct erroneous symbol definitions or uses.

The cross-reference listing produced by CREF provides you with the symbol locations, speeding your search and promoting faster debugging.

The CREF utility requires a minimum of 24 kilobytes of memory, apportioned as follows:

1. 14 K-bytes for code

2. 10 K-bytes for run space

GENERAL INFORMATION

Use the assembler to create a cross-reference file and assign it the filename extension .CRF. CREF accepts this cross-reference file and converts it to an alphabetic listing of the symbols in the file. The cross-reference listing is given the (default) filename extension .REF.

CREF places an ascending sequence of line numbers after each symbol in the listing to indicate where the symbol occurs in the source program. The line number for the symbol definition is flagged by a pound sign (#).

All commands to CREF are entered at the keyboard. Two special command characters (";" and "CTRL-C") are provided as aids in entering CREF commands.

CREATING A CROSS-REFERENCE FILE

Before using CREF to create the cross-reference listing, you must first create a cross-reference file using your assembler.

NOTE

This procedure assumes that you are using Microsoft 8086 assembler. If this is not available, and the user wants to use CREF, the chosen assembler must be able to produce files that conform to the structure indicated in this section. (Also see "Format of CREF-compatible Files" in this section.

To create a cross-reference file during the assembly session, answer the fourth assembler command prompt with the filename by which the cross-reference file will be known.

The fourth assembler prompt is:

```
Cross-reference [NUL.CRF]:
```

Enter a filename in response to this prompt, or the assembler will not create a cross-reference file.

You may also specify which drive or device you want to receive the file, and what filename extension you want the file to have, if different from .CRF.

NOTE

> If you change the filename extension from .CRF to anything else, you must remember to specify the filename extension when naming the file in response to the first CREF prompt.

After assigning a filename in response to the fourth assembler prompt, you will be ready to convert the cross-reference file into a cross-reference listing using CREF.

USING CREF

General

CREF may be invoked by either of two methods:

1. Method 1: Type "CREF" then enter your commands in response to individual prompts.

2. Method 2: Type "CREF" and include all of your commands on the same line.

To summarize:

Method 1 CREF

Method 2 CREF <crffile>,<listing>

Method 1: Prompts

After CREF is invoked, it is loaded into memory and two text prompts are displayed. These prompts and the response criteria are:

Cross-reference [.CRF]:

Enter the name of the cross-reference file that you want CREF to convert into a cross-reference listing. The name of the file is the name you gave your assembler when you directed it to produce the cross reference file.

CREF assumes the filename extension is .CRF. If you do not specify a filename extension when you enter the name of the cross-reference file, CREF looks for a file name you specify with the filename extension .CRF. If your cross-reference file has a different extension, specify that extension when entering the filename.

See "FORMAT OF CREF COMPATIBLE FILES," for a description of what CREF expects to see in the cross-reference file. You will need this information only if your cross-reference file was not produced by a Microsoft assembler.

Listing [crffile.REF]:

Enter the name you want the cross-reference listing file. CREF automatically gives the cross-reference listing the filename extension .REF.

If you want you cross-reference listing to have the same filename as the cross-reference file, but with the filename extension .REF, simply press the carriage-return key when the Listing prompt appears. If the cross-reference listing is to have another name or filename extension, enter a response following the Listing prompt.

If you want the cross-reference listing placed on a drive or device other than the default drive, specify the drive or device when entering your response to the Listing prompt.

The following (two) special command characters are aids in entering CREF commands:

;

Use a single semicolon (;), followed immediately by a carriage return, any time after responding to the Cross-reference prompt, to select the default response to the Listing prompt. This feature saves time and overrides the need to answer the Listing prompt.

If you use the semicolon, CREF gives the listing file the same name as the cross-reference file, and adds the default filename extension .REF. For example:

```
Cross-reference [.CRF]: FUN;
```

CREF processes the cross-reference file named FUN.CRF and produces a listing file named FUN.REF.

CTRL-C

Use CTRL-C to abort the CREF session at any time. If an erroneous response (the wrong filename) or an incorrectly spelled filename is entered, press CTRL-C to exit CREF, then reinvoke CREF to start again. If the error has been typed but not entered, delete the erroneous characters, but only for that line.

Method 2: Command Line

After being invoked, CREF immediately proceeds to convert the cross-reference file into a cross-reference listing. The entries following CREF are responses to the command prompts. The <crffile> and <listing> fields must be separated by a comma.

where: <crffile> is the name of the cross-reference file produced by the assembler. CREF assumes that the filename extension is .CRF, which you may override by specifying a different extension. If the file named for <crffile> does not exist, CREF displays the message:

```
Fatal I/O Error 110
in File: <crffile>.CRF
```

Control then returns to the the operating system (MS-DOS).

<listing> is the name of the file that is to receive the cross-reference listing.

To select the default filename and extension for the listing file, enter a semicolon after the crffile name. For example:

```
CREF FUN;<CR>
```

This example causes CREF to process the cross-reference file FUN.CRF, and produce a listing file named FUN.REF.

To give the listing file a different name, extension, or destination, simply specify these differences when entering the command line. For example:

```
CREF FUN,B:WORK.ARG
```

This example causes CREF to process the cross-reference file named RUN.CRF, and produce a listing file named WORK.ARG, which will be placed on the disk in drive B:.

FORMAT OF CROSS-REFERENCE LISTINGS

The cross-reference listing is an alphabetic list of all the symbols in the program. Each page shows the title of the program or program module, followed by the list of symbols. Following each symbol name is a list of the line numbers where the symbol occurs in your program. The number of the line that contains the definition has a pound sign (#) appended to it.

A sample cross-reference listing appears on the following page:

```
CREF     (vers no.)     (date)

ENTX     PASCAL entry for initializing programs  <--comes from
                                                  TITLE directive

 Symbol Cross-reference     (# is definition)        Cref-1

AAAXQQ . . . . . .      37#   38

BEGHQQ . . . . . .      83    84#  154   176
BEGOQQ . . . . . .      33   162
BEGXQQ . . . . . .     113   126#  164   223

CESXQQ . . . . . .      97    99#  129
CLNEQQ . . . . . .      67    68#
CODE . . . . . .        37   182
CONST. . . . . .       104   104   105   110
CSXEQQ . . . . . .      65    66#  149

DATA . . . . . .        64#   64   100   110
DGROUP . . . . . .     110#  111   111   111   127   153   171
172
DOSOFF . . . . . .      98#  198   199
DOSXQQ . . . . . .     184   204#  219

ENDHQQ . . . . . .      87    88#  158
ENDUQQ . . . . . .      31#  197
ENDXQQ . . . . . .     184   194#
ENTXCM . . . . . .     182#  183   221

FREXQQ . . . . . .     169   170#  178

HDRFQQ . . . . . .      71    72#  151
HEAP . . . . . .        42    44   110
HEAPBEG. . . . . .      54#  153   172
HEAPLOW. . . . . .      43   171

INIUQQ . . . . . .      31   161

MAIN_STARTUP . . . .   109#  111   180
MEMORY . . . . . .      42    48#   48    49   109   110

PNUXQQ . . . . . .      69    70   150

REFEQQ . . . . . .      77    78#
RESEQQ . . . . . .      75    76#  148

SKTOP. . . . . .        59#
SMLSTK . . . . . .     135   137#
STACK. . . . . .        53#   53    60   110
STARTMAIN. . . . . .   163   186#  200
```

FORMAT OF CREF-COMPATIBLE FILES

NOTE

CREF can process files other than those generated by Microsoft assembler, if these files have a similar format.

CREF File Processing (General)

In essence, CREF reads a stream of bytes from the cross-reference file (or source file), sorts them, then emits them as a printable listing file (the .REF file). The symbols are held in memory as a sorted tree. References to the symbols are held in a linked list.

CREF keeps track of line numbers in the source file by the number of end-of-line characters encountered. Therefore, every line in the source file must contain at least an end-of-line character .

CREF attempts to place a heading at the top of every page of the listing. The name it uses as a title is the text passed by your assembler, from a TITLE (or similar) directive in your source program. The title must be followed by a title symbol. If CREF encounters more than one title symbol in the source file, it uses the last title read for all page headings. If CREF does not encounter a title symbol in the file, the title line on the listing is left blank.

Source File Format

First Three Bytes

CREF uses the first three bytes of the source file as format specification data. The rest of the file is processed as a series of records that either begins or ends with a byte that identifies the type of record. (The PAGE directive in the assembler, which accepts arguments for page length and line length, passes this information to the cross-reference file.)

The first three bytes are constructed as follows:

a. **First Byte** - The number of lines (1 to 255) to be printed per page.

b. **Second Byte** - The number of characters per line (1 to 132).

c. **Third Byte** - The page symbol (07) that tells CREF that the preceding two bytes define the listing page size.

If CREF does not find these first three bytes in the file, it uses default values for page size (page length = 58 lines, line length = 80 characters).

Control Symbols

Table 3-1 shows the types of records that CREF recognizes and the byte values and placement it uses to recognize record types.

Each record should have a control symbol (which identifies the record type) as either the first or last byte of the record.

Table 3-1. Recognition of Records

Records That Begin with a Control Symbol

Byte value	Control Symbol	Subsequent Bytes
01	Reference symbol	Record is a reference to a symbol name (1 to 80 characters)
02	Define symbol	Record is a definition of a symbol name (1 to 80 characters)
04	End of line	(none)
05	End of file	1AH

Records That End with a Control Symbol

Byte Value	Control Symbol	Preceding Bytes
06	Title defined	Record is title text (1 to 80 characters)
07	Page length/ line length	One byte for page length followed by one byte for line length

For all record types, the byte value represents a control character, as follows:

01 Control-A
02 Control-B
04 Control-D
05 Control-E
06 Control-F
07 Control-G

The control symbols are defined as follows:

1. **Reference symbol** - Record contains the name of a symbol that is referenced. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.

2. **Define symbol** - Record contains the name of a symbol that is defined. The name may be from 1 to 80 ASCII characters long. Additional characters are truncated.

3. **End of line** - Record is an end of line symbol character only (04H or Control-D).

4. **End of file** - Record is the end of file character (1AH)

5. **Title defined** - ASCII characters define the title to be printed at the top of each listing page. The title may be from 1 to 80 characters long. Additional characters are truncated. The last title definition record encountered is used as the title placed at the top of all pages of the listing. If a title definition record is not encountered, the title line on the listing is left blank.

6. **Page length/line length** - The first byte of the record contains the number of lines (1 to 255) to be printed per page. The second byte contains the number of characters (1 to 232) to be printed per page. The default page length is 58 lines. The default line length is 80 characters.

Following is a summary of the format of these records:

Byte Contents	Record Length
|01|symbol name|	2-81 bytes
|02|symbol name|	2-81 bytes
|04|	1 byte
|05|1A|	2 bytes
|title text|06|	2-81 bytes
|PL|LL|07|	3 bytes

CREF ERROR MESSAGES

All errors cause CREF to abort. After the cause has been found and corrected, CREF must be rerun. The following error messages listed below are displayed by CREF.

