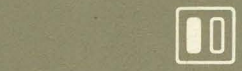

DIGITAL RESEARCH™

CBASIC Compiler™ (CB86™)

Language

Programmer's Guide

for the IBM® Personal Computer
Disk Operating System

CBASIC Compiler™(CB86™) Language Programmer's Guide for the IBM®Personal Computer Disk Operating System

Digital Research
P.O. Box 579
160 Central Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

The CBASIC Compiler (CB86) Language Programmer's Guide for the IBM Personal Computer Disk Operating System was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

****
* First Edition: May 1983 *
****

Foreword

CBASIC Compiler™ is a compiler implementation of the CBASIC® programming language. For the software developer interested in maximizing the execution speed of commercial applications programs, CBASIC Compiler is an excellent choice.

Digital Research designed CBASIC Compiler for use under single-user, multi-user, and concurrent operating systems based on both 8080/8085 and 8086/8088 microprocessor families.

- The 8086/8088 CBASIC Compiler, CB86™ , runs under CP/M-86® , MP/M-86™ , Concurrent CP/M-86™ , CP/NET®, and the IBM® Personal Computer Disk Operating System based on the Intel® 8086, 8088 family of microprocessors.
- The 8080/8085 CBASIC Compiler, CB80™ , runs under the CP/M® (version 2 or later), MP/M™ , and CP/NET® operating systems based on the Intel 8080, 8085, or Zilog® Z80® microprocessor.

The CBASIC Compiler (CB86) Language Programmer's Guide for the IBM Personal Computer Disk Operating System provides a short demonstration program to help you get your CBASIC Compiler system up and running. The manual is divided into five sections.

- Section 1 is an introduction and demonstration program.
- Section 2 describes the compiler, CB86.
- Section 3 describes the linkage editor, LINK86™ .
- Section 4 describes libraries and the librarian program, LIB86™ .
- Section 5 explains the machine-level environment of CBASIC Compiler.

Use this Programmer's Guide in conjunction with the CBASIC Compiler Language Reference Manual. Together, the manuals provide all the information you need to use the CBASIC Compiler to its full potential.

Digital Research is very interested in your comments on programs and documentation. Please use the Software Performance Reports and the Reader Comment Card enclosed in each product package to help us provide you with the best microcomputer software and documentation.

Table of Contents

Table of Contents
(continued)

Appendixes

Tables and Figures

Tables

Figures

Section 1
Getting Started with CBASIC Compiler

A compiler is a computer program that translates high-level programming language instructions into machine-executable code. The compiler takes as input a user-written source program and produces as output a machine-level object program. Some compilers translate a user-written source program into a program that a computer can execute directly. The CBASIC Compiler system, however, uses a link editor and a library in addition to the compiler. Together, the three components translate your CBASIC source-code file into a directly executable program. This approach uses your microcomputer's memory space as efficiently as possible. The system enables you to modularize programs for quick and easy maintenance. The result is a programming system that rivals the performance of systems based on much larger machines.

The primary advantage that compilers provide over other methods of translation is speed. Compiled applications programs execute faster than interpreted programs because the compiler creates a program that the computer can execute directly.

1.1 Components

The three components that make up the CBASIC Compiler system are contained on your CBASIC product disk.

- The compiler, CB86, translates CBASIC source code into relocatable machine code modules. Source programs default to a .BAS filetype unless otherwise specified. CB86 generates .OBJ files. CB86 consists of an executable program and three overlays.
- The link editor, LINK86, combines the relocatable object modules that the compiler creates and routines from the indexed library into a directly executable program with optional overlays. LINK86 generates .EXE files.
- The indexed library, CB86.L86, provides routines that allocate and release memory, determine available memory space, and perform input/output processing. CBASIC Compiler provides a library manager utility program, LIB86.

1.2 A Demonstration

The following demonstration program can help you learn how to compile, link, and run a CBASIC program. The instructions are for CBASIC Compiler on a DOS based system with two floppy-disk drives. You should already be familiar with DOS and a text editor.

Make a back-up copy of your master CBASIC Compiler product disk. Place your operating system disk in drive A and a copy of your CBASIC Compiler disk in drive B.

1) Write the source program.

   Using your text editor, create a file named TEST.BAS on your CBASIC Compiler disk in drive B. Enter the following program into TEST.BAS exactly as it appears below:

```
PRINT
FOR I% = 1 TO 10
    PRINT I%;  "TESTING CBASIC COMPILER!"
NEXT I%
PRINT
PRINT "FINISHED"
END
```

2) Compile the program.

   To start CB86, enter the following command. Be sure drive B is the default drive.

```
B>CB86 TEST
```

   CB86 assumes a filetype of .BAS for the file you specify in the compiler command line unless otherwise specified. A sign-on message, a listing of your source program, and several diagnostic messages display on your terminal.

```
-------------------------------------------------
CB-86 CBASIC Compiler                 Version X.X
Serial No. CB6-0000-654321    All rights reserved
Copyright (c) 1982,1983    Digital Research, Inc.
-------------------------------------------------
end of pass 1
end of pass 2
    1:  0000h PRINT
    2:  000ch FOR I% = 1 TO 10
    3:  0014h     PRINT I%;  "TESTING CBASIC COMPILER!"
    4:  0021h NEXT I%
    5:  002dh PRINT
    6:  0030h PRINT "FINISHED"
    7:  003ah END
end of compilation
no errors detected
code area size:    57         0039h
data area size:    45         002dh
common area size:  0          0000h
symbol table space remaining: 17797
```

The message no errors detected indicates a successful compilation. CB86 creates an object file for the TEST.BAS program. The directory for disk B should have the new file, TEST.OBJ. See Section 2.1 for more information about the various parts of the listing.

3) Link the program.

To start LINK86, enter the following command. Be sure drive B is the default drive.

```
B>LINK86 TEST
```

LINK86 assumes a filetype of .OBJ for the file you specify in the linker command line. A sign-on message and several diagnostic messages display on your terminal.

```
------------------------------------------------
LINK-86 Linkage Editor                 Version X.X
Serial No. XXXX-0000-654321    All Rights Reserved
Copyright (c) 1982,1983     Digital Research, Inc.
------------------------------------------------
CODE    010E0
DATA    002C7

USE FACTOR:  09%
```

If you get no error messages, the program has been linked successfully. LINK86 creates a directly executable program. The directory for disk B should have the new file TEST.EXE.

4) Run the program.

To run the TEST.EXE program, enter the following command. Be sure drive B is the logged-in drive.

```
B>TEST
```

The following output should appear on your terminal:

```
1 TESTING CBASIC COMPILER!
2 TESTING CBASIC COMPILER!
3 TESTING CBASIC COMPILER!
4 TESTING CBASIC COMPILER!
5 TESTING CBASIC COMPILER!
6 TESTING CBASIC COMPILER!
7 TESTING CBASIC COMPILER!
8 TESTING CBASIC COMPILER!
9 TESTING CBASIC COMPILER!
10 TESTING CBASIC COMPILER!

FINISHED
```

End of Section 1

Section 2
The Compiler, CB86

CB86 consists of an executable file and three overlays. Among the files on your CBASIC Compiler product disk are the following four compiler program files:

- CB86.EXE
- CB86.OR1
- CB86.OR2
- CB86.OR3

When compiling a CBASIC program, all four files must be on the same logical disk drive. The drive must be the default drive. The source program file can be on any logical drive.

2.1 Compiling Programs

CB86 takes a CBASIC source program as input and generates a relocatable object file. During compilation, CB86 creates the following temporary work files:

```
<filename>.$PA
<filename>.$QC
<filename>.$DA
```

Unless compilation is unsuccessful, you never see these temporary files listed in a directory. CB86 erases the files automatically when compilation is finished. CB86 also erases the temporary files if they are on disk before you start the compiler.

The size of the temporary files varies with the size of the source program. The amount of temporary space required is approximately equal to the amount of space the source program occupies. If you do not have enough work space on disk for the compiler, you can break up large programs into modules and compile each module separately.

The following example shows a CB86 listing:

```
-------------------------------------------------
CB-86 CBASIC Compiler                    Version X.X
Serial No. CB6-0000-654321     All rights reserved
Copyright (c) 1982,1983     Digital Research, Inc.
-------------------------------------------------
end of pass 1
end of pass 2
    1:  0000h PRINT
    2:  000ch FOR I% = 1 TO 10
    3:  0014h     PRINT I%;  "TESTING THE CBASIC COMPILER!"
    4:  0021h NEXT I%
    5:  002dh PRINT
    6:  0030h PRINT "FINISHED"
    7:  003ah END
end of compilation
no errors detected
code area size:    57        0039h
data area size:    45        002dh
common area size:  0         0000h
symbol table space remaining: 17797
```

Certain phases of the compilation process are combined into a module called a pass. CB86 is a three-pass compiler. Following the sign-on message, CB86 indicates the completion of the first two passes with an end of pass message. The program listing includes the line numbers, relative addresses for the code that each line generates, and the actual source code lines.

In the preceding listing, 1: is an example of a line number. 0000h is a relative address for the relocatable code that the first PRINT statement generates. CB86 prints the total number of compilation errors detected in the program following the message, end of compilation. However, the message, no errors detected, indicates a successful compilation. The next three messages indicate the amount of space CB86 allocates for code and data segments of data. The last message indicates the amount of space remaining in the symbol table. If CB86 detects errors, the relative addresses and the memory allocation messages do not display.

To complete the compilation process, CB86 generates a relocatable object file. The relocatable file has the same filename as the source program and has a .OBJ filetype. The .OBJ file requires approximately the same amount of space as the source program. If the source program contains errors that prevent a successful compilation, CB86 does not generate the .OBJ file.

2.1.1 CB86 Command Lines

The command line starts CB86, specifies the file to compile, and sends special information in the form of command line compiler directives. Refer to Section 2.2.1 for information on command line directives.

The following command line compiles the source program in a file named TEST. CB86 assumes a filetype of .BAS unless otherwise specified.

```
A>CB86 TEST
```

Enter a complete file specification to override the .BAS filetype. The following command line compiles the source program in a file named TEST.PRl. Remember, source files cannot have an .OBJ or .EXE filetype.

```
A>CB86 TEST.PRl
```

Source files can be on any logical disk drive. The following command line compiles the source program TEST.PRl from drive D:

```
A>CB86 D:TEST.PRl
```

If you type an incorrect command line, CB86 responds with the appropriate error message.

2.1.2 Compiler Errors

CB86 reports three different types of compiler errors. The first type, file-system and memory-space errors, includes mistakes such as invalid command lines, read errors, and out-of-memory conditions. CB86 indicates file-system and memory-space errors with literal messages such as disk full and symbol table overflow. Refer to Appendix B, Table B-1, for a complete listing of file-system and memory-space error messages.

The second type, compilation errors, includes misuses of the CBASIC language such as invalid characters, improper data type specifications, and missing delimiters. CB86 prints a caret and a number in the compiler listing of the source program to indicate the occurrence of a compilation error. The number corresponds to an error description listed in Appendix B, Table B-2. The following example shows the compiler listing for a short program that contains one error. The program attempts to assign a string constant to an integer variable. The compiler reports Compilation Error 27.

```
-------------------------------------------------
CB-86 CBASIC Compiler                  Version X.X
Serial No. CB6-0000-654321     All rights reserved
Copyright (c) 1982,1983     Digital Research, Inc.
-------------------------------------------------
end of pass 1
end of pass 2
    1:        I% = "November 11, 1982"
*** error                              ^27
              END
end of compilation
1 errors detected
symbol table space remaining: 14412
```

Notice that the caret points to the string constant. Error message 27 listed in Appendix B reads as follows:

27 Invalid mixed mode. The type of the expression is not permitted.

Depending on what you want the program to do, you can change I% to a string variable or represent the date as an integer. Either change corrects the mistake.

