

basic compiler user's manual

Introduction

Introduction to Compilation

Demonstration Run

Editing

Debugging with the Interpreter

Compiling

Linking

Running a Program

A Compiler/Interpreter Comparison

Error Messages

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software in this package on cassette tape, disk, or any other medium for any purpose other than backing up your software.

© Microsoft, 1981

LIMITED WARRANTY

MICROSOFT shall have no liability or responsibility to purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

CP/M is a registered trademark of Digital Research

Microsoft BASIC Compiler, Version 5.3

July, 1981

ADDENDA TO: The BASIC Compiler User's Manual

There are significant differences between version 5.3 of the Microsoft BASIC Compiler and previous versions. Major differences are listed below:

1. CHAINing with COMMON is supported.
2. Runtime support is now organized so that a single large module contains most of the runtime library.
3. In conjunction with points 1. and 2., now large systems of programs can be created that share common data and use a single runtime environment.
4. Larger programs (16K larger on the average) can be compiled and linked.
5. Programs take up significantly less disk space.

See Appendix D of this manual for a further discussion of these and other changes.

SYSTEM REQUIREMENTS

The Microsoft BASIC Compiler can be used with most microcomputers with a minimum of 48K RAM and one disk drive. We recommend two drives, however, for easier operation. The compiler operates under the CP/M operating system, which is required.

CP/M is a registered trademark of Digital Research

Royalty Information

For those who want to market application programs, use of the BASIC Compiler provides you with three major benefits:

1. Increased speed of execution for most programs,
2. Decreased program size for most programs, and
3. Source code security.

When you distribute a compiled program, you distribute highly optimized machine code, not source code. Consequently, you distribute your program in very compact form and protect your source program from unauthorized alteration.

The policy for distribution of parts of the BASCOM package is as follows:

1. Any application program that you generate by linking to either of the two runtime libraries (BASLIB.REL and OBSLIB.REL) may be distributed without payment of royalties. A copyright notice reading "PORTIONS COPYRIGHTED BY MICROSOFT, 1981" must be displayed on the media.
2. However, the BRUN.COM runtime module cannot be distributed without first entering into a license agreement with Microsoft for such distribution. A copy of the license agreement can be readily obtained by writing to Microsoft. Also, a copyright notice reading "PORTIONS COPYRIGHTED BY MICROSOFT, 1981" must be displayed on the media.
3. All other software in your BASIC Compiler package cannot be duplicated except for purposes of backing up your software. Other duplication of any of the software in the BASIC Compiler package is illegal.

All of the above information is included in the Non-Disclosure Agreement, which must be signed and returned to Microsoft at the time the BASIC Compiler is purchased. In order to provide you any updates or fixes, we must have your completed form on file. Failure to register and sign the non-disclosure agreement voids any warranty expressed or implied.

CONTENTS

CHAPTER 1	INTRODUCTION
1.1	How to Use this Manual
1.2	Contents of the BASIC Compiler Package
	Software Documentation
1.3	Software
	BASCOM - The BASIC Compiler
	L80 - The LINK-80 Linking Loader
	M80 - The MACRO-80 Macro-assembler
	BRUN.COM - The Runtime Module
	BASLIB.REL - The Runtime Library
	OBSLIB.REL - The Alternate Runtime Library
1.4	Documentation
	The BASIC Compiler User's Manual
	The BASIC-80 Reference Manual
	The Utility Software Manual
1.5	Resources for Learning BASIC
CHAPTER 2	INTRODUCTION TO COMPILATION
2.1	Compilation vs. Interpretation
2.2	Vocabulary
2.3	The Program Development Process
CHAPTER 3	DEMONSTRATION RUN
3.1	Compiling
3.2	Linking
3.3	Running a Program
CHAPTER 4	EDITING
CHAPTER 5	DEBUGGING WITH THE BASIC INTERPRETER

CHAPTER 6	COMPILEING
6.1	Command Line Syntax
6.2	Sample Compiler Invocations
6.3	Compiler Switches
CHAPTER 7	LINKING
7.1	Sample Linker Sessions
7.2	Linking to Compiled BASIC .REL Files
7.3	Runtime Support
CHAPTER 8	RUNNING A PROGRAM
CHAPTER 9	COMPILER/INTERPRETER COMPARISON
9.1	Operational Differences
9.2	Language Differences
9.3	Other Differences
CHAPTER 10	ERROR MESSAGES AND DEBUGGING
10.1	BASIC Comptime Error Messages
10.2	BASIC Runtime Error Messages
APPENDIX A	Creating a System of Programs with the BRUN.COM Runtime Module
APPENDIX B	ROM-able Code
APPENDIX C	Memory Map
APPENDIX D	Differences Between Version 5.3 and Previous Versions of the BASIC Compiler

CHAPTER 1

INTRODUCTION

The Microsoft BASIC Compiler is an optimizing compiler designed to complement Microsoft's BASIC-80 interpreter. Since BASIC-80 is the recognized standard for microcomputer BASIC, the BASIC compiler can support programs written for a wide variety of microcomputers.

In addition, the BASIC Compiler allows you to create programs that:

1. Execute faster in most cases than the same interpreted programs,
2. Require less memory in most cases than the same interpreted programs, and are
3. Source-code secure.

These benefits can be critical for real-time applications such as graphics, where execution speed can often make or break an application; business applications, where several CHAINED programs can be supported by a main menu in a single runtime environment; and commercial applications, where software is being sold in a competitive marketplace and source-code security is essential.

There is another major advantage that you gain by owning the compiler. Because the BASIC Compiler has been created to support most of the interpreted BASIC-80 language, the interpreter and the compiler complement each other, and provide you with an extremely powerful BASIC programming environment. In this environment, you can quickly RUN and debug programs from within BASIC-80, and then later compile those programs to increase their speed of execution and to decrease their space in memory.

Although the language supported by the BASIC Compiler is not identical to that supported by the interpreter, the compiler has been designed so that compatibility is maintained where ever possible. Note also, that the file named BRUN.COM contains the majority of the runtime

environment. For this reason, BRUN.COM is called the runtime module. The runtime module is loaded when program execution begins; later execution of CHAINed programs does not require reloading. This allows you to develop a system of related programs that can all be run using the same runtime environment. The runtime environment required by your program need not be saved on disk as part of your executable .COM file. For a system of four programs, this can save at least 48K of disk space--a substantial savings.

This version (5.3) of the BASIC Compiler is substantially different from previous versions. These differences are summarized in Appendix D.

1.1 HOW TO USE THIS MANUAL

The BASIC Compiler User's Manual is designed for users who are unfamiliar with the compiler as a programming tool. Therefore, this manual provides both a step-by-step introduction and a detailed technical guide to the BASIC Compiler and its use. After a few compilations, the User's Manual then serves as both a refresher on procedures and as a technical reference.

This manual assumes that the user has a working knowledge of the BASIC language. For reference information, consult the BASIC-80 Reference Manual. If you need additional information on BASIC programming, refer to Section 1.5 of this manual, RESOURCES FOR LEARNING BASIC.

Organization

This manual contains the following chapters:

Chapter 1, INTRODUCTION. Provides brief descriptions of the contents of the BASIC Compiler package, and gives a list of references for learning BASIC programming.

Chapter 2, AN INTRODUCTION TO COMPIRATION. Gives you an introduction to the vocabulary associated with compilers, a comparison of interpretation and compilation, and an overview of program development with the compiler.

Chapter 3, DEMONSTRATION RUN. Takes you step by step through the compiling, linking, and running of a demonstration program.

Chapter 4, EDITING. Describes how to create a BASIC source program for later compilation, and how to use the %INCLUDE compiler directive.

Chapter 5, DEBUGGING WITH THE INTERPRETER. Describes how to debug the BASIC source file with the BASIC-80 interpreter before compiling it. Note that Chapter 9, A COMPILER/INTERPRETER COMPARISON describes differences between the language supported by the compiler and that supported by the BASIC-80 interpreter.

Chapter 6, COMPILING. Describes use of the BASIC Compiler in detail, including descriptions of the command line syntax and the various compiler options.

Chapter 7, LINKING. Describes how to use LINK-80 to link your programs to needed runtime support. (Note that the Utility Software Manual contains further reference material on LINK-80.)

Chapter 8, RUNNING A PROGRAM. Describes how to run your final executable program.

Chapter 9, A COMPILER/INTERPRETER COMPARISON. Describes all of the language, operational, and other differences between the language supported by the BASIC Compiler and that supported by the BASIC-80 interpreter. It is important to study these differences and to make the necessary editing changes in your BASIC program before you use the compiler.

Chapter 10, ERROR MESSAGES. Describes each error message.

Appendices that show you how to create a system of programs with the BRUN.COM runtime module, and how to generate a ROM-able program are also provided. Two other appendices give you a memory map of the BRUN.COM runtime environment, and describe the differences between this and pre-5.3 versions of the compiler.

NOTATION USED IN THIS MANUAL

For the most part, any punctuation marks or other special characters used, especially in command formats, are to be taken literally. Consider these marks as part of the command format.

However, some special characters used in command formats have special meanings:

- | | | |
|--|-----|--|
| capital letters | FOO | Indicate that the parameter or command must be entered exactly as shown. |
| angle brackets | <> | Indicate that enclosed text specifies a class of parameters. Any parameter that you enter in this position must be a valid member of that parameter class. Hence, <filename> means that you must enter a legal filename. |
| Capital letters enclosed by angle brackets are used to specify non-displayable ASCII characters. For example, <CR> specifies entry of a carriage return. | | |
| square brackets | [] | Indicate that the enclosed parameter is optional. For instance, <filename>[,<filename>] specifies entry of either one filename or two filenames. |
| ellipses | ... | Indicate that the symbols preceding the ellipses can be entered as many times as needed. For example, <filename>... indicates entry of one or more filenames. |

1.2 CONTENTS OF THE BASIC COMPILER PACKAGE

The BASIC Compiler Package contains:

One disk containing the following files:

BASCOM.COM - The BASIC Compiler
BRUN.COM - The Runtime Module
BASLIB.REL -The Runtime Library
OBSLIB.REL - The Alternate Runtime Library
BCLOAD - Runtime load information file
L80.COM - The LINK-80 Linking Loader
M80.COM - The MACRO-80 Macro-assembler
CREF.COM - The Cross-reference Utility
LIB80.COM - The Library Manager
DEMO.BAS - A Demonstration program

A Binder with three Manuals including the following:

The BASIC Compiler User's Manual (this manual)
The BASI -80 Reference Manual
The Utility Software Manual

1.3 SOFTWARE

A description follows of the function of the software on your disk:

1. BASCOM.COM - (The BASIC Compiler) Compiles BASIC source files into relocatable and linkable .REL files.
2. BRUN.COM - (The Runtime Module) A single module containing most of the routines called from your compiled .REL file. So that the entire BRUN.COM module is not loaded into memory at linktime, a dummy module that resolves all of the references to routines in BRUN.COM resides in BASLIB.REL.
3. BASLIB.REL - (The Runtime Library) A collection of routines implementing functions of the BASIC language not found in BRUN.COM. Your .REL file may contain calls to these routines.
4. OBSLIB.REL -(The Old Runtime Library) A collection of modules containing routines that are similar to the routines found in BASLIB.REL and BRUN.COM, above. This library should be used for applications that you wish to make ROM-able, or for those that you want to execute as single .COM files without the BRUN.COM runtime module. This library

does not support CHAIN with COMMON, CLEAR, or RUN <linenumber>. Additional differences are described in Chapter 6, Linking.

5. BCLOAD - (Runtime load information file) Tells at what address to load your program at linktime, and where to find BRUN.COM at runtime.
6. L80.COM - (The Linking Loader) Links and loads compiled .REL files, library modules, and assembly language routines to create an executable .COM file.
7. M80.COM - (The Macro-Assembler) Assembles assembly language routines into .REL files that can later be linked to your compiled .REL file.
8. CREF.COM - (The Cross-Reference Utility) Creates a cross-referenced listing of the use of variables in assembly language programs.
9. LIB80.COM - (The Library Manager) Allows you to create and modify user runtime libraries.
10. DEMO.COM - (A Demonstration Program) Used in Chapter 3 to demonstrate program development with the BASIC Compiler.

1.4 DOCUMENTATION

Three manuals come with the BASIC Compiler package: the BASIC Compiler User's Manual (this manual), the BASIC-80 Reference Manual, and the Utility Software Manual. Each manual provides specific information necessary for the successful creation of an executable compiled BASIC program.

THE BASIC COMPILER USER'S MANUAL

This manual is described above in Section 1.1, How To Use This Manual. See that section for more information.

BASIC-80 REFERENCE MANUAL

The BASIC-80 Reference Manual describes syntax and usage of Microsoft's standard BASIC language. This is the language supported by the BASIC Compiler, with the exceptions noted in Chapter 9 of the BASIC Compiler User's Manual. Note that the BASIC-80 interpreter itself is not supplied as part of the BASIC Compiler package.

The BASIC Compiler supports, in some form, all of the statements and commands described in the BASIC-80 manual, except:

AUTO	CLOAD	CSAVE	CONT	DELETE	EDIT
ERASE	LIST	LLIST	LOAD	MERGE	NEW
RENUM	SAVE				

IMPORTANT

Language, operational, and other differences between the BASIC Compiler and the BASIC-80 interpreter are described in Chapter 9, A BASIC COMPILER/INTERPRETER COMPARISON. You should review the information in that chapter before compiling any of your programs that already run without problem when interpreted by BASIC-80. Only then make any necessary changes.

UTILITY SOFTWARE MANUAL

The Utility Software Manual provides descriptions of the following pieces of software in your BASIC Compiler package:

1. LINK-80
2. MACRO-80
3. LIB-80
4. CREF-80

1.5 RESOURCES FOR LEARNING BASIC

Microsoft provides complete instructions for using the BASIC Compiler. However, no teaching material for BASIC programming has been supplied. The BASIC-80 Reference Manual is strictly a syntax and semantics reference for the Microsoft BASIC-80 language.

If you are new to BASIC and need help learning to program in this language, we suggest the following texts:

1. Dwyer, Thomas A. and Margot Critchfield. BASIC and the Personal Computer. Addison-Wesley, 1978.
2. Simon, David E. BASIC from the Ground Up. Hayden, 1978.
3. Albrecht, Robert L., LeRoy Finkel, and Jerry Brown. BASIC. John Wiley & Sons, 1973.

CHAPTER 2

INTRODUCTION TO COMPIRATION

2.1 COMPIRATION VS. INTERPRETATION

A microprocessor can execute only its own machine instructions; it cannot execute BASIC statements directly. Therefore, before a program can be executed, some type of translation must occur from the statements contained in your BASIC program to the machine language of your microprocessor. Compilers and interpreters are two types of programs that perform this translation. This discussion explains the difference between these two translation schemes, and explains why and when you want to use the compiler.

Interpretation

An interpreter performs translation line by line during runtime. To execute a BASIC statement, the interpreter must analyze the statement, check for errors, then perform the BASIC function requested.

If a statement must be executed repeatedly (inside a FOR/NEXT loop, for example), this translation process must be repeated each time the statement is executed.

In addition, BASIC programs are stored as a linked list of numbered lines, and each line is not available as an absolute memory address during interpretation. Therefore, branches such as GOTOS and GOSUBS cause the interpreter to examine every line number in a program, starting with the first, until the line referred to is found.

Similarly, a list of all variables is maintained by the interpreter. When a reference to a variable is made in a BASIC statement, this list must be searched from the beginning until the variable referred to is found. Thus, absolute memory addresses are not associated with the variables in your program.

Compilation

A compiler, on the other hand, takes a source program and translates it into an object file. The object file contains relocatable machine code. All translation takes place before runtime; no translation of your BASIC source file occurs during the execution of your program. In addition, absolute memory addresses are associated with variables and with the targets of GOTOS and GOSUBs, so that lists of variables or of line numbers do not have to be searched during execution of your program.

Note also, that the compiler is an optimizing compiler. Optimizations such as expression re-ordering or sub-expression elimination are made to either increase speed of execution or to decrease the size of your program.

These factors all combine to measurably increase the execution speed of your program. In most cases, execution of BASIC programs is 3 to 10 times faster than execution of the same program under the interpreter. If maximum use of integer variables is made, execution can be up to 30 times faster.

2.2 VOCABULARY

Before you read any farther in this manual, you need to become familiar with some of the vocabulary that is commonly used when discussing compilers.

To begin with, you should understand that a BASIC program is more commonly called a BASIC "source." This source file is the input file to the compiler and must be in ASCII format. The compiler translates this source and creates as output, a new file, called a relocatable "object" file. These two files have the default extensions .BAS and .REL, respectively.

Other terms that you should know are related to stages in the development and execution of a compiled program. These stages are described below:

Completetime - That period of time during which the compiler is executing, and during which it is compiling a BASIC source file and creating a relocatable object file.

Linktime - That period of time during which the linker is executing, and during which it is loading and linking together relocatable object files and library files.

Runtime - That period of time during which a compiled and linked program is executing. By convention, runtime refers to the execution time of your program and not to the execution time of the compiler or the linker.

You should also learn the following terms pertaining to the linking process and to the runtime support library:

Module - A fundamental unit of code. There are several types of modules, including relocatable and executable modules. Relocatable modules are manipulated by the linker. Your final executable program and BRUN.COM are executable modules. Note that BRUN.COM is special since it is executable only so that you can see its version number. Its main purpose is to serve as a library of routines that can be called at runtime from your compiled program.

Global Reference - A variable name or label in a given module that is referred to by a routine in another module. Global labels are entry points into modules.

Unbound Global Reference - A global reference in a module that is not declared in that module. The linker tries to "resolve" this situation by searching for the declaration of that reference in other modules. These other modules are usually library modules in the runtime library. If the variable or label is found, the address associated with it is substituted for the reference in the first module, and is then said to be "bound." When a variable is not found, it is said to be "undefined."

Relocatable - A module is relocatable if the code within it can be "relocated" and run at different locations in memory. Relocatable modules contain labels and variables represented as offsets relative to the start of the module. These labels and variables are said to be "code relative." When the module is loaded by the linker, an address is associated with the start of the module. The linker then computes an absolute address that is equal to the associated address plus the code relative offset for each label or variable. These new computed values become the absolute addresses that are used in the binary .COM file.

.REL files and library files are all relocatable modules. Note that normally a relocatable module contains global references as well: these are resolved after all local labels and variables have been computed within other relocatable modules. This process of computing absolute relocated values and resolving global references is what is meant by "linking."

Routine - Executable code residing in a module. More than one routine may reside in a module. The BRUN.COM module contains a majority of the library routines needed to implement the BASIC language. A library routine usually corresponds to a feature or sub-feature of the BASIC language.

Runtime Support - The body of routines that may be linked to your compiled .REL file. These routines implement various features of the BASIC language. BRUN.COM, OSLIB.REL, and BASLIB.REL all contain runtime support routines. See Chapter 6, LINKING, for more information on runtime support.

The BRUN.COM Module - A module containing most of the routines needed to implement the BASIC language. It is a peculiarity of the BRUN.COM module that it is an executable .COM file. BRUN.COM, for the most part, is a library of routines: it is made executable so that you can see the version number of the module.

Use of BRUN.COM gives you the following advantages:

1. True CHAINing is allowed.
2. COMMON can be used to communicate between CHAINED programs, not just between subroutines.
3. Linktime is reduced, since unbound globals do not have to be searched for in multiple library modules.
4. The BRUN.COM module is not explicitly loaded at link-time, allowing considerably larger programs to be linked and loaded, since an extra 16K is not contained in your final .COM file.

Note, however, that BRUN.COM must be accessible on disk when you execute your final .COM file.

The BASLIB.REL Runtime Library- A collection of modules containing routines for BASIC functions that often are not used in a program. The transcendental functions, the PRINT USING function, some error handling code, and other miscellaneous functions are contained in this library. These functions are linked to your program only if needed.

BASLIB.REL also contains a module consisting of all the global references in the BRUN.COM module. This module exists so that the routines in BRUN.COM can be linked to your compiled .REL file without BRUN.COM itself being brought into memory at linktime.

The OSLIB.REL Runtime Library - A collection of modules containing routines almost identical in function to similar routines contained in BRUN.COM and BASLIB.REL. However,

this library does not support the CLEAR command, the RUN <line-number> option of the RUN command, and COMMON between CHAINED subprograms. It does support a version of CHAIN that is semantically equivalent to the simple RUN command.

Link Loading - The process in which the LINK-80 linking loader loads modules into memory, computes absolute addresses for labels and variables in relocatable modules, and then resolves all global references by searching the BASLIB.REL runtime library. After loading and linking, the linker saves the modules that it has loaded into memory as a single .COM file on your disk. This entire process is called link loading.

Complete understanding of all the above terms is not essential for continued reading. You may want to refer back to these terms later, as you become familiar with the compiler and with the linker. We now discuss the program development process.

2.3 THE PROGRAM DEVELOPMENT PROCESS

This discussion of the program development process is keyed to figure 2.1. Use it for reference when reading this text.

Program development begins with (1.) the creation of a BASIC source file. The best way to create a BASIC source file is with the editing facilities of BASIC-80, although you can use any general purpose text editor if you wish. Note that files must be SAVED with the ,A option from BASIC-80.

Once you have written a program, you should use BASIC-80 (2.) to debug the program by RUNning it to check for syntax and program logic errors. There are a few differences in the languages understood by the compiler and the interpreter, but for the most part they are identical. Because of this similarity, running a program provides you with a much quicker syntactic and semantic check of your program than does compiling, linking, and finally executing a program. Therefore, you should strive to make the interpreter your chief debugging tool.

After you have debugged your program with the interpreter, (3.) compile it to check out differences that may exist between interpreted and compiled BASIC. The compiler flags all syntax errors as it reads your source file. If compilation is successful, the compiler creates a relocatable .REL file.

The .REL file is not executable, and needs to be linked to the BASLIB.REL runtime library. You may want to include your own assembly language routines to increase the speed of execution of a particular algorithm, or to handle operations that require a more intimate relationship with the microprocessor. For these cases, use MACRO-80, the macro-assembler, (4.) to assemble routines that you can later link to your program. Similarly, separately compiled Microsoft FORTRAN subroutines can be linked to your program. (FORTRAN is a separate product available from Microsoft. Macro-80 is discussed in the Utility Software Manual.) The linker (5.) links all modules needed by your program, and produces as output an executable object file with .COM as the default extension. This file can be (6.) executed like any .COM file by simply typing the file's base name (the file name less its .COM extension).

This program development process is demonstrated in the following chapter, Chapter 3, DEMONSTRATION RUN.