All CREF error messages appear in the following format:

```
Fatal I/O Error <error number>
in File:  <filename>
```

where: filename is the name of the file where the error occurs.

error number is one of the numbers in the following list of errors.

Number	Error		Explanation
101	**Hard data error**	-----	Unrecoverable disk I/O error
101	**Device name error**	-----	Illegal device specification (for example, X:FOO.CRF)
103	**Internal error**	-----	Report to Burroughs
104	**Internal error**	-----	Report to Burroughs
105	**Device offline**	-----	Disk drive door open, no printer attached, and so on.
106	**Internal error**	-----	Report to Burroughs
108	**Disk full**		
110	**File not found**		
111	**Disk is write protected**		
112	**Internal error**	-----	Report to Burroughs
113	**Internal error**	-----	Report to Burroughs
114	**Internal error**	-----	Report to Burroughs
115	**Internal error**	-----	Report to Burroughs

SECTION 4

DEBUG - FILE DEBUGGING UTILITY

WHAT DEBUG CAN DO

This utility provides you with a convenient means for examining, testing, modifying, and debugging machine-language object files. (Note that source files are altered by using EDLIN; DEBUG is the EDLIN counterpart for binary files.)

DEBUG eliminates the need to reassemble a program to see if a problem has been fixed by a minor change. It enables you to alter the contents of a file or CPU register, then immediately rerun the program to check the validity of the changes.

All DEBUG commands may be aborted at any time by pressing <CTRL-C>. To suspend display scrolling, press <CTRL-S>. Pressing any key other than <CTRL-C> or <CTRL-S> restarts the display. All of these commands are consistent with the control functions available at the MS-DOS command level.

USING DEBUG

To invoke this utility from the MS-DOS command mode, use the following syntax:

```
DEBUG  [<filespec> [<arglist>]]
```

An <arglist> may be specified if <filespec> is present. An <arglist> is a list of filename parameters and switches that are to be passed to the program <filespec>. Thus, when <filespec> is loaded into memory, it is loaded as if it had been started with the command:

```
<filespec> <arglist>
```

Here, <filespec> is the file to be debugged, and the <arglist> is the rest of the command line that is used when <filespec> is invoked and loaded into memory.

If a <filespec> is specified, the following is a typical command to start DEBUG:

```
DEBUG FILE.EXE
```

UG then loads FILE.EXE into the lowest available memory segment, rting at 100 hexadecimal. The BX:CX registers are loaded with number of bytes placed into memory.

no <filespec> is specified, then DEBUG is started with the mand:

```
DEBUG
```

UG then returns with its prompt, signaling that it is ready to ept your commands. Since no filename has been specified, rent memory, disk sectors, or disk files can be worked on by ng other commands.

NOTE

The DEBUG prompt is a hyphen (-).

CAUTION

1. When DEBUG (Version 2.0) is started, it sets up a program header at offset 0 in the program work area. You can overwrite this default header if no <filespec> is given to DEBUG. If you are debugging a .COM or .EXE file, however, do not tamper with the program header below address 5CH, or DEBUG will terminate.

2. Do not restart a program after the "Program terminated normally" message is displayed. You must reload the program with the N and L commands in order for it to run properly.

NERAL INFORMATION ON THE COMMANDS

3UG Command Structure

ı DEBUG command consists of a single letter followed by one or ɜ parameters. In addition, the control characters and command ions described previously in this section may be used within JG.

a syntax error occurs in a DEBUG command, DEBUG reprints the nand line and indicates the error with a caret (^) and the word :or."

```
dcs:100 cs:110
    ^ error
```

Any combination of uppercase and lowercase letters may be used in commands and parameters. The DEBUG commands are summarized in table 4-1, and are described later in detail.

Table 4-1. DEBUG Commands

DEBUG Command	Function
A[<address>]	Assemble
C<range> <address>	Compare
D[<range>]	Dump
E<address> [<list>]	Enter
F<range> <list>	Fill
G[=<address> [<address>...]]	Go
H<value> <value>	Hex
I<value>	Input
L[<address> [<drive><record><record>]]	Load
M<range> <address>	Move
N<filename>[<filename>]	Name
O<value> <byte>	Output
Q	Quit
R[<register-name>]	Register
S<range> <list>	Search
T[=<address>][<value>]	Trace
U[<range>]	Unassemble
W[<address> [<drive><record><record>]]	Write

UG Command Parameters

DEBUG commands accept parameters, except the Quit command. meters may be separated by delimiters (spaces or commas), but a miter is required only between two consecutive hexadecimal es. Thus, the following commands are equivalent:

```
dcs:100 110
d cs:100 110
d,cs:100,110
```

e 4-2 defines the DEBUG command parameters.

Table 4-2. DEBUG Command Parameters

meter	Definition
ve>	A 1-digit hexadecimal value to indicate which drive a file will be loaded from or written to. The valid values are 0 to 3. These values designate the drives as follows: 0=A>, 1=B>, 2=C>, 3=D>.
e>	A 2-digit hexadecimal value to be placed in, or read from, an address or register.
ord>	A 1- to 3-digit hexadecimal value used to indicate the logical record number on the disk and the number of disk sectors to be written or loaded. Logical records correspond to sectors, however, their numbering differs since they represent the entire disk space.
ue>	A hexadecimal value up to four digits used to specify a port number or the number of times a command should repeat its functions.
ress>	A two-part designation consisting of either an alphabetic segment register designation or a 4-digit segment address plus an offset value. The segment designation or segment address may be omitted, in which case the default segment is used. DS is the default segment for all commands except G, L, T, U, and W, for which the default segment is CS. All numeric values are hexadecimal.

Table 4-2. DEBUG Command Parameters (Cont.)

Parameter	Definition

Example:

```
CS:0100
04BA:0100
```

The colon is required between a segment designation (whether numeric or alphabetic) and an offset.

<range> Two <address>es: **e.g., <address> <address>**, or one <address>, an L, and a <value>: **e.g., <address> L <value>**, where <value> is the number of lines on which the command should operate, and L80 is assumed. The latter form cannot be used if another hex value follows the <range>, since the hex value would be interpreted as the second <address> of the <range>.

Examples:

```
CS:100 110
CS:100 L 10
CS:100
```

The following is illegal:

```
CS:100 CS:110
       ^ error
```

The limit for <range> is 10000 hex. To specify a <value> of 10000 hex within 4 digits, type 0000 (or 0).

<list> A series of <byte> values or <string>s. <list> must be the last parameter on the command line.

Example:

```
fcs:100 42 45 52 54 41
```

Table 4-2. DEBUG Command Parameters (Cont'd)

rameter	Definition
tring>	Any number of characters enclosed in quote marks. Quote marks may be either single (') or double ("). If the delimiter quote marks must appear within a <string>, the quote marks must be doubled. For example, the following strings are legal: 'This form of "string" is okay.' 'This form of ''string'' is okay.' Similarly, these strings are legal: "This form of 'string' is okay." "This form of ""string"" is okay." However, these strings are illegal: 'This form of 'string' is not valid.' "This form of "string" is not valid." Note that the double quote marks are not necessary in the following strings: "This form of ''string'' is not necessary." 'This form of ""string"" is not necessary.' The ASCII values of the characters in the string are used as a <list> of byte values.

:TAILED DESCRIPTIONS OF THE COMMANDS

: remainder of this section is an alphabetic presentation of the 3UG commands.

Assemble Command

FUNCTION:

Assembles 8086, 8087, and 8088 mnemonics directly into memory.

SYNTAX:

A[<address>]

REMARKS:

If a syntax error is found, DEBUG responds with

^Error

and displays the current assembly address.

All numeric values are hexadecimal and must be entered as 1 to 4 characters. Prefix mnemonics must be specified in front of the opcode to which they refer. They may also be entered on a separate line.

The segment-override mnemonics are CS:, DS:, ES:, and SS:. The mnemonic for the far return is RETF. String manipulation mnemonics must explicitly state the string size. For example, use MOVSW to move word-strings, and use MOVS to move byte-strings.

The assembler automatically generates short, near, or far jumps and calls to the destination address (depending on byte displacement). These jumps and calls may be overridden by the NEAR or FAR prefix. For example:

```
0100:0500  JMP   502; a 2-byte short jump
0100:0502  JMP   NEAR 505; a 3-byte near jump
0100:505   JMP   FAR  50A; a 5-byte far jump
```

The NEAR prefix may be abbreviated to NE, but the FAR prefix cannot be abbreviated.

DEBUG cannot tell whether some operands refer to a word-memory location or a byte-memory location. Therefore, the data type must be explicitly stated with the prefix "WORD PTR" or "BYTE PTR". Acceptable abbreviations are "WO" and "BY". For example:

```
NEG       BYTE PTR [128]

DEC       WO [SI]
```

DEBUG also cannot tell whether an operand refers to a memory location or to an immediate operand. DEBUG uses the common convention that operands enclosed in square brackets refer to memory. For example:

```
MOV       AX,21 ; Load AX with 21H
MOV       AX,[21]; Load AX with the contents
                  ; of memory location 21H
```

Two popular pseudo-instructions are available with Assemble. The DB opcode will assemble byte values directly into memory. The DW opcode will assemble word values directly into memory. For example:

```
DB        1,2,3,4,"THIS IS AN EXAMPLE"
DB        'THIS IS A QUOTE: "'
DB        "THIS IS A QUOTE: '"
DW        1000,2000,3000,"SOTAM"
```

Assemble supports all forms of register indirect commands. For example:

```
ADD       BX,34[BP+2].[SI-1]
POP       [BP+DI]
PUSH      [SI]
```

All opcode synonyms are also supported. For example:

```
LOOPZ 100
LOOPE 100
JA        200
JNBE 200
```

For 8087 opcodes, the WAIT or FWAIT must be explicitly specified. For example:

```
FWAIT FADD ST,ST(3)      ; This line will assemble
                         ; an FWAIT prefix
LD TBYTE PTR [BX]        ; This line will not
```

NCTION:

Compares the block of memory specified by <range> with a block of the same size beginning at <address>.

NTAX:

C<range> <address>

MARKS:

If the two areas of memory are identical, no display appears, and DEBUG returns with the MS-DOS prompt. If differences do exist, they are displayed in this format:

<address1> <byte1> <byte2> <address2>

AMPLES:

The following commands have the same effect:

```
C100,1FF 300
     or
C100L100 300
```

Both commands compare the block of memory from 100 to 1FFH with the block of memory from 300 to 3FFH.

Dump Command

FUNCTION:

Displays the contents of the specified block of memory.

SYNTAX:

D[<range>]

REMARKS:

If a range of addresses is specified, the contents of the range are displayed. If the D command is typed without parameters, 128 bytes are displayed at the first address (DS:100) after that displayed by the previous Dump command.