CB86 identifies the errors in a program logically and accurately. However, CB86 cannot determine what your programming intentions are. Therefore, in some cases, CB86 reports error messages that identify mistakes according to what the compiler logically assumes the program is supposed to do, and not necessarily according to what you want the program to do. The following example shows the compiler listing for another program that contains one error. The keyword PRINT in the first line is misspelled. The compiler reports Compilation Errors 30, 51, 33, and 27.

```
-------------------------------------------------
CB-86 CBASIC Compiler                  Version X.X
Serial No. CB6-0000-654321     All rights reserved
Copyright (c) 1982,1983     Digital Research, Inc.
-------------------------------------------------
end of pass 1
end of pass 2
    1:        PRNIT "TESTING CBASIC!"
*** error                          ^30
*** error                          ^51
*** error                          ^33
*** error                          ^27
    2:        PRINT "FINISHED"
    3:        END
end of compilation
4 errors detected
symbol table space remaining: 14416
```

CB86 does not recognize PRNIT as a misspelled keyword, but as a valid real variable name. At this point in the compilation, CB86 assumes that line 1 is an assignment statement. Therefore, CB86 reports Compilation Errors 30, 51, 33, and 27 with the carets all pointing to the end of the line. If you recognize line 1 as an improper assignment statement, as CB86 does, the four error messages are accurate. The four error messages listed in Appendix B read as follows:

30 Invalid symbol follows a variable, constant, or function reference.

51 An equal sign was expected in assignment. An equal sign is inserted.

33 Invalid symbol encountered in an expression. The symbol is ignored.

27 Invalid mixed mode. The type of the expression is not permitted.

A group of error messages all pointing to one line in your program often indicates that CB86 does not recognize what you want the program to do. Studying the line in error can often reveal a simple solution. When you correct the misspelled keyword in the preceding example and recompile the program, CB86 recognizes a valid PRINT statement and reports the message no errors detected.

The third type of compiler error, internal failures, should never occur during your experience with the CBASIC Compiler. CB86 indicates an internal failure with the following message. The XXX stands for a three-digit number. The ZZZZ stands for a compiler-generated line number.

```
FATAL COMPILER ERROR XXX
NEAR SOURCE LINE ZZZZ
```

In some cases, the second line of this message is not displayed. If the above error message occurs during compilation of your CBASIC program, contact the Digital Research Technical Support Center. Please report the three-digit number and the circumstances under which the error occurs.

2.2 Compiler Directives

Compiler directives are special instructions to CB86. There are two different ways to specify compiler directives: source code compiler directives and command line toggles.

2.2.1 Source Code Compiler Directives

Source code compiler directives are special keywords that do not translate into executable code. All source code compiler directives begin with a percent sign. You cannot place blanks between the percent sign and the rest of the keyword. Only blanks and tab characters can precede a directive. Source code compiler directives cannot appear on the same line with CBASIC statements or functions. CB86 ignores all characters on the same line that are not part of the directive. A source code compiler directive cannot span more than one line. You cannot label source code compiler directives.

CBASIC Compiler has the following six source code compiler directives:

%CHAIN A, B, C, D

- %NOLIST
- %LIST
- %EJECT
- %PAGE
- %INCLUDE
- %DEBUG

Normally, CB86 generates a listing of the source program during compilation. The %NOLIST directive tells CB86 not to list anything that follows the %NOLIST in the program. The %LIST directive tells CB86 to resume the listing. You can use %LIST and %NOLIST any number of times in a program. Toggle B, described in Section 2.2.2, suppresses all listings regardless of any directives in the source code.

The %EJECT directive tells CB86 to continue the program listing at the top of the next page of paper. %EJECT works only when you direct the listing to a printer. CB86 ignores %EJECT if the %NOLIST directive is in effect, or if you direct the listing to the console or a disk file.

The %PAGE directive sets the page length for a listing directed to a printer. The page length you specify must be an unsigned integer placed after the %PAGE keyword, as shown in the following example:

```
%PAGE 40
```

The %INCLUDE directive tells CB86 to include the code from a specified source file along with the original compiling program. The included source file is incorporated into the original program immediately following the %INCLUDE. Specify the filename, the filetype, and the drive that holds the file after the %INCLUDE keyword, as shown in the following examples. CB86 assumes the default drive and a .BAS filetype if not specified otherwise.

A = size of area for real Constants
B = " " " " Code
C = " " " " data stmts
D = " " " " storing Var's

```
%INCLUDE CONDEF
%INCLUDE CONDEF.INC
%INCLUDE D:CONDEF.INC
```

You can nest included files six deep. The maximum nesting depth depends on your particular implementation of the CBASIC Compiler. Refer to Appendix A for current implementation-dependent values.

The %DEBUG directive works with three command line toggles: the I, N, and V toggles. You can switch these three toggles on or off from within the program source code. To turn a toggle on, place the toggle letter after the %DEBUG keyword. To turn a toggle off, place the toggle letter preceded by a minus sign after the %DEBUG keyword. The following examples show variations of the %DEBUG directive:

```
%DEBUG I
%DEBUG -I
%DEBUG INV
%DEBUG -I-N-V
```

2.2.2 CB86 Command Line Toggles

Command line toggles are single-letter compiler directives that you specify in the CB86 command line instead of in the source program. Once a toggle is set, it normally remains set through the entire compilation process. The %DEBUG directive can change the I, N, and V toggles during compilation. Place letters within brackets following the file specification in a CB86 command line. Letters can be lower- or upper-case. If you enter conflicting toggles in a command line, the last one read from left to right takes effect. Certain toggles require an additional parameter enclosed in parentheses. The following examples show several ways to specify command line toggles:

```
A>CB86 TEST [B]

A>CB86 TEST.BAS [B, P, S]

A>CB86 FILE.DAT [BPW(72)]

A>CB86 CALCS.PRG [N] [O] [P]

A>CB86 DATA.OVL [ pon ]
```

CB86 has the fifteen command line toggles listed in the following table:

Table 2-1. CB86 Toggles

Toggle	Instruction
B	Suppress listing of the source file.
C	Change the default %INCLUDE file disk.
F	Send the source listing to a disk file on the same drive as the source file.
I	Interlist the generated code with the source file.
L	Set the page length for printed listings.
N	Generate code for line numbers.
O	Suppress the generation of the object .OBJ file.
P	List the source file on the printer.
R	Change the disk that the .OBJ file is written to.
S	Include symbol name information in the .OBJ file.
T	List the symbol table following the source listing.
U	Generate error messages for undeclared variables.
V	Put source code line numbers into the .SYM file.
W	Set the page width for printed listings.
X	Change the disk used for the work files.

The B toggle tells CB86 not to list the source program on the console screen. However, compiler errors and statistical data concerning size of code and data areas display on the screen. The B toggle overrides other directives that control compiler output.

The C toggle specifies the default drive for an include file. Enclose the new drive specification in parentheses following the C. If a drive has been specified in the %INCLUDE directive, the C toggle has no effect. The C toggle allows program development to be independent of your hardware configuration.

The F toggle tells CB86 to send the source listing to a disk file that is on the same drive as the source file. The new file has the same filename as the source file and has a .LST filetype.

The I toggle interlists compiler-generated code with the original source statements. Compiler-generated code uses standard 8086 mnemonics.

The L toggle changes the page length for a listing directed to a printer. Enclose the new length in parentheses following the L. The length must be an unsigned integer, as in the following example:

```
A>CB86 TEST [L(50)]
```

The N toggle generates code that saves the current line number for each physical line in a source program. The code enables the ERRL function to return the line number when an execution error occurs.

The O toggle tells CB86 not to generate the relocatable object file. If a compiler error occurs, CB86 does not generate the .OBJ file.

The P toggle prints the program listing on the printer. CB86 sends a carriage return and line-feed before printing the first page. CB86 prints the page number and the source filename at the top of each page.

The R toggle specifies which drive to place the .OBJ file on.

The S toggle places all information on program variables and line labels into the .OBJ file. The link editor uses the information to generate a .SYM file. You can use the .SYM file with the Digital Research Symbolic Instruction Debugger, SID-86™ .

The T toggle lists the symbol table immediately following the source program listing.

The U toggle generates an error message if a variable name does not appear in an INTEGER, REAL, or STRING declaration. Use the U toggle to locate misspelled identifiers.

The V toggle places the source code line numbers into the .SYM file to make debugging easier.

The W toggle changes the page width for a listing directed to the printer. Initially, the width is set to 80 columns. Enclose the new width in parentheses following the W. The width must be an unsigned integer.

```
A>CB86 TEST [W(70)]
```

The X toggle specifies a drive for the temporary work files. Normally, CB86 places the work files on the same drive as the source file. Enclose the new drive specification in parentheses following the X. The drive is specified by a single lower- or upper-case letter.

```
A>CB86 TEST [X(D)]
```

CB86 evaluates toggles from left to right. This means a subsequent directive can override any earlier one. In the following example, CB86 sends the listing to the printer.

```
A>CB86 TEST [BP]
```

In the following example, CB86 suppresses the listing.

```
A>CB86 TEST [PB]
```

End of Section 2

Section 3
The Link Editor, LINK86

LINK86 is a linkage editor that combines object files that CB86 generates with object modules from the Run-time Subroutine Library, CB86.L86. LINK86 creates an executable program with optional overlays. You can use LINK86 with other Digital Research 16-bit language translators such as RASM-86™ and PL/I-86™, or with any translator that produces object files using a compatible subset of the Intel 8086 object file format. Your CBASIC Compiler product disk should contain the following two files:

- LINK86.EXE
- CB86.L86

3.1 Linking Files

LINK86 accepts two types of object files as input. The first type has a filetype of .OBJ and contains a single object module. CB86 generates .OBJ files. The second type is library files. Library files have a filetype of .L86 and contain an indexed group of object modules. LINK86 can search library files and include any modules that a compiled program requires in the executable program. LIB86 is a librarian program that generates library files. Refer to Section 4.2 for information on LIB86.

Following the sign-on message, LINK86 displays the Use Factor. The Use Factor is a decimal percentage value indicating the amount of memory that LINK86 uses. The following example shows the console display during linking.

```
-------------------------------------------------
LINK-86 Linkage Editor                  Version X.X
Serial No. XXXX-0000-654321    All Rights Reserved
Copyright (c) 1982,1983     Digital Research, Inc.
-------------------------------------------------
CODE    010E0
DATA    002C7

USE FACTOR:  09%
```

To complete the linking process, LINK86 generates three output files:

- the executable program file with filetype .EXE
- a symbol table file with filetype .SYM
- an optional information file with filetype .MAP

All three files have the same filename as the first file listed in the command line. The .EXE file executes directly under DOS. You can use the .SYM file with the Digital Research symbolic debugging program, SID-86. The .MAP file contains information about

segments and groups. LINK86 displays unresolved symbols at the console. Unresolved symbols are symbols referenced in a program but not defined within the files being linked. You must define all unresolved symbols for the program to run properly.