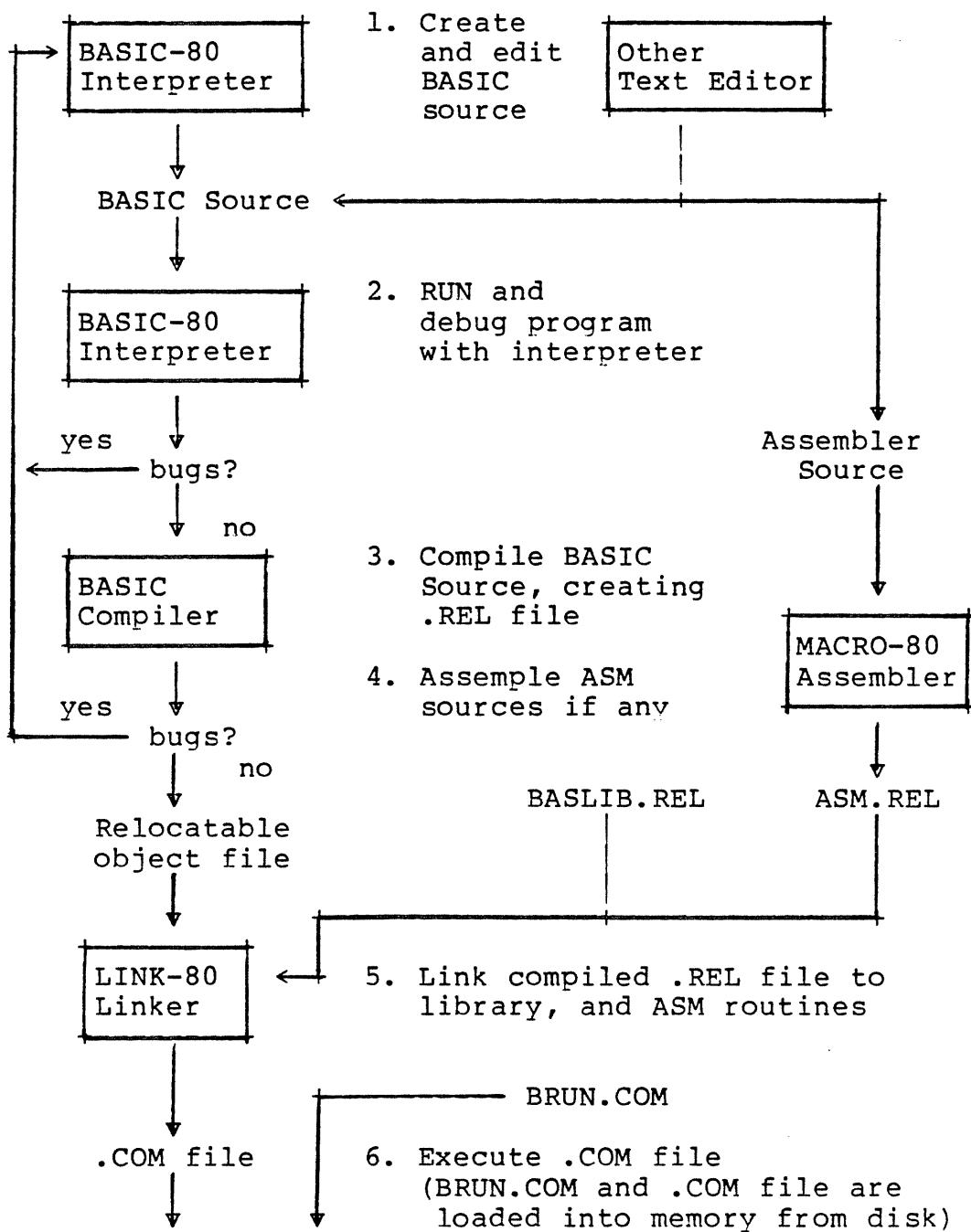


Figure 2.1 The Program Development Process

CHAPTER 3

DEMONSTRATION RUN

IMPORTANT

Before beginning this demonstration run, make a backup copy of your BASIC Compiler disk. Next, COPY CP/M on to your copied disk so that it can be booted up by itself. Store your master disk in a safe place and work with this backup copy.

This chapter provides step by step instructions for using the BASIC Compiler. These steps are outlined using a demonstration program.

We strongly recommend that you compile the demonstration program before compiling any other programs, because this demonstration run gives you an overview of the compilation process. Also, you should read Chapters 4 through 9. They contain important information that is crucial to successful development of a program.

If you enter commands exactly as described in this chapter, you should have a successful session with the BASIC Compiler. If a problem does arise, check and redo each step carefully.

The five steps in developing a program with the BASIC Compiler are:

1. Editing (entering and correcting the BASIC program)
2. Debugging with the Interpreter (using BASIC-80 to RUN your program)
3. Compiling (creating a relocatable object file)
4. Linking (creating an executable object file)
5. Running (executing the program)

Because we have prepared a special debugged demonstration program on disk, you do not have to perform the first two steps in the program development process. Therefore, the demonstration run begins with compilation. Note that we have SAVED the demonstration program on disk with the ,A option, since all files must be in ASCII format to be readable by the compiler.

3.1 COMPIILING

To begin compiling a program, insert a copy of your BASCOM disk in drive A: and boot up your system. The BASCOM disk contains all of the files that you need to carry out this demonstration run, including the demonstration program named DEMO.COM. In this demonstration all files produced by the compiler and by the linker will be placed on this disk. Perform the following steps to compile your program:

1. Enter the BASIC Compiler command line.

Invoke the compiler by typing:

```
BASCOM DEMO,DEMO=DEMO
```

This command line begins compilation of the source file. The source file is the last parameter on the command line, and the .BAS default extension is assumed.

The compiler generates relocatable object code that is stored in the file specified by the first parameter on the command line. This file is created with the default .REL extension.

At the same time, a listing file is written out to your disk. Its file name is that specified by the second parameter on the command line (following the comma). This file is created with the default .PRN extension.

2. Look for error messages.

When the compiler has finished, it displays the message "00000 FATAL ERROR(S)", and program control is returned to CP/M.

At this point, you should see two new files listed in the A: directory: DEMO.REL and DEMO.PRN.

3. Delete the listing file.

You may want to view or print out the listing file (DEMO.PRN) at this juncture in the demonstration run. In any event, you should delete the listing file to gain additional disk space. To do this, type:

ERA DEMO.PRN

Further information on listing files is given in Chapter 6, COMPILING. You are now ready for the next step--Linking.

3.2 LINKING

Linking is accomplished with the LINK-80 linking loader (the file named L80.COM). Perform the following steps to link DEMO.REL to needed runtime support.

1. Invoke LINK-80.

To invoke LINK-80, simply type:

L80

Your computer will search your disk for LINK-80, load it, and then return the asterisk (*) prompt.

If you want to stop the linking process, and you have entered only L80 and nothing more, you can exit to CP/M by entering Control-C.

2. Enter the filename(s) you want loaded and linked.

LINK-80 performs the following operations:

Loads relocatable object (.REL) modules,

Computes absolute addresses for all local references within modules,

Resolves all unbound global references between loaded modules, and

Saves the linked and loaded modules as an executable (.COM) file on disk.

After the asterisk prompt, type the following line to cause loading, linking, and saving of the program DEMO.COM:

DEMO,DEMO/N/E

The first part of the command (DEMO) causes loading of the program called DEMO.REL. The /N switch causes an executable image of the linked file to be saved on your disk with the Name DEMO.COM. This occurs after an automatic search of the BASLIB.REL runtime library. The file is only saved after a /E or a /G switch is entered on the command line. You may enter as many command lines as needed before you enter a /E or /G switch. Note that the /E switch, causes an Exit back to CP/M. If you substitute /G for /E here, you cause execution of the new .COM file after linking. In either case, BASLIB.REL is automatically searched to satisfy any unbound global references before linking ends.

3. Wait.

The linking process requires several minutes. During this time, the following messages will appear on your screen:

```
DATA <program-start> <program-end> <bytes>
      <free-bytes> BYTES FREE
      [<start-address> <program-end> <num-of-pages>]
```

This information is described in Chapter 7, LINKING.

4. Examine your directory

Type as follows:

DIR A:

You should see a file named DEMO.COM. This is an executable file.

3.3 RUNNING A PROGRAM

Once you have compiled and linked your program, it is simple to run it. From CP/M, enter the program filename, less its .COM extension. In the case of DEMO.COM, type:

DEMO

The speed of execution of your program should be quite fast relative to execution of the same program with the BASIC-80 interpreter. Compare speeds of execution by running the BASIC source program with the interpreter.

LEARNING MORE ABOUT DEVELOPING A PROGRAM

You have successfully developed and run a simple BASIC program. You are now ready to learn the more technical details that you need to know to compile other BASIC programs. Chapters 4-8 contain more extensive descriptions of each of the steps you followed in this chapter. Chapter 9 describes all of the language, operational, and other differences between the BASIC Compiler and the BASIC interpreter.

CHAPTER 4

EDITING A SOURCE PROGRAM

The creation of your BASIC source program requires the use of a text editor. Most any text editor will do, but the obvious choice is the line editor available from within BASIC-80. If you have previous experience with BASIC-80, then there is little need to learn how to use a new editor.

It is important to note that the compiler expects its source file in ASCII format. If you edit a file from within BASIC-80, it must be SAVED with the ,A option; otherwise, the compiler will attempt to read a tokenized encoding of your BASIC program. For more information on editing, saving, and loading files with BASIC-80, you should refer to the BASIC-80 Reference Manual.

The BASIC Compiler supports a useful feature that is not available when you run a BASIC program under the interpreter. This is the %INCLUDE <filename> compiler directive. It is called a compiler directive rather than a BASIC command because it is not really a part of the BASIC language. Rather, it is a command to the compiler, thus its distinctive "%" prefix.

The %INCLUDE <filename> directive allows you to switch compilation of BASIC source files in mid-stream. It switches from the source file you specify on invoking the compiler, to the file you specify as <filename> in the %INCLUDE directive (the <filename> parameter does not require surrounding quotes). When compilation of the external file is complete, the compiler switches back to the original BASIC source and continues compilation.

This process is equivalent to having the text of <filename> expanded at the location of the %INCLUDE directive in your BASIC source. (Note that %INCLUDEs cannot be nested.) Any file that is %INCLUDED in your BASIC program is called an INCLUDE file. All INCLUDE files must be SAVED with the ,A option if edited within BASIC-80. If you use another editor, this is not the case.

You may want to create %INCLUDE files for any COMMON declarations existing in more than one program, or for subroutines that you might have in an external library of subroutines. Note that the BASIC-80 interpreter does not support the %INCLUDE directive, thus a syntax error occurs when %INCLUDE is encountered during interpretation.

To make INCLUDE files easily included in large numbers of programs, you may want to edit the INCLUDE file so that it has no line numbers. The compiler supports sequences of lines without line numbers if the /C is used during compilation. However, the BASIC-80 interpreter does not allow you to create lines without line numbers, so you need an external editor to do so. Also, line numbers must exist for any lines that are targets for GOTOS or GOSUBS.

A word here about the differences between the languages supported by the interpreter and the compiler. The interpreter supports a number of editing and file manipulation commands that are useful mainly when creating a program. Examples are LOAD, SAVE, LIST, and EDIT. These are operational commands not supported by the compiler. Some differences also exist for some of the other statements and functions. Realize that it is during editing that you should account for language differences. See Chapter 9, A COMPILER/INTERPRETER COMPARISON for a full description of these differences.

Note also, that the interpreter cannot accept physical lines greater than 254 characters in length. A physical line is the unit of input to the interpreter. Interpreter logical lines can contain as many physical lines as desired.

In contrast to the interpreter, the BASIC Compiler accepts logical lines of up to only 253 characters in length. If you are using an external editor, you can create logical lines containing sequences of physical lines by ending your lines with an underscore. The underscore removes the significance of the carriage return in the <CR><LF> sequence that ends each line (underscore characters in quoted strings do not count). This results in just a linefeed being presented to the compiler. The linefeed, <LF>, is the line continuation character understood by the compiler and the interpreter. The ASCII key code for a linefeed is Control-J.

CHAPTER 5

DEBUGGING WITH THE BASIC INTERPRETER

You should use BASIC-80 to interpret your BASIC source, and thus to check for syntax and program logic errors. Note that debugging with BASIC-80 is an optional step.

It is possible that you do not have the Microsoft BASIC-80 interpreter, and only own the BASIC Compiler. If this is the case, you must edit your program with any general purpose text editor and check for any errors at compiletime. We strongly urge you to complement the compiler with the BASIC-80 interpreter because the combination of the two gives you an extremely powerful BASIC programming environment.

You may use some commands or functions in your compiled program that execute differently with the interpreter. In those cases, you need to use the compiler for debugging. Note that %INCLUDE is the only statement supported by the compiler that is not supported in some form by the BASIC-80 interpreter. Also, the interpreter does not support double precision transcendental functions as does the BASIC Compiler.

Nevertheless, the language supported by the compiler is intended to be as similar to BASIC-80 as possible. This allows you to make BASIC-80 your prime debugging tool, and to save you debugging time by avoiding lengthy compilations and links. Also, the RUN, CONT, and TRON/TROFF statements make BASIC-80 a very powerful interactive debugging tool. See your BASIC-80 Reference Manual for more information on these statements.

Note that the interpreter stops execution of a program when an error is encountered. Any subsequent errors are not caught until the first detected error is corrected and the program re-RUN. This differs from the compiler where all lines are scanned and all detected errors are reported at compiletime.

CHAPTER 6

COMPILING

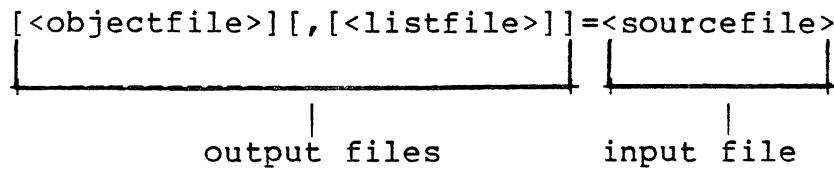
After creating a BASIC source program that you have debugged with the interpreter, your next step is compilation. This chapter covers:

1. Compiler command line syntax,
2. Sample compiler invocations, and
3. Compiler switches.

6.1 COMMAND LINE SYNTAX

Unlike the BASIC-80 interpreter, the compiler is not interactive. It accepts only a single command line containing filenames and extensions, appropriate punctuation, optional device designations, and switches. The placement of these elements when entering the command line determines which processes the compiler performs. To allow users of single-drive system configurations to use the compiler, the command line can be separated into two command lines if desired: one to invoke the compiler and the other to specify compilation parameters.

The general format for the BASIC Compiler command line is:



<objectfile> Specifies the name of the relocatable (.REL) object file,

<listfile> Specifies the name of the listing (.PRN) file,

<sourcefile> Specifies the name of the BASIC (.BAS) source file.

When filenames are entered as parameters, the compiler reads them according to the syntax described above, and assigns them to the appropriate input and output parameters.

Note that the above syntax is concise and accurate, but can be fairly cryptic. We will clear up questions in the following paragraphs, by examining several sample compiler invocations.

6.2 SAMPLE COMPILER INVOCATIONS

You can specify on the compiler command line, creation of four possible combinations of files. These are listed below:

1. A .REL (relocatable object) file only.
2. A .PRN (listing) file only.
3. Both a .REL and a .PRN file.
4. Neither a .REL file nor a .PRN file.

Sample compiler invocations are given below for these combinations of file productions.

1. How to Generate both Object and Listing Files

To generate both object and listing files, invoke the compiler as shown below:

```
BASCOM <objectfile>,<listfile>=<sourcefile>
```

The <objectfile> and <listfile> parameters default to the currently logged drive. You may prefix the file specifications for these parameters with optional device designations.

At the end of your compilation, the following message is displayed:

```
<number-of-errors> FATAL ERROR(S)  
<free-bytes> BYTES FREE
```

2. How to Generate an Object (.REL) File Only

The simplest way to create only a .REL file is to invoke the compiler as shown below:

```
BASCOM =<sourcefile>
```

The above example creates an <objectfile> (not explicitly specified) on the same disk as that containing the <sourcefile>. The <objectfile> will have the same base name as your <sourcefile>. For example, if your <sourcefile> is named A:PROG.BAS, then the <objectfile> will be created with the name A:PROG.REL. Another way to generate only an <objectfile> is to enter:

```
BASCOM <objectfile>=<sourcefile>
```

In this invocation, <objectfile> defaults to the disk in the currently logged drive. This may or may not be the disk on which <sourcefile> resides. An optional device designation may also be given to either <objectfile> or <sourcefile>.

3. How to Generate a Listing (.PRN) File Only

To create only a listing file, invoke the BASIC Compiler as follows:

```
BASCOM ,<listfile>=<sourcefile>
```

The generated <listfile> contains a line-by-line listing of the BASIC source. Also, the object code generated for each BASIC statement is disassembled and listed along with the corresponding BASIC statements in your program. If you use the /N compiler switch described later in this section, listing of the object code is suppressed. Note that the actual .REL file is not in a human-readable form.

As an alternative, you may have the listing file printed out on a line printer. There are two ways to do this. The first way is to enter Control-P (to turn on the printer), then enter TYPE DEMO.PRN. The listing file is simultaneously printed on the line printer and displayed on your screen. When the file has been printed, enter Control-P again (to turn off the printer).

Another way to print out a listing file is to enter the command line once again, but this time with the name of the line printer device (LST:) in place of the listing filename:

```
BASCOM ,LST:=<sourcefile>
```

The second method is the faster of the two since it does not require the creation of a disk file.

When you examine your listing, notice the two hexadecimal numbers preceding each line of the source program. The first number is the relative address of the code associated with that line, using the start of the program as 0. The second number is the cumulative data area needed so far during the compilation. These two columns are totaled at the end of the listing. The left column total is the actual size of the generated .REL file in bytes. The right column total is the total data area required in bytes.

4. How to Suppress Generation of Any Output Files

To perform a syntax check of your <sourcefile>, and to suppress generation of either an <objectfile> or a <listfile>, invoke the compiler as follows:

```
BASCOM ,=<sourcefile>
```

In the above example, the compiler simply compiles the source program and reports the number of errors and the number of free bytes. This is the fastest way to perform a syntax check of your program with the compiler. RUNning a program with the interpreter allows you to perform an accurate syntax check only insofar as the language of the BASIC-80 supports the same language as the BASIC Compiler.

You may want to create output files on a disk other than the defaults provided by the compiler, or you may want to create output files with different extensions or base names than that of your BASIC source file. To do so, you must actually specify the filenames with the desired extensions or device designations, as described below:

Filename Extensions

You may append up to three-characters to filenames as filename extensions. These extensions may contain any alphanumeric character, given in any position in the extension. Lowercase letters are converted to uppercase. Extensions must be preceded by a period (.).

Keep in mind that the BASIC Compiler and L80 recognize certain extensions by default. If you give your filenames unique extensions, you must always remember to include the extension as part of the filename for any filename parameter.

When filename extensions are omitted, default extensions are assumed.

The relevant default filename extensions under CP/M are:

<u>EXTENSION</u>	<u>TYPE OF FILE</u>
.BAS	BASIC source file
.REL	Relocatable object file
.COM	Executable object file
.PRN	Listing file
.MAC	MACRO-80 source file

Device Designations

Each command line field may include device designations that instruct the compiler where to find files or where to output files.

The device designation is placed in front of a filename. For example:

B:DEMO

A device designation may be up to three alphanumeric characters. Note also that the device name must always include the colon (:).

For the input file, (the <sourcefile>), the device designation indicates from which device the file is read. For output files (<objectfile>, <listfile>), the device designation indicates where the files are written.

Device names supported under CP/M are:

DESIGNATIONS	DEVICES
A:, B:, C:, etc.	Disk Drives
LST:	Line Printer
TTY:	CRT (or Teletype)

When device names are omitted, the command scanner defaults to the currently logged disk drive. The only exception to this occurs if a drive is specified as the device for <sourcefile>, but no filenames are specified for <objectfile> or <listfile>. In this case, the compiler writes the output files to the drive specified for the <sourcefile>.

Take for example, the following command line:

BASCOM =B:DEMO

This command line directs the compiler to write the object file to the disk in drive B:, regardless of the location of the currently logged drive.

In all other cases, the default device is the currently logged drive. This may, or may not be the disk on which the compiler resides.

For instance, in the following examples, if A: is the currently logged drive, then the output files are written to drive A::.

```
BASCOM DEMO,DEMO=B:DEMO
BASCOM ,DEMO=B:DEMO
```

When the compiler has finished compilation, it exits to C/PM and the currently logged drive.

Device Names as Filenames

Giving a device name in place of a filename is a command line option. The result of this option depends on which device you specify, and for which command line parameter. Figure 6.1 illustrates the possibilities:

DEVICE	<objectfile>	<listfile>	<sourcefile>
A:, B:, C:, D:	writes file to drive specified	writes file to drive specified	N/A
LST:	N/A (unreadable file format)	writes listing to line printer	N/A (output only)
TTY:	N/A (unreadable file format)	"writes" listing to CRT	Reads statements from keyboard

N/A = Not Allowed

Figure 6.1 Effects of Using Device Designations in Place of File Names

Of special interest is the interactive ability you gain by using the ,TTY:=TTY: command line. In this mode, you can type single BASIC statements at your terminal to check them individually for syntax errors. No disk files are created or read.

6.3 COMPILER SWITCHES

In addition to specifying filenames, extensions, and devices to direct the compiler to produce object and listing files, you can direct BASCOM to perform additional or alternate functions by adding switches to the command line.

Switches may be placed after source file names or after other switches, as in the following command line:

```
BASCOM FOO,FOO=FOO/T/4/X
```

Switches signal special instructions to be used during compilation. The switch tells the compiler to "switch on" a special function or to alter a normal compiler function. More than one switch may be used, but all must begin with a slash (/). Do not confuse these switches with the linker switches.

Compiler switches fall into one of three categories:

1. Conventions
2. Error Trapping
3. Special Code

Conventions

The BASIC Compiler allows you to specify which of two lexical and execution conventions you want applied during compilation: version 4.51 or version 5.0. You need to use the lexical convention switches only if you have older programs that you are trying to convert to version 5.0 BASIC conventions. You specify which conventions you want with either or both of the switches /4 and /T.

Error Trapping

If your BASIC source program contains error trapping routines that involve the ON ERROR GOTO statement plus some form of a RESUME statement, you need to use one of the two error trapping switches, /E and /X. Error trapping routines require line numbers in the binary (.REL) file. If you do not use one of the error trapping switches, the compiler does not place line numbers in the binary file, and a fatal compiler error will result.

Special Code

The BASIC Compiler can generate special code for special uses or situations. Be aware that some of these special code switches cause BASIC Compiler to generate larger and slower code. Examples of special code switches are /D, /S, and /O.

Let's go over the compiler switches by category. First, we'll give you a chart that summarizes the function of each switch. Following that, you'll find detailed descriptions of each switch.

Table 6.1 Compiler Switches

CATEGORY	SWITCH	ACTION
Conventions	/4	Use Microsoft 4.51 lexical conventions (not allowed together with /C).
	/T	Use 4.51 execution conventions.
	/C	Relax line numbering constraints (Not allowed together with /4). *Use /4/T together for 4.51 lexical and execution conventions.
Error Trapping	/E	Program has ON ERROR GOTO with RESUME <line number>.
	/X	Program has ON ERROR GOTO with RESUME, RESUME 0, or RESUME NEXT.
Special Code	/Z	Use Z80 opcodes.
	/N	Suppress listing of disassembled object code in the listing file.
	/O	Substitute the OBSLIB.REL runtime library for BASLIB.REL as the default runtime library searched by the linker after a linker /E or /G switch.
	/D	Generate debug code for runtime error checking.
	/S	Write quoted strings to .REL file on disk and not to data area in RAM.

Each of the switches shown in table 6.1 is explained in detail in the following pages.

CONVENTIONS

The convention switches may be given together (/4/T) to request 4.51 lexical and execution conventions. The individual action of each switch is described below:

Switch Action

- | | |
|----|---|
| /4 | The /4 switch directs the compiler to use the lexical conventions of the Microsoft 4.51 BASIC-80 interpreter. Lexical conventions are the rules that the compiler uses to recognize the BASIC language. |
|----|---|

The following conventions are observed:

1. Spaces are not significant.
2. Variables with embedded reserved words are illegal.
3. Variable names are restricted to two significant characters.

The /4 switch is needed to correctly compile a source program in which spaces do not delimit reserved words, as in the following statement.

FORI=ATOBSTEPC

Without the /4 switch, the compiler would assign the variable "ATOBSTEPC" to the variable "FORI". With the /4 switch set, the compiler recognizes the line as a FOR statement.

We recommend that you edit such programs to 5.0 lexical standards, rather than compile them with the /4 switch. Delimiting reserved words with spaces causes no increase in generated code while greatly improving program readability.

NOTE

The /4 and /C switches may not be used together.

/T The /T switch tells the compiler to use BASIC-80 Version 4.51 execution conventions. Execution conventions refer to the implementation of BASIC functions and commands and what they actually do at runtime. With /T specified, the following 4.51 execution conventions are switched on:

1. FOR/NEXT loops are always executed at least one time.
2. TAB, SPC, POS, and LPOS perform according to 4.51 conventions.
3. Automatic floating point to integer conversions use truncation instead of rounding, except in the case where a floating point number is being converted to an integer in an INPUT statement.
4. The INPUT statement leaves the variables in the input list unchanged if only a carriage return is entered. If a "?Redo from start" message is issued, then a valid input list must be given. A carriage return in this case generates another "?Redo from start" message.

/C The /C switch tells the compiler to relax line numbering constraints. When /C is specified, line numbers in your source file may be in any order, or they may be eliminated entirely.

With /C, lines are compiled normally, but unnumbered lines cannot be targets for GOTOS or GOSUBS. Be aware that while /C is set, the underline character causes the remainder of the physical line to be ignored. Also, /C causes the underline character to act as a line feed so that the next physical line becomes a continuation of the current logical line. (See Chapter 4 for more information on physical and logical lines.)