The dump is displayed in two portions: a hexadecimal dump (each byte is shown in hexadecimal form) and an ASCII dump (the bytes are shown as ASCII characters). Nonprinting characters are denoted by a period (.) in the ASCII portion of the display. Each display line contains 16 bytes, with a hyphen between the eighth and ninth bytes. Each displayed line begins on a 16-byte boundary.

If you type the command:

```
dcs:100 110
```

DEBUG displays the dump in the following format:

```
06DA:0100 52 2E 20 4D 41 54 4F 53 ... 20    C. DICKENS...
```

If you type the command:

```
D
```

the display is formatted as described previously. Each line of the display begins with an address, incremented by 16 above the address on the previous line. Each subsequent D (typed without parameters) displays the bytes immediately following those last displayed.

If you type the command:

```
DCS:100 L 20
```

the display is formatted as described previously, but 20H bytes are displayed.

If then you type the command:

```
DCS:100 115
```

the display is formatted as described previously, but all the bytes of lines 100H to 115H in the CS segment are displayed.

FUNCTION:

Enters byte values into memory at the specified <address>.

SYNTAX:

E<address>[<list>]

REMARKS:

If the optional <list> of values is typed, the replacement of byte values occurs automatically. (If an error occurs, no byte values are changed.)

If the <address> is typed without the optional <list>, DEBUG displays the address and its contents, then repeats the address on the next line and waits for your input. At this point, the Enter command waits for you to perform one of the following actions:

1. **Replace a byte value with a value you type.** Simply type the value after the current value. If the typed value is not one or two legal hexadecimal digits, illegal or extra characters are not echoed.

2. **Press the SPACE bar to advance to the next byte.** To change the value, simply type the new value as described in the preceding step. If you space beyond an 8-byte boundary, DEBUG starts a new display line with the address displayed at the beginning.

3. **Type a hyphen (-) to return to the preceding byte.** If you decide to change a byte before the current position, typing the hyphen returns the current position to the previous byte. When the hyphen is typed, a new line is started with the address and its byte value displayed.

4. **Press the <cursor return> key to terminate the Enter command.** The cursor-return key may be pressed at any byte position.

EXAMPLES:

If the following command is typed:

```
ECS:100
```

DEBUG displays:

```
04BA:0100 EB.[]
```

To change this value to 41, type 41 as shown:

```
04BA:0100 EB.41[]
```

To step through the subsequent bytes, press the SPACE bar to see:

```
04BA:0100 EB.41 10.  00.  BC.[]
```

To change BC to 42:

```
04BA:0100 EB.41 10.  00.  BC.42[]
```

Now, realizing that 10 should be 6F, type the hyphen as many times as needed to return to byte 0101 (value 10), then replace 10 with 6F:

```
04BA:0100 EB.41 10.  00.  BC.42[]
04BA:0102 00.[]
04BA:0101 10.6F[]
```

Pressing the cursor-return key ends the Enter command and returns to the DEBUG command level.

FUNCTION:

Fills the addresses in the <range> with the values in the <list>.

SYNTAX:

F<range> <list>

REMARKS:

If the <range> contains more bytes than the number of values in the <list>, the <list> is used repeatedly until all bytes in the <range> are filled. If the <list> contains more values than the number of bytes in the <range>, the extra values in the <list> are ignored. If any of the memory in the <range> is not valid (bad or nonexistent), the error will occur in all succeeding locations.

EXAMPLES:

If the following command is typed:

```
F04BA:100 L 100 52 2E 20 4D 41 54 4F 53 20 20
```

DEBUG fills memory locations 04BA:100 through 04BA:1FF with the bytes specified. The 10 values are repeated until all 100H bytes are filled.

FUNCTION:

Executes a program currently in memory.

SYNTAX:

G[=<address>[<address>...]]

REMARKS:

If only the Go command is typed, the program executes as if the program had run outside DEBUG.

If =<address> is set, execution begins at the address specified. The equal sign (=) is required so that DEBUG can distinguish the start =<address> from the breakpoint <address>es.

With the other optional addresses set, execution stops at the first <address> encountered (regardless of the position of that address in the list of addresses) in order to halt execution or program branching. When program execution reaches a breakpoint, the registers, flags, and the last decoded instruction are displayed . (The result is the same as if you had typed the Register command for the breakpoint address.)

Up to 10 breakpoints may be set. Breakpoints may be set only at addresses containing the first byte of an 8086 opcode. If more than 10 breakpoints are set, DEBUG returns the BP Error message.

The user stack pointer must be valid and have 6 bytes available for this command. The G command uses an IRET instruction to cause a jump to the program under test. The user stack pointer is set, and the user flags, code segment register, and instruction pointer are pushed onto the user stack. (Thus, if the user stack is not valid or is too small, the operating system may crash.) An interrupt code (0CCH) is placed at the specified breakpoint address(es). When an instruction with the breakpoint code is encountered, all breakpoint addresses are restored to their original instructions. If execution is not halted at one of the breakpoints, the interrupt codes are not replaced by the original instructions.

EXAMPLES:

If the following command is typed:

```
GCS:7550
```

the program currently in memory executes up to the address 7550 in the CS segment. Next, DEBUG displays registers and flags, then the Go command is terminated.

After a breakpoint has been encountered, if you type the Go command again, then the program executes just as if you had typed the filename at the MS-DOS command level. The only difference is that program execution begins at the instruction after the breakpoint rather than at the usual start address.

FUNCTION:

Performs hexadecimal arithmetic on the two specified parameters.

SYNTAX:

H<value> <value>

REMARKS:

First, DEBUG adds the two parameters, then subtracts the second parameter from the first. The results of the arithmetic are displayed on one line: first the sum, then the difference.

EXAMPLES:

If the following command is typed:

```
H19F 10A
```

DEBUG performs the calculations and then displays the result:

```
02A9   0095
```

FUNCTION:

Inputs and displays 1 byte from the port specified by <value>.

SYNTAX:

I<value>

REMARKS:

A 16-bit port address is allowed.

EXAMPLES:

Assume that you type the command:

```
I2F8
```

and the byte at the port is 42H, DEBUG inputs the byte and displays the value:

```
42
```

FUNCTION:

Loads a file from disk into memory.

SYNTAX:

L[<address> [<drive> <record> <record>]]

REMARKS:

Set BX:CX to the number of bytes read. The file must have been named either when DEBUG was started or with the N command. Both the DEBUG invocation and the N command format a filename properly in the normal format of a file control block at CS:5C.

If the L command is typed without any parameters, DEBUG loads the file into memory beginning at address CS:100 and sets BX:CX to the number of bytes loaded. If the L command is typed with an address parameter, loading begins at the memory <address> specified. If L is typed with all parameters, absolute disk sectors are loaded, not a file. The <record>s are taken from the <drive> specified (the drive designation is numeric here--0=A:, 1=B:, 2=C:, etc.); DEBUG begins loading with the first <record> specified, and continues until the number of sectors specified in the second <record> have been loaded.

EXAMPLES:

After the following commands are typed:

```
A:DEBUG
>NFILE.COM
```

To load FILE.COM, type:

```
L
```

DEBUG loads the file and then displays the DEBUG prompt. Assume that you want to load only portions of a file or certain records from a disk. To load these portions, type:

```
L04BA:100 2 0F 6D
```

DEBUG then loads into memory 109 (6D hex) records beginning with logical record number 15 at memory address 04BA:0100. When the records have been loaded, DEBUG simply returns the > prompt.

If the file has an .EXE extension, it is relocated to the load address specified in the header of the .EXE file: the <address> parameter is always ignored for .EXE files. The header itself is stripped off the .EXE file before it is loaded into memory. Thus the size of an .EXE file on disk differs from its size in memory.

If the file named by the Name command or specified when DEBUG is started is a .HEX file, then typing the L command with no parameters causes DEBUG to load the file, beginning at the address specified in the .HEX file. If the L command includes the option <address>, DEBUG adds the <address> specified in the L command to the address found in the .HEX file, to determine the starting address for loading the file.

FUNCTION:

Moves the block of memory specified by <range> to the location beginning at the specified <address>.

SYNTAX:

M<range> <address>

REMARKS:

Overlapping moves (i.e., moves where part of the block overlaps some of the current addresses) are always performed without loss of data. Addresses that can be overwritten are moved first. The sequence for moves from higher addresses to lower addresses is to move the data beginning at the lowest address of the block and work toward the highest. The sequence for moves from lower addresses to higher addresses is to move the data beginning at the highest address of the block and work toward the lowest.

Note that if the addresses in the block being moved will not have new data written to them, the data there before the move will remain. The M command copies the data from one area into another, in the sequence described, and writes over the new addresses. This action is why the sequence of the move is important.

EXAMPLES:

Assume that you type:

```
MCS:100 110 CS:500
```

DEBUG first moves address CS:110 to address CS:510, then CS:10F to CS:50F, and so on until CS:100 is moved to CS:500. You should type the D command, using the <address> typed for the M command, to review the results of the move.

FUNCTION:

Applies user-selected names to files, for later use.

SYNTAX:

N<filename>[<filename>...]

REMARKS:

The Name command performs two functions. First, Name is used to assign a filename for a later Load or Write command. Thus, if you start DEBUG without naming any file to be debugged, the N<filename> command must be typed before a file can be loaded. Second, Name is used to assign filename parameters to the file being debugged. In this case, Name accepts a list of parameters that are used by the file being debugged.

These two functions overlap. Consider the following set of DEBUG commands:

```
>NFILE1.EXE
>L
>G
```

Because of the effects of the Name command, Name performs the following steps:

1. (N)ame assigns FILE1.EXE to the filename to be used in any later Load or Write commands.

2. (N)ame also assigns FILE1.EXE as the first filename parameter used by any program that is later debugged.

3. (L)oad loads FILE1.EXE into memory.

4. (G)o causes FILE1.EXE to be executed with FILE1.EXE as the single filename parameter (that is, FILE1.EXE is executed as if FILE1.EXE had been typed at the command level).

A more useful chain of commands might look like this one:

```
>NFILE1.EXE
>L
>NFILE2.DAT FILE3.DAT
>G
```

Here, Name sets FILE1.EXE as the filename for the subsequent Load command. The Load command loads FILE1.EXE into memory; and the Name command is used again, this time to specify the parameters to be used by FILE1.EXE. Finally, when the Go command is executed, FILE1.EXE is executed as if FILE1 FILE2.DAT FILE3.DAT had been typed at the MS-DOS command level. Note that if a Write command is executed at this point, FILE1.EXE (the file being debugged) would be saved with the name FILE2.DAT!! To avoid such undesired results, always execute a Name command before either a Load or a Write.

Four regions of memory can be affected by the Name command:

```
CS:5C  FCB for file 1
CS:6C  FCB for file 2
CS:80  Count of characters
CS:81  All characters typed
```

A File Control Block (FCB) for the first filename parameter given to the Name command is set up at CS:5C. If a second filename parameter is typed, then an FCB is set up for it beginning at CS:6C. The number of characters typed in the Name command (exclusive of the first character, "N") is given at location CS:80. The actual stream of characters given by the Name command (again, exclusive of the letter "N") begins at CS:81. Note that this stream of characters may contain switches and delimiters that would be legal in any command typed at the MS-DOS command level.