To halt LINK86 during processing, press CTRL-Break. Under DOS Version 1 you can halt LINK86 only during console output. LINK86 immediately returns control to DOS. Under DOS Version 2, you can set break mode and halt LINK86 at any time during processing.

3.1.1 LINK86 Command Lines

The command line starts LINK86 and specifies the files to link. LINK86 automatically searches the Run-time Subroutine Library, CB86.L86, for any required routines. LINK86 command lines use the following general format. Items enclosed in braces are optional.

```
LINK86 {filespec =} filespec1 {,filespec2... ,filespecN}
```

The following example links the file named TEST.OBJ and generates an executable file named TEST.EXE and a symbol table file named TEST.SYM. LINK86 assumes a filetype of .OBJ if not specified otherwise.

```
A>LINK86 TEST
```

You can rename the output files in the LINK86 command line using an equal sign. The following command line links the file named TEST.OBJ but generates output files with the filename TESTPGM.

```
A>LINK86 TESTPGM=TEST
```

You can specify which drive holds the .OBJ file to link, and you can specify a drive for LINK86 to write the output files to. The following command line produces the same output files as the previous example, but LINK86 links the TEST.OBJ file from drive D and writes the TESTPGM output files to drive A.

```
A>LINK86 A:TESTPGM=D:TEST
```

You can specify a filetype other than .OBJ or .L86 for files in a command line if the files are object files. LINK86 aborts the linking process if any of the files are not proper object files. In the following example, LINK86 assumes ONE.A, TWO.B, and THREE.C are object files and links them.

```
A>LINK86 TESTPGM=ONE.A, TWO.B, THREE.C
```

You can link several relocatable files into one executable program. However, when combining several files, only one file can contain executable statements. This file is the main program. All other files must contain only multiple-line functions. In the following command line, TEST.OBJ is the executable main program. ONE.OBJ, TWO.OBJ, and THREE.OBJ contain multiple-line functions.

LINK86 links all four object files into one executable program named TEST.EXE.

```
A>LINK86, TEST, ONE, TWO, THREE
```

LINK86 can link any number of object files until the number of symbols contained in the files exhausts the space reserved for the symbol table. However, the length of a command line entered from the console cannot exceed 128 characters. In cases where a command line exceeds 128 characters, you can place the command line in a disk file and use the INPUT command line option. The command line disk file is also handy if you want to avoid having to type a long and complicated command line over and over.

Create command line disk files with a .INP filetype using any text editor. List each input file with options after the equal sign as you would in an ordinary command line. Do not include the characters LINK86 in the disk file. You can place tab characters, carriage returns, and line-feeds anywhere in a command line file. The following example shows the beginning of a command line file named NEWCOM.INP.

```
PROGNEW= FILE1, FILE2, FILE3, FILE4, FILE5, FILE6,
FILE7, FILE8, FILE9, FILE10, LIB1.L86,
FILE11[MAP], LIB2.L86[SEARCH],
  .
  .
  .
```

To tell LINK86 to read the rest of its command line from the file NEWCOM.INP, use the following command. LINK86 assumes a .INP filetype.

```
A>LINK86 NEWCOM [INPUT]
```

3.1.2 LINK86 Errors

In case of a command line error, LINK86 echoes the command line tail up to the point where the error occurs and follows the error with a question mark. In the following example, LINK86 detects a misplaced semicolon where it expects a comma:

```
A>LINK86 ONE, TWO, THREE; FOUR
-----------------------------------------------
LINK-86 Linkage Editor                Version X.X
Serial No. XXXX-0000-654321    All Rights Reserved
Copyright (c) 1982,1983     Digital Research, Inc.
-----------------------------------------------

SYNTAX ERROR
 ONE, TWO, THREE;?

A>
```

In the next example LINK86 detects a filename that exceeds the eight-character limit.

```
A>LINK86 TESTPROGRAM
-------------------------------------------------
LINK-86 Linkage Editor                  Version X.X
Serial No. XXXX-0000-654321    All Rights Reserved
Copyright (c) 1982,1983     Digital Research, Inc.
-------------------------------------------------

SYNTAX ERROR
 TESTPROGRAM?

A>
```

LINK86 reports errors that occur during linking with a literal message. Refer to Appendix C for a list of LINK86 error messages.

3.2 LINK86 Command Line Options

LINK86 has a number of command line options that you can use with compiled CBASIC programs. Options are keywords that send special instructions to LINK86. You specify options within brackets in a LINK86 command line. The following table lists the options, a brief description of each, and a keyword abbreviation.

Table 3-1. LINK86 Command Line Options

Keyword	Abbr	Meaning
FILL	F	Zero fill and include uninitialized data in .EXE file.
NOFILL	NOF	Do not include uninitialized data in .EXE file.
INPUT	I	Read command line from disk file.
LIBSYMS	LI	Include symbols from library files in .SYM file.
NOLIBSYMS	NOLI	Do not include symbols from library files in .SYM file.
LOCALS	LO	Include local symbols in .SYM file.
NOLOCALS	NOLO	Do not include local symbols in .SYM file.

Table 3-1. (continued)

Keyword	Abbr	Meaning
MAP	M	Generate a .MAP file.
SEARCH	S	Search library and link only referenced modules.
$C		Specify .EXE file destination.
$L		Specify .L86 file location for libraries linked automatically, such as CB86.L86.
$M		Specify .MAP file destination.
$O		Specify .OBJ or .L86 file location for files linked in the command line.
$S		Specify .SYM file destination.

You can specify either the keyword or the abbreviation in a command line. Each option, except for INPUT and SEARCH, affects one of the LINK86 output files. Sections 3.2.1 through 3.2.5 explain the use of the LINK86 command line options in more detail. See Section 3.1.1 for information on INPUT.

3.2.1 Uninitialized Data

Uninitialized data often occurs at the end of a section of a .EXE file. The FILL option directs LINK86 to include uninitialized data in the .EXE file and fill it with zeros. The NOFILL option directs LINK86 to omit the uninitialized data from the .EXE file.

The FILL and NOFILL options do not affect uninitialized data that occur within a section of .EXE file. LINK86 always fills uninitialized data inside a section of a .EXE file with zeros. The following examples show the proper use of the FILL and NOFILL options. FILL is the default option.

```
A>LINK86 TESTPGM [FILL]

A>LINK86 TESTPGM [NOFILL]
```

Using the FILL option often results in a larger .EXE file. However, LINK86 usually processes faster when the FILL option is in effect.

3.2.2 Symbol File Options

The following command options affect the contents of the .SYM file that LINK86 creates:

- LOCALS / NOLOCALS
- LIBSYMS / NOLIBSYMS

You must place the symbol file option keyword after the first file to which it applies. A symbol file option remains in effect until you change it. LINK86 processes the command line from left to right.

The LOCALS option directs LINK86 to place any local symbols from the object files in the .SYM file. The NOLOCALS option directs LINK86 to ignore the local symbols in the object files. LINK86 defaults to the LOCALS option. The following command line creates a .SYM file containing local symbols from TEST2.OBJ and TEST3.OBJ, but not from TEST1.OBJ.

```
A>LINK86 TEST1[NOLOCALS], TEST2[LOCALS], TEST3
```

The LIBSYMS option directs LINK86 to place symbols from library files into the .SYM file. The NOLIBSYMS option directs LINK86 not to include library symbols in the .SYM file. LINK86 defaults to the NOLIBSYMS option. The following command line creates a .SYM file containing symbols from LIB2.L86, but not from LIB1.L86 or LIB3.L86.

```
A>LINK86 LIB1.L86, LIB2.L86[LIBSYMS], LIB3.L86[NOLIBSYMS]
```

3.2.3 MAP File Options

The MAP option directs LINK86 to generate a .MAP file that contains information about the segments in a .EXE file. Five MAP option parameters control the amount of information that LINK86 puts into the .MAP file. Specify MAP option parameters enclosed in brackets after the keyword MAP.

- OBJMAP/NOOBJMAP
- L86MAP/NOL86MAP
- ALL

The OBJMAP parameter directs LINK86 to put .OBJ file segment information into the .MAP file. The NOOBJMAP parameter suppresses .OBJ file segment information. Similarly, the L86MAP parameter directs LINK86 to put library file segment information into the .MAP file. The NOL86MAP parameter suppresses library file segment information. The ALL parameter directs LINK86 to put both .OBJ and library file segment information in the .MAP file.

Once you instruct LINK86 to create a .MAP, you can change the MAP option parameters at different points in the command line, as shown in the following example:

```
A>LINK86 TEST1[MAP[ALL]], TEST2, LIB1.L86 [S,MAP[NOL86MAP]]
```

If you specify the MAP option with no parameters, LINK86 uses OBJMAP and NOL86MAP as defaults.

3.2.4 Library File Options

The SEARCH option directs LINK86 to search a library file and include referenced modules in the .EXE file. Note that LINK86 does not search .L86 files automatically. If you do not specify the SEARCH option after a library file name, LINK86 includes all the modules from the library file in the .EXE file. The following command line combines TEST1.OBJ, TEST2.OBJ, and any modules from LIB1.L86 referenced in the object files:

```
A>LINK86 TEST1, TEST2, LIB1.L86[SEARCH]
```

3.2.5 Redirecting LINK86 File I/O

LINK86 assumes all files specified in a command line are on the default drive, unless you explicitly specify otherwise. Likewise, LINK86 writes all output files to the default drive unless you specify otherwise. The following command line shows how to specify different drives input and output files:

```
A>LINK86 E:NEWTEST = D:TEST1, D:TEST2, B:LIB1.L86
```

In the preceding example, the files TEST1.OBJ and TEST2.OBJ are on drive D and the LIB1.L86 file is on drive B. LINK86 writes the output files to drive E.

Alternatively, you can use the I/O options $C, $L, $M, $O, and $S to override the default drive specifications.

- $C<drivespec> specifies the .EXE file destination.
- $L<drivespec> specifies the .L86 file location for files linked automatically.
- $M<drivespec> specifies the .MAP file destination.
- $O<drivespec> specifies the .OBJ or .L86 file location for files specified in the command line.
- $S<drivespec> specifies the .SYM file destination.

The <drivespec> can be any letter from A to P corresponding to one of sixteen logical drives. You can also direct the .MAP or .SYM output files to the console. Enter an X after the option to direct the file to the console. You can enter a Z after the option to suppress the generation of an output file.

A given I/O option remains in effect, as LINK86 processes a command line from left to right until it encounters a change in that option. The following example shows how to use the I/O options:

```
A>LINK86 TEST1[$CDSZMXOB], TEST2, TEST3[$OA], LIB1.L86[$LF]
```

The preceding example tells LINK86 to read TEST1.OBJ and TEST2.OBJ from drive B, TEST3.OBJ from drive A, and LIB1.L86 from drive F. The command line tells LINK86 to write the .EXE file to drive D, the .MAP file to the console, and to suppress generation of the .SYM file.

You must separate I/O options from other command line options with commas, as shown in this example:

```
A>LINK86 TEST1[NOLOCALS,$SB], TEST2, LIB1.L86[SEARCH,$LC]
```

The preceding command line tells LINK86 to include LIB1.L86 referenced library modules from drive C and write the .SYM file without local symbols to drive B. LINK86 reads the TEST1.OBJ and TEST2.OBJ files from the default drive.