There are three advantages to using /C:

1. Elimination of line numbers increases program readability.
2. The BASIC Compiler optimizes over entire blocks of code rather than single lines (for example in FOR...NEXT loops.)
3. BASIC source code can more easily be included in a file with %INCLUDE.

Note that /C and /4 may not be used together.

ERROR TRAPPING

The error trapping switches allow you to use ON ERROR GOTO statements in your program. These statements can aid you greatly in debugging your BASIC programs. Note, however, that extra code is generated by the compiler to handle ON ERROR GOTO statements.

Switch	Action
/E	The /E switch tells the compiler that the program contains an ON ERROR GOTO/RESUME <line-number> construction. To handle ON ERROR GOTO properly, the compiler must generate extra code for the GOSUB and RETURN statements. Also a line number address table (one entry per line number) must be included in the binary file, so that each runtime error message includes the number of the line in which the error occurs. To save memory space and execution time, do not use this switch unless your program contains an ON ERROR GOTO statement.

NOTE

If a RESUME statement other than RESUME <line-number> is used with the ON ERROR GOTO statement, use the /X switch instead.

/X The /X switch tells the BASIC Compiler that the program contains one or more RESUME, RESUME NEXT, or RESUME 0 statements.

The /X switch performs all the functions of the /E switch, so the two need never be used at the same time. For instance, the /X switch, like the /E switch, causes a line number address table (one entry per statement) to be included in your binary object file, so that each runtime error message includes the number of the line in which the error occurs. Nevertheless, the /X switch performs additional functions not performed by the /E switch.

Note that to handle RESUME statements properly, the compiler cannot optimize across statements. Therefore, do not use /X unless your program contains RESUME statements other than RESUME <line-number>.

SPECIAL CODE

Switch Action

/Z The /Z switch tells the compiler to use Z80 opcodes whenever possible. When the /Z switch is set, additional Z80 opcodes are allowed, and Z80 mnemonics are used when listing these instructions. All other opcodes are listed using 8080 mnemonics.

/N The /N switch suppresses listing of the disassembled object code for each source line. Instead, you get a simple BASIC source listing plus the relative locations of your code and the size of your accumulated data area. If this switch is not set, the source listing produced by the compiler contains the disassembled object code generated by each statement. Use this switch when you want a shorter listing file, and want to list your BASIC source file along with the code relative locations of your program and the size of your accumulated data area.

/O The /O switch tells the compiler to substitute the OBSLIB.REL runtime library for BASLIB.REL as the default runtime library searched by the linker. When you use this switch you cannot use the BRUN.COM module.

Note that you can create ROM-able code when you link to OBSLIB.REL, something you cannot do if you link to BASLIB.REL. Also, .COM files created by linking to OBSLIB.REL do not need BRUN.COM on disk at runtime.

/D The /D switch causes debugging and error handling code to be generated at runtime. Use of /D allows you to use TRON and TROFF in the compiled file. Without /D set, TRON and TROFF are ignored.

With /D, the BASIC Compiler generates somewhat larger and slower code to perform the following checks:

1. Arithmetic overflow. All arithmetic operations, integer and floating point, are checked for overflow and underflow.
2. Array bounds. All array references are checked to see if the subscripts are within the bounds specified in the DIM statement.
3. Line numbers. The generated binary code includes line numbers so that the runtime error listing can indicate on which line each error occurs.
4. RETURN. Each RETURN statement is checked for a prior GOSUB statement.

Without the /D switch set, array bound errors, RETURN without GOSUB errors, and arithmetic overflow errors do not generate error messages at compile time. At runtime, no error messages are generated either, and erroneous program execution results. Use the /D switch to make sure that you have thoroughly debugged your program.

/S The /S switch forces the compiler to write quoted strings greater than 4 characters in length to your .REL file on disk as they are encountered, rather than retaining them in memory during the compilation of your program. If this switch is not set, and your program contains a large number of long quoted strings, you may run out of memory at compiletime.

Although the /S switch allows programs with many quoted strings to take up less memory at compiletime, it may increase the amount of memory needed in the runtime environment, since multiple instances of identical strings will exist in your program. Without /S, references to multiple identical strings are combined so that only one instance of the string is necessary in your final compiled program.

CHAPTER 7

LINKING

To load and link a compiled program, use the Microsoft LINK-80 Linking Loader. Refer to the LINK-80 section of the Utility Software Manual for information on how to use the linker, before you read this chapter. This chapter supplements the Utility Software Manual, by providing:

1. Sample linker sessions,
2. A discussion of linking compiled BASIC programs, and
3. A discussion of the BASIC runtime support environment.

We begin with some sample linker sessions.

7.1 SAMPLE LINKER SESSIONS

A simple link might look like this on your screen:

```
>L80  
*PROG.COM/N,PROG.REL/E
```

The caret (>) is the CP/M prompt; the asterisk (*) is the linker prompt. Note that linker switches have no relation whatsoever to the compiler switches discussed in the preceding chapter.

If you use default extensions, a link session might look like this:

```
>L80  
*PROG/N,PROG/E
```

The L80 invocation line can also be used for specifying linker parameters. So, the following command would perform the same functions as the preceding example:

```
>L80 PROG/N,PROG/E
```

In any of the above cases, the /E switch tells the linker to exit to CP/M and store a .COM file on disk. Before exiting, the linker automatically searches BASLIB.REL on the currently logged drive for any as yet undefined global references. The final linked .COM file has the name specified by your last <filename>/N command. The /N switch is essential if you want to create a .COM file.

The /G switch is similar to the /E switch. The only difference between the two is that the /G switch causes execution of the .COM file after it is stored on disk. In either case, you must specify the name of the file to store on disk. If you do not, no .COM file is stored.

If you choose to link an assembly language routine to your BASIC program, a sample linker invocation might look like this:

```
>L80  
*PROG,MYASM,PROG/N/E
```

In the above case, MYASM.REL is the name of the assembly language routine and PROG.REL is the name of your program. The routine MYASM.REL cannot be assembled with an END <label> statement since the linker will assume that <label> is the start address of a separate program. The linker will refuse to link two programs together since their two separate start addresses will conflict.

When you link a BASIC .REL file to BASLIB.REL, the BCLOAD file must be on disk in the currently logged drive. If it is not, the following error message is generated:

```
?BCLOAD not found, please create header file
```

More information about BCLOAD can be found later in this chapter.

When your linking session is complete, the following message is generated:

```
DATA <program-start> <program-end> <bytes>
<free-bytes> BYTES FREE
[<start-address> <program-end> <num-of-pages>]
```

The values displayed provide the information shown in Figure 7.1 for a program linked to BASLIB.REL and using BRUN.COM. If you link to OBSLIB.REL and use the /P and /D linker switches, some of this information is not accurate.

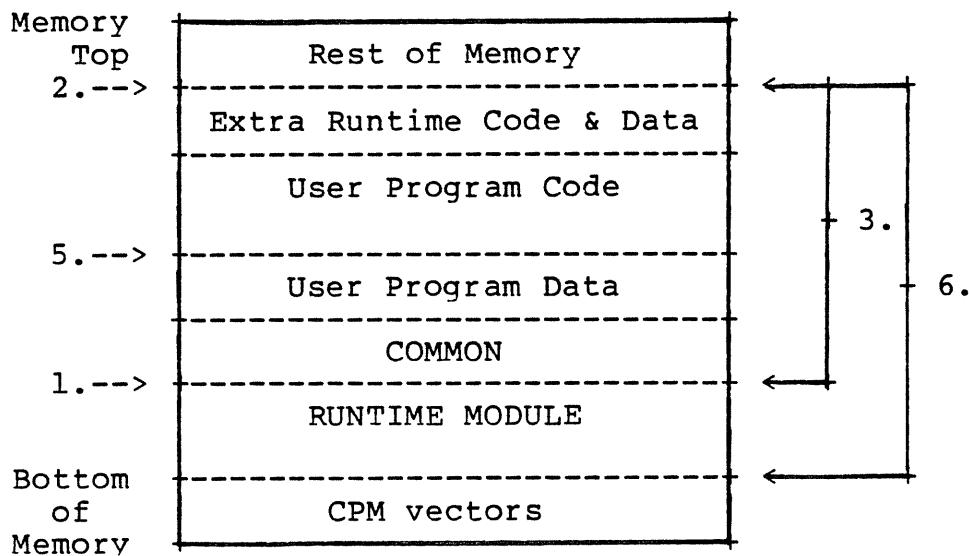


Figure 7.1 Link Data Map

1. <program-start> - Hexadecimal address of the beginning of your program.
2. <program-end> - Hexadecimal address of the end of your program.
3. <bytes> - Decimal size of program in bytes.
4. <free-bytes> - Decimal size of unused memory in bytes during linking.
5. <start-address> - Hexadecimal start address of your program (not necessarily the same as <program-start>).
6. <num-of-pages> - Decimal number of 256-byte pages used by program.

For programs linked to BASLIB.REL and using the BRUN.COM runtime module, the size of your .COM file in bytes is equal to:

```
<program-end> - <start-address> + 128
```

At runtime, remember that BRUN.COM also resides in memory along with your program. The 128 bytes in the above equation is for a small relocator routine that begins every .COM file. When you invoke a program, this relocator routine is the first routine executed. All it does is move the rest of your .COM file to the start address shown above. Execution of your program then begins. The first thing your program does is load the runtime module to establish the runtime support environment.

We now discuss linking to compiled BASIC .REL files.

7.2 LINKING TO COMPILED BASIC .REL FILES

Because of the way the BASIC runtime environment is implemented with the BRUN.COM runtime module, there are a number of peculiarities that you must account for at linktime.

First of all, before you can link any BASIC .REL file, you must have the file BCLOAD on the currently logged disk. BCLOAD contains two pieces of information: the hexadecimal load address of your program, and the drive in which to find BRUN.COM at runtime.

BCLOAD looks like this if you TYPE it out:

```
+4000      [Program Load Address]
:          [A:, B:, C:, etc., or : for default]
```

At runtime, you must have BRUN.COM on the disk specified in BCLOAD or an error is generated. Note that the plus sign (+) is necessary to tell the linker to write the .COM file beginning at the start address of your program instead of the program load address. (The start address is the address at which your program begins execution.) The default location of the BRUN.COM runtime module is the currently logged drive. You can alter BCLOAD, before linktime, to specify the disk on which you want BRUN.COM to reside at runtime.

There are two other peculiarities associated with linking programs that require the BRUN.COM runtime module. Namely, these linking procedures may not work:

L80 FOO/G

L80 FOO/E followed by SAVE xxx FOO.COM

L80 FOO/G may not work if BRUN.COM does not reside on the disk you have specified in BCLOAD. CHAINing of programs does not work properly if you use SAVE after a link.

We now move to a discussion of the runtime support that is linked to your program.

7.3 RUNTIME SUPPORT

Once you have compiled a .REL file, you need to link your program to modules that contain runtime support routines. Runtime support is the body of routines that, in essence, implement the BASIC language. Your compiled .REL file, on the other hand, implements the particular algorithm that makes your program a unique BASIC program.

Runtime support is essential to the execution of all compiled BASIC programs. It is found in BRUN.COM and the runtime library. As a rule, only a portion of all possible runtime routines is linked to your .REL file. The length of time necessary to link all these needed runtime support routines is often a problem on microcomputers.

Partly for this reason, the BRUN.COM runtime module contains all of the more frequently used routines in one module. Since they all reside in one module, they are linked all at once, and need not be searched for in later linker searches. Note that the BRUN.COM module is automatically linked to every program via a dummy module in BASLIB.REL: it is not present in memory at linktime. Thus, a minimal program at runtime is at least 16K long. If your program uses other less frequently used routines, these routines are searched for and found in BASLIB.REL. At linktime, you cannot use the /P and /D linker switches, since they will cause errors at program runtime.

When you specify the /O switch at runtime, the alternate runtime library (OBSDLIB.REL) is substituted for BASLIB.REL as the default library to be searched at linktime. At linktime you can then use /P and /D as described in the Utility Software Manual. Note that when OBSDLIB.REL is selected as the library to be searched, BRUN.COM is not used by your program at all.

There are several advantages to using OBSLIB.REL:

1. Programs not using BRUN.COM can be put in ROM, since separate instruction and data areas can be created when linking to routines in OBSLIB.REL with the /P and /D switches.
2. For small and simple programs, you may be able to compile and link smaller programs than the 16K minimum required to accommodate the BRUN.COM module. This can be of importance in compiling a program for a ROM-based application, where space can be a critical factor.
3. Execution of a compiled and linked .COM file does not require the existence of BRUN.COM on disk at runtime.

There are, however, some distinct disadvantages to using OBSLIB.REL:

1. COMMON is not supported between programs.
2. The CHAIN command is semantically equivalent to the RUN command.
3. COMMON and CHAIN commands cannot be used to support a system of programs sharing common data. (See 1. and 2. above.)
4. The CLEAR command is not implemented.
5. The RUN <linenumber> option to RUN is not implemented.
6. The linker cannot load programs as large as those that use the BRUN.COM module.
7. All required runtime support functions are included in every .COM file generated, thus increasing the size of each of your .COM files. This is not the case for .COM files using the BRUN.COM runtime module.

For more information on using CHAIN and COMMON with a system of programs, see Appendix A. For more information on ROM-able code, see Appendix B.

CHAPTER 8

RUNNING A COMPILED PROGRAM

To run a compiled program, simply enter the filename without its .COM filename extension. For example:

```
DEMO
```

The above command causes execution of the program DEMO.COM. At runtime, BRUN.COM must be accessible from disk. BRUN.COM is loaded from the disk in the drive you specify in BCLOAD at linktime.

Programs can also be executed immediately after linking is complete by using the /G linker switch. This works only if BRUN.COM is on the disk you have selected in BCLOAD.

The executable binary file can also be executed from within a program, as in the following statement:

```
10 RUN "PROG"
```

The default extension is .COM. The .COM file can be a program created in any programming language. The CHAIN command is used in a similar fashion. In either case, an executable binary file is loaded. The BRUN.COM runtime module is not reloaded when you use CHAIN; it is when you use RUN.

It is important to realize that the bulk of the runtime environment is taken up by the BRUN.COM runtime module. This module is automatically loaded when you initially invoke an executable .COM file requiring BRUN.COM. When you RUN a program, the .COM file is loaded into memory and BRUN.COM is also loaded to create a fresh runtime environment. Both files reside in memory simultaneously.

CHAPTER 9

A COMPILER/INTERPRETER COMPARISON

There are differences between the languages supported by the BASIC Compiler and the BASIC-80 interpreter that must be taken into account when compiling existing or new BASIC programs. This is why we strongly recommend that you compile the demonstration program in Chapter 3 first; read Chapters 4-8; and only then begin compiling other programs.

The differences between the languages supported by the BASIC Compiler and the BASIC interpreter fall into three categories: operational differences, language differences, and other differences. The tables on the next page serve as a reference guide to these differences. All commands and functions except %INCLUDE are described in the BASIC-80 Reference Manual. Where differences exist, those commands and functions are also discussed in the following paragraphs.

9.1 OPERATIONAL DIFFERENCES

Those BASIC-80 commands used to operate in the BASIC-80 programming environment are not acceptable input to the compiler. These include the following:

AUTO	CLOAD	CSAVE	CONT	DELETE
EDIT	LIST	LLIST	LOAD	MERGE
NEW	RENUM	SAVE		

9.2 LANGUAGE DIFFERENCES

Most programs that run under the BASIC-80 interpreter will compile under the BASIC Compiler with little or no change. However, it is necessary to note differences in the following commands:

CALL	%INCLUDE
CHAIN	ON ERROR GOTO
CLEAR	REM
COMMON	RESUME
DEFxxx	RUN
DIM	STOP
END	TRON/TROFF
ERASE	USRn
FOR/NEXT	WHILE/WEND

These differences are described below:

1. CALL

The CALL statement allows you to call and transfer program control to a precompiled FORTRAN-80 subroutine, or to an assembly language routine that you have created with MACRO-80. The format of the CALL Statement is:

```
CALL <variable-name> [<argument-list>...]
```

The <variable-name> parameter is the name of the subroutine that you wish to call. This name must be 1 to 6 characters long and must be recognized by LINK-80 as a global symbol. That is, <variable-name> must be the name of the subroutine in a FORTRAN SUBROUTINE statement, or a PUBLIC symbol in an assembly language routine. Refer to the MACRO-80 Reference Manual and the FORTRAN-80 Reference Manual for definitions of these terms. (See NOTE below.)

The <argument-list> parameter is optional. It contains arguments that are passed to an assembly language or FORTRAN subroutine.

Example: 120 CALL MYSUBR (I,J,K)

NOTE

FORTRAN-80 is a separate product available from Microsoft and is not part of the BASIC Compiler package. If you do not have FORTRAN-80, then the CALL statement can only be used with assembly language subroutines.

Further information on assembly language subroutines is contained in the discussion of the USR function that follows in this chapter. Also, more information is provided on creating and interfacing assembly language routines in the Utility Software Manual.

2. CHAIN

The BASIC Compiler does not support the ALL, MERGE, DELETE, and <line number> options to CHAIN. If you wish to pass variables, it is recommended that the COMMON statement be used. Note that files are left open during CHAINing.

3. CLEAR

The BASIC Compiler supports the CLEAR command as described in the BASIC-80 Reference Manual, with the restriction that <expression1> and <expression2> must be integer expressions. If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 256 bytes and the default top of memory is the current top of memory. The CLEAR statement performs the following actions:

Closes all files
Clears all COMMON and user variables
Resets the stack and string space
Releases all disk buffers

See Appendix C for a memory map showing the location of the stack, string space, and disk buffers discussed above.

Note that CLEAR is supported only for programs using the BRUN.COM module, and not for programs linked to the OSLIB.REL runtime library.

4. COMMON

The BASIC Compiler supports a modified version of the COMMON statement. The COMMON statement must appear in a program before any executable statements. A list of non-executable statements follows:

COMMON
DEFDBL, DEFINT, DEFSNG, DEFSTR
DIM
OPTION BASE
REM
%INCLUDE

All other statements are executable. Arrays in COMMON must be declared in preceding DIM statements.

The standard form of the COMMON statement is referred to as blank COMMON. FORTRAN-style named COMMON areas are also supported; however, the named COMMON variables are not preserved across CHAINS.

The syntax for named COMMON is as follows:

COMMON /<name>/ <list of variables>

The parameter <name> is 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with FORTRAN and assembly

language routines without having to explicitly pass parameters in the CALL statement.

IMPORTANT

For blank COMMON statements communicating between CHAINing and CHAINED-to programs, both the size of the COMMON area, and the order of variables must be the same.

To ensure that COMMON areas can be shared between programs, place blank COMMON declarations in a single INCLUDE file and use the %INCLUDE statement in each program. For example:

```
MENU.BAS
10 %INCLUDE COMDEF
.
.
.
1000 CHAIN "PROG1"
```

```
PROG1.BAS
10 %INCLUDE COMDEF
.
.
.
2000 CHAIN "MENU"
```

```
COMDEF.BAS
100 DIM A(100),B$(200)
110 COMMON I,J,K,A()
120 COMMON A$,B$(),X,Y,Z
```

5. DEFINT/SNG/DBL/STR

DEFxxx statements designate the storage class and data type of variables listed as parameters. The compiler does not "execute" DEFxxx statements as it does a PRINT statement, for example.

Instead, the compiler allocates memory for storage of designated variables, and assigns them one of the following data types:

1. INTeger,
2. SiNGle precision floating point,
3. DouBLe precision floating point, or
4. STRing.

A DEFxxx statement takes effect as soon as it is encountered in your program during compilation. Once the type has been defined for the listed variables, that type remains in effect either until the end of the program or until another DEFxxx statement alters the type of the variable. Unlike the interpreter, the compiler cannot circumvent the DEFxxx statement by directing flow of control around it with a GOTO. For variables given with a precision designator (i.e., %, !, #, as in A%=B), the type is not affected by the DEFxxx statement.

6. DIM

The DIM statement is similar to the DEFxxx statement in that it is scanned rather than executed. That is, DIM takes effect when it is encountered at compiletime and remains in effect until the end of the program: it cannot be re-executed at runtime. If the default dimension (10) has already been established for an array variable, and that variable is later encountered in a DIM statement, an "Array Already Dimensioned" error results. Therefore, the practice of putting a collection of DIM statements in a subroutine at the end of your program generates fatal errors. In that case, the compiler sees the DIM statement only after it has already assigned the default dimension to arrays declared earlier in the program.

Also note that the values of the subscripts in a DIM statement must be integer constants; they may not be variables, arithmetic expressions, or floating point values. For example, each of the following DIM statements is illegal:

```
DIM A1(I)
DIM A1(3+4)
DIM A1(3.4E5)
```

7. END

During execution of a compiled program, an END statement closes files and returns control to the operating system. The compiler assumes an END statement at the end of the program, so "running off the end" (omitting an END statement at the end of the program) produces proper program termination by default.

8. ERASE

The ERASE statement is not implemented for the compiler. ERASE in BASIC-80 allows you to re-dimension arrays, something that is not done in the compiled environment.

9. FOR/NEXT

Double precision FOR/NEXT loops can be used with the compiler. Also, FOR/NEXT loops must be statically nested. Static nesting means that each FOR must have a single corresponding NEXT.

Static nesting also means that each FOR/NEXT pair must reside within an outer FOR/NEXT pair. Therefore, the following construction is not allowed:

```
FOR I
  |
  FOR J
    |
    FOR K
      |
      NEXT J
        |
        NEXT K
      |
      NEXT I
```

This construction is the correct form:

```
FOR I
  |
  FOR J
    |
    FOR K
      |
      NEXT K
    |
    NEXT J
  |
  NEXT I
```

Also, you should not direct program flow into a FOR/NEXT loop with a GOTO statement. The result of such a jump is undefined, as in the following example:

```
50 GOTO 100
:
:
90 FOR I = 1 to 10
:
100 PRINT "INLOOP"
:
200 NEXT I
```

10. %INCLUDE

The format of the %INCLUDE compiler directive is:

```
%INCLUDE <filename>
```

%INCLUDE allows the compiler to include source code from an alternate BASIC file. These BASIC source files may be subroutines, single lines, or any type of partial program. No assembly language or FORTRAN files are allowed as arguments to the %INCLUDE statement. Note that <filename> does not require quotes and that the default extension is .BAS.

The programmer should take care that any variables in the included files match their counterparts in the main program, and that included lines do not contain GOTOS to non-existent lines, END statements, or similarly erroneous code.

These further restrictions must be observed:

(a.) The INCLUDED file must be SAVED with the ,A option if created from within BASIC-80.

(b.) The INCLUDED lines must be in ascending order.

(c.) The lowest line number of the included lines must be higher than the line number of the INCLUDE statement in the main program.

(d.) The range of line numbers in the INCLUDED file must numerically precede subsequent line numbers in the main program. These restrictions are removed if the main program is compiled with the /C switch set, since line numbers need not be in ascending order in this case. For more information, see Section 6.3, Compiler Switches.

(e.) %INCLUDE directives cannot be nested inside INCLUDE files. This means that %INCLUDE can only be used in the file containing your main BASIC program.

(f.) The %INCLUDE directive must be the last statement on a line, as in the following statement:

```
999 DEFINT I-N : %INCLUDE COMMON.BAS
```

11. ON ERROR GOTO

If a program contains ON ERROR GOTO and RESUME <line number> statements, the /E compilation switch must be given in the compiler command line. If the RESUME NEXT, RESUME, or RESUME 0 form is used, the /X switch must be used instead.

The basic function of these switches is to allow the compiler to function correctly when error trapping routines are included in a program. See Section 6.3, Compiler Switches, for a detailed explanation of these switches. Note, however, that the use of these switches increases the size of the .REL and .COM files.

12. REM

REM statements are REMarks starting with a single quotation mark or the word REM. Since REM statements do not take up time or space during execution, REM may be used as freely as desired. This practice is encouraged for improving the readability of your programs.

13. RESUME

See the preceding discussion of ON ERROR GOTO.

14. RUN

The compiler supports both the RUN and RUN <line number> forms of the RUN statement. The BASIC Compiler does not support the "R" option with RUN. If this feature is desired, the CHAIN statement should be used. Note that RUN is used to execute .COM files created by the BASIC Compiler, and does not support the execution of BASIC source files as does the interpreter.