EXAMPLES:

A typical use of the Name command is:

```
DEBUG PROG.COM
-NPARAM1 PARAM2/C
-G
-
```

In this case, the Go command executes the file in memory as if the following command line had been typed:

```
PROG PARAM1 PARAM2/C
```

Testing and debugging therefore reflect a normal run-time environment for PROG.COM.

FUNCTION:

Sends the specified <byte> to the output port specified by <value>.

SYNTAX:

O<value> <byte>

REMARKS:

A 16-bit port address is allowed.

EXAMPLES:

Type:

```
O2F8 4F
```

DEBUG outputs the byte value 4F to output port 2F8.

Quit Command

FUNCTION:

Terminates the DEBUG utility.

SYNTAX:

Q

REMARKS:

The Q command takes no parameters and exits DEBUG without saving the file currently operating. Control returns to the MS-DOS command level. command level.

EXAMPLES:

To end the debugging session, type:

Q<cursor return>

DEBUG has been terminated, and control returns to the MS-DOS command level.

FUNCTION:

Displays the contents of one or more CPU registers.

SYNTAX:

R[<register-name>]

REMARKS:

If no <register-name> is typed, the R command dumps the register save area and displays the contents of all registers and flags.

If a <register-name> is typed, the 16-bit value of that register is displayed in hexadecimal format, and then a colon appears as a prompt. You then either type a <value> to change the register, or simply press the cursor-return key if no change is desired.

The only valid <register-name>s are:

AX	BP	SS	
BX	SI	CS	
CX	DI	IP	(IP and PC both refer
DX	DS	PC	to the instruction
SP	ES	F	pointer.)

Any other entry for <register-name> results in a BR Error message.

If F is entered as the <register-name>, DEBUG displays each flag with a two-character alphabetic code. To alter any flag, type the opposite two-letter code. The flags are either set or cleared.

Table 4-3 shows the flags with their SET and CLEAR codes.

Table 4-3. Flags and Their Codes

Flag Name	Set	Clear
Overflow	OV	NV
Direction	DN Decrement	UP Increment
Interrupt	EI Enabled	DI Disabled
Sign	NG Negative	PL Plus
Zero	ZR	NZ
Auxiliary Carry	AC	NA
Parity	PE Even	PO Odd
Carry	CY	NC

Whenever you type the command RF, the flags are displayed in the order just shown. The flags appear in a row at the beginning of a line. At the end of the list of flags, DEBUG displays a hyphen (-). You may enter new flag values as alphabetic pairs. These new flag values can be entered in any order. You do not have to leave spaces between the flag entries. To exit the R command, press the cursor-return key. Flags not given new values remain unchanged.

If more than one value is entered for a flag, DEBUG returns a DF Error message. If you enter a flag code other than those just listed, DEBUG returns a BF Error message. In both cases, the flags up to the error in the list are changed; flags at and after the error are not.

At startup, the segment registers are set to the bottom of free memory, the instruction pointer is set to 0100H, all flags are cleared, and the remaining registers are set to zero.

EXAMPLES:

Type:

```
R
```

DEBUG displays all registers, flags, and the decoded instruction for the current location. If the location is CS:11A, the display is similar to the following example:

```
AX=0E00  BX=00FF  CX=0007  DX=01FF  SP=039D  BP=0000
SI=005C  DI=0000  DS=04BA  ES=04BA  SS=04BA  CS=04BA
IP=011A       NV  UP  DI  NG  NZ  AC  PE  NC
04BA:011A  CD21           INT         21
```

If you type:

```
RF
```

DEBUG displays the flags:

```
NV UP DI NG NZ AC PE NC - _
```

Now, type any valid flag designation, in any order, with or without spaces.

For example:

```
NV UP DI NG NZ AC PE NC - PLEICY<cursor return>
```

DEBUG responds only with the DEBUG prompt. To see the changes, type either the R or RF command:

```
RF
NV UP EI PL NZ AC PE CY - _
```

Press the cursor-return key to leave the flags this way, or to specify different flag values.

FUNCTION:

Searches the designated <range> for the specified <list> of bytes.

SYNTAX:

S<range> <list>

REMARKS:

The <list> may contain one or more bytes, each separated by a space or comma. If the <list> contains more than one byte, only the first address of the byte string is returned. If the <list> contains only one byte, all addresses of the byte in the <range> are displayed.

EXAMPLES:

If you type:

```
SCS:100 110 41
```

DEBUG displays a response similar to the following:

```
04BA:0104
04BA:010D
-[]
```

NCTION:

Executes one instruction and displays the contents of all registers and flags, and the decoded instruction.

NTAX:

T[=<address>][<value>]

MARKS:

If the optional =<address> is typed, tracing occurs at the =<address> specified. The optional <value> causes DEBUG to execute and trace the number of steps specified by <value>.

The T command uses the hardware trace mode of the 8086 or 8088 microprocessor. Consequently, you may also trace instructions stored in ROM (Read Only Memory).

AMPLES:

If you type:

```
T
```

DEBUG returns a display of the registers, flags, and decoded instruction for that one instruction. If the current position is 04BA:011A, DEBUG may return the display:

```
AX=0E00   BX=00FF   CX=0007   DX=01FF   SP=039D   BP=0000
SI=005C   DI=0000   DS=04BA   ES=04BA   SS=04BA   CS=04BA
IP=011A   NV   UP   DI   NG   NZ   AC   PE   NC
04BA:011A    CD21        INT             21
```

If you type:

```
T=011A 10
```

DEBUG executes 16 (10 hex) instructions beginning at 011A in the current segment, and then displays all registers and flags for each instruction as it is executed. The display scrolls upward until the last instruction is executed. Then the display stops, and you can see the register and flag values for the last few instructions performed. Remember that <CTRL-S> suspends the display at any point, so that you can study the registers and flags for any instruction.

Unassemble Command

FUNCTION:

Disassembles bytes and displays their corresponding source statements (with addresses and byte values).

SYNTAX:

U[<range>]

REMARKS:

The display of disassembled code looks like a listing of an assembler file. If you type the U command without parameters, 20 hexadecimal bytes are disassembled at the first address after that displayed by the previous Unassemble command. If you type the U command with the <range> parameter, then DEBUG disassembles all bytes in the range. If the <range> is given as an <address> only, then 20H bytes are disassembled instead of 80H.

EXAMPLES:

Type:

```
U04BA:100 L10
```

DEBUG disassembles 16 bytes beginning at address 04BA:0100:

```
04BA:0100  206472        AND     [SI+72],AH
04BA:0103  69            DB      69
04BA:0104  7665          JBE     016B
04BA:0106  207370        AND     [BP+DI+70],DH
04BA:0109  65            DB      65
04BA:010A  63            DB      63
04BA:010B  69            DB      69
04BA:010C  66            DB      66
04BA:010D  69            DB      69
04BA:010E  63            DB      63
04BA:010F  61            DB      61
```

If you type:

```
U04ba:0100 0108
```

the display will show:

```
04BA:0100  206472        AND     [SI+72],AH
04BA:0103  69            DB      69
04BA:0104  7665          JBE     016B
04BA:0106  207370        AND     [BP+DI+70],DH
```

If the bytes in some addresses are altered, the disassembler alters the instruction statements. The U command can be typed for the changed locations, the new instructions viewed, and the disassembled code used to edit the source file.

FUNCTION:

Writes the file being debugged to a disk file.

SYNTAX:

W[<address>[<drive> <record> <record>]]

REMARKS:

If you type W with no parameters, BX:CX must already be set to the number of bytes to be written; the file is written beginning from CS:100. If the W command is typed with just an address, then the file is written beginning at that address. If a G or T command has been used, BX:CX must be reset before using the Write command without parameters. Note that if a file is loaded and modified, the name, length, and starting address are all set correctly to save the modified file (as long as the length has not changed).

The file must have been named either when DEBUG was invoked, or with the N command. See the subsection "Name Command" described previously. Both the DEBUG invocation and the N command will correctly format a file name in the normal format of a file control block at CS:5C.

If the W command is typed with parameters, the write begins from the specified memory address. The file is written to the specified <drive> (the drive designation is numeric here--0=A>, 1=B>, 2=C>, etc.). DEBUG writes the file beginning at the logical record number specified by the first <record>. DEBUG continues to write the file until the number of sectors specified in the second <record> have been written.

CAUTION

Writing to absolute sectors is EXTREMELY dangerous because the process bypasses the file handler.

EXAMPLES:

If you type:

```
W
```

DEBUG writes the file to disk and then displays the DEBUG prompt, as shown in the following two examples:

```
W
-[]
```

If you type:

```
WCS:100 1 37 2B
```

DEBUG writes out the contents of memory, beginning with the address CS:100 to the disk in drive B>. The data written out start in disk logical record number 37H and consist of 2BH records. When the write is complete, DEBUG displays the prompt:

```
WCS:100 1 37 2B
-[]
```

EBUG ERROR MESSAGES

ring the DEBUG session, you may receive any of the following ror messages. Each error terminates the DEBUG command under ich it occurred, but does not terminate DEBUG itself.

ror Code	Definition
BF	**Bad flag** You attempted to alter a flag, but the characters typed are not one of the acceptable pairs of flag values. See the Register command for the list of acceptable flag entries.
BP	**Too many breakpoints** You specified more than 10 breakpoints as parameters to the G command. Retype the Go command with 10 or fewer breakpoints.
BR	**Bad register** You typed the R command with an invalid register name. See the Register command for the list of valid register names.
DF	**Double flag** You typed two values for one flag. You may specify a flag value only once per RF command.

SECTION 5

FC - FILE COMPARISON UTILITY

WHAT FC CAN DO

It is sometimes useful to compare files on your disk. If you have copied a file, and later want to compare copies to see which one is current, you can use the File Comparison (FC) Utility.

FC compares the contents of two files. The differences between the two files can be output to the console or to a third file. The files being compared may be either source files (containing the input statements of a programming language) or binary files (the output of an assembler, linker, or compiler).

The comparisons are made in one of two ways: line by line or byte by byte. The line-by-line comparison isolates blocks of lines that differ between the two files, and displays those blocks of lines. The byte-by-byte comparison displays the bytes that differ between the two files.

FC uses a large amount of memory as buffer (storage) space to hold source files. If the source files are larger than available memory, FC compares the portions that can be loaded into the buffer space. If no lines match in the portions of the files in the buffer space, FC displays the message:

```
FILES ARE DIFFERENT
```

For binary files larger than available memory, FC compares both files completely, overlaying the portion in memory with the next portion from disk. All differences are output in the same manner as those files that completely fit in memory.

NOTE

All file specifications use the following syntax. For detailed information, see the subsection "USING FC" described in this section.