3.3 Producing Overlays

LINK86 can produce overlay files that a CBASIC CHAIN statement can execute. Overlay files have a .OVR filetype. The 8086 version of CBASIC Compiler does not handle chaining and overlays like the 8080 version. In the 8086 version, the root program always resides in memory. Once you chain from the root program to an overlay, you cannot effectively chain back to the root. Chaining back to the root causes the entire program to restart from the beginning. Certain data elements and the stack are reinitialized, and all COMMON data from the first execution is lost. Overlays can only chain effectively to other overlays.

To generate an overlay, enclose the filename of the object file in parentheses within the LINK86 command line. The following command line creates an executable file named TEST.EXE and one overlay named ONE.OVR:

```
A>LINK86 TEST(ONE)
```

The TEST.EXE file that the preceding example generates is the root program. A CHAIN statement in the TEST.EXE program transfers control to the overlay ONE.OVR. The root program contains all library routines and COMMON data for the entire program.

The following command line generates an executable program named TESTPGM.EXE and two overlays named ONE.OVR and TWO.OVR:

```
A>LINK86 TESTPGM=TEST(ONE)(TWO)
```

You can combine several object files into one overlay. The following command line generates an executable program named TEST.EXE and three overlays named A.OVR, C.OVR, and F.OVR:

```
A>LINK86 TEST(A,B) (C,D,E) (F)
```

You can specify names for the overlay files in the command line. The following command line generates the TESTPGM.EXE program and two overlays named FIRST.OVR and SECOND.OVR:

```
A>LINK86 TESTPGM=TEST (FIRST=A) (SECOND=B,C)
```

3.4 The Linking Operation

This section provides a brief introduction to how LINK86 works. For a more detailed description you can refer to the Digital Research Programmer's Utilities Guide for 8086 machines. You should have a working knowledge of microprocessors and operating systems to understand the LINK86 system.

Object files contain a number of segments that have four attributes: segment name, class name, align type, and combine type. The CBASIC Compiler automatically assigns the four attributes to various segments of a CBASIC program at compile time. You cannot manipulate CBASIC programs at the attribute level. You can use RASM-86 to write assembly language programs that maintain direct control over segmentation and grouping. RASM-86 is an assembler that uses a subset of Intel ASM-86 assembly language. RASM-86 generates object files compatible for linking with LINK86. RASM-86 is available from Digital Research.

- LINK86 uses the segment name to identify and combine all program parts that belong together in one segment.
- LINK86 uses the class name to position combined segments in the .EXE file.
- The align type indicates the type of boundary on which a segment begins. Align types are byte, word, and paragraph. LINK86 uses the align type with the segment name and class name to combine all program parts that belong together in one segment and to position segments in the .EXE file.
- The combine type determines how LINK86 combines program parts from different files that have the same segment name. Combine types are PUBLIC, COMMON, and STACK.

The LINK86 process involves two distinct phases. In Phase 1, LINK86 collects all the program parts with the same segment name and class name. LINK86 then combines the segments according to the align and combine attributes. In Phase 2, LINK86 organizes the segments into the proper groups such as CGROUP (code) and DGROUP (data) and assigns the grouped segments to a section of the .EXE file.

- Segments in the CGROUP are placed in the CODE section of the .EXE file.
- Segments in the DGROUP are placed in the DATA section of the .EXE file.
- Segments not part of the CGROUP or DGROUP are placed in the .EXE file according to class name.

LINK86 omits any segment that does not have sufficient positioning information.

End of Section 3

Section 4
The Library

A library file consists of one or more object modules. To speed up the linking process, a library file contains an index. The index contains all the public symbols that are in each module, enabling LINK86 to determine which routines in CB86.L86 are required to create the executable program. CBASIC Compiler provides a library file to use with LINK86 and a library manager utility program to create your own library files. Your CBASIC Compiler product disk should contain the following two files:

- CB86.L86
- LIB86.EXE

4.1 CB86.L86

The file CB86.L86 is a library file that contains modules to allocate and release memory, determine available memory space, and perform arithmetic operations and input/output processing. All library files have a .L86 filetype.

LINK86 first reads the object files you specify in the command line. LINK86 then searches the index of CB86.L86 for any symbols that remain unresolved. LINK86 links only those modules from CB86.L86 that contain definitions of the unresolved symbols.

For example, if a module in one of your programs requires the square root subroutine, LINK86 searches the index of the CB86.L86 file for the symbol ?RSQR. Assuming that this symbol is not defined anywhere in your program, LINK86 links the module from CB86.L86 that contains the definition of ?RSQR. LINK86 links any module from the library that contains a required symbol definition.

The CBASIC Compiler indexed library file provides four routines for use in assembly modules that enable you to allocate and release memory, and to determine the amount of space that is available for allocation.

- The ?GETS routine allocates space. The routine requires that the number of bytes of memory to allocate pass in register BX. The maximum number of bytes the routine can allocate is 32,762. ?GETS returns a pointer to a contiguous block of memory in register AX. There is no restriction on what the allocated memory space can contain if the adjacent space at either end of the allocated area is not modified.

- The ?RELS routine releases previously allocated memory. The routine requires that the address of the space to release passes in register BX. ?RELS does not return a value.

- The ?MFRE routine returns the size of the largest contiguous area available for allocation using the ?GETS routine. The value returned is an integer placed in in register AX.

- The ?IFRE routine returns the total amount of unallocated dynamic memory. The returned value is an integer placed in register AX. A negative value indicates a number larger than 32,767.

4.2 The Library Manager Utility, LIB86

LIB86 is a versatile library manager for developing library files to use with LINK86. LIB86 can perform the following six tasks:

- create a library file from a group of object files
- append modules to an existing library file
- replace modules in an existing library file
- delete modules from an existing library file
- select specific modules from a library file
- display library information

LIB86 processes each input file and generates output files according to instructions in the command line. Input files can have filetype .OBJ or .L86. .OBJ files contain only one module. .L86 files contain one or more modules. LIB86 assumes a .OBJ filetype if not specified otherwise. To conclude processing, LIB86 displays the total number of modules that it processes and the Use Factor. The Use Factor is a decimal percentage value indicating the amount of memory that LIB86 uses.

To halt LIB86 during processing, press CTRL-Break. Under DOS Version 1, you can halt LIB86 only during console output. LIB86 immediately returns control to DOS. Under DOS Version 2, you can set break mode and halt LIB86 at any time during processing.

4.2.1 LIB86 Command Lines

The command line starts LIB86 and specifies the input files to process. A LIB86 command line uses the following general format:

A>LIB86 <new filespec>=<filespec> {[options]} {,<filespec>}

LIB86 checks for errors and displays a literal message, as described in Appendix E.

LIB86 has fourteen command line options. Options are keywords that send special instructions to LIB86. You specify options within brackets in a LIB86 command line. The following table lists the LIB86 command line options, a brief description of each, and a keyword abbreviation.

Table 4-1. LIB86 Command Line Options

Option	Abbr	Purpose
DELETE	D	Delete a module from a library file.
EXTERNALS	E	Show EXTERNALS in a library file.
INPUT	I	Read commands from input file.
MAP	MA	Create a module map.
MODULES	MO	Show modules in a library file.
NOALPHA	N	Show modules in order of occurrence.
PUBLICS	P	Show PUBLICS in a library file.
REPLACE	R	Replace a module in a library file.
SEGMENTS	SEG	Show segments in a module.
SELECT	SEL	Select a module from a library file.
XREF	X	Create a cross-reference file.
$O		Specify input file location.
$X		Specify .XRF file destination.
$M		Specify .MAP file destination.

You can specify either the keyword or the abbreviation in a command line. Sections 4.2.2 through 4.2.8 explain the use of command line options in more detail.

LIB86 can process any number of files. However, the length of a command line cannot exceed 128 characters. In cases where a command line exceeds 128 characters, you can either shorten filenames, or you can place the command line in a disk file and use the INPUT option.

Create command line disk files with a .INP filetype using any text editor. Enter the new library name before an equal sign and list each input file with options after the equal sign as you would in an ordinary command line. Do not include the characters LIB86 in the disk file. You can place tab characters, carriage returns, and line-feeds anywhere in a command line file. The following example shows the beginning of a command line file named LIBRARY1.INP:

```
LIBRARY1 = SUBTOT [XREF], ADD2, SUB45, MULT, DIV2,
           NET1, NET2, NET3,
           TOTAL, GROSS1, GROSS2, GROSS3,
           CHART1, CHART2, CHART3,
           .
           .
           .
```

To use the command line disk file and start LIB86, specify the .INP file as shown below. LIB86 assumes a .INP filetype.

```
A>LIB86 LIBRARY1 [INPUT]
```

The preceding example specifies the INPUT option, telling LIB86 to read the rest of the command line from the LIBRARY1.INP disk file.

4.2.2 Creating a Library File

The following example creates a library named TEST.L86 from the input files ONE.OBJ, TWO.OBJ, and THREE.OBJ. Notice that you do not have to specify the .L86 filetype for the library name. LIB86 assumes a filetype of .OBJ for input files unless specified otherwise. Remember, .OBJ files contain only one module.

```
A>LIB86 TEST=ONE,TWO,THREE
```

You can create one large library file from several smaller library files. The following example creates a new large library file named TESTLIB.L86 from the the input files LIB1.L86 and LIB2.L86. Remember, .L86 files contain more than one module.

```
A>LIB86 TESTLIB=LIB1.L86,LIB2.L86
```

You can combine .OBJ and .L86 files in one command line to create a library, as in the following example:

```
A>LIB86 MATHLIB = SQRT, TRIGLIB.L86
```

The preceding example creates a library file named MATHLIB.L86 from the input files SQRT.OBJ, and TRIGLIB.L86.

4.2.3 Appending an Existing Library

To add a module to an existing library, specify the existing library filename on both sides of the equal sign. Then, list the input files that you want to append. You must include the .L86 filetype for the library filename on the right side of the equal sign. The following example appends the files ONE.OBJ and LIB1.L86 to the existing library file TESTLIB.L86:

```
A>LIB86 TESTLIB=TESTLIB.L86,ONE,LIB1.L86
```

You can rename an appended library file, as shown in the following example:

```
A>LIB86 NEWTEST=TESTLIB.L86,ONE,LIB1.L86
```

The preceding example appends the files ONE.OBJ and LIB1.L86 to the existing library TESTLIB.L86, creating a new library file named NEWTEST.L86.

4.2.4 Replacing Library Modules

Use the REPLACE option to replace a module in an existing library file. The following command line replaces the module ONE with the file NEWONE.OBJ in the library file TESTLIB.L86. Notice the proper use of brackets.

```
A>LIB86 TESTLIB=TESTLIB.L86 [REPLACE [ONE=NEWONE]]
```

If you want to replace a module but maintain the same module name, specify the name only once after the REPLACE keyword. The following example replaces the module ONE with a new ONE.OBJ file in the library TESTLIB.L86 and renames the the library NEWLIB.L86:

```
A>LIB86 NEWLIB=TESTLIB.L86 [REPLACE [ONE]]
```

You can replace several modules with one command line. Separate each REPLACE specification after the keyword REPLACE with commas, as shown in the following example:

```
A>LIB86 NEWLIB=TESTLIB.L86 [REPLACE [ONE=NEW1, TWO=NEW2]]
```

You cannot use the command options DELETE and SELECT with REPLACE in the same command line.