Other .COM files not created with the BASIC Compiler are executable with the RUN statement. These can be .COM files created in other languages besides BASIC.

15. STOP

The STOP statement is identical to the END statement, except that it terminates your program at a point that is not necessarily its end. It also prints a message telling you at which hexadecimal address you have stopped. If the /D, /E, or /X compiler switches are turned on, then the message prints the line number at which you have stopped. As with the END statement, STOP closes all open files and returns control to the operating system. STOP is normally used for debugging purposes.

16. TRON/TROFF

In order to use TRON/TROFF, the compiler /D Debug switch must be switched on. Otherwise, TRON and TROFF are ignored and a warning message is generated.

17. USRn Functions

Although the USRn function is implemented in the compiler to call machine language subroutines, there is no way to pass parameters, except through the use of POKEs to protected memory locations that are later accessed by the machine language routine.

When the compiler sees X = USRn (0), it generates the following code:

```
CALL      $U%+const  
SHLD      X%
```

If you have compiled the program with the /Z switch on, then the compiler generates instead similar Z80 code:

```
CALL      $U%+const  
LD       (X%),HL
```

During execution, the program encounters this code, jumps to the address of the CALL, performs the steps of your subroutine and returns. Your routine should place the integer result of the routine in the H,L register pair prior to returning to the compiled BASIC program. On return, as shown above, the contents of the H,L register pair are placed in the location of the variable X. Any other parameters to be passed must be PEEKed from the main BASIC program, and POKEd into protected memory locations. With this method of passing parameters, the USRn function is quite usable. You must take responsibility, though, to ensure that your code and any variables you use are protected.

If you do not want to use the above method of passing parameters, you have two other choices:

1. If your machine language routine is short enough, you can store it by making the first string defined in the program contain the ASCII values corresponding to the hexadecimal values of your routine. Use the CHR\$ function to insert ASCII values in the string. You can then find the start of your routine by using the VARPTR function. For example, for the string A\$, VARPTR(A\$) will return the address of the length of the string. The next two addresses are (first) the least significant byte and (then) the most significant byte of the actual address of the string. This set-up of the string space for the compiler differs from the set-up for the interpreter in this respect. Thus, to find the actual start address of your routine, you would use the following BASIC instructions:

```
A$ = "String containing routine"  
I% = VARPTR(A$)  
AD% = PEEK(I% + 2) * 256 + PEEK(I% + 1)  
AD% is the start address of your routine.
```

Note that strings move around in the string space, so any absolute references must be adjusted to reflect the current memory location of the routine. To make your code position independent for the Z80, you should use relative, rather than absolute jumps.

2. The second method is to reset the default value of the load address in the BCLOAD file. The BCLOAD file's main purpose is to direct loading of your executable program in memory after BRUN.COM has been loaded. By increasing the load address by 100H, for example, 256 bytes of free protected space are created between the end of BRUN.COM and the start of the loading area. Machine language routines or data can then be safely POKEd into this area.

A better alternative is to use MACRO-80 to assemble your subroutines. Then, your subroutines can be linked directly to the compiled program and referenced using the CALL statement.

18. WHILE/WEND

WHILE/WEND constructions should be statically nested. Static nesting means that each WHILE/WEND pair, when nested within other FOR/NEXT or WHILE/WEND pairs, cannot reside partly in, and partly outside, the nesting pair. For example, the following construction is not allowed:

```
FOR I = 1 to 10
  A = COUNT
  WHILE A = 1
    NEXT I
    A = A - 1
  WEND
```

You should also not direct program flow into a WHILE/WEND loop without entering through the WHILE statement. See FOR/NEXT, above, for an example of this restriction.

9.3 OTHER DIFFERENCES

Other differences between BASIC-80 and the BASIC Compiler include the following:

1. Expression Evaluation - The BASIC Compiler performs optimizations, if possible, when evaluating expressions.
2. Use of Integer Variables - The BASIC Compiler can make optimum use of integer variables as loop control variables. This allows some functions (and programs) to execute up to 30 times faster than when interpreted.
3. Double Precision Arithmetic Functions - The BASIC Compiler implements double precision arithmetic functions, including all of the transcendental functions.
4. String Space Implementation - To increase the speed of garbage collection, the implementation of the string space for the compiler differs from its implementation for the interpreter.

EXPRESSION EVALUATION

During expression evaluation, the BASIC Compiler converts operands of different types to the type of the more precise operand.

```
QR=J%+A!+Q#
```

The above expression causes J% to be converted to single precision and added to A!. This double precision result is added to Q#.

The BASIC Compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter, the following statements yield 10000 for M%.

```
I%=20000  
J%=20000  
K%=-30000  
M%=I%+J%-K%
```

That is, J% is added to I%. Because the number is too large, it converts the result into a floating point number. K% is then converted to a floating point number and subtracted. The result, 10000, is found, and converted back to an integer and saved as M%.

The BASIC Compiler, however, must make type conversion decisions during compilation. It cannot defer until actual values are known. Thus, the compiler generates code to perform the entire operation in integer mode and arithmetic overflow occurs. If the /D Debug switch is set, the error is detected. Otherwise, an incorrect answer is produced.

Besides the above type conversion decisions, the compiler performs certain valid optimizing algebraic transformations before generating code. For example, the following program could produce an incorrect result when run:

```
I%=20000  
J%=-18000  
K%=20000  
M%=I%+J%+K%
```

If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I%+K% first and then adds J%, overflow does occur. The compiler follows the rules of operator precedence, and parentheses may be used to direct the order of evaluation. No other guarantee of evaluation order can be made.

INTEGER VARIABLES

To produce the fastest and most compact object code possible, you should make maximum use of integer variables. For example, the following program executes approximately 30 times faster by replacing "I", the loop control variable, with "I%" or by declaring I an integer variable with DEFINT.

```
FOR I=1 TO 10
  A(I)=0
NEXT I
```

Also, it is especially advantageous to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

DOUBLE PRECISION ARITHMETIC FUNCTIONS

The BASIC Compiler allows you to use double precision floating point numbers as operands for arithmetic functions, including all of the transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, SQR). Only single precision arithmetic functions are supported by the interpreter.

Your program development strategy when designing a program with double precision arithmetic functions should be the following:

1. Implement your BASIC program using single precision operands for all functions that you later intend to be double precision.
2. Debug your program with the interpreter to determine the soundness of your algorithm before converting variables to double precision.
3. Declare all desired variables as double precision. Your algorithm should be sound at this point.
4. Compile and link your program. It should implement the algorithm that you have already debugged with the interpreter, now with double the precision in your arithmetic functions.

STRING SPACE IMPLEMENTATION

The compiler and interpreter differ in their implementation and maintenance of the string space. Using PEEK, POKE, VARPTR, or assembly language routines to change string descriptors may result in a String Space Corrupt error. See more information on the string space in the discussion of the USR function earlier in this chapter.

CHAPTER 10

ERROR MESSAGES

During development of a BASIC program with the BASIC Compiler, three different kinds of errors may occur: BASIC Compiler fatal errors, BASIC Compiler warning errors, and BASIC runtime errors. This chapter lists error codes, error numbers, and error messages for each type of error.

10.1 BASIC COMPILETIME ERROR MESSAGES

For errors that occur at compiletime, the compiler outputs the line containing the error, an arrow beneath that line pointing to the place in the line where the error occurred, and a two-character code for the error. In some cases, the compiler reads ahead on a line to determine whether an error has actually occurred. In those cases, the arrow points a few characters beyond the error, or to the end of the line.

The BASIC Completetime errors described below are divided into Fatal Errors and Warning Errors.

FATAL ERRORS

<u>CODE</u>	<u>MESSAGE</u>
BS	Bad Subscript Illegal dimension value Wrong number of subscripts
CD	Duplicate COMMON variable
CN	COMMON array not dimensioned
CO	COMMON out of order
DD	Array Already Dimensioned
FD	Function Already Defined
FN	FOR/NEXT Error FOR loop index variable already in use FOR without NEXT NEXT without FOR
IN	INCLUDE Error %INCLUDE file not found
LL	Line Too Long
LS	String Constant Too Long
OM	Out of Memory Array too big Data memory overflow Too many statement numbers Program memory overflow
OV	Math Overflow
SN	Syntax error - caused by one of the following: Illegal argument name Illegal assignment target Illegal constant format Illegal debug request Illegal DEFxxx character specification Illegal expression syntax Illegal function argument list Illegal function name Illegal function formal parameter Illegal separator Illegal format for statement number Illegal subroutine syntax Invalid character Missing AS Missing equal sign

Missing GOTO or GOSUB
Missing comma
Missing INPUT
Missing line number
Missing left parenthesis
Missing minus sign
Missing operand in expression
Missing right parenthesis
Missing semicolon
Missing slash
Name too long
Expected GOTO or GOSUB
String assignment required
String expression required
String variable required
Illegal syntax
Variable required
Wrong number of arguments
Formal parameters must be unique
Single variable only allowed
Missing TO
Illegal FOR loop index variable
Illegal COMMON name
Missing THEN
Missing BASE
Illegal subroutine name

SQ Sequence Error
Duplicate statement number
Statement out of sequence

TC Too Complex
Expression too complex
Too many arguments in function call
Too many dimensions
Too many variables for LINE INPUT
Too many variables for INPUT

TM Type Mismatch
Data type conflict
Variable must be of same type

UC Unrecognizable Command
Statement unrecognizable
Command not implemented

UF Function Not Defined

WE WHILE/WEND Error
WHILE without WEND
WEND without WHILE

/0 Division by Zero

/E Missing "/E" Switch
/X Missing "/X" Switch

WARNING ERRORS

<u>CODE</u>	<u>MESSAGE</u>
ND	Array not Dimensioned
SI	Statement Ignored Statement ignored Unimplemented command

10.2 BASIC RUNTIME ERROR MESSAGES

The following errors may occur at program runtime. The error numbers match those issued by the BASIC-80 interpreter. The compiler runtime system prints long error messages followed by an address, unless /D, /E, or /X is specified in the compiler command line. In those cases, the error message is also followed by the number of the line in which the error occurred.

<u>NUMBER</u>	<u>MESSAGE</u>
2	Syntax Error A line is encountered that contains an incorrect sequence of characters in a DATA statement.
3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	Out of Data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal Function Call A parameter that is out of range is passed to a math or string function. A function call error may also occur as the result of: A negative or unreasonably large subscript A negative or zero argument with LOG A negative argument to SQR A negative mantissa with a non-integer exponent A call to a USR function for which the starting address has not yet been given An improper argument to ASC, CHR\$, MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING\$, SPACE\$, INSTR, or ON...GOTO A string concatenation that is longer than 255 characters
6	Floating Overflow or Integer Overflow The result of a calculation is too large to be represented within the range allowed for floating point numbers.

- 9 Subscript Out of Range
An array element is referenced with a subscript that is outside the dimensions of the array.
- 11 Division by Zero
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power.
- 14 Out of String Space
String variables exceed the allocated amount of string space.
- 20 RESUME without Error
A RESUME statement is encountered before an error trapping routine is entered.
- 21 Unprintable Error
An error message is not available for the error condition that exists. This is usually caused by an ERROR with an undefined error code.
- 50 Field Overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
- 51 Internal Error
An internal malfunction occurs in the BASIC Compiler. Report to Microsoft the conditions under which the message appeared.
- 52 Bad File Number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
- 53 File Not Found
A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.
- 54 Bad File Mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN with a file mode other than I, O, or R.
- 55 File Already Open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

- 57 Disk I/O Error
An I/O error occurred on a disk I/O operation.
The operating system cannot recover from the error.
- 58 File Already Exists
The filename specified in a NAME statement is identical to a filename already in use on the disk.
- 61 Disk Full
All disk storage space is in use.
- 62 Input Past End
An INPUT statement reads from a null (empty) file, or from a file in which all data has already been read. To avoid this error, use the EOF function to detect the end of file.
- 63 Bad Record Number
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or is equal to 0.
- 64 Bad File Name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).
- 67 Too Many Files
The 255 file directory maximum is exceeded by an attempt to create a new file with SAVE or OPEN.

The following additional runtime error messages are fatal and cannot be trapped:

Internal Error - String Space Corrupt

Internal Error - String Space Corrupt during G.C.

Internal Error - No Line Number

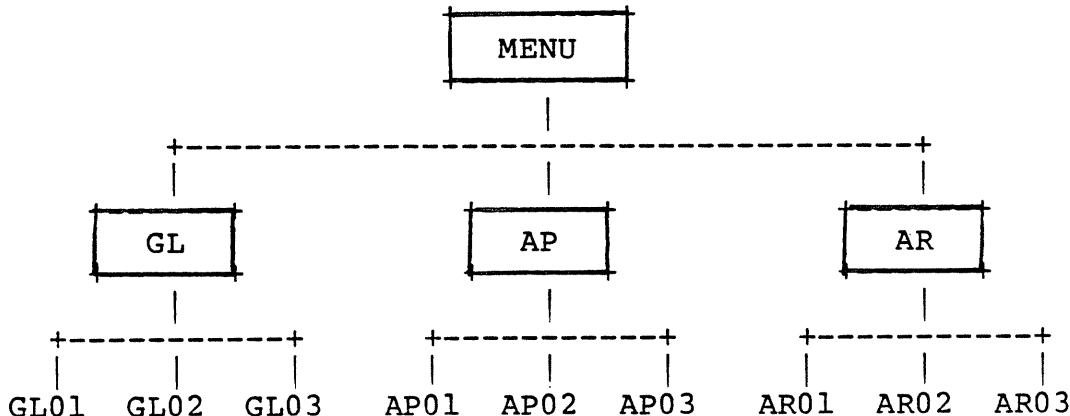
The first two errors usually occur because a string descriptor has been improperly modified. (G.C stands for garbage collection.) The last error occurs when the error address cannot be found in the line number table during error trapping.

APPENDIX A

Creating a System of Programs with the BRUN.COM Runtime Module

The CHAINing with COMMON feature and the BRUN.COM runtime module are designed for creating large systems of BASIC programs that interact with each other. A hypothetical system will be described to show the interactions in a large system design. In particular, the distinction between CHAIN and RUN will be highlighted.

Consider the following integrated accounting system containing three packages for general ledger, accounts payable, and accounts receivable. Entry into each package is controlled by a main menu program. The system structure is shown below:



In order to use CHAINing with COMMON effectively, it is important to logically structure the system and the COMMON information. In the system pictured above, COMMON information exists within each of the packages GL, AP, and AR. Each package contains a system of three separately compiled programs. Furthermore, there may be COMMON information between MENU and each of the packages. Note that there may be overlapping sets of COMMON information. The compiler's COMMON statement is not as flexible as the interpreter's: COMMON areas must be the same size in programs that CHAIN to each other.

Two solutions to this problem of communicating between programs are given below, though others are possible:

1. Use the same COMMON declarations in all programs so that all common information may be shared, or
2. Use the same set of COMMON declarations within each of the three packages with no common information shared via COMMON with the other packages or the main MENU program. In this case, there are three sets of COMMON declarations, one for each package.

For a large integrated set of systems of programs, the second method gives more flexibility with the compiler. Since program control is switched from package to package through the main MENU, there is little loss of flexibility with this method. Any common information that could be obtained in MENU should be obtained in the main program for each of the packages GL, AP, and AR. This is the same approach you would use with a single package.

For the above diagram, the use of CHAIN and RUN in each of the major programs is outlined in the following program fragments:

MENU.BAS

```
1000 IF MENU=1 THEN RUN "GL"
1010 IF MENU=2 THEN RUN "AP"
1020 IF MENU=3 THEN RUN "AR"
```

GL.BAS

<pre>10 %INCLUDE GLCOMDEF 1000 CHAIN "GL01" 1010 CHAIN "GL02" 1020 CHAIN "GL03" 1030 IF MENU=YES THEN RUN "MENU"</pre>	General Ledger (GL) COMMON declarations
--	--

AP.BAS

<pre>10 %INCLUDE APCOMDEF 1000 CHAIN "AP01" 1010 CHAIN "AP02" 1020 CHAIN "AP03" 1030 IF MENU=YES THEN RUN "MENU"</pre>	Accounts Payable (AP) COMMON declarations
--	--

AR.BAS

<pre>10 %INCLUDE ARCOMDEF 1000 CHAIN "AR01" 1010 CHAIN "AR02" 1020 CHAIN "AR03" 1030 IF MENU=YES THEN RUN "MENU"</pre>	Accounts Receivable (AR) COMMON declarations
--	---

Each of the lower level programs XXYY (XX=GL, AP, AR, YY = 01, 02, 03) should CHAIN back to the package main program XX.

The RUN statement in the above programs loads the specified program as a normal .COM file and starts execution. For compiled BASIC programs, a new copy of the BRUN.COM runtime module is reloaded. This allows a new system of CHAINed programs to be started. During CHAINS, the BRUN.COM runtime module is in control, like the BASIC interpreter during interpretation, and BRUN.COM is not reloaded.

APPENDIX B

ROM-able Code

To create a program that can be burned into ROM, you should note the following:

1. Constant data and instructions can go into ROM.
2. Variable data cannot go into ROM.

Therefore, it is necessary that ROM-able code have separate data and instruction areas. You can specify these areas at linktime by using the /D and /P switches (D for Data and P for Program). See the Utility Software Manual for more information on the use of these switches.

Unfortunately, you cannot use the /P and /D switches if you choose to link a program that uses the BRUN.COM runtime module. Furthermore, any program that requires BRUN.COM cannot be put into ROM.

The only way that you can put a compiled BASIC program into ROM is by linking to the OSLIB.REL runtime library. This library is searched by default at linktime only if at compiletime you compile with the /O switch.

The disadvantages of using OSLIB.REL are discussed in Chapter 7.

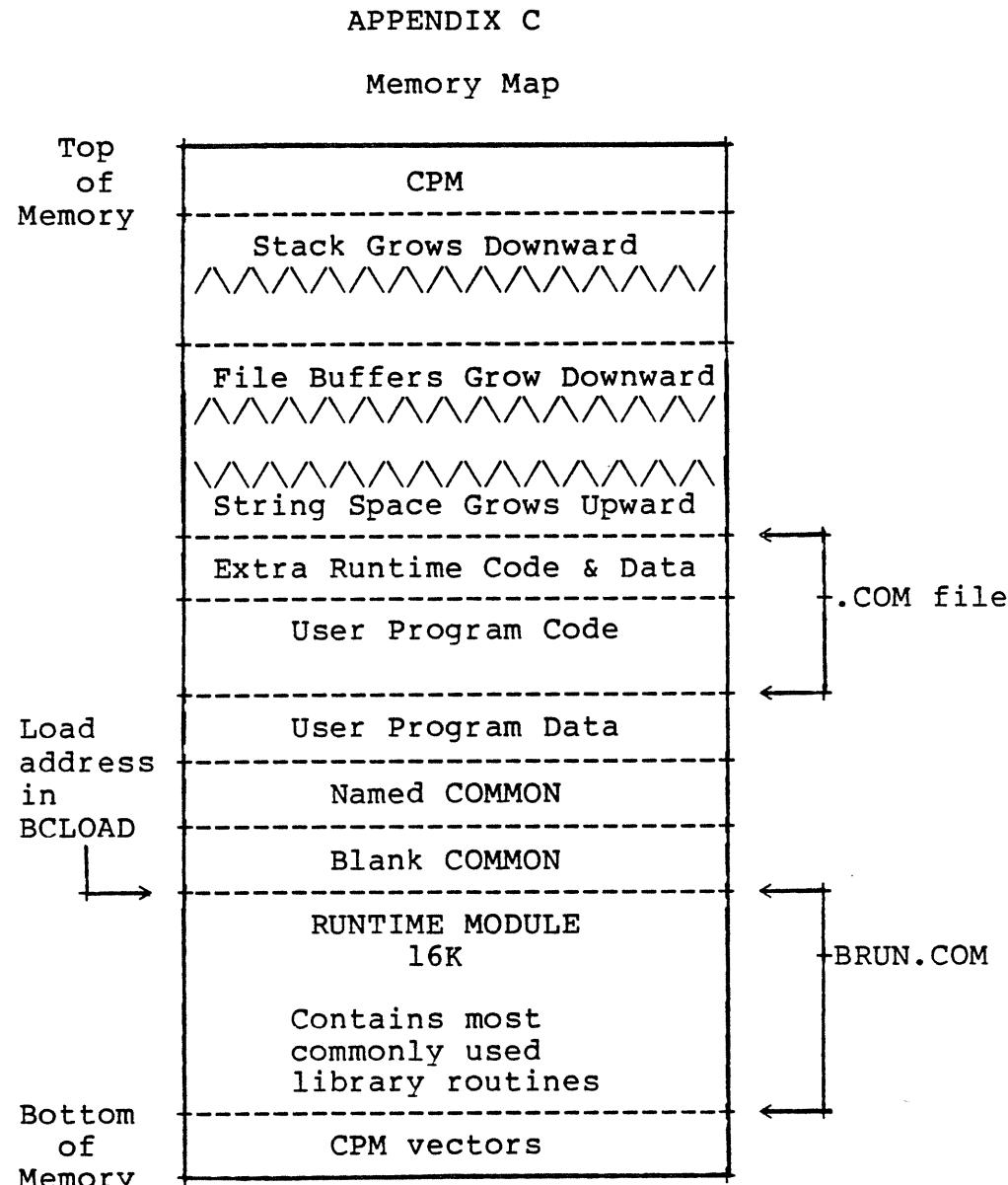


Figure 1 Runtime Memory Map

Runtime memory map of a program using the BRUN.COM runtime module.

APPENDIX D

Differences between Version 5.3 and Previous Versions of the BASIC Compiler

Described below are the major differences between this version of the BASIC Compiler (5.3) and previous versions of the compiler:

1. Your compiled programs now rely on a large runtime module for most of the runtime support that you need during program execution. This module is named BRUN.COM.
2. What used to be called BASLIB.REL, is now called OBSDLIB.REL (short for Old BaSlib). The runtime library on your disk called BASLIB.REL contains a dummy module containing references to all the routines in the BRUN.COM module. BRUN.COM is never in memory at linktime.
3. The COMMON statement now works between CHAINed subprograms, as well as between functions in the same program.
4. The CHAIN statement is no longer semantically equivalent to RUN, and true chaining is allowed. Note that CHAIN <filename> does not cause reloading of the runtime module. In fact, BRUN.COM acts much like the interpreter in this case, supervising the change of control from one program to the next.
5. The CLEAR command is now implemented.
6. The RUN <line-number> form of the RUN command is now implemented.

As a result of the above changes to the BASIC compiler package, the royalty requirements have been altered. The old runtime library (what used to be BASLIB.REL and is now OBSDLIB.REL) can be used in your applications without payment of royalties. However, notice must exist within your application that portions of your software are copyrighted by Microsoft.

However, any distribution of the BRUN.COM runtime module requires payment of royalties. Examine your non-disclosure agreement or contact Microsoft for more specific information on the nature of royalty payments.