[d:]<filename>[<.ext>]

where:

d: is the letter designating a disk drive. If the drive designation is omitted, FC defaults to the (current) default drive of the operating system

filename is the one- to eight-character name of the file.

.ext is a one- to three-character extension to the filename.

JSING FC

The syntax of FC is as follows:

FC [/# /B /W /C] <filename1> <filename2>

C matches the first file (filename1) against the econd(filename2) and reports any differences between them. Both ilenames can be pathnames. For example:

FC B:/YING/YANG/FILE1.TXT /YANG/FILE2.TXT

C takes FILE1.TXT in the /YING/YANG directory of disk B and ompares it with FILE2.TXT in the /YANG directory. Since no rive is specified for filename2, FC assumes that the /YANG irectory is on the disk in the default drive.

FC SWITCHES

Four switches can be used with the File Comparison Utility:

/B Forces a binary comparison of both files. The two files are compared byte by byte, with no attempt to resynchronize after a mismatch. The mismatches are printed as follows:

```
--ADDRS----F1----F2-
xxxxxxxx   yy    zz
```

where: xxxxxxxx is the relative address of the pair of bytes from the beginning of the file. Addresses start at 00000000; yy and zz are the mismatched bytes from file1 and file2, respectively. If one of the files contains less data than the other, a message is displayed. For example, if file1 ends before file2, then FC displays:

```
***Data left in F2***
```

/# # stands for a number from 1 to 9. This switch specifies the number of lines required to match, for the files to be considered as matching again after a difference has been found. If this switch is not specified, it defaults to 3. This switch is used only in source comparisons.

'W Causes FC to compress whites (tabs and spaces) during the comparison. Thus, multiple contiguous whites in any line are considered as a single white space. Note that although FC compresses whites, it does not ignore them. The two exceptions are beginning and ending whites in a line, which are ignored. For example (note that an underscore represents a white):

```
___More__data_to_be_found____
```

will match with

```
More_data_to_be_found
```

and with

```
_______More_______data_to_be_____found_______
```

but will not match with

```
____Moredata_to_be_found
```

This switch is used only in source comparisons.

C Causes the matching process to ignore the case of letters. All letters in the files are considered uppercase letters. For example,

```
Much_MORE_data_IS_NOT_FOUND
```

matches

```
much_more_data_is_not_found
```

If both switches and /C options are specified, then FC compresses whites and ignores case: For example,

```
___DATA_was_found____
```

matches:

```
data_was_found
```

This switch is used only in source comparisons.

DIFFERENCE REPORTING

FC reports the differences between the two files you specify by displaying the first filename, followed by the lines that differ between the files, followed by the first line to match in both files. FC then displays the name of these file followed by the lines that are different, followed by the first line that matches. The default for the number of lines to match between the files is 3. (If you want to change this default, specify the number of lines with the /# switch.) For example:

```
        ...
        ...

----------<filename1>
<difference>
<1st line to match file2 in file1>

----------<filename2>
<difference>
<1st line to match file1 in file2>

-----------------------------------------------

        ...
        ...
```

FC continues to list each difference. If there are too many differences (involving too many lines), the program simply reports that the files are different and stops.

If no matches are found after the first difference, FC displays:

```
*** Files are different ***
```

and returns to the MS-DOS default drive prompt (A>, for example).

REDIRECTING FC OUTPUT TO A FILE

The differences and matches between the two files you specify are displayed on your screen unless you redirect the output to a file.

To compare File1 and File2, and send the FC output to a file named DIFFER.TXT, type:

```
FC File1 File2 > DIFFER.TXT
```

The differences and matches between File1 and File2 will be put into DIFFER.TXT on the default drive.

FILE COMPARISON EXAMPLES

Example 1

Assume that the following two ASCII files currently reside on disk:

ALPHA.ASM FILE A	BETA.ASM FILE B
A	A
B	B
C	C
D	G
E	H
F	I
G	J
H	1
I	2
M	P
N	Q
O	R
P	S
Q	T
R	U
S	V
T	4
U	5
V	W
W	X
X	Y
Y	Z
Z	

To compare the two files and display the differences on the screen, type:

```
FC ALPHA.ASM BETA.ASM
```

FC compares ALPHA.ASM with BETA.ASM and displays the differences on the workstation screen. All other defaults remain intact.

(The defaults are: do not use tabs, spaces, or comments for matches, and do a source comparison on the two files.)

The output will appear as follows on the workstation screen (but the notes do not appear):

```
----------ALPHA.ASM
D                          NOTE: ALPHA file
E                          contains defg;
F                          BETA contains g.
G

----------BETA.ASM
G


--------------------------------


----------ALPHA.ASM
M                          NOTE: ALPHA file
N                          contains mno; BETA
O                          contains j12.
P

----------BETA.ASM
J
1
2
P


--------------------------------


----------ALPHA.ASM
W                          NOTE: ALPHA file
                           contains w; BETA
----------BETA.ASM         contains 45w.
4
5
W
```

Example 2

You can output the differences to the printer using the same two source files. In this example, four successive lines must be the same to constitute a match. Type:

```
FC -4 ALPHA.ASM BETA.ASM > PRN
```

The following output appears on the printer:

```
----------ALPHA.ASM
D
E
F
G
H
I
M
N              NOTE: p is the first of
O              a string of four matches.
P

----------BETA.ASM
G
H
I
J
1
2
P


------------------------------


----------ALPHA.ASM
W                        NOTE: w is the first of a
                         string of four matches.
----------BETA.ASM
4
5
W
```

Example 3

This example forces a binary comparison and then displays the differences on the workstation screen using the same two source files that were used in the previous examples. Type:

```
FC /B ALPHA.ASM BETA.ASM
```

The /B switch in this example forces a binary comparison. This switch, and any others, must be typed before the filenames in the FC command line. The following display should appear:

```
  ADDRS      F1    F2
--------------------------
00000009     44    47
0000000C     45    48
0000000F     46    49
00000012     47    4A
00000015     48    31
00000018     49    32
0000001B     4D    50
0000001E     4E    51
00000021     4F    52
00000024     50    53
00000027     51    54
0000002A     52    55
0000002D     53    56
00000030     54    34
00000033     55    35
00000036     56    57
00000039     57    58
0000003C     58    59
0000003F     59    5A
00000042     5A    1A
```

FC ERROR MESSAGES

When the File Comparison Utility detects an error, one or more of the following error messages is displayed:

Incorrect DOS version
You are running FC under a version of MS-DOS that is not 2.0 or higher.

Invalid parameter:<option>
One of the switches you have specified is invalid.

File not found:<filename>
FC could not find the filename you specified.

Read error in:<filename>
FC could not read the entire file.

Invalid number of parameters
You have specified the wrong number of options on the FC command line.

SECTION 6

LIB - LIBRARY MANAGER UTILITY

WHAT LIB CAN DO

The Library Manager utility (LIB) creates and modifies library files that are used with the Linker utility (LINK - see section 7). The LIB utility can:

1. Add object files to a library.

2. Delete modules from a library.

3. Extract modules from a library.

4. Place the extracted modules into separate object files.

LIB provides a means of creating both general and special libraries for a variety of programs or for specific programs. With LIB, you can create a library for a language compiler or a single program (which would permit fast linking and possibly more efficient execution).

You can modify individual modules within a library by extracting the modules, making changes, then adding the modules to the library again. You can also replace an existing module with a different module or with a new version of an existing module.

The command scanner in LIB is the same as the one used in the Microsoft MS-LINK, MS-Pascal, MS-FORTRAN, and other Microsoft 16-bit products. The command syntax is straightforward, and LIB prompts you for any needed commands that you have not supplied.

The LIB utility requires a minimum of 38 kilobytes of memory, apportioned as follows:

1. 28 K bytes for code

2. 10 K bytes for run space.

LIB performs two basic actions: deleting modules from a library file and changing object files into modules and appending them to a library file. These two actions underlie five library manager functions:

1. Deleting a module.

2. Extracting a module and placing it in a separate object file.

3. Appending an object file to a library, as a module.

4. Replacing a module in the library file with a new module.

5. Creating a library file.

During each library session, LIB first deletes or extracts modules, then appends new ones. In a single operation, LIB reads each module into memory, checks it for consistency, and writes it back to the file.

If you delete a module, LIB reads in that module but does not write it back to the file. When LIB writes back the next module to be retained, it places the module at the end of the last module written.

This procedure effectively "closes up" the disk space to keep the library file from growing larger than necessary. When LIB has read through the whole library file, it appends any new modules to the end of the file.

Finally, LIB creates the index that LINK uses to find modules and symbols in the library file. (If the user requests, LINK outputs a cross-reference listing of the public symbols in the library.) For example:

```
LIB PASCAL+HEAP-HEAP;
```

First deletes the library module HEAP from the library file, then adds the file HEAP.OBJ as the last module in the library. This order of execution prevents confusion in LIB when a new version of a module replaces an old version in the library file. Note that the replace function is simply the delete-append functions in succession. Note, also, that you can specify delete, append, or extract functions in any order. The order is insignificant to the LIB command scanner.

USING LIB

General

LIB commands are usually entered on your keyboard. As an option, answers to the command prompts may be contained in a response file. Several command characters are provided; some are required parts of the LIB commands, and others assist you in entering LIB commands.

Invoking LIB

LIB may be invoked by one of three methods:

1. Method 1: Type "LIB" then enter your commands as answers to individual prompts.

2. Method 2: Type "LIB"; and include all of your commands on one line.

3. Method 3: Create a response file that contains all necessary commands; then type "LIB" followed by the location and name of the response file.

To summarize:

Method 1 LIB

Method 2 LIB <library><operations>,<listing>

Method 3 LIB @<filespec>

Method 1: Prompts

After invocation, LIB is loaded into memory. It then diplays three text prompts, one at a time. Answer the prompts to command LIB to perform the necessary tasks.

Table 6-1 summarizes the command prompts and command characters which are described in more detail in the subsections "COMMAND PROMPTS" and "COMMAND CHARRACTERS," respectively.

Table 6-1. Command Prompts and Characters

Prompt	Response
Library file:	List filename of library to be manipulated (default: filename extension .LIB).
Operation:	List command character(s) followed by module name(s) or object filename(s) (default action: no changes; default object filename extension: .OBJ).
List file:	List filename for a cross-reference listing file (default: NUL; no file).

ummary of Command Characters

Character	Action
+	Append an object file as the last module.
-	Delete a module from the library.
*	Extract a module and place in an object file.
;	Use default responses to remaining prompts.
&	Extend current physical line; repeat command prompt.
Control-C	Abort library session.