LIB86 displays an error message if it cannot find a specified module in the library file.

4.2.5 Deleting Library Modules

Use the DELETE option to delete modules from an existing library file as shown below. The following example deletes the module TWO from the library file TESTLIB.L86. Notice the proper use of brackets.

```
A>LIB86 TESTLIB=TESTLIB.L86 [DELETE [TWO]]
```

You can delete several modules with one command line. Separate modules after the keyword DELETE with commas. The following example deletes three modules to create a new library named NEWLIB.L86:

```
A>LIB86 NEWLIB=TESTLIB.L86 [DELETE [ONE, TWO, FIVE]]
```

You can delete a group of contiguous library modules using a hyphen, as shown below:

```
A>LIB86 NEWLIB=TESTLIB.L86 [DELETE [ONE - FIVE]]
```

The preceding command line deletes all modules from module ONE through module FIVE.

You cannot use the command options REPLACE and SELECT with DELETE in one command line.

LIB86 displays an error message if it cannot find a specified module in a library file.

4.2.6 Selecting Modules

Use the SELECT option to select specific modules from an existing library to create a new library. The following example creates a new library named NEWLIB.L86 that consists of three modules selected from OLDLIB.L86. Notice the proper use of brackets.

```
A>LIB86 NEWLIB=OLDLIB.L86 [SELECT [TWO, FOUR, FIVE]]
```

You can select a group of contiguous library modules using a hyphen, as shown below. The following example creates a new library that consists of five modules selected from an existing library, assuming the modules ONE, TWO, THREE, FOUR, and FIVE are contiguous in the library file:

```
A>LIB86 NEWLIB=OLDLIB [SELECT [ONE - FIVE]]
```

You cannot use the command options DELETE and REPLACE with SELECT in one command line.

LIB86 displays an error message if it cannot find a specified module in a library file.

4.2.7 Displaying Library Information

LIB86 can produce two types of listing files: a cross-reference file and a library module map. A cross-reference file contains an alphabetized list of all public, external, and segment name symbols in a library file. Following each symbol is a list of all modules that contain the symbol. LIB86 marks the module or modules that define the symbol with a pound sign, #. LIB86 encloses segment names in slashes, //. For example, the segment CODE would appear as /CODE/.

You can use the XREF option to create a cross-reference listing for a specified library file. The following example creates a cross-reference file named TESTLIB.XRF for the TESTLIB.L86 library file:

```
A>LIB86 TESTLIB.L86 [XREF]
```

A module map contains an alphabetized list of all modules in a library file. Following each module name is a list of all segments in the module and the length of each segment. A module map also includes a list of all public and external symbols specified in the module.

Use the MAP option to create a module map for a specified library file. The following example creates a module map named TESTLIB.MAP for the TESTLIB.L86 library file:

```
A>LIB86 TESTLIB.L86 [MAP]
```

Normally, LIB86 alphabetizes the names of modules in a module map listing. Use the NOALPHA option to produce a module map that lists module names in order of occurrence as shown below:

```
A>LIB86 TESTLIB.L86 [MAP, NOALPHA]
```

You can use LIB86 to create partial library information maps using the MODULES, SEGMENTS, PUBLICS, and EXTERNALS options. You can use the four options in any combination. The following example creates a module map that contains only public and external symbols:

```
A>LIB86 TESTLIB [PUBLICS, EXTERNALS]
```

You can combine the SELECT command with any of the MAP options described above to generate partial library information maps as shown in the following examples:

```
A>LIB86 TESTLIB.L86 [XREF, [SELECT [ONE, TWO, THREE]]

A>LIB86 MATHLIB.L86 [MAP, NOALPHA, SELECT [SQRT, LOG, TAN]]

A>LIB86 LIBRARY1.L86 [MODULES, SEGMENTS, SELECT [ONE - FIVE]
```

4.2.8 Redirecting Library File I/O

LIB86 assumes all files specified in a command line are on the default drive. Therefore, you must specify the drive for any input file that is not on the default drive. Likewise, LIB86 writes all output files to the default drive unless you specify otherwise. The following command line shows how to specify different drives for input and output library files.

```
A>LIB86 E:NEWLIB1 = LIBRARY1.L86, D:ONE, D:TWO, B:THREE
```

In the preceding example, the existing library file is on the default drive. LIB86 appends the files ONE.OBJ and TWO.OBJ from drive D, and the file THREE.OBJ from drive B to LIBRARY1.L86. LIB86 writes the new library file, NEWLIB1.L86, to drive E.

Alternatively, you can use the I/O options $O, $X, and $M to override the default drive specifications.

- $O<drivespec> specifies input .OBJ or .L86 file location.
- $X<drivespec> specifies output .XRF file destination.
- $M<drivespec> specifies output .MAP file destination.

The $O option remains in effect as LIB86 processes a command line from left to right until it encounters another $O. This is a useful feature if you want to create a library from several groups of files on different drives. The following example tells LIB86 to read the input files A1.OBJ, A2.OBJ, and A3.OBJ from drive C, and A4.OBJ, A5.OBJ, A6.OBJ, and A7.OBJ from drive D:

```
A>LIB86 NEWLIB1 = A1[$OC], A2, A3, A4[$OD], A5, A6, A7
```

You can direct .XRF and .MAP files to the console or printer in addition to any logical drive. Enter an X after the option to direct the file to the console. Enter a Y after the option to direct the file to the printer.

You can specify multiple I/O options in one command line as shown below:

```
A>LIB86 LIBRARY1.L86 [MAP, XREF, $OCXEMX]
```

The preceding example tells LIB86 to read the existing library file from drive C, write the .XRF file to drive E, and send the .MAP file to the console screen.

End of Section 4

Section 5
Machine-level Environment

To understand the machine-level environment of CBASIC Compiler, you should have a working knowledge of DOS and a familiarity with 8086 microprocessor architecture. As you read this section, have your operating system manuals ready for quick reference.

5.1 Memory Allocation

Figure 5-1 shows memory allocation for a CBASIC program loaded according to Intel's Small Memory Model. Notice that the CBASIC Compiler system supports program groups to the 64K addressable segment range. The limitation provides a great deal of program flexibility without having to reinitialize segment registers. LINK86 displays an error message when a program group for the resulting .EXE file exceeds the 64K limit.

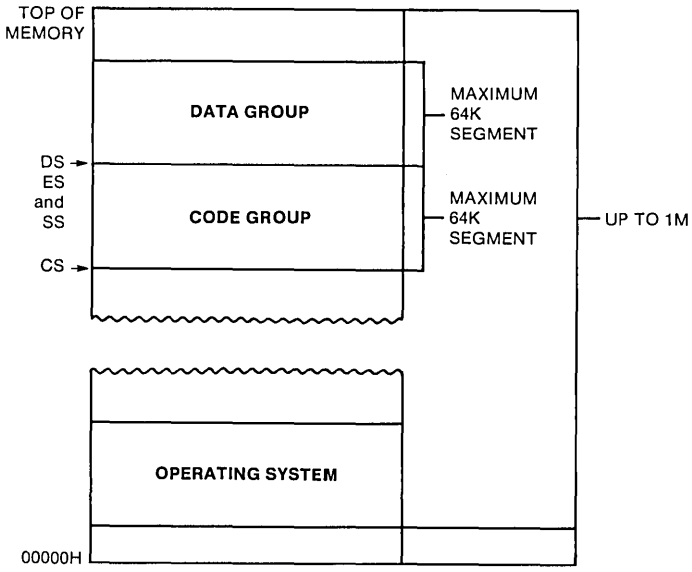

Figure 5-1. Memory Allocation

5.2 Run-time Organization

The 16-bit CBASIC Compiler system generates directly executable command files with a filetype of .EXE. A CBASIC command file contains both a header record and the program memory image. The CBASIC Compiler system defines a separate DGROUP (data group) and CGROUP (code group) in the memory image portion of each command file. The operating system maintains a base page in the data area of the program.

Figure 5-2 roughly depicts memory allocation within the DGROUP and CGROUP for an executable root program and one overlay. Notice that CBASIC Compiler stores all COMMON data in the root and maintains its own stack just below the Dynamic Storage Area.

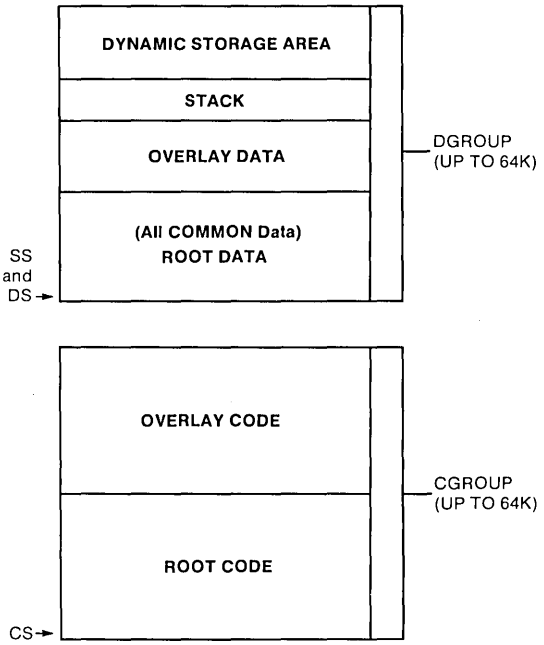

Figure 5-2. Run-time Organization

The CB86 version of 16-bit CBASIC Compiler does not handle chaining and overlays like the CB80 version. In the CB86 version, the root program always resides in memory. Once you chain from the root program to an overlay, you cannot effectively chain back to the root. Chaining back to the root causes the entire program to restart from the beginning. Certain data elements, including the stack, are reinitialized and all COMMON data from the first execution is lost. Overlays can only chain effectively to other overlays. Your root program must contain the COMMON declarations for all overlays.

To conserve disk space and minimize overlay load time, you can link all library code in the root program explicitly. Specify the CB86.L86 file in the LINK86 command line. Any library routines that an overlay needs are included as part of the overlay, unless those routines are already part of the root.

5.3 Internal Data Representation

CBASIC machine-level representation varies somewhat for real numbers, integers, strings, and arrays.

- REAL NUMBERS are stored in binary coded decimal (BCD) floating-point form. Each real number occupies eight bytes of memory storage space, as shown in Figure 5-3. The high-order bit in the first byte (byte 0) contains the sign of the number. The remaining seven bits in byte 0 contain a decimal exponent. The exponent is a binary number representing a power of ten. The number is biased by 40H. Therefore, an exponent value of 42H represents an actual exponent of 2. Bytes 1 through 7 contain the mantissa. Two BCD digits occupy each of the seven bytes in the mantissa. The most significant digit of the number is stored in byte 7, furthest from the exponent. The floating decimal point is always situated to the left of the most significant digit.