INDEX

%INCLUDE	4-1, 9-9
,A - save option	3-2, 4-1
,TTY:=TTY:	6-7
/4 switch (compiler)	6-10 to 6-11
/C switch (compiler)	6-10, 6-12
/D switch (compiler)	6-10, 6-14, 9-10, 9-13
/D switch (linker)	7-5
/E switch (compiler)	6-10, 6-13, 9-9
/E switch (linker)	7-1
/G switch (linker)	7-1
/N switch (compiler)	6-10, 6-14
/N switch (linker)	7-1
/O switch (compiler)	6-14
/P switch (linker)	7-5, 9-11
/S switch (compiler)	6-10, 6-15
/T switch (compiler)	6-10, 6-12
/X switch (compiler)	6-10, 6-13, 9-9
/Z switch (compiler)	6-10, 6-14
4.51 execution switch - /T	6-10, 6-12
4.51 lexical switch - /4	6-10 to 6-11
Array Variables	9-14
ASCII - source file format	3-2, 4-1
BASIC Compiler procedures	3-2
BASIC Compiler User's Manual	1-7
BASIC Learning Resources	1-8
BASIC Runtime Errors	10-5
BASIC Statements not implemented	1-8
BASIC-80 Reference Manual	1-8
BASLIB.REL	2-4
BCLOAD	7-4
BCLOAD - Format	7-4
BRUN.COM	2-4, 9-11
CALL	9-2, 9-10 to 9-11
CAPITAL LETTERS	1-5
CHAIN	9-3
CHR\$	9-11
CLEAR	9-4
Code Relative	2-3
Command line syntax	6-1
Commands not implemented	9-2
COMMON	9-4
COMMON - blank	9-4
COMMON - named	9-4
Compilation	3-2

Compiler Fatal Errors	10-2
Compiletime	2-2
Compiletime Error Messages . . .	10-1
Compiling - output files	6-2
Compiling - technical details .	6-1
Contents of Package	1-6
Convention switches	6-8
Copyright Requirement	4
Debug code switch - /D	6-10, 6-14
DEFDBL	9-5
DEFINT	9-5
DEFSNG	9-5
DEFSTR	9-5
Device Designations	6-6
Device names as files	6-7
Devices as Parameters	6-7
Differences	9-2
DIM	9-6
Documentation	1-7
Double Precision Arithmetic .	9-14
Editing - technical details . . .	9-1
Ellipses (...)	1-5
END	9-7
Error Messages	10-1
Error Trapping	9-9
Error Trapping switches	6-8
Errors - Fatal	10-2
Errors - Runtime	10-5
Errors - Warning	10-4
Expression Evaluation	9-13
Extensions - filename	6-5
Fatal Errors - Compiler	10-2
Filename Extensions	6-5
FOR/NEXT	9-7
Global Reference	2-3
Global Reference - Unbound . .	2-3
Global Reference - Undefined .	2-3
How to use this manual	1-3
INCLUDE	4-1, 9-9
Integer Variables	9-14
Language Differences	9-2
Learning BASIC	1-8
Line Length	4-2
Line number switch - /C	6-10, 6-12
Link Loading	2-5, 3-3
Link Loading - basic steps . .	3-3
Linking - restrictions	7-4
Linking - technical details .	7-1
Linktime	2-2
Long string switch - /S	6-10, 6-15

Manual descriptions	1-7
Manuals - BASIC-80 Reference .	1-8
Manuals - Compiler User's . .	1-7
Manuals - User's Manual . . .	1-3
Manuals - Utility Software . .	1-8
Memory Map	C-1
Message - finished linking . .	3-4
Module	2-3
Module - BRUN.COM	2-4
Non-Disclosure Agreement . . .	4
Notation used in Manual . . .	1-5
OBSDLIB.REL	2-4, 6-14
OBSDLIB.REL - Advantages . . .	7-5
OBSDLIB.REL - Disadvantages . .	7-6
ON ERROR GOTO	9-9
ON ERROR GOTO switch - /E . .	6-10, 6-13
ON ERROR GOTO switch - /X . .	6-10, 6-13
Operational Differences . . .	9-2
Operators	9-13
Optimizations	2-2
Other Differences	9-12
Overflow	9-13
POKE	9-10 to 9-11
Procedures - BASIC Compiler .	3-2
Program Development Process .	2-6
Relocatable	2-3
REM	9-9
Resources for BASIC	1-8
RESUME	9-9
Routine	2-3
Royalty Information	4
RUN	9-9
Running a Compiled Program .	8-1
Running a Program	3-5
Runtime	2-2
Runtime Errors - BASIC . . .	10-5
Runtime library - BASLIB.REL .	2-4
Runtime library - OBSDLIB.REL .	2-4
Runtime support	2-4
Runtime support routines . . .	2-4
SAVE	3-2, 4-1
Source file - format	3-2, 4-1
Special Code switches	6-9
Statements not implemented . .	9-2
Static nesting	9-7
STOP	9-10
String Space	9-14
Subroutine	9-2
Subscripts	9-6
SUPPRESS code switch - /N . .	6-10, 6-14
Switch - /4 (compiler)	6-10 to 6-11
Switch - /C (compiler)	6-10, 6-12
Switch - /D (compiler)	6-10, 6-14

Switch - /D (linker)	7-5
Switch - /E (compiler)	6-10, 6-13
Switch - /E (linker)	7-1
Switch - /G (linker)	7-1
Switch - /N (compiler)	6-10, 6-14
Switch - /N (linker)	7-1
Switch - /O (compiler)	6-14
Switch - /P (linker)	7-5
Switch - /S (compiler)	6-10, 6-15
Switch - /T (compiler)	6-10, 6-12
Switch - /X (compiler)	6-10, 6-13
Switch - /Z (compiler)	6-10, 6-14
Switch - Conventions	6-8
Switch - Error Trapping	6-8
Switch - Special Code	6-9
Syntax - Command Line	6-1
Syntax Notation	1-5
System Requirements	3
TROFF	9-10
TRON	9-10
Unbound Global Reference	2-3
Undefined Global Reference	2-3
USRn Functions	9-10
Utility Software Manual	1-8
VARPTR	9-11
Warning errors	10-4
Warranty	2
WHILE...WEND	9-12
Z80 switch - /Z	6-10, 6-14
<> (angle brackets)	1-5
[] (square brackets)	1-5
... (ellipses)	1-5

basic-80 reference manual

This manual is a reference for Microsoft's BASIC-80 language, release 5.0 and later.

There are significant differences between the 5.0 release of BASIC-80 and the previous releases (release 4.51 and earlier). If you have programs written under a previous release of BASIC-80, check Appendix A for new features in 5.0 that may affect execution.

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy Microsoft BASIC on cassette tape, disk, or any other medium for any purpose other than personal convenience.

© Microsoft, 1979

LIMITED WARRANTY

MICROSOFT shall have no liability or responsibility to purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of this product. This product will be exchanged within twelve months from date of purchase if defective in manufacture, labeling or packaging, but except for such replacement the sale or subsequent use of this program is without warranty or liability.

THE ABOVE IS A LIMITED WARRANTY AND THE ONLY WARRANTY MADE BY MICROSOFT. ANY AND ALL WARRANTIES FOR MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY EXCLUDED.

To report software bugs or errors in the documentation, please complete and return the Problem Report at the back of this manual.

CP/M is a registered trademark of Digital Research

BASIC-80 Reference Manual

CONTENTS

INTRODUCTION

- CHAPTER 1 General Information About BASIC-80
- CHAPTER 2 BASIC-80 Commands and Statements
- CHAPTER 3 BASIC-80 Functions

- APPENDIX A New Features in BASIC-80, Release 5.0
- APPENDIX B BASIC-80 Disk I/O
- APPENDIX C Assembly Language Subroutines
- APPENDIX D BASIC-80 with the CP/M Operating System
- APPENDIX E BASIC-80 with the ISIS-II Operating System
- APPENDIX F BASIC-80 with the TEKDOS Operating System
- APPENDIX G BASIC-80 with the Intel SBC and MDS Systems
- APPENDIX H Standalone Disk BASIC
- APPENDIX I Converting Programs to BASIC-80
- APPENDIX J Summary of Error Codes and Error Messages
- APPENDIX K Mathematical Functions
- APPENDIX L Microsoft BASIC Compiler
- APPENDIX M ASCII Character Codes

Introduction

BASIC-80 is the most extensive implementation of BASIC available for the 8080 and Z80 microprocessors. In its fifth major release (Release 5.0), BASIC-80 meets the ANSI qualifications for BASIC, as set forth in document BSRX3.60-1978. Each release of BASIC-80 consists of three upward compatible versions: 8K, Extended and Disk. This manual is a reference for all three versions of BASIC-80, release 5.0 and later. This manual is also a reference for Microsoft BASIC-86 and the Microsoft BASIC Compiler. BASIC-86 is currently available in Extended and Disk Standalone versions, which are comparable to the BASIC-80 Extended and Disk Standalone versions.

There are significant differences between the 5.0 release of BASIC-80 and the previous releases (release 4.51 and earlier). If you have programs written under a previous release of BASIC-80, check Appendix A for new features in 5.0 that may affect execution.

The manual is divided into three large chapters plus a number of appendices. Chapter 1 covers a variety of topics, largely pertaining to information representation when using BASIC-80. Chapter 2 contains the syntax and semantics of every command and statement in BASIC-80, ordered alphabetically. Chapter 3 describes all of BASIC-80's intrinsic functions, also ordered alphabetically. The appendices contain information pertaining to individual operating systems; plus lists of error messages, ASCII codes, and math functions; and helpful information on assembly language subroutines and disk I/O.

CHAPTER 1

GENERAL INFORMATION ABOUT BASIC-80

1.1 INITIALIZATION

The procedure for initialization will vary with different implementations of BASIC-80. Check the appropriate appendix at the back of this manual to determine how BASIC-80 is initialized with your operating system.

1.2 MODES OF OPERATION

When BASIC-80 is initialized, it types the prompt "Ok". "Ok" means BASIC-80 is at command level, that is, it is ready to accept commands. At this point, BASIC-80 may be used in either of two modes: the direct mode or the indirect mode.

In the direct mode, BASIC statements and commands are not preceded by line numbers. They are executed as they are entered. Results of arithmetic and logical operations may be displayed immediately and stored for later use, but the instructions themselves are lost after execution. This mode is useful for debugging and for using BASIC as a "calculator" for quick computations that do not require a complete program.

The indirect mode is the mode used for entering programs. Program lines are preceded by line numbers and are stored in memory. The program stored in memory is executed by entering the RUN command.

1.3 LINE FORMAT

Program lines in a BASIC program have the following format (square brackets indicate optional):

nnnnn BASIC statement[:BASIC statement...] <carriage return>

At the programmer's option, more than one BASIC statement may be placed on a line, but each statement on a line must be separated from the last by a colon.

A BASIC program line always begins with a line number, ends with a carriage return, and may contain a maximum of:

72 characters in 8K BASIC-80
255 characters in Extended and Disk BASIC-80.

In Extended and Disk versions, it is possible to extend a logical line over more than one physical line by use of the terminal's <line feed> key. <Line feed> lets you continue typing a logical line on the next physical line without entering a <carriage return>. (In the 8K version, <line feed> has no effect.)

1.3.1 Line Numbers

Every BASIC program line begins with a line number. Line numbers indicate the order in which the program lines are stored in memory and are also used as references when branching and editing. Line numbers must be in the range 0 to 65529. In the Extended and Disk versions, a period (.) may be used in EDIT, LIST, AUTO and DELETE commands to refer to the current line.

1.4 CHARACTER SET

The BASIC-80 character set is comprised of alphabetic characters, numeric characters and special characters.

The alphabetic characters in BASIC-80 are the upper case and lower case letters of the alphabet.

The numeric characters in BASIC-80 are the digits 0 through 9.

The following special characters and terminal keys are recognized by BASIC-80:

<u>Character</u>	<u>Name</u>
	Blank
=	Equal sign or assignment symbol
+	Plus sign
-	Minus sign
*	Asterisk or multiplication symbol
/	Slash or division symbol
^	Up arrow or exponentiation symbol
(Left parenthesis
)	Right parenthesis
%	Percent
#	Number (or pound) sign
\$	Dollar sign
!	Exclamation point
[Left bracket
]	Right bracket
,	Comma
:	Period or decimal point
'	Single quotation mark (apostrophe)
;	Semicolon
:	Colon
&	Ampersand
?	Question mark
<	Less than
>	Greater than
\	Backslash or integer division symbol
@	At-sign
	Underscore
<rubout>	Deletes last character typed.
<escape>	Escapes Edit Mode subcommands. See Section 2.16.
<tab>	Moves print position to next tab stop. Tab stops are every eight columns.
<line feed>	Moves to next physical line.
<carriage return>	Terminates input of a line.

1.4.1 Control Characters

The following control characters are in BASIC-80:

- | | |
|-----------|---|
| Control-A | Enters Edit Mode on the line being typed. |
| Control-C | Interrupts program execution and returns to BASIC-80 command level. |
| Control-G | Rings the bell at the terminal. |
| Control-H | Backspace. Deletes the last character typed. |
| Control-I | Tab. Tab stops are every eight columns. |
| Control-O | Halts program output while execution continues. A second Control-O restarts output. |
| Control-R | Retypes the line that is currently being typed. |
| Control-S | Suspends program execution. |
| Control-Q | Resumes program execution after a Control-S. |
| Control-U | Deletes the line that is currently being typed. |

1.5 CONSTANTS

Constants are the actual values BASIC uses during execution. There are two types of constants: string and numeric.

A string constant is a sequence of up to 255 alphanumeric characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"  
"$25,000.00"  
"Number of Employees"
```

Numeric constants are positive or negative numbers. Numeric constants in BASIC cannot contain commas. There are five types of numeric constants:

1. Integer constants Whole numbers between -32768 and +32767. Integer constants do not have decimal points.
2. Fixed Point constants Positive or negative real numbers, i.e., numbers that contain decimal points.

3. Floating Point constants

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). The allowable range for floating point constants is 10-38 to 10+38.
Examples:

235.988E-7 = .0000235988
2359E6 = 2359000000

(Double precision floating point constants use the letter D instead of E. See Section 1.5.1.)

4. Hex constants

Hexadecimal numbers with the prefix &H. Examples:

&H76
&H32F

5. Octal constants

Octal numbers with the prefix &O or &. Examples:

&O347
&1234

1.5.1 Single And Double Precision Form For Numeric Constants

In the 8K version of BASIC-80, all numeric constants are single precision numbers. They are stored with 7 digits of precision, and printed with up to 6 digits.

In the Extended and Disk versions, however, numeric constants may be either single precision or double precision numbers. With double precision, the numbers are stored with 16 digits of precision, and printed with up to 16 digits.

A single precision constant is any numeric constant that has:

1. seven or fewer digits, or
2. exponential form using E, or
3. a trailing exclamation point (!)

A double precision constant is any numeric constant that has:

1. eight or more digits, or
2. exponential form using D, or
3. a trailing number sign (#)

Examples:

<u>Single Precision Constants</u>	<u>Double Precision Constants</u>
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

1.6 VARIABLES

Variables are names used to represent values that are used in a BASIC program. The value of a variable may be assigned explicitly by the programmer, or it may be assigned as the result of calculations in the program. Before a variable is assigned a value, its value is assumed to be zero.

1.6.1 Variable Names And Declaration Characters

BASIC-80 variable names may be any length, however, in the 8K version, only the first two characters are significant. In the Extended and Disk versions, up to 40 characters are significant. The characters allowed in a variable name are letters and numbers, and the decimal point is allowed in Extended and Disk variable names. The first character must be a letter. Special type declaration characters are also allowed -- see below.

A variable name may not be a reserved word. The Extended and Disk versions allow embedded reserved words; the 8K version does not. If a variable begins with FN, it is assumed to be a call to a user-defined function. Reserved words include all BASIC-80 commands, statements, function

names and operator names.

Variables may represent either a numeric value or a string. String variable names are written with a dollar sign (\$) as the last character. For example: A\$ = "SALES REPORT". The dollar sign is a variable type declaration character, that is, it "declares" that the variable will represent a string.

In the Extended and Disk versions, numeric variable names may declare integer, single or double precision values. (All numeric values in 8K are single precision.) The type declaration characters for these variable names are as follows:

```
% Integer variable  
! Single precision variable  
# Double precision variable
```

The default type for a numeric variable name is single precision.

Examples of BASIC-80 variable names follow.

In Extended and Disk versions:

```
PI#      declares a double precision value  
MINIMUM! declares a single precision value  
LIMIT%    declares an integer value
```

In 8K, Extended and Disk versions:

```
N$      declares a string value  
ABC     represents a single precision value
```

In the Extended and Disk versions of BASIC-80, there is a second method by which variable types may be declared. The BASIC-80 statements DEFINT, DEFSTR, DEFSNG and DEFDBL may be included in a program to declare the types for certain variable names. These statements are described in detail in Section 2.12.

1.6.2 Array Variables

An array is a group or table of values referenced by the same variable name. Each element in an array is referenced by an array variable that is subscripted with an integer or an integer expression. An array variable name has as many subscripts as there are dimensions in the array. For example V(10) would reference a value in a one-dimension array, T(1,4) would reference a value in a two-dimension array, and so on. The maximum number of dimensions for an

array is 255. The maximum number of elements per dimension is 32767.

1.6.3 Space Requirements

VARIABLES: BYTES

INTEGER	2
SINGLE PRECISION	4
DOUBLE PRECISION	8

ARRAYS: BYTES

INTEGER	2 per element
SINGLE PRECISION	4 per element
DOUBLE PRECISION	8 per element

STRINGS:

3 bytes overhead plus the present contents of the string.

1.7 TYPE CONVERSION

When necessary, BASIC will convert a numeric constant from one type to another. The following rules and examples should be kept in mind.

1. If a numeric constant of one type is set equal to a numeric variable of a different type, the number will be stored as the type declared in the variable name. (If a string variable is set equal to a numeric value or vice versa, a "Type mismatch" error occurs.)

Example:

```
10 A% = 23.42
20 PRINT A%
RUN
23
```

2. During expression evaluation, all of the operands in an arithmetic or relational operation are converted to the same degree of precision, i.e., that of the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

Examples:

```
10 D# = 6#/7      The arithmetic was performed
```

```
20 PRINT D#      in double precision and the
RUN             result was returned in D#
    .8571428571428571 as a double precision value.
```

```
10 D = 6#/7      The arithmetic was performed
20 PRINT D        in double precision and the
RUN             result was returned to D (single
    .857143         precision variable), rounded and
                      printed as a single precision
                      value.
```

3. Logical operators (see Section 1.8.3) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an "Overflow" error occurs.
4. When a floating point value is converted to an integer, the fractional portion is rounded.
Example:

```
10 C% = 55.88
20 PRINT C%
RUN
56
```

5. If a double precision variable is assigned a single precision value, only the first seven digits, rounded, of the converted number will be valid. This is because only seven digits of accuracy were supplied with the single precision value. The absolute value of the difference between the printed double precision number and the original single precision value will be less than 6.3E-8 times the original single precision value.
Example:

```
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
2.04 2.039999961853027
```

1.8 EXPRESSIONS AND OPERATORS

An expression may be simply a string or numeric constant, or a variable, or it may combine constants and variables with operators to produce a single value.

Operators perform mathematical or logical operations on values. The operators provided by BASIC-80 may be divided into four categories:

1. Arithmetic
2. Relational
3. Logical
4. Functional

1.8.1 Arithmetic Operators

The arithmetic operators, in order of precedence, are:

<u>Operator</u>	<u>Operation</u>	<u>Sample Expression</u>
^	Exponentiation	X^Y
-	Negation	-X
* , /	Multiplication, Floating Point Division	X*Y X/Y
+ , -	Addition, Subtraction	X+Y

To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

<u>Algebraic Expression</u>	<u>BASIC Expression</u>
$X+2Y$	$X+Y*2$
$X - \frac{Y}{Z}$	$X-Y/Z$
$\frac{XY}{Z}$	$X*Y/Z$
$\frac{X+Y}{Z}$	$(X+Y)/Z$
$(X^2)^Y$	$(X^2)^Y$
X^{YZ}	$X^(Y^Z)$
$X(-Y)$	$X*(-Y)$ Two consecutive operators must be separated by parentheses.

1.8.1.1 Integer Division And Modulus Arithmetic -

Two additional operators are available in Extended and Disk versions of BASIC-80: Integer division and modulus arithmetic.

Integer division is denoted by the backslash (\). The operands are rounded to integers (must be in the range -32768 to 32767) before the division is performed, and the quotient is truncated to an integer.

For example:

```
10\4 = 2  
25.68\6.99 = 3
```

The precedence of integer division is just after multiplication and floating point division.

Modulus arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division. For example:

```
10.4 MOD 4 = 2 (10/4=2 with a remainder 2)  
25.68 MOD 6.99 = 5 (26/7=3 with a remainder 5)
```

The precedence of modulus arithmetic is just after integer division.

1.8.1.2 Overflow And Division By Zero -

If, during the evaluation of an expression, a division by zero is encountered, the "Division by zero" error message is displayed, machine infinity with the sign of the numerator is supplied as the result of the division, and execution continues. If the evaluation of an exponentiation results in zero being raised to a negative power, the "Division by zero" error message is displayed, positive machine infinity is supplied as the result of the exponentiation, and execution continues.

If overflow occurs, the "Overflow" error message is displayed, machine infinity with the algebraically correct sign is supplied as the result, and execution continues.

1.8.2 Relational Operators

Relational operators are used to compare two values. The result of the comparison is either "true" (-1) or "false" (0). This result may then be used to make a decision regarding program flow. (See IF, Section 2.26.)

<u>Operator</u>	<u>Relation Tested</u>	<u>Expression</u>
=	Equality	X=Y
<>	Inequality	X<>Y
<	Less than	X<Y
>	Greater than	X>Y
<=	Less than or equal to	X<=Y
>=	Greater than or equal to	X>=Y

(The equal sign is also used to assign a value to a variable. See LET, Section 2.30.)

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression

X+Y < (T-1)/Z

is true if the value of X plus Y is less than the value of T-1 divided by Z. More examples:

```
IF SIN(X)<0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

1.8.3 Logical Operators

Logical operators perform tests on multiple relations, bit manipulation, or Boolean operations. The logical operator returns a bitwise result which is either "true" (not zero) or "false" (zero). In an expression, logical operations are performed after arithmetic and relational operations. The outcome of a logical operation is determined as shown in the following table. The operators are listed in order of precedence.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Just as the relational operators can be used to make decisions regarding program flow, logical operators can connect two or more relations and return a true or false value to be used in a decision (see IF, Section 2.26). For example:

```
IF D<200 AND F<4 THEN 80
IF I>10 OR K<0 THEN 50
IF NOT P THEN 100
```

Logical operators work by converting their operands to sixteen bit, signed, two's complement integers in the range -32768 to +32767. (If the operands are not in this range, an error results.) If both operands are supplied as 0 or -1, logical operators return 0 or -1. The given operation is

performed on these integers in bitwise fashion, i.e., each bit of the result is determined by the corresponding bits in the two operands.

Thus, it is possible to use logical operators to test bytes for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port. The OR operator may be used to "merge" two bytes to create a particular binary value. The following examples will help demonstrate how the logical operators work.

63 AND 16=16 63 = binary 111111 and 16 = binary
10000, so 63 AND 16 = 16

15 AND 14=14 15 = binary 1111 and 14 = binary 1110,
so 15 AND 14 = 14 (binary 1110)

-1 AND 8=8 -1 = binary 1111111111111111 and
8 = binary 1000, so -1 AND 8 = 8

4 OR 2=6 4 = binary 100 and 2 = binary 10,
so 4 OR 2 = 6 (binary 110)

10 OR 10=10 10 = binary 1010, so 1010 OR 1010 =
1010 (10)

-1 OR -2=-1 -1 = binary 1111111111111111 and
-2 = binary 1111111111111110,
so -1 OR -2 = -1. The bit
complement of sixteen zeros is
sixteen ones, which is the
two's complement representation of -1.

NOT X==-(X+1) The two's complement of any integer
is the bit complement plus one.

1.8.4 Functional Operators

A function is used in an expression to call a predetermined operation that is to be performed on an operand. BASIC-80 has "intrinsic" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC-80's intrinsic functions are described in Chapter 3.

BASIC-80 also allows "user defined" functions that are written by the programmer. See DEF FN, Section 2.11.