Method 2: Command Line

Type all commands on one line. The entries following LIB are responses to the command prompts. The <library> and <operations> fields and all operations entries must be separated by one of three command characters (+, -, or *). If a cross-reference listing is desired, the name of the file must be separated from the last operations entry by a comma. Use the following syntax:

```
LIB <library><operations>,<listing>
```

where: library is the name of a library file. LIB assumes that the filename extension is .OBJ, which you may override by specifying a different extension. If the filename supplied for the library field does not exist, LIB prompts you:

```
Library file does not exist. Create?
```

Enter Yes (or any response beginning with Y) to create a new library file. Enter No (or any other response not beginning with Y) to abort the library session.

operations is deleting a module, appending an object file as a module, or extracting a module as an object file from the library file. Use the three command characters which are plus sign, minus sign, and asterisk (+,-,*)to tell LIB what to do with each module or object file.

listing is the name of the file to receive the cross-reference listing of public symbols in the library modules. The list is compiled after all module manipulation has taken place.

To select the default for remaining field(s), enter the semicolon command character.

If you enter a Library filename followed by a semicolon, LIB reads through the library file and performs a consistency check without altering modules.

If you enter a Library filename followed by a comma and a List and filename, LIB performs its consistency check on the library file then produce the cross-reference listing file.

The three examples that follow should be helpful:

Example 1:

```
LIB PASCAL-HEAP+HEAP;
```

This command causes LIB to delete the module HEAP from the library file PASCAL.LIB, then append the object file HEAP.OBJ as the last module of PASCAL.LIB (the module will be named HEAP).

NOTE

If you have many operations to perform during a library session, use the ampersand (&) command character to extend the line so that you can enter additional object filenames and module names. Always include one of the operation command characters (+, -, *) before the name of each module or object file.

Example 2:

```
LIB PASCAL<CR>
```

This command causes LIB to perform a consistency check on the library file PASCAL.LIB. No other action is performed.

Example 3:

```
LIB PASCAL,PASCROSS.PUB
```

This command causes LIB to perform a consistency check on the library file PASCAL.LIB, then output a cross-reference listing file named PASCROSS.PUB.

Method 3: Response File

To invoke link using Method 3, use the syntax:

```
LIB @<filespec>
```

where: filespec is the name of a response file. A response file contains answers to the LIB prompts; enabling you to conduct the LIB session without interactive (direct) responses to the LIB prompts (described fully in the subsection "USING LIB."

CAUTION

Before using method 3 to invoke LIB, you must first create the response file.

A response file contains text lines, one for each prompt. Responses must appear in the same order as the command prompts.

Use command characters in the response file in the same manner as they are used for responses entered on the keyboard.

When the library session begins, each prompt is displayed in turn with the responses from the response file. If the response file does not contain answers for all the prompts, LIB uses the default responses. The modules currently in the library file are not changed, and no cross-reference listing file is created.

If you enter a library filename followed by a semicolon, LIB reads through the library file and performs a consistency check without altering modules.

If you enter a library filename followed by a carriage return, comma, and list filename, LIB performs a consistency check on the library file, then produces the cross-reference listing file.

Example:

```
PASCAL<CR>
+CURSOR+HEAP-HEAP*FOIBLES<CR>
CROSSLST<CR>
```

This response file causes LIB to:

1. Delete the module HEAP from the PASCAL.LIB library file.

2. Extract the module FOIBLES and place it in an object file named FOIBLES.OBJ.

3. Append the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library.

4. Create a cross-reference file named CROSSLST.

COMMAND PROMPTS

LIB is invoked by entering responses to three text prompts. After entering your response to the current prompt, the next prompt appears. After the last prompt has been answered, LIB performs its library management functions without further command. When the library session is finished, LIB exits to MS-DOS. If the MS-DOS prompt is displayed, the library session has been successfully completed. If the library session is unsuccessful, LIB returns the appropriate error message.

LIB prompts you for the name of the library file, the operation(s) to perform, and the name for the cross-reference listing file, if any.

Library file: Enter the name of the library file that you want to manipulate. LIB assumes that the filename extension is .LIB. You can override this assumption by supplying a filename extension when you enter the library filename. Because LIB can manage only one library file at a time, enter only one filename is allowed in response to this prompt. Additional responses, except the semicolon command character, are ignored.

If you enter a library filename followed by a semicolon command character, LIB performs a consistency check before returning to the operating system. Any errors in the file will be reported.

If the filename you enter does not exist, LIB returns the prompt:

```
Library file does not exist. Create?
```

Enter Y to create a new file. LIB checks this response to verify that letter Y is the first character; if any other character is entered first, LIB returns to MS-DOS.

Operation: Enter one of the three command characters for manipulating modules (+, -, *), followed immediately (no space) by the module name or the object filename. The plus sign (+) appends an object file as the last module in the library file (see "COMMAND CHARACTERS" in this section for a description of command characters). A minus sign deletes a module from the library file. An asterisk extracts a module from the library and places it in a separate object file with the filename taken from the module name (and filename extension .OBJ).

When you have a large number of modules to manipulate (more than can be typed on one line), enter an ampersand (&) as the last character on the line. LIB repeats the Operation prompt, which permits you to enter additional module names and object filenames.

With LIB, operations on modules and object files can be entered in any order.

The command character descriptions provide more information about order of execution and actions of LIB on each module.

List file: To generate a cross-reference list of those public symbols in the library file modules after manipulations, enter a filename (the one into which LIB is to place the cross-reference listing). If no filename is entered, no cross-reference listing is generated (a NUL file).

Because the response to the list file prompt is a filespec, you can specify, along with the filename, a drive (or device) designation and a filename extension. The list file is not given a default filename extension. To give the file a filename extension, specify it when entering the filename.

The cross-reference listing file contains two lists. The first is an alphabetic listing of all public symbols. Each symbol name is followed by the name of its module. The second is an alphabetic list of the modules in the library. Under each module name is an alphabetic listing of the public symbols in that module.

COMMAND CHARACTERS

LIB provides six command characters. Three are required in response to the Operation prompt; the remaining three provide additional helpful LIB commands.

+ The plus sign followed by an object filename appends the object file to the last library module named in response to the Library file prompt. When LIB sees the plus sign, it assumes that the filename extension is .OBJ. To override this assumption, specify a different filename extension.

  LIB strips the drive designation and extension from the object file specification, leaving only the filename. For example, if the object file to be appended as a module to a library is:

  ```
  B:CURSOR.OBJ
  ```

  The following operation-prompt response:

  ```
  +B:CURSOR.OBJ
  ```

  causes LIB to strip off B: and .OBJ, leaving only CURSOR, which becomes a module named CURSOR in the library.

  NOTE

  > The distinction between an object file and a module (or object module) is that the file possesses a drive designation (even if it is the default drive) and a filename extension; object modules possess neither.

  The minus sign followed by a module name deletes that module from the library file. LIB then "closes up" the file space left empty by the deletion. This cleanup action keeps the library file from expanding with empty space. New modules, even replacement modules, are added to the end of the file, not the space vacated by deleted modules.

* The asterisk followed by a module name causes that module to be extracted from the library file and placed into a separate object file. (Here, extract means, to copy the module to a separate object file.) The module still exists in the library. LIB uses the module name as the filename and adds the default drive designation and filename extension .OBJ. For example, if the module to be extracted is:

    CURSOR

    and the current default disk drive is A:, a response to the operation prompt of:

    *CURSOR

    causes LIB to extract the module named CURSOR from the library file and set it up as an object file with the file specification of:

    default drive:CURSOR.OBJ

    (The drive designation and filename extension cannot be overridden. You can, however, rename the file, by supplying a new filename extension, or copy the file to a new disk drive, by supplying a new filename and filename extension.)

; Use a single semicolon (;) followed immediately by a carriage return any time after responding to the first prompt (from Library file on), to select default responses to the remaining prompts. This feature saves time and overrides the need to answer additional prompts.

NOTE

Once the semicolon has been entered, you can no longer respond to any of the prompts for that library session. Therefore, do not use the semicolon to skip over prompts. Use the cursor-return key for this purpose.

Example:

```
Library file: FUN <CR>
Operation: +CURSOR;<CR>
```

The remaining prompt(s) do not appear, and LIB uses the default value (no cross-reference file).

& Use the ampersand to extend the current physical line. This command character is needed only for the Operation prompt. LIB can perform many functions during a single library session. The number of modules that can be appended is limited only by the disk space available. The number of modules that can be replaced or extracted is also limited only by disk space. The number of modules that can be deleted is limited only by the number of modules in the library file. However, the line length for a response to any prompt is limited to the line length of the system. For a large number of responses to the Operation prompt, place an ampersand at the end of a line. LIB again displays the Operation prompt, then you may enter more responses. You may use the ampersand character as many times as required. For example:

```
Library file: FUN<CR>
Operation: +CURSOR-HEAP+HEAP*FOIBLES&
Operation: *INIT+ASSUME+RIDE;<CR>
```

LIB deletes the module HEAP, extracts the modules FOIBLES and INIT (creating two files, FOIBLES.OBJ and INIT.OBJ), then appends the object files CURSOR, HEAP, ASSUME, and RIDE. Note, however, that LIB allows the user to enter the Operation reponses in any order.

CTRL-C Use Control-C at any time to abort the library session. If an erroneous response (such as a wrong or misspelled filename or module name) is entered, first press CTRL-C to exit LIB; then reinvoke LIB and start again. If the error has been typed but not entered, delete the erroneous characters, but only for that line.

LIB ERROR MESSAGES

When LIB encounters an error, one of the following messages is displayed:

<symbol> is a multiply defined PUBLIC. Proceed?
Cause: Two modules define the same public symbol. The user must confirm that the old symbol definition has been removed. A "No" response leaves the library in an undetermined state.

Cure: Remove the PUBLIC declaration from one of the object modules and recompile or reassemble.

Allocate error on VM.TMP
Cause: Insufficient disk space.

Cannot create extract file
Cause: No room in directory for extract file.

Cannot create list file
Cause: No room in directory for library file.

Cannot nest response file
Cause: '@filespec' in response (or indirect) file.

Cannot open VM.TMP
Cause: No room for VM.TMP in disk directory.

Cannot write library file
Cause: Insufficient disk space.

Close error on extract file
Cause: Insufficient space, because of an internal error. Contact your Burroughs representative.

Fatal Error: Cannot open input file
Cause: Mistyped object file name.

Fatal Error: Module is not in the library
Cause: Trying to delete a module that is not in the library.

Input file read error
Cause: Bad object module or faulty disk.

Invalid object module/library
Cause: Bad object and/or library.

Library Disk is full
Cause: No more room on diskette.

Listing file write error
Cause: Insufficient disk space.

No library file specified
Cause: No response to Library File prompt.

Read error on VM.TMP
Cause: Disk not ready for read.

Symbol table capacity exceeded
Cause: Too many public symbols (about 30 K characters in symbols).

Too many object modules
Cause: More than 500 object modules.

Too many public symbols
Cause: Exceeded maximum of 1024 public symbols.

Write error on library/extract file
Cause: Insufficient disk space.

Write error on VM.TMP
Cause: Insufficient disk space.