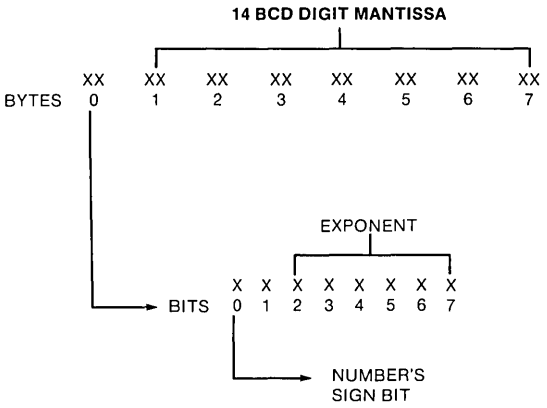

Figure 5-3. Real Number Storage

- INTEGERS are stored in two bytes of memory space with the low-order byte first, as shown in Figure 5-4. Integers are represented as 16-bit, two's complement binary numbers. Integer values range from -32768 to +32767, inclusive.

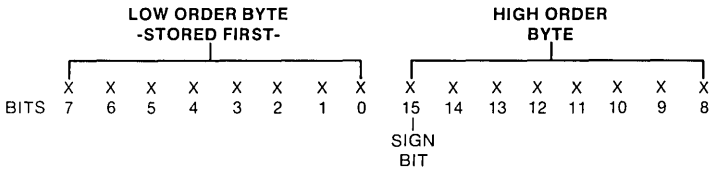

Figure 5-4. Integer Storage

- STRINGS are stored as a sequence of ASCII characters. The length of a string is stored in the first two bytes followed by the actual ASCII values, as shown in the following figure. The high-order length byte is stored first. The maximum number of characters in a string is 32,762. CBASIC Compiler allocates space in the Dynamic Storage Area for strings. A pointer in the Data Area is an address in the DSA for the actual string.

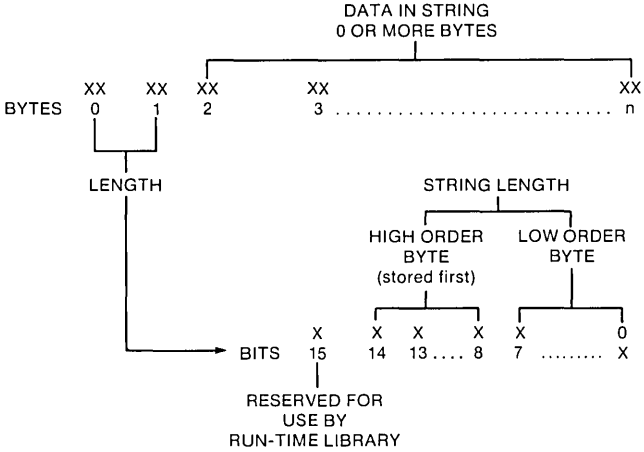

Figure 5-5. String Storage

- ARRAYS, both numeric and string, are allocated space in the Dynamic Storage Area as required. Eight bytes are reserved for each element of an array containing real numbers and two bytes for each element of an integer array. String arrays are allocated two bytes for each entry plus the sum of all the string elements.

At some point in a program it might be necessary to free memory space allocated to arrays that are no longer needed in the program. Freeing numeric array space requires that you simply redimension the array to zero. However, freeing string array space is a two step process. First, you must set all string array elements to null. Set all string array elements equal to a string variable that has never been assigned a value. Use a variable such as NULL$. Be sure that NULL$ has never been assigned a value. Do not set NULL$ equal to "". Second, you must redimension the string array to zero after assigning each element in the array to NULL$.

5.4 Parameter Passing and Returning Values

CBASIC Compiler passes all parameters on the hardware stack. When a program calls a routine, CBASIC places each parameter on the stack reading from left to right. The last entry on the stack is the return address. All values must conform to the format described in Section 5.5.

An assembly language routine can return integer, real, or string values to a CBASIC program. Before returning to the CBASIC program, all parameters passed on the stack must be removed and the stack pointer adjusted accordingly.

Integers return in register AX. Real numbers return using a pointer in register AX that points to an eight byte area containing the real value to return. The AX register contains the address of the first exponent byte of the number being returned.

Strings return using a pointer in register AX. Strings must have been allocated using CBASIC Compiler dynamic storage management routines. The allocation bit of a returning string should be set to 1. This ensures that the space can be reclaimed when no longer needed.

If you use STD instruction in your assembly language routine, you must clear the direction flag upon exit from the routine. The direction flag is initially clear upon entry to a routine.

End of Section 5

Appendix A
Implementation-dependent Values

The following implementation-dependent values apply to CB86.

Table A-1. Implementation-dependent Values

Parameter	Value	Minimum
Initial page width for compiler output	80	-
Initial page length for compiler output	66	-
Maximum number of errors maintained	95	-
Maximum nesting of INCLUDE	6	4
Maximum number of formal parameters	15	15
Maximum number of subscripts in an array	15	15
Maximum unique identifier length	50	31
Maximum number of characters in string constant	255	255
Maximum length of global and external names	6	6
Maximum nesting of FOR loops	13	-
Maximum nesting of WHILE loops	39	-
Number of files that can be open at one time	20	12
File buffer size in bytes	128	-

The minimum values are the minimum that are used in any CB86 implementation.

The following extensions exist in CB86 to provide compatibility with CBASIC-86. Note that future versions of CB86 might not support these extensions.

- The LPRINTER statement accepts a WIDTH option to be consistent with CBASIC. The width is ignored.
- Integer and real data is initialized to 0; strings are initialized to null strings. See Section 5.5.
- The INPUT prompt string can be any expression; the first operand must be a string constant.
- An OPEN or CREATE statement accepts a RECS field for compatibility with CBASIC. The expression is ignored.
- You can use the reserved words LT, GT, GE, LE, EQ, and NE in place of the relational operators <, >, <=, >=, =, and <>.
- CB86 supports the following form of an IF statement:

  IF <expression> THEN <label>

  but the <label> must be a numeric label.

End of Appendix A

Appendix B
Compiler Error Messages

The compiler prints the following messages when a file system error or memory space error occurs. In each case, control returns to the operating system.

Table B-1. File System and Memory Space Errors

Error	Meaning
COULD NOT OPEN FILE: <filename>	The filename cannot be found in the file system directory.
%INCLUDES NESTED TOO DEEP: <filename>	The filename occurs in an %INCLUDE directive that exceeds the allowed nesting of %INCLUDE directives.
SYMBOL TABLE OVERFLOW	The available memory for symbol table space has been exceeded. Break the program into modules or use shorter symbol names.
INVALID FILE NAME: <filename>	The filename is not valid for your operating system.
DISK READ ERROR	The operating system reports a disk read error.
CREATE ERROR: <filename>	The file cannot be created. Normally this means there is no directory space on the disk.

Table B-1. (continued)

Error	Meaning
DISK FULL	The operating system reports that no additional space is available to write temporary or output files. The directory is full or the disk is out of space.
INVALID COMMAND LINE	The command line is incorrect. The compiler prints a greater-than sign, >, one blank space, and all command line characters beginning with the first character in error. If no characters remain in the command line when an error occurs, the compiler does not print the > or the space.
MISSING SOURCE FILE NAME	The command line processor reports that you did not specify a source file.
CLOSE OR DELETE ERROR	The operating system reports that it cannot close a file. This occurs if disks are switched during compilation.

If the compiler detects an internal failure, the following error message appears:

```
FATAL COMPILER ERROR XXX
NEAR SOURCE LINE ZZZZ
```

where XXX is a three-digit number. If the preceding error message occurs during compilation of your CBASIC program, contact the Digital Research Technical Support Center. Please report the three-digit number and the circumstances under which it occurs.

The following error messages indicate the occurrence of compilation errors. Compilation error messages display within the source code listing. CB86 does not create the object file if a compilation error occurs.

Table B-2. Compilation Error Messages

Error	Meaning
1	Invalid character in the source program. The character is ignored.
2	Invalid string constant. The string is too long or contains a carriage return.
3	Invalid numeric constant. An integer constant of zero is assumed.
4	Undefined compiler directive. This source line is ignored.
5	The %INCLUDE directive is missing a filename. This source line is ignored.
6	Statements found after an END.
7	The program attempts to divide-by-zero in the evaluation of an integer constant expression, such as I% = 7/0.
8	Variable used without being defined, and the U toggle used during compilation.
9	The DEF statement is not terminated by a carriage return. A carriage return is inserted.
10	A right parenthesis is missing from the parameter list. A right parenthesis is inserted.
11	A comma is missing in the parameter list. A comma is inserted.
12	An identifier is missing in the parameter list.
13	The same name is used twice in a parameter list.
14	A DEF statement occurs within a multiple-line function. Multiple-line functions cannot be nested. The statement is ignored.

Table B-2. (continued)

Error	Meaning
15	A variable is missing.
16	The function name is missing following the keyword DEF. The DEF statement is ignored.
17	A function name is used previously. The DEF statement is ignored.
18	A FEND statement is missing. A FEND is inserted.
19	There are too many parameters in a multiple-line function.
20	Inconsistent identifier usage. An identifier cannot be used as both a label and a variable.
21	Additional data exists in the source file following an END statement. This is the logical end of the program.
22	Data statements must begin on a new line. The remainder of this statement is treated as a remark.
23	There are no variables or function names in a declaration statement, or a reserved word appears in the list of identifiers.
24	A function name appears in a declaration within a multiple-line function other than the multiple-line function that defines this function name.
25	A function call has incorrect number of parameters.
26	A left parenthesis is missing. A left parenthesis is inserted.
27	Invalid mixed mode. The type of the expression is not permitted.
28	Unary operator cannot be used with this operand.
29	Function call has improper type of parameter.

Table B-2. (continued)

Error	Meaning
30	Invalid symbol follows a variable, constant, or function reference.
31	This symbol cannot occur at this location in an expression. The symbol is ignored.
32	Operator is missing. Multiplication operator inserted.
33	Invalid symbol encountered in an expression. The symbol is ignored.
34	A right parenthesis is missing. A right parenthesis is inserted.
35	A subscripted variable is used with the incorrect number of subscripts.
36	An identifier is used as a simple variable with previous usage as a subscripted variable.
37	An identifier is used as a subscripted variable with previous usage as an unsubscripted variable.
38	A string expression is used as a subscript in an array reference.
39	A constant is missing.
40	Invalid symbol found in declaration list. The symbol is skipped.
41	A carriage return is missing in a declaration statement. A carriage return is inserted.
42	A comma is missing in declaration list. A comma is inserted.
43	A common declaration cannot occur in a multiple-line function. The statement is ignored.
44	An identifier appears in a declaration twice in the main program or within the same multiple-line function.

Table B-2. (continued)

Error	Meaning
45	The number of dimensions specified for an array exceeds the maximum number allowed. A value of one is used. This might generate additional errors in the program.
46	Right parenthesis is missing in the dimension specification within a declaration. A right parenthesis is inserted.
47	The same identifier is placed in COMMON twice.
48	An invalid subscripted variable reference encountered in a declaration statement. An integer constant is required. A value of 1 is used.
49	An invalid symbol found following a declaration, or the symbol in the first statement in the program is invalid. The symbol is ignored.
50	An invalid symbol encountered at the beginning of a statement or following a label.
51	An equal sign is missing in an assignment. An equal sign is inserted.
52	A name used as a label previously used at this level as either a label or variable.
53	Unexpected symbol follows a simple statement. The symbol is ignored.
54	A statement is not terminated with a carriage return. Text is ignored until the next carriage return.
55	A function name is used in the left part of an assignment statement outside of a multiple-line function. Only when the function is being compiled can its name appear on the left of an assignment statement.
56	A predefined function name is used as the left part of an assignment statement.
57	In an IF statement, a THEN is missing. A THEN is inserted.