1.8.5 String Operations

Strings may be concatenated using +. For example:

```
10 A$="FILE" : B$="NAME"
20 PRINT A$ + B$
30 PRINT "NEW " + A$ + B$
RUN
FILENAME
NEW FILENAME
```

Strings may be compared using the same relational operators that are used with numbers:

= <> < > <= >=

String comparisons are made by taking one character at a time from each string and comparing the ASCII codes. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. Examples:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/12/78"      where B$ = "8/12/78"
```

Thus, string comparisons can be used to test string values or to alphabetize strings. All string constants used in comparison expressions must be enclosed in quotation marks.

1.9 INPUT EDITING

If an incorrect character is entered as a line is being typed, it can be deleted with the RUBOUT key or with Control-H. Rubout surrounds the deleted character(s) with backslashes, and Control-H has the effect of backspacing over a character and erasing it. Once a character(s) has been deleted, simply continue typing the line as desired.

To delete a line that is in the process of being typed, type Control-U. A carriage return is executed automatically after the line is deleted.

To correct program lines for a program that is currently in memory, simply retype the line using the same line number. BASIC-80 will automatically replace the old line with the new line.

More sophisticated editing capabilities are provided in the Extended and Disk versions of BASIC-80. See EDIT, Section 2.16.

To delete the entire program that is currently residing in memory, enter the NEW command. (See Section 2.41.) NEW is usually used to clear memory prior to entering a new program.

1.10 ERROR MESSAGES

If BASIC-80 detects an error that causes program execution to terminate, an error message is printed. In the 8K version, only the error code is printed. In the Extended and Disk versions, the entire error message is printed. For a complete list of BASIC-80 error codes and error messages, see Appendix J.

CHAPTER 2

BASIC-80 COMMANDS AND STATEMENTS

All of the BASIC-80 commands and statements are described in this chapter. Each description is formatted as follows:

Format: Shows the correct format for the instruction.
See below for format notation.

Versions: Lists the versions of BASIC-80
in which the instruction is available.

Purpose: Tells what the instruction is used for.

Remarks: Describes in detail how the instruction
is used.

Example: Shows sample programs or program segments
that demonstrate the use of the instruction.

Format Notation

Wherever the format for a statement or command is given, the following rules apply:

1. Items in capital letters must be input as shown.
2. Items in lower case letters enclosed in angle brackets (< >) are to be supplied by the user.
3. Items in square brackets ([]) are optional.
4. All punctuation except angle brackets and square brackets (i.e., commas, parentheses, semicolons, hyphens, equal signs) must be included where shown.
5. Items followed by an ellipsis (...) may be repeated any number of times (up to the length of the line).

2.1 AUTO

Format: AUTO [<line number>[,<increment>]]

Versions: Extended, Disk

Purpose: To generate a line number automatically after every carriage return.

Remarks: AUTO begins numbering at <line number> and increments each subsequent line number by <increment>. The default for both values is 10. If <line number> is followed by a comma but <increment> is not specified, the last increment specified in an AUTO command is assumed.

If AUTO generates a line number that is already being used, an asterisk is printed after the number to warn the user that any input will replace the existing line. However, typing a carriage return immediately after the asterisk will save the line and generate the next line number.

AUTO is terminated by typing Control-C. The line in which Control-C is typed is not saved. After Control-C is typed, BASIC returns to command level.

Example: AUTO 100,50 Generates line numbers 100,
 150, 200 ...

 AUTO Generates line numbers 10,
 20, 30, 40 ...

2.2 CALL

Format: CALL <variable name>[(<argument list>)]

Version: Extended, Disk

Purpose: To call an assembly language subroutine.

Remarks: The CALL statement is one way to transfer program flow to an external subroutine. (See also the USR function, Section 3.40)

<variable name> contains an address that is the starting point in memory of the subroutine. <variable name> may not be an array variable name. <argument list> contains the arguments that are passed to the external subroutine. <argument list> may contain only variables.

The CALL statement generates the same calling sequence used by Microsoft's FORTRAN, COBOL and BASIC compilers.

Example:

```
110 MYROUT=&HD000
120 CALL MYROUT(I,J,K)
```

•
•
•

NOTE: For a BASIC Compiler program, line 110 is not needed because the address of MYROUT will be assigned by the linking loader at load time.

2.3 CHAIN

Format: CHAIN [MERGE] <filename>[,,[<line number exp>]
 [,ALL][,DELETE<range>]]

Version: Disk

Purpose: To call a program and pass variables to it from
 the current program.

Remarks: <filename> is the name of the program that is
 called. Example:

 CHAIN"PROG1"

<line number exp> is a line number or an expression that evaluates to a line number in the called program. It is the starting point for execution of the called program. If it is omitted, execution begins at the first line.
Example:

 CHAIN"PROG1",1000

<line number exp> is not affected by a RENUM command.

With the ALL option, every variable in the current program is passed to the called program. If the ALL option is omitted, the current program must contain a COMMON statement to list the variables that are passed. See Section 2.7.
Example:

 CHAIN"PROG1",1000,ALL

If the MERGE option is included, it allows a subroutine to be brought into the BASIC program as an overlay. That is, a MERGE operation is performed with the current program and the called program. The called program must be an ASCII file if it is to be MERGED. Example:

 CHAIN MERGE"OVRLAY",1000

After an overlay is brought in, it is usually desirable to delete it so that a new overlay may be brought in. To do this, use the DELETE option. Example:

 CHAIN MERGE"OVRLAY2",1000,DELETE 1000-5000

The line numbers in <range> are affected by the RENUM command.

NOTE: The CHAIN statement with MERGE option leaves the files open and preserves the current OPTION BASE setting.

NOTE: If the MERGE option is omitted, CHAIN does not preserve variable types or user-defined functions for use by the chained program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEFFN statements containing shared variables must be restated in the chained program.

NOTE: The Microsoft BASIC compiler does not support the ALL, MERGE, DELETE, and <line number exp> options to CHAIN. Thus, the statement format is CHAIN <filename>. If you wish to maintain compatibility with the BASIC compiler, it is recommended that COMMON be used to pass variables and that overlays not be used. The CHAIN statement leaves the files open during CHAINing.

2.4 CLEAR

Format: `CLEAR [,,[<expression1>][,<expression2>]]`

Versions: `8K, Extended, Disk`

Purpose: To set all numeric variables to zero, all string variables to null, and to close all open files; and, optionally, to set the end of memory and the amount of stack space.

Remarks: `<expression1>` is a memory location which, if specified, sets the highest location available for use by BASIC-80.

`<expression2>` sets aside stack space for BASIC. The default is 256 bytes or one-eighth of the available memory, whichever is smaller.

NOTE: In previous versions of BASIC-80, `<expression1>` set the amount of string space, and `<expression2>` set the end of memory. BASIC-80, release 5.0 and later, allocates string space dynamically. An "Out of string space error" occurs only if there is no free memory left for BASIC to use.

NOTE: The BASIC Compiler supports the CLEAR statement with the restriction that `<expression1>` and `<expression2>` must be integer expressions. If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 256 bytes, and the default top of memory is the current top of memory. The CLEAR statement performs the following actions:

Closes all files
Clears all COMMON and user variables
Resets the stack and string space
Releases all disk buffers

Examples: `CLEAR`

`CLEAR ,32768`

`CLEAR ,,2000`

`CLEAR ,32768,2000`

2.5 CLOAD

Formats: CLOAD <filename>

CLOAD? <filename>

CLOAD* <array name>

Versions: 8K (cassette), Extended (cassette)

Purpose: To load a program or an array from cassette tape into memory.

Remarks: CLOAD executes a NEW command before it loads the program from cassette tape. <filename> is the string expression or the first character of the string expression that was specified when the program was CSAVEd.

CLOAD? verifies tapes by comparing the program currently in memory with the file on tape that has the same filename. If they are the same, BASIC-80 prints Ok. If not, BASIC-80 prints NO GOOD.

CLOAD* loads a numeric array that has been saved on tape. The data on tape is loaded into the array called <array name> specified when the array was CSAVE*ed.

CLOAD and CLOAD? are always entered at command level as direct mode commands. CLOAD* may be entered at command level or used as a program statement. Make sure the array has been DIMensioned before it is loaded. BASIC-80 always returns to command level after a CLOAD, CLOAD? or CLOAD* is executed. Before a CLOAD is executed, make sure the cassette recorder is properly connected and in the Play mode, and the tape is positioned correctly.

See also CSAVE, Section 2.9.

NOTE: CLOAD and CSAVE are not included in all implementations of BASIC-80.

Example: CLOAD "MAX2"

Loads file "M" into memory.

2.6 CLOSE

Format: CLOSE[[#]<file number>[, [#]<file number...>]]

Version: Disk

Purpose: To conclude I/O to a disk file.

Remarks: <file number> is the number under which the file was OPENed. A CLOSE with no arguments closes all open files.

The association between a particular file and file number terminates upon execution of a CLOSE. The file may then be reOPENed using the same or a different file number; likewise, that file number may now be reused to OPEN any file.

A CLOSE for a sequential output file writes the final buffer of output.

The END statement and the NEW command always CLOSE all disk files automatically. (STOP does not close disk files.)

Example: See Appendix B.

2.7 COMMON

Format: COMMON <list of variables>

Version: Disk

Purpose: To pass variables to a CHAINed program.

Remarks: The COMMON statement is used in conjunction with the CHAIN statement. COMMON statements may appear anywhere in a program, though it is recommended that they appear at the beginning. The same variable cannot appear in more than one COMMON statement. Array variables are specified by appending "()" to the variable name. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Example:

```
100 COMMON A,B,C,D(),G$  
110 CHAIN "PROG3",10  
    .  
    .  
    .
```

NOTE: The BASIC Compiler supports a modified version of the COMMON statement. The COMMON statement must appear in a program before any executable statements. The current non-executable statements are:

```
COMMON  
DEFDBL, DEFINT, DEFSNG, DEFSTR  
DIM  
OPTION BASE  
REM  
%INCLUDE
```

Arrays in COMMON must be declared in preceding DIM statements,

The standard form of the COMMON statement is referred to as blank COMMON. FORTRAN style named COMMON areas are also supported; however, the variables are not preserved across CHAINS. The syntax for named COMMON is as follows:

```
COMMON /<name>/ <list of variables>
```

where <name> is 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with FORTRAN and assembly language routines without having to pass explicit parameters in the CALL statement.

The blank COMMON size and order of variables must be the same in the CHAINing and CHAINed-to programs. With the BASIC Compiler, the best way to insure this is to place all blank COMMON declarations in a single include file and use the %INCLUDE statement in each program. For example:

MENU.BAS

```
10 %INCLUDE COMDEF
.
.
. 1000 CHAIN "PROG1"
```

PROG1.BAS

```
10 %INCLUDE COMDEF
.
.
. 2000 CHAIN "MENU"
```

COMDEF.BAS

```
100 DIM A(100),B$(200)
110 COMMON I,J,K,A,()
120 COMMON A$,B$,(),X,Y,Z
```

2.8 CONT

Format: CONT

Versions: 8K, Extended, Disk

Purpose: To continue program execution after a Control-C has been typed, or a STOP or END statement has been executed.

Remarks: Execution resumes at the point where the break occurred. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt (?) or prompt string).

CONT is usually used in conjunction with STOP for debugging. When execution is stopped, intermediate values may be examined and changed using direct mode statements. Execution may be resumed with CONT or a direct mode GOTO, which resumes execution at a specified line number. With the Extended and Disk versions, CONT may be used to continue execution after an error.

CONT is invalid if the program has been edited during the break. In 8K BASIC-80, execution cannot be CONTinued if a direct mode error has occurred during the break.

Example: See example Section 2.61, STOP.

2.9 CSAVE

Formats: CSAVE <string expression>

CSAVE* <array variable name>

Versions: 8K (cassette), Extended (cassette)

Purpose: To save the program or an array currently in memory on cassette tape.

Remarks: Each program or array saved on tape is identified by a filename. When the command CSAVE <string expression> is executed, BASIC-80 saves the program currently in memory on tape and uses the first character in <string expression> as the filename. <string expression> may be more than one character, but only the first character is used for the filename.

When the command CSAVE* <array variable name> is executed, BASIC-80 saves the specified array on tape. The array must be a numeric array. The elements of a multidimensional array are saved with the leftmost subscript changing fastest.

CSAVE may be used as a program statement or as a direct mode command.

Before a CSAVE or CSAVE* is executed, make sure the cassette recorder is properly connected and in the Record mode.

See also CLOAD, Section 2.5.

NOTE: CSAVE and CLOAD are not included in all implementations of BASIC-80.

Example: CSAVE "TIMER"

Saves the program currently in memory on cassette under filename "T".

2.10 DATA

Format: DATA <list of constants>

Versions: 8K, Extended, Disk

Purpose: To store the numeric and string constants that are accessed by the program's READ statement(s). (See READ, Section 2.54)

Remarks: DATA statements are nonexecutable and may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line (separated by commas), and any number of DATA statements may be used in a program. The READ statements access the DATA statements in order (by line number) and the data contained therein may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program.

<list of constants> may contain numeric constants in any format, i.e., fixed point, floating point or integer. (No numeric expressions are allowed in the list.) String constants in DATA statements must be surrounded by double quotation marks only if they contain commas, colons or significant leading or trailing spaces. Otherwise, quotation marks are not needed.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement.

DATA statements may be reread from the beginning by use of the RESTORE statement (Section 2.57).

Example: See examples in Section 2.54, READ.

2.11 DEF FN

Format: DEF FN<name>[(<parameter list>)]=<function definition>

Versions: 8K, Extended, Disk

Purpose: To define and name a function that is written by the user.

Remarks: <name> must be a legal variable name. This name, preceded by FN, becomes the name of the function. <parameter list> is comprised of those variable names in the function definition that are to be replaced when the function is called. The items in the list are separated by commas. <function definition> is an expression that performs the operation of the function. It is limited to one line. Variable names that appear in this expression serve only to define the function; they do not affect program variables that have the same name. A variable name used in a function definition may or may not appear in the parameter list. If it does, the value of the parameter is supplied when the function is called. Otherwise, the current value of the variable is used.

The variables in the parameter list represent, on a one-to-one basis, the argument variables or values that will be given in the function call. (Remember, in the 8K version only one argument is allowed in a function call, therefore the DEF FN statement will contain only one variable.)

In Extended and Disk BASIC-80, user-defined functions may be numeric or string; in 8K, user-defined string functions are not allowed. If a type is specified in the function name, the value of the expression is forced to that type before it is returned to the calling statement. If a type is specified in the function name and the argument type does not match, a "Type mismatch" error occurs.

A DEF FN statement must be executed before the function it defines may be called. If a function is called before it has been defined, an "Undefined user function" error occurs. DEF FN is illegal in the direct mode.

Example:

```
.  
410 DEF FNAB(X,Y)=X^3/Y^2  
420 T=FNAB(I,J)
```

.
. .
Line 410 defines the function FNAB. The function is called in line 420.

2.12 DEFINT/SNG/DBL/STR

Format: DEF<type> <range(s) of letters>
where <type> is INT, SNG, DBL, or STR

Versions: Extended, Disk

Purpose: To declare variable types as integer, single precision, double precision, or string.

Remarks: A DEFtype statement declares that the variable names beginning with the letter(s) specified will be that type variable. However, a type declaration character always takes precedence over a DEFtype statement in the typing of a variable.

If no type declaration statements are encountered, BASIC-80 assumes all variables without declaration characters are single precision variables.

Examples: 10 DEFDBL L-P All variables beginning with the letters L, M, N, O, and P will be double precision variables.

10 DEFSTR A All variables beginning with the letter A will be string variables.

10 DEFINT I-N,W-Z
All variable beginning with the letters I, J, K, L, M, N, W, X, Y, Z will be integer variables.

2.13 DEF USR

Format: DEF USR[<digit>]=<integer expression>

Versions: Extended, Disk

Purpose: To specify the starting address of an assembly language subroutine.

Remarks: <digit> may be any digit from 0 to 9. The digit corresponds to the number of the USR routine whose address is being specified. If <digit> is omitted, DEF USR0 is assumed. The value of <integer expression> is the starting address of the USR routine. See Appendix C, Assembly Language Subroutines.

Any number of DEF USR statements may appear in a program to redefine subroutine starting addresses, thus allowing access to as many subroutines as necessary.

Example:

```
.  
. .  
200 DEF USR0=24000  
210 X=USR0(Y^2/2.89)  
. .  
. .
```

2.14 DELETE

Format: **DELETE**[<line number>] [-<line number>]

Versions: Extended, Disk

Purpose: To delete program lines.

Remarks: BASIC-80 always returns to command level after a DELETE is executed. If <line number> does not exist, an "Illegal function call" error occurs.

Examples: DELETE 40 Deletes line 40

DELETE 40-100 Deletes lines 40 through 100, inclusive

DELETE-40 Deletes all lines up to
and including line 40

2.15 DIM

Format: DIM <list of subscripted variables>

Versions: 8K, Extended, Disk

Purpose: To specify the maximum values for array variable subscripts and allocate storage accordingly.

Remarks: If an array variable name is used without a DIM statement, the maximum value of its subscript(s) is assumed to be 10. If a subscript is used that is greater than the maximum specified, a "Subscript out of range" error occurs. The minimum value for a subscript is always 0, unless otherwise specified with the OPTION BASE statement (see Section 2.46).

The DIM statement sets all the elements of the specified arrays to an initial value of zero.

Example:

```
10 DIM A(20)
20 FOR I=0 TO 20
30 READ A(I)
40 NEXT I
.
.
.
```

2.16 EDIT

Format: EDIT <line number>

Versions: Extended, Disk

Purpose: To enter Edit Mode at the specified line.

Remarks: In Edit Mode, it is possible to edit portions of a line without retying the entire line. Upon entering Edit Mode, BASIC-80 types the line number of the line to be edited, then it types a space and waits for an Edit Mode subcommand.

Edit Mode Subcommands

Edit Mode subcommands are used to move the cursor or to insert, delete, replace, or search for text within a line. The subcommands are not echoed. Most of the Edit Mode subcommands may be preceded by an integer which causes the command to be executed that number of times. When a preceding integer is not specified, it is assumed to be 1.

Edit Mode subcommands may be categorized according to the following functions:

1. Moving the cursor
2. Inserting text
3. Deleting text
4. Finding text
5. Replacing text
6. Ending and restarting Edit Mode

NOTE

In the descriptions that follow, <ch> represents any character, <text> represents a string of characters of arbitrary length, [i] represents an optional integer (the default is 1), and \$ represents the Escape (or Altmode) key.

1. Moving the Cursor

Space Use the space bar to move the cursor to the right. [i]Space moves the cursor i spaces to the right. Characters are printed as you space over them.

Rubout In Edit Mode, [i]Rubout moves the cursor i spaces to the left (backspaces). Characters are printed as you backspace over them.

2. Inserting Text

I I<text>\$ inserts <text> at the current cursor position. The inserted characters are printed on the terminal. To terminate insertion, type Escape. If Carriage Return is typed during an Insert command, the effect is the same as typing Escape and then Carriage Return. During an Insert command, the Rubout, Delete, or Underscore key on the terminal may be used to delete characters to the left of the cursor. Rubout will print out the characters as you backspace over them. Delete and Underscore will print an Underscore for each character that you backspace over. If an attempt is made to insert a character that will make the line longer than 255 characters, a bell (Control-G) is typed and the character is not printed.

X The X subcommand is used to extend the line. X moves the cursor to the end of the line, goes into insert mode, and allows insertion of text as if an Insert command had been given. When you are finished extending the line, type Escape or Carriage Return.

3. Deleting Text

D [i]D deletes i characters to the right of the cursor. The deleted characters are echoed between backslashes, and the cursor is positioned to the right of the last character deleted. If there are fewer than i characters to the right of the cursor, iD deletes the remainder of the line.

H H deletes all characters to the right of the cursor and then automatically enters insert mode. H is useful for replacing statements at the end of a line.

4. Finding Text

S The subcommand [i]S<ch> searches for the ith

occurrence of <ch> and positions the cursor before it. The character at the current cursor position is not included in the search. If <ch> is not found, the cursor will stop at the end of the line. All characters passed over during the search are printed.

K The subcommand [i]K<ch> is similar to [i]S<ch>, except all the characters passed over in the search are deleted. The cursor is positioned before <ch>, and the deleted characters are enclosed in backslashes.

5. Replacing Text

C The subcommand C<ch> changes the next character to <ch>. If you wish to change the next i characters, use the subcommand iC, followed by i characters. After the ith new character is typed, change mode is exited and you will return to Edit Mode.

6. Ending and Restarting Edit Mode

<cr> Typing Carriage Return prints the remainder of the line, saves the changes you made and exits Edit Mode.

E The E subcommand has the same effect as Carriage Return, except the remainder of the line is not printed.

Q The Q subcommand returns to BASIC-80 command level, without saving any of the changes that were made to the line during Edit Mode.

L The L subcommand lists the remainder of the line (saving any changes made so far) and repositions the cursor at the beginning of the line, still in Edit Mode. L is usually used to list the line when you first enter Edit Mode.

A The A subcommand lets you begin editing a line over again. It restores the original line and repositions the cursor at the beginning.

NOTE

If BASIC-80 receives an unrecognizable command or illegal character while in Edit Mode, it prints a bell (Control-G) and the command or character is ignored.

Syntax Errors

When a Syntax Error is encountered during execution of a program, BASIC-80 automatically enters Edit Mode at the line that caused the error. For example:

```
10 K = 2(4)
RUN
?Syntax error in 10
10
```

When you finish editing the line and type Carriage Return (or the E subcommand), BASIC-80 reinserts the line, which causes all variable values to be lost. To preserve the variable values for examination, first exit Edit Mode with the Q subcommand. BASIC-80 will return to command level, and all variable values will be preserved.

Control-A

To enter Edit Mode on the line you are currently typing, type Control-A. BASIC-80 responds with a carriage return, an exclamation point (!) and a space. The cursor will be positioned at the first character in the line. Proceed by typing an Edit Mode subcommand.

NOTE

Remember, if you have just entered a line and wish to go back and edit it, the command "EDIT." will enter Edit Mode at the current line. (The line number symbol "." always refers to the current line.)

2.17 END

Format: END

Versions: 8K, Extended, Disk

Purpose: To terminate program execution, close all files and return to command level.

Remarks: END statements may be placed anywhere in the program to terminate execution. Unlike the STOP statement, END does not cause a BREAK message to be printed. An END statement at the end of a program is optional. BASIC-80 always returns to command level after an END is executed.

Example: 520 IF K>1000 THEN END ELSE GOTO 20

2.18 ERASE

Format: ERASE <list of array variables>

Versions: Extended, Disk

Purpose: To eliminate arrays from a program.

Remarks: Arrays may be redimensioned after they are ERASED, or the previously allocated array space in memory may be used for other purposes. If an attempt is made to redimension an array without first ERASING it, a "Redimensioned array" error occurs.

NOTE: The Microsoft BASIC compiler does not support ERASE.

Example:

```
.  
. .  
450 ERASE A,B  
460 DIM B(99)  
. .  
. .
```

2.19 ERR AND ERL VARIABLES

When an error handling subroutine is entered, the variable ERR contains the error code for the error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF...THEN statements to direct program flow in the error trap routine.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. To test if an error occurred in a direct statement, use IF 65535 = ERL THEN ... Otherwise, use

```
IF ERR = error code THEN ...
```

```
IF ERL = line number THEN ...
```

If the line number is not on the right side of the relational operator, it cannot be renumbered by RENUM. Because ERL and ERR are reserved variables, neither may appear to the left of the equal sign in a LET (assignment) statement. BASIC-80's error codes are listed in Appendix J. (For Standalone Disk BASIC error codes, see Appendix H.)

2.20 ERROR

Format: ERROR <integer expression>

Versions: Extended, Disk

Purpose: 1) To simulate the occurrence of a BASIC-80 error; or 2) to allow error codes to be defined by the user.

Remarks: The value of <integer expression> must be greater than 0 and less than 255. If the value of <integer expression> equals an error code already in use by BASIC-80 (see Appendix J), the ERROR statement will simulate the occurrence of that error, and the corresponding error message will be printed. (See Example 1.)

To define your own error code, use a value that is greater than any used by BASIC-80's error codes. (It is preferable to use the highest available values, so compatibility may be maintained when more error codes are added to BASIC-80.) This user-defined error code may then be conveniently handled in an error trap routine. (See Example 2.)

If an ERROR statement specifies a code for which no error message has been defined, BASIC-80 responds with the message UNPRINTABLE ERROR. Execution of an ERROR statement for which there is no error trap routine causes an error message to be printed and execution to halt.

Example 1: LIST
10 S = 10
20 T = 5
30 ERROR S + T
40 END
Ok
RUN
String too long in line 30

Or, in direct mode:

Ok
ERROR 15 (you type this line)
String too long (BASIC-80 types this line)
Ok

Example 2:

```
.  
. .  
110 ON ERROR GOTO 400  
120 INPUT "WHAT IS YOUR BET";B  
130 IF B > 5000 THEN ERROR 210  
. .  
. .  
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"  
410 IF ERL = 130 THEN RESUME 120  
. .  
. .
```

2.21 FIELD

Format: FIELD[#]<file number>,<field width> AS <string variable>...

Version: Disk

Purpose: To allocate space for variables in a random file buffer.

Remarks: To get data out of a random buffer after a GET or to enter data before a PUT, a FIELD statement must have been executed.

<file number> is the number under which the file was OPENed. <field width> is the number of characters to be allocated to <string variable>. For example,

FIELD 1, 20 AS N\$, 10 AS ID\$, 40 AS ADD\$

allocates the first 20 positions (bytes) in the random file buffer to the string variable N\$, the next 10 positions to ID\$, and the next 40 positions to ADD\$. FIELD does NOT place any data in the random file buffer. (See LSET/RSET and GET.)

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was OPENed. Otherwise, a "Field overflow" error occurs. (The default record length is 128.)

Any number of FIELD statements may be executed for the same file, and all FIELD statements that have been executed are in effect at the same time.

Example: See Appendix B.

NOTE: Do not use a FIELDed variable name in an INPUT or LET statement. Once a variable name is FIELDed, it points to the correct place in the random file buffer. If a subsequent INPUT or LET statement with that variable name is executed, the variable's pointer is moved to string space.

2.22 FOR...NEXT

Format: FOR <variable>=x TO y [STEP z]

•

•

•

NEXT [<variable>][,<variable>...]

where x, y and z are numeric expressions.

Versions: 8K, Extended, Disk

Purpose: To allow a series of instructions to be performed in a loop a given number of times.

Remarks: <variable> is used as a counter. The first numeric expression (x) is the initial value of the counter. The second numeric expression (y) is the final value of the counter. The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by STEP. A check is performed to see if the value of the counter is now greater than the final value (y). If it is not greater, BASIC-80 branches back to the statement after the FOR statement and the process is repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR...NEXT loop. If STEP is not specified, the increment is assumed to be one. If STEP is negative, the final value of the counter is set to be less than the initial value. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step.

Nested Loops

FOR...NEXT loops may be nested, that is, a FOR...NEXT loop may be placed within the context of another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

The variable(s) in the NEXT statement may be

omitted, in which case the NEXT statement will match the most recent FOR statement. If a NEXT statement is encountered before its corresponding FOR statement, a "NEXT without FOR" error message is issued and execution is terminated.

Example 1: 10 K=10
20 FOR I=1 TO K STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
1 20
3 30
5 40
7 50
9 60
Ok

Example 2: 10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I

In this example, the loop does not execute because the initial value of the loop exceeds the final value.

Example 3: 10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
1 2 3 4 5 6 7 8 9 10
Ok

In this example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set. (Note: Previous versions of BASIC-80 set the initial value of the loop variable before setting the final value; i.e., the above loop would have executed six times.)

2.23 GET

Format: GET [#]<file number>[,<record number>]

Version: Disk

Purpose: To read a record from a random disk file into a random buffer.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the next record (after the last GET) is read into the buffer. The largest possible record number is 32767.

Example: See Appendix B.

NOTE: After a GET statement, INPUT# and LINE INPUT# may be done to read characters from the random file buffer.

2.24 GOSUB...RETURN

Format: GOSUB <line number>

.

.

.

RETURN

Versions: 8K, Extended, Disk

Purpose: To branch to and return from a subroutine.

Remarks: <line number> is the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement(s) in a subroutine cause BASIC-80 to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. Subroutines may appear anywhere in the program, but it is recommended that the subroutine be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, it may be preceded by a STOP, END, or GOTO statement that directs program control around the subroutine.

Example:

```
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

2.25 GOTO

Format: GOTO <line number>

Versions: 8K, Extended, Disk

Purpose: To branch unconditionally out of the normal program sequence to a specified line number.