SECTION 7

LINK - LINKER UTILITY

WHAT LINK CAN DO

The Linker utility is a tool that:

1. Combines separately produced 8086 object-code modules into one relocatable load module ("run" file).

2. Searches library files for definitions of unresolved external references.

3. Resolves external cross-references.

4. Produces a listing that shows both the resolution of external references and error messages.

5. Uses available memory, as much as possible; when available memory is exhausted, LINK creates a disk file and becomes a virtual linker.

NOTE

To compile and link programs, read this entire section before using LINK. If not, skip this section.

The LINK utility requires a minimum of 50 kilobytes of memory, apportioned as follows:

1. 40 K bytes for code and data and

2. 10 K bytes for run space.

GENERAL INFORMATION

When you write a program, it is in source code. This source code is converted into object modules by a compiler. These object modules must be correctly combined with other object modules, by the link process, to produce the program (executable file) that you will subsequently run.

You may want to link (combine) several programs and run them together. Each program may refer to a symbol that is defined in another object module. This symbol definition is called an external reference.

As LINK combines modules, it makes certain that all external references between object modules are defined. LINK can search several library files for the definitions of any external references that are not defined in the object modules.

LINK also produces a list file that shows external references resolved and displays any error messages. LINK uses available memory as much as possible.

NOTE

When available memory is exhausted, LINK creates a temporary diskfile named VM.TMP.

CAUTION

Do not use VM.TMP as a filename for any of your files. If LINK requires a VM.TMP file, it deletes the VM.TMP already on disk and creates a new VM.TMP. The contents of the previous VM.TMP are lost.

SPECIAL LINK TERMS

A few terms terms used in this section are explained to help you understand how LINK works. Generally, if you are linking object modules compiled from BASIC, Pascal, FORTRAN, or any other high-level language, you will not need to know these terms. However, if you are writing and compiling programs in assembly language, you will need to understand the following terms.

In MS-DOS, memory can be divided into segments, classes and groups.

A segment is a contiguous area of memory up to 64 K-bytes in length. A segment may be located anywhere on a 16 byte (paragraph) boundary. The contents of a segment are addressed by a segment-register/offset pair.

A class is a collection of segments. Each segment has a segment name and a class name. LINK loads all segments into memory by class name, from the first segment encountered to the last. All segments assigned to the same class are loaded into contiguous memory areas.

During processing, LINK refers to segments by their memory addresses. LINK performs this task by finding groups of segments.

A group is a collection of segments that fit within a 64 K-byte area of memory. These segments need not be contiguous to form a group (see illustration). The address of any group is the address of the lowest segment in that group. At linking time, LINK analyzes the groups, then refers to the segments by the memory address of that group. A program may consist of one or more groups.

When writing in assembly language, you may assign the group and class names in your program. In high-level languages (BASIC,COBOL, FORTRAN, Pascal), the naming is done automatically by the compiler.

Refer to the MS-DOS Programmer Manual for information on how to assign group and class names and how to use for LINK combining and arranging segments in memory.

LINK FILES USAGE

LINK uses files in the following ways:

1. Works with one or more input files.

2. Produces two output files.

3. May create a temporary disk file.

4. May be directed to search up to eight library files.

For each type of file, the user may supply a three-part file specification. The format for LINK file specifications is the same as that for a disk file:

```
[d:]<filename>[<.ext>]
```

where: d: is the drive designation. Permissible drive designations for LINK are A: through O:. The colon is always required as part of the drive designation.

filename is any legal filename of one to eight characters.

.ext is a one- to three-character extension to the filename. The period is a required part of the extension.

Input File Extensions

If no filename extensions are supplied in the input (object) file specifications, LINK recognizes the following extensions by default:

.OBJ	Object
.LIB	Library

Output File Extensions

LINK appends the following default extensions to the output (run and list) files:

.EXE	Run (may not be overridden)
.MAP	List (may be overridden)

VM.TMP (Temporary) File

INK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, LINK creates a temporary file named VM.TMP and stores it on the disk in the default drive. If LINK creates VM.TMP, it displays the message:

```
VM.TMP has been created.
Do not change disk in drive <d:>
```

When this message is displayed, you must not remove the disk from the default drive until the link session ends. If the disk is removed, LINK operates unpredictably and may display the error message:

```
Unexpected end of file on VM.TMP
```

The contents of VM.TMP are written to the file named following the "Run File:" prompt. VM.TMP is a working file only and is deleted at the end of the linking session.

CAUTION

Do not use VM.TMP as a filename for any of your files. If LINK requires a VM.TMP file, it deletes the VM.TMP already on disk and creates a new VM.TMP. The contents of the previous VM.TMP are lost.

ISING LINK

ieneral

INK may be invoked in any of the three following ways:

- Method 1: Type LINK, then enter your commands in response to individual prompts.
- Method 2: Type LINK, and include all of your commands on the same line.
- Method 3: Create a response file that contains all of your commands then type LINK and indicate where to locate that file.

To summarize:

Method 1	LINK
Method 2	LINK <filenames>[-switches]
Method 3	LINK @<filespec>

lethod 1: Prompts

fter being invoked, LINK is loaded into memory. It then isplays four text prompts, one at a time. Answer the prompts to ndicate what tasks LINK is to perform.

t the end of each line, you may type one or more switches, receded by the switch delimiter (/).

able 7-1 summarizes the command prompts, which are described in etail later in this section.

Table 7-1. Command Prompts

Prompt	Responses
Object Modules [.OBJ]:	List .OBJ files to be linked. They must be separated by blank spaces or plus signs (+). If a plus sign is the last character typed, the prompt reappears. No default exists; a response is required.

Table 7-1. Command Prompts (Cont.)

Prompt	Responses
Run File [Object-file.EXE]:	Type filename for executable object code. The default is first-object-filename.EXE. (You cannot change the output extension.)
List File [Run-file.MAP]:	Type filename for listing. The default is RUN filename.
Libraries [ ]:	List filenames to be searched, separated by spaces or plus signs (+). If a plus sign is the last character typed, the prompt reappears. The default is no search. (Extensions will be changed to .LIB.)

Method 2: Command Line

Type all commands on one line. The entries following LINK are responses to the command prompts. The entry fields for the different prompts must be separated by commas. Use the following syntax:

```
LINK (object-list>,<runfile>,<listfile>,<lib-list>[/switch...]
```

where: object-list is a list of object modules, separated by plus signs.

runfile is the name of the file that will receive the executable output.

listfile is the name of the file that will receive the listing.

lib-list is the list of library modules to be searched.

/switch refers to optional switches, which may be placed following any of the response entries (just before any of the commas or after the <lib-list>, as shown).

To select the default for a field, type a second comma with no space between the two.

Example:

```
LINK
FUN+TEXT+TABLE+CARE-P-M,,FUNLIST,COBLIB.LIB
```

This command loads LINK, and then loads object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ. LINK then pauses as a result of the /P switch. When any key is pressed, the object modules are linked, and a global symbol map is produced as a result of the /M switch. LINK defaults to FUN.EXE run file, creates a list file named FUNLIST.MAP, and searches the Library file COBLIB.LIB.

Method 3: Response File

To invoke LINK using Method 3, use the syntax:

```
LINK @<filespec>
```

where: filespec is the name of a response file. A response file contains answers to the LINK prompts (shown in Method 1) and may also contain any of the switches. When naming a response file, the use of filename extensions is optional. Method 3 permits the command that invokes LINK to be entered from the keyboard or from within a batch file without requiring further action.

To use this option, create a response file containing several lines of text, each of which is the response to a LINK prompt. The responses must be in the same order as the LINK prompts discussed in Method 1. Long responses to the "Object Modules:" or "Libraries:" prompt may be typed on several lines by using a plus sign (+) to continue on the next line.

Use switches and command characters in the response file the same way they are used for responses typed on the keyboard.

When the LINK session begins, each prompt will be displayed in the order of the responses from the response file. If the response file does not contain answers for all the prompts, (in the form of filenames, the semicolon command character, or carriage returns), LINK displays the prompt that does not have a response, and then waits for you to type a legal response. When a legal response has been typed, the link session continues.

Example:

```
FUN TEXT TABLE CARE
/PAUSE/MAP
FUNLIST
COBLIB.LIB
```

This response file tells LINK to load the four object modules named FUN, TEXT, TABLE, and CARE. LINK pauses before producing a public symbol map to permit you to change disks (see discussion "LINE SWITCHES" later in this section before using this feature). When any key is pressed, the output files are named FUN.EXE and FUNLIST.MAP. LINK searches the library file COBLIB.LIB, and uses the default settings for the switches.

COMMAND CHARACTERS

LINK provides the following three command "characters":

1. The plus sign (+).

2. The semicolon (;).

3. The abort "character" (CTRL-C).

Plus Sign

Use the plus sign (+) to separate entries and extend the current line in response to the "Object Modules:" and "Libraries:" prompts. (A blank space may be used to separate object modules.) To enter a large number of responses (each may be very long), extend the line by typing a plus-sign/<cursor-return> at the end of the line. If the plus-sign/<cursor-return> is the last entry following these two prompts, LINK prompts you for more module names. When the "Object Modules:" or "Libraries:" prompt appears again, continue to type your responses. When all the modules to be linked and libraries to be searched have been listed, be sure the response line ends with a module name and a <cursor-return> and not a plus-sign/<cursor-return>.

Examples:

```
Object Modules [.OBJ]: FUN TEXT
TABLE CARE+<cursor-return>

Object Modules [.OBJ]:
FOO+FLIPFLOP+JUNQUE+<cursor-return>

Object Modules [.OBJ]:
CORSAIR<cursor-return>
```

Semicolon

To select default responses to the remaining prompts, use a single semicolon (followed immediately by a cursor-return) at any time after the first prompt (Run File:). This feature saves time and overrides the need to repeatedly press the <cursor-return> key.

NOTE

After the semicolon has been typed and entered by pressing the <cursor-return> key, you can no longer respond to any of the prompts for that link session. Therefore, do not use the semicolon to skip prompts. To skip prompts, use the <cursor-return> key.

Example:

```
Object Modules [.OBJ]:
FUN TEXT TABLE CARE<cursor-return>

Run Module [FUN.EXE]: ;<cursor-return>
```

No other prompts appear, and LINK uses the default values (including FUN.MAP for the list file).

Abort Character

Abort is not a single character, but the familiar MS-DOS "break" command, which is implemented by holding the CTRL key down, pressing "C", then releasing both keys. Use the <CTRL-C> key to abort the link session at any time. If an erroneous response is typed, (such as a wrong or misspelled filename) press <CTRL-C> to exit LINK. Then restart LINK. If the error has been typed but you have not pressed the <cursor-return> key, you may delete the erroneous characters with the backspace key (but only for that line).

COMMAND PROMPTS

LINK asks you to respond to four text prompts. When you have typed a response to a prompt and pressed <cursor-return>, the next prompt appears. When the last prompt has been answered, linking automatically begins without further command. When the link session is finished, LINK exits to the operating system. When the operating system prompt appears, LINK has successfully completed its tasks. If the link session is not successful, LINK displays the appropriate error message.