Table B-2. (continued)

Error	Meaning
58	A WEND statement is missing. A WEND is inserted.
59	A carriage return or colon is missing at the end of a WHILE loop header.
60	In a FOR loop header, the index is missing. The compiler skips to end of this statement.
61	In a FOR loop header, a TO is missing. A TO is inserted.
62	An equal sign is missing in a FOR loop header assignment. An equal sign is inserted.
63	Carriage return or colon is missing at end of FOR loop header.
64	A NEXT statement is missing. A NEXT is inserted.
65	Not used.
66	The variable that follows NEXT does not match the FOR loop index.
67	NEXT statement encountered without a corresponding FOR loop header.
68	WEND statement encountered without a corresponding WHILE loop header.
69	FEND statement encountered without a corresponding DEF statement. This error indicates that the end of the source program was detected while within a multiple-line function.
70	The PRINT USING string is not of type string.
71	A delimiter is missing in a PRINT statement. A comma is inserted.
72	A semicolon is missing in an INPUT prompt. A semicolon is inserted.
73	A delimiter is missing in an INPUT statement. A comma is inserted.

Table B-2. (continued)

Error	Meaning
74	A semicolon is missing following a file reference. A semicolon is inserted.
75	The prompt in an INPUT statement is not of type string.
76	In an INPUT LINE statement, the variable following the keyword LINE is not a string variable.
77	In an INPUT statement, a comma is missing between variables. A comma is inserted.
78	The keyword AS is missing in an OPEN or CREATE statement. AS is inserted.
79	The filename in an OPEN or CREATE statement is not a string expression.
80	A delimiter is missing in a READ statement. A comma is inserted.
81	In a GOTO, GOSUB, or ON statement, a label is missing. This token can be an identifier previously used as a variable.
82	The label in a GOTO statement is not defined. If the label is used in a function, it must be defined in that function.
83	A delimiter is missing in a file READ statement. A comma is inserted.
84	In a READ LINE statement, the variable following the keyword LINE is not a string variable.
85	The label in an IF END statement is not defined.
86	A pound sign, #, is missing in an IF END statement. A pound sign is inserted.
87	A THEN is missing in an IF END statement. A THEN is inserted.
88	In a PRINT statement, the semicolon is missing following a USING string. A semicolon is inserted.

Table B-2. (continued)

Error	Meaning
89	In an ON statement, a GOTO or GOSUB is missing. A GOTO is assumed.
90	The index of a FOR loop header is of type string. The index must be an integer or real number.
91	The expression following the keyword TO in a FOR loop header is of type string. The expression must be an integer or real value.
92	The expression following the keyword STEP in a FOR loop header is of type string. The expression must be an integer or real value.
93	A variable in a DIM statement is defined previously as other than a subscripted variable.
94	An identifier is missing as an array name in a DIM statement. The entire statement is ignored.
95	A left parenthesis is missing in a DIM statement. A left parenthesis is inserted.
96	A right parenthesis is missing in a DIM statement. A right parenthesis is inserted.
97	The maximum number of dimensions allowed with a subscripted variable is exceeded.
98	A comma is missing in a POKE statement. A comma is inserted.
99	The index of a FOR loop header is not a simple variable.
100	In a CALL statement, a multiple-line function name is missing.
101	A file PRINT statement is terminated with a comma or semicolon.
102	A DIM statement is missing for this subscripted variable.

Table B-2. (continued)

Error	Meaning
103	A comma is missing in the label list associated with an ON GOTO or ON GOSUB statement. A comma is inserted.
104	A GOTO is missing in an ON ERROR statement. A GOTO is inserted.
105	A comma is missing in a PUT statement. A comma is inserted.
106	The expression in an IF statement is of type string. An integer or real expression is required.
107	The expression in a WHILE loop header is of type string. An integer or real expression is required.
108	In an OPEN or CREATE statement, the filename is missing.
109	In an OPEN or CREATE statement, the expression following the reserved word AS is missing.
110	A multiple-line function calls itself.
111	A semicolon separates expressions in a file PRINT statement. A comma is substituted for the semicolon.
112	A file PRINT statement does not have an expression list.
113	A TAB function is used in a file PRINT statement expression list.
114	Label used as a variable in a list of expressions.
115	A GO not followed by a TO or SUB. GOTO is assumed.
116	An OPEN or CREATE statement specifies both UNLOCKED and LOCKED access control.
117	A CREATE statement uses the READ-ONLY access control.

End of Appendix B

Appendix C
LINK86 Error Messages

During the course of operation, LINK86 can display error messages. The error messages and a brief explanation of their cause are described in Table C-1.

Table C-1. LINK86 Error Messages

Message	Meaning
CANNOT CLOSE	An output file cannot be closed. The disk might be write-protected.
COMMAND TOO LONG	Too many characters in a command line INPUT (INP) file.
DIRECTORY FULL	There is no directory space for the output files or intermediate files.
DISK READ ERROR	A file cannot be read properly.
DISK WRITE ERROR	A file cannot be written properly, probably due to a full disk.
GROUP OVER 64K	LINK86 attempted to put more than 64K bytes into a single group. This can occur in either the code or data group.
LINK86 ERROR	Internal LINK86 error.
MORE THAN ONE MAIN PROGRAM	There is more than one main program among the files being linked. When linking multiple CBASIC object files, only one file (the main program) can contain executable statements. All other files must contain only multiple-line functions.
MULTIPLE DEFINITION	The specified symbol is defined in more than one of the modules being linked.

Table C-1. (continued)

Message	Meaning
NO FILE	The indicated file cannot be found.
NO STACK SEGMENT	There is no stack segment defined for the .EXE file created by LINK86.
OBJECT FILE ERROR	LINK86 detected an error in the object file. This is caused by a translator error or by a bad disk file. Try regenerating the file.
SEGMENT ATTRIBUTE ERROR	The align type or combine type of the indicated segment is not the same as the type of the segment in a previously linked file. Regenerate the object file after changing the segment attributes as needed.
SEGMENT COMBINATION ERROR	Attempt to combine segments that cannot be combined, such as LOCAL segments. Change the segment attributes and relink.
SEGMENT OVER 64K	LINK86 attempted to put more than 64K bytes in a single segment.
SYMBOL TABLE OVERFLOW	LINK86 ran out of symbol table space. Reduce the number and/or length of symbols in the program, or relink on a system with more memory available.
TARGET OUT OF RANGE	LINK86 attempts to resolve a reference made to a symbol outside the acceptable referencing range.
UNDEFINED SYMBOLS	The symbols following this message are referenced but not defined in any of the modules being linked.

End of Appendix C

Appendix D
Execution Error Messages

The following warning message might be printed during execution of a CB86 program:

```
IMPROPER INPUT - REENTER
```

This message occurs when the fields you enter from the console do not match the fields specified in the INPUT statement. Following this message, you must reenter all values required by the input statement.

Execution errors cause a two-letter code to be printed. The following table contains valid CB86 error codes.

If an error occurs with a code consisting of an asterisk followed by a letter, such as *R, a CB86 library has failed. Please notify Digital Research of the circumstances under which the error occurs.

Table D-1. Execution Error Codes

Code	Error
AC	The argument in an ASC function is a null string.
BN	The value following the BUFF option in an OPEN or CREATE statement is less than 1 or greater than 128.
CE	The file being closed cannot be found in the directory. This occurs if the file has been changed by the RENAME function.
CM	The file specified in a CHAIN statement cannot be found in the selected directory. If no filetype is present, the compiler assumes a type of .OVR.
CT	The filetype of the file specified in a CHAIN statement is not .EXE or .OVR.
CU	A CLOSE statement specifies a file identification number that is not active.
CX	Overlay does not fit in overlay area.
CY	Disk read error during loading of overlay.

Table D-1. (continued)

Code	Error
DE	File to delete cannot be found in the directory.
DF	An OPEN or CREATE statement uses a file identification number that is already used.
DU	A DELETE statement specifies a file identification number that is not active.
DW	The operating system reports that there is no disk or directory space available for the file being written to, and no IF END statement is in effect for the file identification number.
DZ	Division by zero is attempted.
EF	Attempt to read past the end-of-file, and no IF END statement is in effect for the file identification number.
ER	Attempt to write a record of length greater than the maximum record size specified in the OPEN or CREATE statement for this file.
FR	Attempt to rename a file to a filename that already exists.
FU	Attempt to access a file that was not open.
IF	A filename in an OPEN or CREATE statement, or with the RENAME function, is invalid for your operating system.
IR	A record number of zero is specified in a READ or PRINT statement.
LN	The argument in the LOG function is zero or negative.
ME	The operating system reports an error during an attempt to create or extend a file. Normally, this means the disk directory is full.
MP	The third parameter in a MATCH function is zero or negative.
NE	A negative value is specified for the operand to the left of the power operator.

Table D-1. (continued)

Code	Error
NF	A file identification is less than 1 or greater than the maximum number of files allowed. See Appendix A, Table A-1.
NN	An attempt to print a numeric expression with a PRINT USING statement fails because there is not a numeric field in the USING string.
NS	An attempt to print a string expression with a PRINT USING statement fails because there is not a string field in the USING string.
OD	A READ statement is executed, but there are no DATA statements in the program, or all data items in all the DATA statements have been read.
OE	Attempt to OPEN a file that does not exist, and for which no IF END statement is in effect.
OF	An overflow occurs during a real arithmetic calculation.
OM	The program runs out of dynamically allocated memory during execution.
PU	An attempt was made to nest PRINT USING. This can only happen if a multiple-line function that contains a PRINT USING statement is called in the output list of another PRINT USING statement.
RB	Random access is attempted to a file activated with the BUFF option specifying more than one buffer.
RE	Attempt to read past the end of a record in a fixed file.
RU	A random read or print is attempted to a stream file.
SL	A concatenation operation results in a string greater than the maximum allowed string length.
SQ	Attempt to calculate the square root of a negative number.
SS	The second parameter of a MID$ function is zero or negative, or the last parameter of a LEFT$, RIGHT$, or MID$ is negative.

Table D-1. (continued)

Code	Error
TL	A tab statement contains a parameter less than 1.
UN	A PRINT USING statement is executed with a null edit string, or a backslash escape character, \, is the last character in an edit string.
WR	Attempt to write to a stream file after it is read, but before it is read to the end-of-file.

End of Appendix D

Appendix E
LIB86 Error Messages

LIB86 can produce the following error messages during processing. With each message, LIB86 displays additional information appropriate to the error, such as the filename or module name, to help isolate the location of the problem.

Table E-1. LIB86 Error Messages

Error	Meaning
CANNOT CLOSE	LIB86 cannot close an output file. Make sure the disk is not write-protected.
DIRECTORY FULL	There is not enough directory space for the output files. Erase unnecessary files or use a disk with more space.
DISK FULL	There is not enough disk space for the output files. Erase unnecessary files or use a disk with more space.
DISK READ ERROR	LIB86 detects a disk error while reading the indicated file. Try regenerating the file.
INVALID COMMAND SWITCH	LIB86 encounters an unrecognized switch in the command line. Retype the command line or edit the .INP file.
LIB86 ERROR n	Internal LIB86 error.