Remarks: If <line number> is an executable statement, that statement and those following are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after <line number>.

Example:

```
LIST
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 10
60 DATA 5,7,12
Ok
RUN
R = 5           AREA = 78.5
R = 7           AREA = 153.86
R = 12          AREA = 452.16
?Out of data in 10
Ok
```

2.26 IF...THEN[...ELSE] AND IF...GOTO

Format: IF <expression> THEN <statement(s)> | <line number>
[ELSE <statement(s)> | <line number>]

Format: IF <expression> GOTO <line number>
[ELSE <statement(s)> | <line number>]

Versions: 8K, Extended, Disk

NOTE: The ELSE clause is allowed only in Extended and Disk versions.

Purpose: To make a decision regarding program flow based on the result returned by an expression.

Remarks: If the result of <expression> is not zero, the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number. If the result of <expression> is zero, the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution continues with the next executable statement. (ELSE is allowed only in Extended and Disk versions.) Extended and Disk versions allow a comma before THEN.

Nesting of IF Statements

In the Extended and Disk versions, IF...THEN...ELSE statements may be nested. Nesting is limited only by the length of the line. For example

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X  
    THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a legal statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example

```
IF A=B THEN IF B=C THEN PRINT "A=C"  
    ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

If an IF...THEN statement is followed by a line number in the direct mode, an "Undefined line" error results unless a statement with the specified line number had previously been entered in the indirect mode.

NOTE: When using IF to test equality for a value that is the result of a floating point computation, remember that the internal representation of the value may not be exact. Therefore, the test should be against the range over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns true if the value of A is 1.0 with a relative error of less than 1.0E-6.

Example 1: 200 IF I THEN GET#1,I

This statement GETs record number I if I is not zero.

Example 2: 100 IF(I<20)*(I>10) THEN DB=1979-1:GOTO 300
110 PRINT "OUT OF RANGE"

•
•
•

In this example, a test determines if I is greater than 10 and less than 20. If I is in this range, DB is calculated and execution branches to line 300. If I is not in this range, execution continues with line 110.

Example 3: 210 IF IOFLAG THEN PRINT A\$ ELSE LPRINT A\$

This statement causes printed output to go either to the terminal or the line printer, depending on the value of a variable (IOFLAG). If IOFLAG is zero, output goes to the line printer, otherwise output goes to the terminal.

2.27 INPUT

Format: INPUT[;][<"prompt string">];]<list of variables>

Versions: 8K, Extended, Disk

Purpose: To allow input from the terminal during program execution.

Remarks: When an INPUT statement is encountered, program execution pauses and a question mark is printed to indicate the program is waiting for data. If <"prompt string"> is included, the string is printed before the question mark. The required data is then entered at the terminal.

A comma may be used instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT "ENTER BIRTHDATE",B\$ will print the prompt with no question mark.

If INPUT is immediately followed by a semicolon, then the carriage return typed by the user to input data does not echo a carriage return/line feed sequence.

The data that is entered is assigned to the variable(s) given in <variable list>. The number of data items supplied must be the same as the number of variables in the list. Data items are separated by commas.

The variable names in the list may be numeric or string variable names (including subscripted variables). The type of each data item that is input must agree with the type specified by the variable name. (Strings input to an INPUT statement need not be surrounded by quotation marks.)

Responding to INPUT with too many or too few items, or with the wrong type of value (numeric instead of string, etc.) causes the message "?Redo from start" to be printed. No assignment of input values is made until an acceptable response is given.

In the 8K version, INPUT is illegal in the direct mode.

Examples: 10 INPUT X
 20 PRINT X "SQUARED IS" X^2
 30 END
 RUN
 ? 5 (The 5 was typed in by the user
 in response to the question mark.)
 5 SQUARED IS 25
 Ok

LIST

```
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 PRINT
60 GOTO 20
Ok
RUN
WHAT IS THE RADIUS? 7.4 (User types 7.4)
THE AREA OF THE CIRCLE IS 171.946

WHAT IS THE RADIUS?
etc.
```

2.28 INPUT#

Format: INPUT#<file number>,<variable list>

Version: Disk

Purpose: To read data items from a sequential disk file and assign them to program variables.

Remarks: <file number> is the number used when the file was OPENed for input. <variable list> contains the variable names that will be assigned to the items in the file. (The variable type must match the type specified by the variable name.) With INPUT#, no question mark is printed, as with INPUT.

The data items in the file should appear just as they would if data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns and line feeds are ignored. The first character encountered that is not a space, carriage return or line feed is assumed to be the start of a number. The number terminates on a space, carriage return, line feed or comma.

If BASIC-80 is scanning the sequential data file for a string item, leading spaces, carriage returns and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of a string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string, and will terminate on a comma, carriage or line feed (or after 255 characters have been read). If end of file is reached when a numeric or string item is being INPUT, the item is terminated.

Example: See Appendix B.

2.29 KILL

Format: KILL <filename>

Version: Disk

Purpose: To delete a file from disk.

Remarks: If a KILL statement is given for a file that is currently OPEN, a "File already open" error occurs.

KILL is used for all types of disk files: program files, random data files and sequential data files.

Example: 200 KILL "DATA1"

See also Appendix B.

2.30 LET

Format: [LET] <variable>=<expression>

Versions: 8K, Extended, Disk

Purpose: To assign the value of an expression to a variable.

Remarks: Notice the word LET is optional, i.e., the equal sign is sufficient when assigning an expression to a variable name.

Example:

```
110 LET D=12
120 LET E=12^2
130 LET F=12^4
140 LET SUM=D+E+F
.
.
.
```

or

```
110 D=12
120 E=12^2
130 F=12^4
140 SUM=D+E+F
.
.
.
```

2.31 LINE INPUT

Format: LINE INPUT[;][<"prompt string">];<string variable>

Versions: Extended, Disk

Purpose: To input an entire line (up to 254 characters) to a string variable, without the use of delimiters.

Remarks: The prompt string is a string literal that is printed at the terminal before input is accepted. A question mark is not printed unless it is part of the prompt string. All input from the end of the prompt to the carriage return is assigned to <string variable>. However, if a line feed/carriage return sequence (this order only) is encountered, both characters are echoed; but the carriage return is ignored, the line feed is put into STRING variable>, and data input continues.

If LINE INPUT is immediately followed by a semicolon, then the carriage return typed by the user to end the input line does not echo a carriage return/line feed sequence at the terminal.

A LINE INPUT may be escaped by typing Control-C. BASIC-80 will return to command level and type Ok. Typing CONT resumes execution at the LINE INPUT.

Example: See Example, Section 2.32, LINE INPUT#.

2.32 LINE INPUT#

Format: LINE INPUT#<file number>,<string variable>

Version: Disk

Purpose: To read an entire line (up to 254 characters), without delimiters, from a sequential disk data file to a string variable.

Remarks: <file number> is the number under which the file was OPENed. <string variable> is the variable name to which the line will be assigned. LINE INPUT# reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT# reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved.)

LINE INPUT# is especially useful if each line of a data file has been broken into fields, or if a BASIC-80 program saved in ASCII mode is being read as data by another program.

Example:

```
10 OPEN "O",1,"LIST"
20 LINE INPUT "CUSTOMER INFORMATION? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "I",1,"LIST"
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN
CUSTOMER INFORMATION? LINDA JONES      234,4      MEMPHIS
LINDA JONES      234,4      MEMPHIS
Ok
```

2.33 LIST

Format 1: LIST [<line number>]

Versions: 8K, Extended, Disk

Format 2: LIST [<line number>[-[<line number>]]]

Versions: Extended, Disk

Purpose: To list all or part of the program currently in memory at the terminal.

Remarks: BASIC-80 always returns to command level after a LIST is executed.

Format 1: If <line number> is omitted, the program is listed beginning at the lowest line number. (Listing is terminated either by the end of the program or by typing Control-C.) If <line number> is included, the 8K version will list the program beginning at that line; and the Extended and Disk versions will list only the specified line.

Format 2: This format allows the following options:

1. If only the first number is specified, that line and all higher-numbered lines are listed.
2. If only the second number is specified, all lines from the beginning of the program through that line are listed.
3. If both numbers are specified, the entire range is listed.

Examples: Format 1:

- | | |
|----------|--|
| LIST | Lists the program currently
in memory. |
| LIST 500 | In the 8K version, lists
all programs lines from
500 to the end.
In Extended and Disk,
lists line 500. |

Format 2:

- | | |
|---------------|---|
| LIST 150- | Lists all lines from 150
to the end. |
| LIST -1000 | Lists all lines from the
lowest number through 1000. |
| LIST 150-1000 | Lists lines 150 through
1000, inclusive. |

2.34 LLIST

Format: LLIST [<line number>[-[<line number>]]]

Versions: Extended, Disk

Purpose: To list all or part of the program currently in memory at the line printer.

Remarks: LLIST assumes a 132-character wide printer.

BASIC-80 always returns to command level after an LLIST is executed. The options for LLIST are the same as for LIST, Format 2.

NOTE: LLIST and LPRINT are not included in all implementations of BASIC-80.

Example: See the examples for LIST, Format 2.

2.35 LOAD

Format: LOAD <filename>[,R]

Version: Disk

Purpose: To load a file from disk into memory.

Remarks: <filename> is the name that was used when the file was SAVED. (With CP/M, the default extension .BAS is supplied.)

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the designated program. However, if the "R" option is used with LOAD, the program is RUN after it is LOADED, and all open data files are kept open. Thus, LOAD with the "R" option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using their disk data files.

Example: LOAD "STRTRK",R

2.36 LPRINT AND LPRINT USING

Format: LPRINT [<list of expressions>]

 LPRINT USING <string exp>;<list of expressions>

Versions: Extended, Disk

Purpose: To print data at the line printer.

Remarks: Same as PRINT and PRINT USING, except output goes to the line printer. See Section 2.49 and Section 2.50.

 LPRINT assumes a 132-character-wide printer.

NOTE: LPRINT and LLIST are not included in all implementations of BASIC-80.

2.37 LSET AND RSET

Format: LSET <string variable> = <string expression>
 RSET <string variable> = <string expression>

Version: Disk

Purpose: To move data from memory to a random file buffer
(in preparation for a PUT statement).

Remarks: If <string expression> requires fewer bytes than
were FIELDed to <string variable>, LSET
left-justifies the string in the field, and RSET
right-justifies the string. (Spaces are used to
pad the extra positions.) If the string is too
long for the field, characters are dropped from
the right. Numeric values must be converted to
strings before they are LSET or RSET. See the
MKI\$, MKS\$, MKD\$ functions, Section 3.25.

Examples: 150 LSET A\$=MKSS\$(AMT)
 160 LSET D\$=DESC(\$)

See also Appendix B.

NOTE: LSET or RSET may also be used with a non-fielded
string variable to left-justify or right-justify
a string in a given field. For example, the
program lines

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N\$ in a 20-character
field. This can be very handy for formatting
printed output.

2.38 MERGE

Format: MERGE <filename>

Version: Disk

Purpose: To merge a specified disk file into the program currently in memory.

Remarks: <filename> is the name used when the file was SAVED. (With CP/M, the default extension .BAS is supplied.) The file must have been SAVED in ASCII format. (If not, a "Bad file mode" error occurs.)

If any lines in the disk file have the same line numbers as lines in the program in memory, the lines from the file on disk will replace the corresponding lines in memory. (MERGEing may be thought of as "inserting" the program lines on disk into the program in memory.)

BASIC-80 always returns to command level after executing a MERGE command.

Example: MERGE "NUMBR5"

2.39 MID\$

Format: `MID$(<string exp1>,n[,m])=<string exp2>`

where `n` and `m` are integer expressions and
`<string exp1>` and `<string exp2>` are string
expressions.

Versions: Extended, Disk

Purpose: To replace a portion of one string with another
string.

Remarks: The characters in `<string exp1>`, beginning at
position `n`, are replaced by the characters in
`<string exp2>`. The optional `m` refers to the
number of characters from `<string exp2>` that
will be used in the replacement. If `m` is
omitted, all of `<string exp2>` is used. However,
regardless of whether `m` is omitted or included,
the replacement of characters never goes beyond
the original length of `<string exp1>`.

Example: `10 A$="KANSAS CITY, MO"`
`20 MID$(A$,14)="KS"`
`30 PRINT A$`
RUN
`KANSAS CITY, KS`

`MID$` is also a function that returns a substring
of a given string. See Section 3.24.

2.40 NAME

Format: NAME <old filename> AS <new filename>

Version: Disk

Purpose: To change the name of a disk file.

Remarks: <old filename> must exist and <new filename> must not exist; otherwise an error will result. After a NAME command, the file exists on the same disk, in the same area of disk space, with the new name.

Example: Ok
NAME "ACCTS" AS "LEDGER"
Ok

In this example, the file that was formerly named ACCTS will now be named LEDGER.

2.41 NEW

Format: NEW

Versions: 8K, Extended, Disk

Purpose: To delete the program currently in memory and clear all variables.

Remarks: NEW is entered at command level to clear memory before entering a new program. BASIC-80 always returns to command level after a NEW is executed.

2.42 NULL

Format: `NULL <integer expression>`

Versions: `8K, Extended, Disk`

Purpose: To set the number of nulls to be printed at the end of each line.

Remarks: For 10-character-per-second tape punches, `<integer expression>` should be ≥ 3 . When tapes are not being punched, `<integer expression>` should be 0 or 1 for Teletypes and Teletype-compatible CRTs. `<integer expression>` should be 2 or 3 for 30 cps hard copy printers. The default value is 0.

Example:

```
Ok
NULL 2
Ok
100 INPUT X
200 IF X<50 GOTO 800
.
.
.
```

Two null characters will be printed after each line.

2.43 ON ERROR GOTO

Format: ON ERROR GOTO <line number>

Versions: Extended, Disk

Purpose: To enable error trapping and specify the first line of the error handling subroutine.

Remarks: Once error trapping has been enabled all errors detected, including direct mode errors (e.g., Syntax errors), will cause a jump to the specified error handling subroutine. If <line number> does not exist, an "Undefined line" error results. To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC-80 to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

NOTE: If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example: 10 ON ERROR GOTO 1000

2.44 ON...GOSUB AND ON...GOTO

Format: ON <expression> GOTO <list of line numbers>

 ON <expression> GOSUB <list of line numbers>

Versions: 8K, Extended, Disk

Purpose: To branch to one of several specified line numbers, depending on the value returned when an expression is evaluated.

Remarks: The value of <expression> determines which line number in the list will be used for branching. For example, if the value is three, the third line number in the list will be the destination of the branch. (If the value is a non-integer, the fractional portion is rounded.)

In the ON...GOSUB statement, each line number in the list must be the first line number of a subroutine.

If the value of <expression> is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement. If the value of <expression> is negative or greater than 255, an "Illegal function call" error occurs.

Example: 100 ON L-1 GOTO 150,300,320,390

2.45 OPEN

Format: OPEN <mode>,[#]<file number>,<filename>,[<reclen>]

Version: Disk

Purpose: To allow I/O to a disk file.

Remarks: A disk file must be OPENed before any disk I/O operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:

O specifies sequential output mode

I specifies sequential input mode

R specifies random input/output mode

<file number> is an integer expression whose value is between one and fifteen. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.

<filename> is a string expression containing a name that conforms to your operating system's rules for disk filenames.

<reclen> is an integer expression which, if included, sets the record length for random files. The default record length is 128 bytes. See also page A-3.

NOTE: A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

Example: 10 OPEN "I",2,"INVEN"

See also Appendix B.

2.46 OPTION BASE

Format: OPTION BASE n
where n is 1 or 0

Versions: 8K, Extended, Disk

Purpose: To declare the minimum value for array
subscripts.

Remarks: The default base is 0. If the statement

 OPTION BASE 1

is executed, the lowest value an array subscript
may have is one.

2.47 OUT

Format: OUT I,J
where I and J are integer expressions in the range 0 to 255.

Versions: 8K, Extended, Disk

Purpose: To send a byte to a machine output port.

Remarks: The integer expression I is the port number, and the integer expression J is the data to be transmitted.

Example: 100 OUT 32,100

2.48 POKE

Format: **POKE I,J**
 where I and J are integer expressions

Versions: 8K, Extended, Disk

Purpose: To write a byte into a memory location.

Remarks: The integer expression I is the address of the memory location to be POKEd. The integer expression J is the data to be POKEd. J must be in the range 0 to 255. In the 8K version, I must be less than 32768. In the Extended and Disk versions, I must be in the range 0 to 65536.

With the 8K version, data may be POKEd into memory locations above 32768 by supplying a negative number for I. The value of I is computed by subtracting 65536 from the desired address. For example, to POKE data into location 45000, I = 45000-65536, or -20536.

The complementary function to POKE is PEEK. The argument to PEEK is an address from which a byte is to be read. See Section 3.27.

POKE and PEEK are useful for efficient data storage, loading assembly language subroutines, and passing arguments and results to and from assembly language subroutines.

Example: 10 POKE &H5A00,&HFF

2.49 PRINT

Format: PRINT [<list of expressions>]

Versions: 8K, Extended, Disk

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is printed. If <list of expressions> is included, the values of the expressions are printed at the terminal. The expressions in the list may be numeric and/or string expressions. (Strings must be enclosed in quotation marks.)

Print Positions

The position of each printed item is determined by the punctuation used to separate the items in the list. BASIC-80 divides the line into print zones of 14 spaces each. In the list of expressions, a comma causes the next value to be printed at the beginning of the next zone. A semicolon causes the next value to be printed immediately after the last value. Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma or a semicolon terminates the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions terminates without a comma or a semicolon, a carriage return is printed at the end of the line. If the printed line is longer than the terminal width, BASIC-80 goes to the next physical line and continues printing.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign. Single precision numbers that can be represented with 6 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1E-7 is output as .0000001 and 1E-8(-7) is output as 1E-08. Double precision numbers that can be represented with 16 or fewer digits in the unscaled format no less accurately than they can be represented in the scaled format, are output using the unscaled format. For example, 1D-15 is output as .000000000000001 and 1D-16 is output as 1D-16.

A question mark may be used in place of the word PRINT in a PRINT statement.

Example 1: 10 X=5
20 PRINT X+5, X-5, X*(-5), X^5
30 END
RUN
10 0 -25 3125
Ok

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

Example 2: LIST
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
40 PRINT
50 GOTO 10
Ok
RUN
? 9
9 SQUARED IS 81 AND 9 CUBED IS 729

? 21
21 SQUARED IS 441 AND 21 CUBED IS 9261

?

In this example, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line, and line 40 causes a blank line to be printed before the next prompt.

Example 3: 10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
Ok
RUN
5 10 10 20 15 30 20 40 25 50
Ok

In this example, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

2.50 PRINT USING

Format: PRINT USING <string exp>;<list of expressions>

Versions: Extended, Disk

Purpose: To print strings or numbers using a specified format.

Remarks
and
Examples: <list of expressions> is comprised of the string expressions or numeric expressions that are to be printed, separated by semicolons. <string exp> is a string literal (or variable) comprised of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

"!" Specifies that only the first character in the given string is to be printed.

"\n spaces\" Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters will be printed; with one space, three characters will be printed, and so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string will be left-justified in the field and padded with spaces on the right.
Example:

```
10 A$="LOOK":B$="OUT"
30 PRINT USING "!" ;A$ ;B$
40 PRINT USING "\ \" ;A$ ;B$
50 PRINT USING "\ \" ;A$ ;B$ ;" !"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

"&" Specifies a variable length string field. When the field is specified with "&", the string is output exactly as input. Example:

```
10 A$="LOOK":B$="OUT"
20 PRINT USING "!" ;A$;
30 PRINT USING "&";B$
RUN
LOUT
```

Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

- # A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has fewer digits than positions specified, the number will be right-justified (preceded by spaces) in the field.
- . A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
0.78
```

```
PRINT USING "###.##";987.654
987.65
```

```
PRINT USING "##.##   ";10.2,5.3,66.789,.234
10.20    5.30    66.79    0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

- + A plus sign at the beginning or end of the format string will cause the sign of the number (plus or minus) to be printed before or after the number.

- A minus sign at the end of the format field will cause negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##   "; -68.95, 2.4, 55.6, -.9
-68.95      +2.40      +55.60      -0.90
```

```
PRINT USING "##.##-   "; -68.95, 22.449, -7.01
68.95-      22.45      7.01-
```

- ** A double asterisk at the beginning of the format string causes leading spaces in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "***#.#   "; 12.39, -0.9, 765.1
*12.4      *-0.9      765.1
```

- \$\$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The \$\$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with \$\$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$##.##"; 456.78
$456.78
```

- ***\$ The ***\$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces will be asterisk-filled and a dollar sign will be printed before the number. ***\$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "***$##.##"; 2.34
***$2.34
```

- , A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^) format.

```
PRINT USING "###,.##"; 1234.5
1,234.50
```

```
PRINT USING "###.##,"; 1234.5
1234.50,
```

^^^^ Four carats (or up-arrows) may be placed after the digit position characters to specify exponential format. The four carats allow space for E+xx to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position will be used to the left of the decimal point to print a space or a minus sign.

```
PRINT USING "#.##^^^";234.56  
2.35E+02
```

```
PRINT USING ".###^__-";888888  
.8889E+06
```

```
PRINT USING "+.##^^^";123  
+.12E+03
```

- An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##!";12.34  
!12.34!
```

The literal character itself may be an underscore by placing "__" in the format string.

% If the number to be printed is larger than the specified numeric field, a percent sign is printed in front of the number. If rounding causes the number to exceed the field, a percent sign will be printed in front of the rounded number.