LINK prompts the user for the names of Object, Run, and List files, and for Libraries. The prompts are listed in order of appearance. For prompts that can default to preset responses, the default response is shown in square brackets ([ ]) following the prompt. The "Object Modules:" prompt has no preset filename response and so you must type in a filename.

The command prompts are used as follows:

Object Modules [.OBJ]: Type a list of the object modules to be linked. LINK assumes (by default) that the filename extension is .OBJ. If an object module has any other filename extension, the extension must be supplied. Otherwise, the extension may be omitted.

Modules must be separated by plus signs (+).

Since LINK loads segments into classes in the order encountered, use this information to set the order in which the object modules are to be read by LINK. Refer to the MS-DOS Programmer's Manual for more information on this process.

Run File [First-Object-filename.EXE]:

Typing a filename creates a file for storing the run (executable) file that results from the link session. All run files receive the filename extension .EXE even if you specify an extension other than .EXE.

If no response is typed to the "Run File:" prompt, LINK uses the first filename typed in response to the "Object Modules:" prompt as the RUN filename.

Example:

```
Run File [FUN.EXE]: B:PAYROLL/P
```

This response directs LINK to create the run file PAYROLL.EXE on drive B: and then pauses to allow insertion of a new disk to receive the run file.

List File [Run-Filename.MAP]:

The list file contains an entry for each segment in the input (object) modules. Each entry also shows the addressing in the run file.

The default response is the run filename with the default filename extension .MAP.

Libraries [ ]:

The valid responses are up to eight library filenames or a carriage return. (A carriage return means no library search.) Library files must have been created by a library utility. LINK assumes by default that the filename extension is .LIB for library files.

Library filenames must be separated by spaces or plus signs (+).

LINK searches library files in the order listed to resolve external references. When it finds the module that defines the external symbol, LINK processes that module as another object module.

If LINK cannot find a library file on the disks currently in the disk drives, it will display the message:

```
Cannot find library <library-name>
Type new drive letter:
```

Press the letter for the drive designation (for example, B).

LINK SWITCHES

The six LINK switches control various LINK functions. These switches must be typed at the end of a prompt response, regardless of which method is used to invoke LINK. Switches may be grouped at the end of one response or scattered at the end of several. If more than one switch is typed at the end of a response, each switch must be preceded by the switch delimiter (/).

All LINK switches may be abbreviated. The only restriction is that an abbreviation must be sequential, from the first letter typed to the last. No gaps or transpositions are permitted. The following example shows legal and illegal abbreviations for the /DSALLOCATE switch:

LEGAL	ILLEGAL
/D	/DSL
/DS	/DAL
/DSA	/DLC
/DSALLOCA	/DSALLOCT

/DSALLOCATE

Using the /DSALLOCATE switch tells LINK to load all data at the high end of the Data Segment (DS). Otherwise, LINK loads all data at the low end of the Data Segment. At run time, the DS pointer is set to the lowest possible address to allow the entire DS segment to be used. The /DSALLOCATE switch in combination with a low load (the default value) means that the /HIGH switch is not used and permits the user application to dynamically allocate any available memory below the area specifically allocated within DGroup. Yet that memory remains addressable by the same DS pointer. This dynamic allocation is needed for Pascal and FORTRAN programs.

NOTE

Your application program may dynamically allocate up to 64 K-bytes (or the actual amount of memory available), less the amount allocated within DGroup.

/HIGH

Use of the /HIGH switch causes LINK to place the run file as high as possible in memory. Otherwise, LINK places the run file as low as possible.

CAUTION

Do not use the /HIGH switch with Pascal or FORTRAN programs.

/LINENUMBERS

The /LINENUMBERS switch tells LINK to include in the list file the line numbers and addresses of the source statements in the input modules. Otherwise, line numbers are not included in the list file.

NOTE

Not all compilers produce object modules that contain line number information. If such a compiler is being used, LINK cannot include line numbers.

/MAP

/MAP directs LINK to list alphabetically all public (global) symbols defined in the input modules. For each symbol, LINK lists its value and its segment:offset location in the run file. The symbols are listed at the end of the list file. If /MAP is not supplied, LINK will list only errors (including undefined globals).

PAUSE

The /PAUSE switch causes LINK to suspend the link session when the switch is encountered. Normally, LINK performs the linking session from beginning to end without stopping. This switch allows the user to change disks before LINK outputs the run (.EXE) file.

When LINK encounters the /PAUSE switch, it displays the message:

```
About to generate .EXE file
Change disks <hit any key>
```

LINK resumes processing when the user presses any key.

CAUTION

Do not remove the disk that is to receive the list file or the disk used for the VM.TMP file, if one has been created.

STACK:<number>

number represents any positive numeric value (in hexadecimal radix) up to 65536 bytes. If a value from 1 to 511 is typed, LINK uses 512. If the /STACK switch is not used for a link session, LINK automatically calculates the necessary stack size.

The object modules of a compiler or assembler should provide information that enables the linker to compute the required stack size.

At least one object (input) module must contain a stack allocation statement, or LINK displays the following error message:

```
WARNING: NO STACK STATEMENT
```

SAMPLE LINK SESSION

This sample shows the type of information displayed during a LINK session.

In response to the MS-DOS prompt, type:

```
LINK
```

The system displays the following messages and prompts (your answers are underlined):

```
Microsoft Object Linker V2.01 (Large)
(C) Copyright 1983, by Microsoft Inc.

Object Modules [.OBJ]: DOSBIO SYSINIT
Run File [DOSBIO.EXE]:
List File [NUL.MAP]: DOSBIO /MAP
Libraries [.LIB]: ;
```

NOTES

1. By specifying /MAP, you get both an alphabetic listing and a chronological listing of public symbols.

2. By responding PRN to the "List File:" prompt, you can redirect your output to the printer.

3. By specifying the /LINE switch, you receive a listing of all line numbers for all modules. (Note that the /LINE switch can generate a large volume of output.)

4. By pressing <cursor-return> in response to the "Libraries:" prompt, you initiate an automatic library search.

After LINK locates all libraries, the linker map displays a list of segments in the order of their appearance within the load module. The list might look like this:

Start	Stop	Length	Name
00000H	009ECH	09EDH	CODE
009F0H	01166H	0777H	SYSINITSEG

The information in the Start and Stop columns shows the 20-bit hexadecimal address of each segment relative to location zero (the beginning of the load module).

The addresses displayed are not the absolute addresses at which these segments are loaded. Consult the MS-DOS Programmer's Manual for information on how to determine the location of relative zero and the absolute address of a segment.

Because the /MAP switch has been used, LINK displays the public symbols by name and value. For example:

```
ADDRESS         PUBLICS_BY_NAME
009F:0012       BUFFERS
009F:0005       CURRENT_DOS_LOCATION
009F:0011       DEFAULT_DRIVE
009F:000B       DEVICE_LIST
009F:0013       FILES
009F:0009       FINAL_DOS_LOCATION
009F:000F       MEMORY_SIZE
009F:0000       SYSINIT

ADDRESS         PUBLICS BY VALUE
009F:0000       SYSINIT
009F:0005       CURRENT_DOS_LOCATION
009F:0009       FINAL_DOS_LOCATION
009F:000B       DEVICE_LIST
009F:000F       MEMORY_SIZE
009F:0011       DEFAULT_DRIVE
009F:0012       BUFFERS
009F:0013       FILES
```

LINK ERROR MESSAGES

All LINK errors cause the link session to abort. After the cause has been found and corrected, LINK must be rerun. The following error messages are displayed by LINK:

ATTEMPT TO ACCESS DATA OUTSIDE OF SEGMENT BOUNDS, POSSIBLY BAD OBJECT MODULE
: A bad object file probably exists.

BAD NUMERIC PARAMETER
: Numeric value is not in digits.

CANNOT OPEN TEMPORARY FILE
: LINK is unable to create the VM.TMP file because the disk directory is full. Insert a new disk. Do not remove the disk that will receive the List.MAP file.

ERROR: DUP RECORD TOO COMPLEX
: DUP record in assembly language module is too complex. Simplify assembly language DUP record.

ERROR: FIXUP OFFSET EXCEEDS FIELD WIDTH
: An assembly language instruction refers to an address with a short instruction instead of a long instruction. Edit assembly language source and reassemble.

INPUT FILE READ ERROR
: A bad object file probably exists.

INVALID OBJECT MODULE
: An object module is incorrectly formed or incomplete (as when assembly is prematurely stopped).

SYMBOL DEFINED MORE THAN ONCE
: LINK has found two or more modules that define a single symbol name.

PROGRAM SIZE OR NUMBER OF SEGMENTS EXCEEDS CAPACITY OF LINKER

The total size may not exceed 384 K-bytes, and the number of segments may not exceed 255.

REQUESTED STACK SIZE EXCEEDS 64K

Specify a size greater than or equal to 64 K-bytes with the /STACK switch.

SEGMENT SIZE E)XCEEDS 64K

64 K-bytes is the addressing system limit.

SYMBOL TABLE CAPACITY EXCEEDED

Very many and/or very long names were typed, exceeding the limit of approximately 25 K-bytes.

TOO MANY EXTERNAL SYMBOLS IN ONE MODULE

The limit is 256 external symbols per module.

TOO MANY GROUPS

The limit is 10 groups.

TOO MANY LIBRARIES SPECIFIED

The limit is 8 libraries.

TOO MANY PUBLIC SYMBOLS

The limit is 1024 public symbols.

TOO MANY SEGMENTS OR CLASSES

The limit is 256 (segments and classes taken together).

UNRESOLVED EXTERNALS: <list>
The external symbols listed have no defining module among the modules or library files specified.

VM READ ERROR
This disk error is not generated by LINK.

WARNING: NO STACK SEGMENT
None of the object modules specified contains a statement allocating stack space, but the user has typed the /STACK switch.

WARNING: SEGMENT OF ABSOLUTE OR UNKNOWN TYPE
A bad object module exists, or an attempt has been made to link modules that LINK cannot handle (e.g., an absolute object module).

WRITE ERROR IN TMP FILE
No more disk space remains to expand VM.TMP file.

WRITE ERROR ON RUN FILE
This message usually indicates insufficient disk space for the run file.

Documentation Evaluation Form

Title: ET 2000 Series Utilities Programmer's Reference Manual

Form No: 1171501

Date: June 1984

Burroughs Corporation is interested in receiving your comments and suggestions regarding this manual. Comments will be utilized in ensuing revisions to improve this manual.

Please check type of Comment/Suggestion:

☐ Addition ☐ Deletion ☐ Revision ☐ Error ☐ Other

Comments:

From:

Name _____

Title _____

Company _____

Address _____

_____

Phone Number _____ Date _____

Remove form and mail to:

Burroughs Corporation
Corporate Documentation
Planning, East
209 W. Lancaster Ave.
Paoli, PA 19301, U.S.A.