Table E-1. (continued)

Error	Meaning
MODULE NOT FOUND	The indicated module name, which appeared in a REPLACE, SELECT, or DELETE switch, could not be found. Retype the command line, or edit the .INP file.
MULTIPLE DEFINITION	The indicated symbol is defined as PUBLIC in more than one module. Correct the problem in the source file.
NO FILE	LIB86 could not find the indicated file.
RENAME ERROR	LIB86 cannot rename a file. Check that the disk is not write-protected.
SYMBOL TABLE OVERFLOW	There is not enough memory for the symbol table. Reduce the number of switches in the command line (MAP and XREF both use symbol table space), or use a system with more memory.
SYNTAX ERROR	LIB86 detected a syntax error in the command line, probably due to an improper filename or an invalid command option. LIB86 echoes the command line up to the point where it found the error. Retype the command line or edit the .INP file.

End of Appendix E

Appendix F
CBASIC to CBASIC Compiler Conversion Aid

In this appendix, CBASIC refers to the compiled/interpreted implementation of the CBASIC language, and CBASIC Compiler refers to the compiled implementation of the CBASIC language explained in this manual.

This conversion aid helps you convert your CBASIC programs to CBASIC Compiler. When you compile your source code in CBASIC Compiler, pay close attention to all error messages. This is the fastest way to determine any necessary changes. Most programs recompile with no conversion. If any problems arise, call the Digital Research Technical Support Center (408-375-6262) for assistance.

F.1 Subscripted Variables (Arrays)

CBASIC allows you to use a dimensioned variable name (an array) as a simple or unsubscripted variable. CBASIC treats these as separate and distinct variables. CBASIC Compiler does not allow a dimensioned variable without the array index.

CBASIC	CBASIC Compiler
`DIM A% (20)`	`DIM A% (20)`

`FOR I% = 1 to 20`
`  A% (I%) = 0`
`NEXT I%`
	`FOR I% = 1 to 20`
	`  A% (I%) = 0`
	`NEXT I%`

`A% = 100`	`A% = 100` `(error message #36)`

CBASIC Compiler issues error message #36 for the statement A% = 100 because the statement uses an identifier as a simple variable that was previously used as a subscripted variable.

CBASIC	CBASIC Compiler
A% = 100	A% = 100
DIM A% (20)	DIM A% (20) (error message #93)
For I% = 1 to 20 A% (I%) =0 NEXT I%	For I% = 1 to 20 A% (I%) =0 (error message #37) NEXT I%
END	

CBASIC Compiler issues error message #93 for the statement DIM A% (20) because a variable in a DIM statement is previously defined as other than a subscripted variable. CBASIC Compiler issues error message #37 for the statement A% = 100 because an identifier used as a subscripted variable was previously used as an unsubscripted variable.

To correct the error, change the unsubscripted variable to a different variable name of the same type. Choose a new variable that differs from all other variable names in your program.

F.2 FILE Statement

The FILE statement in CBASIC opens a file present on the referenced disk. The FILE statement can also create a file of the name you specify. However, CBASIC Compiler does not use the FILE statement. Use the OPEN, SIZE, and CREATE statements to open and create files.

CBASIC	CBASIC Compiler
FILE NAME$	IF SIZE (NAME$) <> 0 \ THEN OPEN NAME$ AS FILE.NO% \ ELSE CREATE NAME$ AS FILE.NO%

In the CBASIC Compiler example, if there is a file NAME$, the file is opened as usual. If there is no file NAME$, or the length of the file is zero (determined by the SIZE statement), the IF statement passes control to the CREATE statement, which creates the file NAME$. Both the OPEN and CREATE statements require a file reference number (FILE.NO%). However, the FILE statement does not need a file reference number.

When you convert a FILE statement, choose a file number that does not conflict with any other file reference numbers already in your program. Remember to modify the PRINT and READ statements that access the file to reflect the new file number.

F.3 SAVEMEM

The SAVEMEM statement, which executes routines written to the assembler in CBASIC, has no meaning in CBASIC Compiler. The CBASIC Language Reference Manual tells how to use assembler routines and explains how to link the routines to CBASIC Compiler programs.

F.4 CHAIN Statement

The CHAIN statement in CBASIC and CBASIC Compiler passes control from the program executing in memory to the program specified in the CHAIN statement. The CHAIN statement syntax is identical for both CBASIC and CBASIC Compiler.

CBASIC	CBASIC Compiler
CHAIN <string expression>	CHAIN <string expression>

The string expression following the keyword CHAIN must evaluate to a file specification, which is an overlay file the root program can chain to. If the filespec does not include the filetype, CBASIC assumes a filetype of .INT.

CBASIC Compiler (CB80) assumes a .OVL filetype for overlays. In CB80, the root program always has a .COM filetype. An overlay file with filetype .OVL cannot be the root of a chaining sequence. The string expression in a CHAIN statement must evaluate to a filespec with a .COM filetype when chaining to the root. A program compiled with CB80 can chain to a .COM file other than one generated with LK80.

CBASIC Compiler (CB86) assumes a .OVR filetype for overlays. In CB86, the root program resides in memory at all times. Once you chain from the root program to an overlay, you cannot effectively chain back to the root. Chaining back to the root causes the entire program to restart from the beginning. Certain data elements, including the stack, are reinitialized and all COMMON data from the first execution is lost. For programs compiled with CB86, overlays can only chain effectively to other overlays. Your root program must contain the COMMON declarations for all overlays.

F.5 String Lengths

CBASIC Compiler allows string lengths up to 32K. CBASIC Compiler uses two bytes to give this expanded string length; CBASIC uses one byte. To set strings to null in CBASIC Compiler, see the Programmer's Guide.

If your program uses the SADD function with PEEK and POKE to pass a string to an assembly language routine, you must change your program to accommodate the two-byte length indicator in CBASIC Compiler.

CBASIC	CBASIC Compiler
`LEN% = PEEK (SADD(STRING$))` `END`	`LEN% = (PEEK (SADD(STRING$)) AND 07FH \` `+ PEEK (SADD(STRING$) + 1)) * 256`

F.6 PEEK and POKE

The PEEK function in CBASIC and CBASIC Compiler returns the contents of the memory location specified in the PEEK function call. Memory locations in CBASIC Compiler might not contain the same information that CBASIC programs expect. You might have to change the memory location your program is examining, or remove the PEEK statement from your program.

The POKE statement behaves the same in CBASIC Compiler as it does in CBASIC. However, the memory locations in CBASIC Compiler differ from the memory locations in CBASIC. If your program contains a POKE statement to a location in a CBASIC program, it might insert the value at the wrong address when used in a CBASIC Compiler program. In particular, the statements,

```
POKE 0110H, 0
```

or

```
POKE 272, 0
```

used in CBASIC to adjust the console width, must be removed. Use the POKE statement carefully because the actual location of code is determined by the link editor.

F.7 FOR-NEXT Loops

When using FOR-NEXT loops in CBASIC, the NEXT statement can terminate more than one loop. CBASIC Compiler does not allow this construct. You must use a separate NEXT statement for each FOR statement that begins a loop.

CBASIC	CBASIC Compiler

`FOR I% = 1 TO 100`
`FOR J% = 1 TO 100`
`.`
`. (statements)`
`.`
`NEXT J%, I%`
	`FOR I% = 1 TO 100`
	`FOR J% = 1 TO 100`
	`.`
	`. (statements)`
	`.`
	`NEXT J%`
	`NEXT I%`

Also, CBASIC executes all statements in the FOR-NEXT loop at least once. CBASIC Compiler executes the statements in a FOR-NEXT loop zero or more times, depending on the values of the loop indexes. This is potentially troublesome. Examine the logic of your programs, and make any necessary changes.

F.8 Console Width

To facilitate cursor addressing, CBASIC Compiler generates a carriage return only upon executing a PRINT statement not terminated by a comma or semicolon. This is analogous to setting the CBASIC console width to zero by a POKE to 272. CBASIC automatically generates a carriage return when the console width has been exceeded. Therefore, CBASIC programs that assume the cursor returns when the console width is exceeded might not execute correctly in CBASIC Compiler.

F.9 FRE

In CBASIC Compiler, FRE returns a binary value that represents the number of bytes of available memory. In CBASIC, the binary value represents a real value. Programs that use FRE must interpret negative values correctly, because CBASIC Compiler arithmetic routines interpret binary values in excess of 32,767 as negative numbers. In general, negative values indicate ample available memory.

The following statement can determine whether adequate memory is available.

```
IF (FRE > 0) AND (FRE < MIN.MEMORY%) THEN \
    CALL LOW.MEMORY.WARNING
```

F.10 READ and INPUT Statements for Integers

READ and INPUT statements handle integers differently in the two languages. CBASIC accepts all numeric values as real numbers, and then converts to integers if required. CBASIC Compiler accepts integers directly.

CBASIC	CBASIC Compiler
DATA 10.7, 1E2	DATA 10.7, 1E2
READ A%,B%	READ A%,B%
The values of A% and B% after the READ are:	The values of A% and B% after the READ are:
A% = 11 B% = 100	A% = 10 B% = 1

With CBASIC Compiler, conversion stops at the first character not a part of a valid integer.

F.11 Function and Variable Names

CBASIC Compiler requires that function names, variables, and statement labels be unique. In CBASIC, all functions must start with the letters FN, and labels must be numeric constants. Thus, no problems should occur when you convert programs from CBASIC to CBASIC Compiler. Remember that variables and arrays might conflict as described in Section F.1.

F.12 Labels

CBASIC Compiler places all program labels, including unreferenced labels, in a symbol table. CBASIC does not put unreferenced labels in the symbol table.

A label in a multiple-line function is local to the function. This is not the same in CBASIC.

CBASIC	CBASIC Compiler
DEF FN.A	DEF FN.A
100 PRINT "HELLO"	100 PRINT "HELLO"
FEND	FEND
GOTO 100	GOTO 100
	(error message #82)

CBASIC Compiler issues error message #82 because the label in a GOTO statement is undefined. The label used in a function must be defined in that function.

F.13 Warning Messages

CBASIC Compiler produces no warning messages during the execution of a program. All errors are fatal and execution terminates unless you use an ON ERROR GOTO statement to trap the error.

F.14 New Reserved Words

CBASIC Compiler incorporates new reserved words with some of the newly implemented features. If your CBASIC programs use these words as variables, rename them to a different variable name. The following is a list of reserved words unique to CBASIC Compiler.

ATTACH	GET	PUT
%DEBUG	INITIALIZE	READONLY
DETACH	INKEY	REAL
ERR	INTEGER	SHIFT
ERRL	LOCK	STRING
ERROR	LOCKED	STRING$
ERRX	MOD	UNLOCK
EXTERNAL	PUBLIC	UNLOCKED

End of Appendix F

Index

Reader Comment Card

We welcome your comments and suggestions. They help us provide you with better product documentation.

Date _____ First Edition: May 1983

1. What sections of this manual are especially helpful?

2. What suggestions do you have for improving this manual? What information is missing or incomplete? Where are examples needed?

3. Did you find errors in this manual? (Specify section and page number.)

CBASIC Compiler™ (CB-86™) Language Programmer's Guide
for the IBM® Personal Computer Disk Operating System

COMMENTS AND SUGGESTIONS BECOME THE PROPERTY OF DIGITAL RESEARCH.

From: _____

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 182 / PACIFIC GROVE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL RESEARCH™

P.O. Box 579
Pacific Grove, California
93950

Attn: Publications Production