```
PRINT USING "#.##";111.22  
$111.22
```

```
PRINT USING ".##";.999  
$1.00
```

If the number of digits specified exceeds 24, an "Illegal function call" error will result.

2.51 PRINT# AND PRINT# USING

Format: PRINT#<filenumber>,[USING<string exp> ;]<list of exps>

Version: Disk

Purpose: To write data to a sequential disk file.

Remarks: <file number> is the number used when the file was OPENed for output. <string exp> is comprised of formatting characters as described in Section 2.50, PRINT USING. The expressions in <list of expressions> are the numeric and/or string expressions that will be written to the file.

PRINT# does not compress data on the disk. An image of the data is written to the disk, just as it would be displayed on the terminal with a PRINT statement. For this reason, care should be taken to delimit the data on the disk, so that it will be input correctly from the disk.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

PRINT#1,A;B;C;X;Y;Z

(If commas are used as delimiters, the extra blanks that are inserted between print fields will also be written to disk.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the disk, use explicit delimiters in the list of expressions.

For example, let A\$="CAMERA" and B\$="93604-1". The statement

PRINT#1,A\$;B\$

would write CAMERA93604-1 to the disk. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT# statement as follows:

PRINT#1,A\$;",";B\$

The image written to disk is

CAMERA,93604-1

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to disk surrounded by explicit quotation marks, CHR\$(34).

For example, let A\$="CAMERA, AUTOMATIC" and B\$=" 93604-1". The statement

PRINT#1,A\$;B\$

would write the following image to disk:

CAMERA, AUTOMATIC 93604-1

and the statement

INPUT#1,A\$,B\$

would input "CAMERA" to A\$ and "AUTOMATIC 93604-1" to B\$. To separate these strings properly on the disk, write double quotes to the disk image using CHR\$(34). The statement

PRINT#1,CHR\$(34);A\$;CHR\$(34);CHR\$(34);B\$;CHR\$(34)

writes the following image to disk:

"CAMERA, AUTOMATIC"" 93604-1"

and the statement

INPUT#1,A\$,B\$

would input "CAMERA, AUTOMATIC" to A\$ and " 93604-1" to B\$.

The PRINT# statement may also be used with the USING option to control the format of the disk file. For example:

PRINT#1,USING"\$\$##.##,";J;K;L

For more examples using PRINT#, see Appendix B.

See also WRITE#, Section 2.68.

2.52 PUT

Format: PUT [#]<file number>[,<record number>]

Version: Disk

Purpose: To write a record from a random buffer to a random disk file.

Remarks: <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will have the next available record number (after the last PUT). The largest possible record number is 32767. The smallest record number is 1.

Example: See Appendix B.

NOTE: PRINT#, PRINT# USING, and WRITE# may be used to put characters in the random file buffer before a PUT statement.

In the case of WRITE#, BASIC-80 pads the buffer with spaces up to the carriage return. Any attempt to read or write past the end of the buffer causes a "Field overflow" error.

2.53 RANDOMIZE

Format: RANDOMIZE [<expression>]

Versions: Extended, Disk

Purpose: To reseed the random number generator.

Remarks: If <expression> is omitted, BASIC-80 suspends program execution and asks for a value by printing

Random Number Seed (-32768 to 32767) ?

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 5
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767) ? 3 (user
types 3)
    .88598   .484668   .586328   .119426   .709225
Ok
RUN
Random Number Seed (-32768 to 32767) ? 4 (user
types 4 for new sequence)
    .803506   .162462   .929364   .292443   .322921
Ok
RUN
Random Number Seed (-32768 to 32767) ? 3 (same
sequence as first RUN)
    .88598   .484668   .586328   .119426   .709225
Ok
```

2.54 READ

Format: READ <list of variables>

Versions: 8K, Extended, Disk

Purpose: To read values from a DATA statement and assign them to variables. (See DATA, Section 2.10.)

Remarks: A READ statement must always be used in conjunction with a DATA statement. READ statements assign variables to DATA statement values on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a "Syntax error" will result.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in <list of variables> exceeds the number of elements in the DATA statement(s), an OUT OF DATA message is printed. If the number of variables specified is fewer than the number of elements in the DATA statement(s), subsequent READ statements will begin reading data at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

To reread DATA statements from the start, use the RESTORE statement (see RESTORE, Section 2.57)

Example 1:

```
.  
. .  
80 FOR I=1 TO 10  
90 READ A(I)  
100 NEXT I  
110 DATA 3.08,5.19,3.12,3.98,4.24  
120 DATA 5.08,5.55,4.00,3.16,3.37  
. .  
. .
```

This program segment READs the values from the DATA statements into the array A. After execution, the value of A(1) will be 3.08, and so on.

Example 2: LIST

```
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
Ok
RUN
CITY           STATE           ZIP
DENVER,        COLORADO       80211
Ok
```

This program READs string and numeric data from the DATA statement in line 30.

2.55 REM

Format: REM <remark>

Versions: 8K, Extended, Disk

Purpose: To allow explanatory remarks to be inserted in a program.

Remarks: REM statements are not executed but are output exactly as entered when the program is listed.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution will continue with the first executable statement after the REM statement.

In the Extended and Disk versions, remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM.

WARNING: Do not use this in a data statement as it would be considered legal data.

Example:

```
•  
•  
•  
120 REM CALCULATE AVERAGE VELOCITY  
130 FOR I=1 TO 20  
140 SUM=SUM + V(I)  
•  
•  
•
```

or, with Extended and Disk versions:

```
•  
•  
•  
120 FOR I=1 TO 20      'CALCULATE AVERAGE VELOCITY  
130 SUM=SUM+V(I)  
140 NEXT I  
•  
•  
•
```

2.56 RENUM

Format: RENUM [*[<new number>]*] [*[,<old number>]*] [*[,<increment>]*]

Versions: Extended, Disk

Purpose: To renumber program lines.

Remarks: <new number> is the first line number to be used in the new sequence. The default is 10. <old number> is the line in the current program where renumbering is to begin. The default is the first line of the program. <increment> is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB and ERL statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message "Undefined line xxxxx in yyyy" is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyy may be changed.

NOTE: RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An "Illegal function call" error will result.

RENUM 300,,50 Renumbers the entire program. The first new line number will be 300. Lines will increment by 50.

RENUM 1000,900,20 Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

2.57 RESTORE

Format: RESTORE [<line number>]

Versions: 8K, Extended, Disk

Purpose: To allow DATA statements to be reread from a specified line.

Remarks: After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
      .
      .
      .
```

2.58 RESUME

Formats: RESUME

RESUME 0

RESUME NEXT

RESUME <line number>

Versions: Extended, Disk

Purpose: To continue program execution after an error recovery procedure has been performed.

Remarks: Any one of the four formats shown above may be used, depending upon where execution is to resume:

RESUME Execution resumes at the
 or statement which caused the
RESUME 0 error.

RESUME NEXT Execution resumes at the statement immediately following the one which caused the error.

RESUME <line number> Execution resumes at <line number>.

A RESUME statement that is not in an error trap routine causes a "RESUME without error" message to be printed.

Example: 10 ON ERROR GOTO 900

1

1

1

900 IF (ERR=230) AND (ERL=90) THEN PRINT "TRY AGAIN": RESUME 80

1

1

1

2.59 RUN

Format 1: RUN [<line number>]

Versions: 8K, Extended, Disk

Purpose: To execute the program currently in memory.

Remarks: If <line number> is specified, execution begins on that line. Otherwise, execution begins at the lowest line number. BASIC-80 always returns to command level after a RUN is executed.

Example: RUN

Format 2: RUN <filename>[,R]

Version: Disk

Purpose: To load a file from disk into memory and run it.

Remarks: <filename> is the name used when the file was SAVED. (with CP/M and ISIS-II, the default extension .BAS is supplied.)

RUN closes all open files and deletes the current contents of memory before loading the designated program. However, with the "R" option, all data files remain OPEN.

Example: RUN "NEWFIL",R

See also Appendix B.

NOTE: The BASIC Compiler supports the RUN and RUN <line number> forms of the RUN statement. The BASIC Compiler does not support the "R" option with RUN. If you want this feature, the CHAIN statement should be used.

2.60 SAVE

Format: `SAVE <filename>[,A | ,P]`

Version: Disk

Purpose: To save a program file on disk.

Remarks: `<filename>` is a quoted string that conforms to your operating system's requirements for filenames. (With CP/M, the default extension .BAS is supplied.) If `<filename>` already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADED), any attempt to list or edit it will fail.

Examples: `SAVE"COM2",A`
 `SAVE"PROG",P`

See also Appendix B.

2.61 STOP

Format: STOP

Versions: 8K, Extended, Disk

Purpose: To terminate program execution and return to command level.

Remarks: STOP statements may be used anywhere in a program to terminate execution. When a STOP is encountered, the following message is printed:

Break in line nnnn

Unlike the END statement, the STOP statement does not close files.

BASIC-80 always returns to command level after a STOP is executed. Execution is resumed by issuing a CONT command (see Section 2.8).

Example:

```
10 INPUT A,B,C
20 K=A^2*5.3:L=B^3/.26
30 STOP
40 M=C*K+100:PRINT M
RUN
? 1,2,3
BREAK IN 30
Ok
PRINT L
 30.7692
Ok
CONT
 115.9
Ok
```

2.62 SWAP

Format: SWAP <variable>,<variable>

Versions: Extended, Disk

Purpose: To exchange the values of two variables.

Remarks: Any type variable may be SWAPPED (integer, single precision, double precision, string), but the two variables must be of the same type or a "Type mismatch" error results.

Example: LIST

```
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
Ok
    ONE FOR ALL
    ALL FOR ONE
Ok
```

2.63 TRON/TROFF

Format: TRON

TROFF

Versions: Extended, Disk

Purpose: To trace the execution of program statements.

Remarks: As an aid in debugging, the TRON statement (executed in either the direct or indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace flag is disabled with the TROFF statement (or when a NEW command is executed).

Example: TRON

Ok

LIST

```
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
```

Ok

RUN

```
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
```

Ok

TROFF

Ok

2.64 WAIT

Format: WAIT <port number>, I[,J]
where I and J are integer expressions

Versions: 8K, Extended, Disk

Purpose: To suspend program execution while monitoring
the status of a machine input port.

Remarks: The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern. The data read at the port is exclusive OR'ed with the integer expression J, and then AND'ed with I. If the result is zero, BASIC-80 loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If J is omitted, it is assumed to be zero

CAUTION: It is possible to enter an infinite loop with the WAIT statement, in which case it will be necessary to manually restart the machine.

Example: 100 WAIT 32,2

2.65 WHILE...WEND

Format: WHILE <expression>
 .
 .
 [<loop statements>]
 .
 .
 WEND

Versions: Extended, Disk

Purpose: To execute a series of statements in a loop as long as a given condition is true.

Remarks: If <expression> is not zero (i.e., true), <loop statements> are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks <expression>. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE/WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a "WHILE without WEND" error, and an unmatched WEND statement causes a "WEND without WHILE" error.

Example: 90 'BUBBLE SORT ARRAY A\$
100 FLIPS=1 'FORCE ONE PASS THRU LOOP
110 WHILE FLIPS
115 FLIPS=0
120 FOR I=1 TO J-1
130 IF A\$(I)>A\$(I+1) THEN
 SWAP A\$(I),A\$(I+1):FLIPS=1
140 NEXT I
150 WEND

2.66 WIDTH

Format: **WIDTH [LPRINT] <integer expression>**

Versions: **Extended, Disk**

Purpose: To set the printed line width in number of characters for the terminal or line printer.

Remarks: If the LPRINT option is omitted, the line width is set at the terminal. If LPRINT is included, the line width is set at the line printer.

<integer expression> must have a value in the range 15 to 255. The default width is 72 characters.

If **<integer expression>** is 255, the line width is "infinite," that is, BASIC never inserts a carriage return. However, the position of the cursor or the print head, as given by the POS or LPOS function, returns to zero after position 255.

Example: 10 PRINT "ABCDEFGHIJKLMNPQRSTUVWXYZ"
RUN
ABCDEFGHIJKLMNPQRSTUVWXYZ
Ok
WIDTH 18
Ok
RUN
ABCDEFGHIJKLMNPQR
STUVWXYZ
Ok

2.67 WRITE

Format: `WRITE[<list of expressions>]`

Version: Disk

Purpose: To output data at the terminal.

Remarks: If <list of expressions> is omitted, a blank line is output. If <list of expressions> is included, the values of the expressions are output at the terminal. The expressions in the list may be numeric and/or string expressions, and they must be separated by commas.

When the printed items are output, each item will be separated from the last by a comma. Printed strings will be delimited by quotation marks. After the last item in the list is printed, BASIC inserts a carriage return/line feed.

WRITE outputs numeric values using the same format as the PRINT statement, Section 2.49.

Example: `10 A=80:B=90:C$="THAT'S ALL"`
 `20 WRITE A,B,C$`
 RUN
 `80, 90, "THAT'S ALL"`
 Ok

2.68 WRITE#

Format: `WRITE#<file number>,<list of expressions>`

Version: Disk

Purpose: To write data to a sequential file.

Remarks: <file number> is the number under which the file was OPENed in "O" mode. The expressions in the list are string or numeric expressions, and they must be separated by commas.

The difference between WRITE# and PRINT# is that WRITE# inserts commas between the items as they are written to disk and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. A carriage return/line feed sequence is inserted after the last item in the list is written to disk.

Example: Let `A$="CAMERA"` and `B$="93604-1"`. The statement:

`WRITE#1,A$,B$`

writes the following image to disk:

`"CAMERA","93604-1"`

A subsequent INPUT# statement, such as:

`INPUT#1,A$,B$`

would input "CAMERA" to A\$ and "93604-1" to B\$.

CHAPTER 3

BASIC-80 FUNCTIONS

The intrinsic functions provided by BASIC-80 are presented in this chapter. The functions may be called from any program without further definition.

Arguments to functions are always enclosed in parentheses. In the formats given for the functions in this chapter, the arguments have been abbreviated as follows:

X and Y	Represent any numeric expressions
I and J	Represent integer expressions
X\$ and Y\$	Represent string expressions

If a floating point value is supplied where an integer is required, BASIC-80 will round the fractional portion and use the resulting integer.

NOTE

With the BASIC-80 and BASIC-86 interpreters, only integer and single precision results are returned by functions. Double precision functions are supported only by the BASIC compiler.

3.1 ABS

Format: ABS(X)

Versions: 8K, Extended, Disk

Action: Returns the absolute value of the expression X.

Example: PRINT ABS(7*(-5))
35
Ok

3.2 ASC

Format: ASC(X\$)

Versions: 8K, Extended, Disk

Action: Returns a numerical value that is the ASCII code of the first character of the string X\$. (See Appendix M for ASCII codes.) If X\$ is null, an "Illegal function call" error is returned.

Example: 10 X\$ = "TEST"
20 PRINT ASC(X\$)
RUN
84
Ok

See the CHR\$ function for ASCII-to-string conversion.

3.3 ATN

Format: ATN(X)

Versions: 8K, Extended, Disk

Action: Returns the arctangent of X in radians. Result is in the range -pi/2 to pi/2. The expression X may be any numeric type, but the evaluation of ATN is always performed in single precision.

Example:

```
10 INPUT X
20 PRINT ATN(X)
RUN
? 3
1.24905
Ok
```

3.4 CDBL

Format: CDBL(X)

Versions: Extended, Disk

Action: Converts X to a double precision number.

Example:

```
10 A = 454.67
20 PRINT A;CDBL(A)
RUN
454.67 454.6700134277344
Ok
```

3.5 CHR\$

Format: CHR\$(I)

Versions: 8K, Extended, Disk

Action: Returns a string whose one element has ASCII code I. (ASCII codes are listed in Appendix M.) CHR\$ is commonly used to send a special character to the terminal. For instance, the BEL character could be sent (CHR\$(7)) as a preface to an error message, or a form feed could be sent (CHR\$(12)) to clear a CRT screen and return the cursor to the home position.

Example: PRINT CHR\$(66)

B

Ok

See the ASC function for ASCII-to-numeric conversion.

3.6 CINT

Format: CINT(X)

Versions: Extended, Disk

Action: Converts X to an integer by rounding the fractional portion. If X is not in the range -32768 to 32767, an "Overflow" error occurs.

Example: PRINT CINT(45.67)

46

Ok

See the CDBL and CSNG functions for converting numbers to the double precision and single precision data type. See also the FIX and INT functions, both of which return integers.

3.7 COS

Format: COS(X)

Versions: 8K, Extended, Disk

Action: Returns the cosine of X in radians. The calculation of COS(X) is performed in single precision.

Example:

```
10 X = 2*COS(.4)
20 PRINT X
RUN
 1.84212
Ok
```

3.8 CSNG

Format: CSNG(X)

Versions: Extended, Disk

Action: Converts X to a single precision number.

Example:

```
10 A# = 975.3421#
20 PRINT A#; CSNG(A#)
RUN
 975.3421  975.342
Ok
```

See the CINT and CDBL functions for converting numbers to the integer and double precision data types.

3.9 CVI, CVS, CVD

Format: CVI(<2-byte string>)
 CVS(<4-byte string>)
 CVD(<8-byte string>)

Version: Disk

Action: Convert string values to numeric values. Numeric values that are read in from a random disk file must be converted from strings back into numbers. CVI converts a 2-byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

Example:

```
.  
. .  
70 FIELD #1,4 AS N$, 12 AS B$, ...  
80 GET #1  
90 Y=CVS(N$)  
. .  
. .
```

See also MKI\$, MKS\$, MKD\$, Section 3.25 and Appendix B.

3.10 EOF

Format: EOF(<file number>)

Version: Disk

Action: Returns -1 (true) if the end of a sequential file has been reached. Use EOF to test for end-of-file while INPUTting, to avoid "Input past end" errors.

Example:

```
10 OPEN "I",1,"DATA"  
20 C=0  
30 IF EOF(1) THEN 100  
40 INPUT #1,M(C)  
50 C=C+1:GOTO 30  
. .  
. .
```

3.11 EXP**Format:** EXP(X)**Versions:** 8K, Extended, Disk**Action:** Returns e to the power of X. X must be <=87.3365. If EXP overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.**Example:**
10 X = 5
20 PRINT EXP (X-1)
RUN
54.5982
Ok**3.12 FIX****Format:** FIX(X)**Versions:** Extended, Disk**Action:** Returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT is that FIX does not return the next lower number for negative X.**Examples:**
PRINT FIX(58.75)
58
Ok
PRINT FIX(-58.75)
-58
Ok

3.13 FRE

Format: FRE(0)
FRE(X\$)

Versions: 8K, Extended, Disk

Action: Arguments to FRE are dummy arguments. FRE returns the number of bytes in memory not being used by BASIC-80.

FRE("") forces a garbage collection before returning the number of free bytes. BE PATIENT: garbage collection may take 1 to 1-1/2 minutes. BASIC will not initiate garbage collection until all free memory has been used up. Therefore, using FRE("") periodically will result in shorter delays for each garbage collection.

Example: PRINT FRE(0)
14542
Ok

3.14 HEX\$

Format: HEX\$(X)

Versions: Extended, Disk

Action: Returns a string which represents the hexadecimal value of the decimal argument. X is rounded to an integer before HEX\$(X) is evaluated.

Example: 10 INPUT X
20 A\$ = HEX\$(X)
30 PRINT X "DECIMAL IS " A\$ " HEXADECIMAL"
RUN
? 32
32 DECIMAL IS 20 HEXADECIMAL
Ok

See the OCT\$ function for octal conversion.

3.15 INKEY\$**Format:** INKEY\$**Action:** Returns either a one-character string containing a character read from the terminal or a null string if no character is pending at the terminal. No characters will be echoed and all characters are passed through to the program except for Control-C, which terminates the program. (With the BASIC Compiler, Control-C is also passed through to the program.)**Example:**
1000 'TIMED INPUT SUBROUTINE
1010 RESPONSE\$=""
1020 FOR I%=1 TO TIMELIMIT%
1030 A\$=INKEY\$: IF LEN(A\$)=0 THEN 1060
1040 IF ASC(A\$)=13 THEN TIMEOUT%=0 : RETURN
1050 RESPONSE\$=RESPONSE\$+A\$
1060 NEXT I%
1070 TIMEOUT%=1 : RETURN**3.16 INP****Format:** INP(I)**Versions:** 8K, Extended, Disk**Action:** Returns the byte read from port I. I must be in the range 0 to 255. INP is the complementary function to the OUT statement, Section 2.47.**Example:** 100 A=INP(255)

3.17 INPUT\$

Format: INPUT\$(X[,#]Y)

Version: Disk

Action: Returns a string of X characters, read from the terminal or from file number Y. If the terminal is used for input, no characters will be echoed and all control characters are passed through except Control-C, which is used to interrupt the execution of the INPUT\$ function.

Example 1: 5 'LIST THE CONTENTS OF A SEQUENTIAL FILE IN
HEXADECIMAL
10 OPEN "I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX\$(ASC(INPUT\$(1,#1)));
40 GOTO 20
50 PRINT
60 END

Example 2: .
. .
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X\$=INPUT\$(1)
120 IF X\$="P" THEN 500
130 IF X\$="S" THEN 700 ELSE 100
. . .

3.18 INSTR

Format: **INSTR([I,]X\$,Y\$)**

Versions: Extended, Disk

Action: Searches for the first occurrence of string Y\$ in X\$ and returns the position at which the match is found. Optional offset I sets the position for starting the search. I must be in the range 1 to 255. If I>LEN(X\$) or if X\$ is null or if Y\$ cannot be found, INSTR returns 0. If Y\$ is null, INSTR returns I or 1. X\$ and Y\$ may be string variables, string expressions or string literals.

Example: 10 X\$ = "ABCDEB"
 20 Y\$ = "B"
 30 PRINT INSTR(X\$,Y\$);INSTR(4,X\$,Y\$)
RUN
 2 6
Ok

NOTE: If I=0 is specified, error message "ILLEGAL ARGUMENT IN <line number>" will be returned.

3.19 INT

Format: INT(X)

Versions: 8K, Extended, Disk

Action: Returns the largest integer <=X.

Examples: PRINT INT(99.89)
99
Ok

PRINT INT(-12.11)
-13
Ok

See the FIX and CINT functions which also return integer values.

3.20 LEFT\$

Format: LEFT\$(X\$, I)

Versions: 8K, Extended, Disk

Action: Returns a string comprised of the leftmost I characters of X\$. I must be in the range 0 to 255. If I is greater than LEN(X\$), the entire string (X\$) will be returned. If I=0, the null string (length zero) is returned.

Example: 10 A\$ = "BASIC-80"
20 B\$ = LEFT\$(A\$,5)
30 PRINT B\$
BASIC
Ok

Also see the MID\$ and RIGHT\$ functions.

3.21 LEN

Format: LEN(X\$)

Versions: 8K, Extended, Disk

Action: Returns the number of characters in X\$. Non-printing characters and blanks are counted.

Example: 10 X\$ = "PORTLAND, OREGON"
20 PRINT LEN(X\$)
16
Ok

3.22 LOC

Format: LOC(<file number>)

Version: Disk

Action: With random disk files, LOC returns the record number just read or written from a GET or PUT. If the file was opened but no disk I/O has been performed yet, LOC returns a 0. With sequential files, LOC returns the number of sectors (128 byte blocks) read from or written to the file since it was OPENed.

Example: 200 IF LOC(1)>50 THEN STOP

3.23 LOG

Format: LOG(X)

Versions: 8K, Extended, Disk

Action: Returns the natural logarithm of X. X must be greater than zero.

Example: PRINT LOG(45/7)
1.86075
Ok

3.24 LPOS

Format: LPOS(X)

Versions: Extended, Disk

Action: Returns the current position of the line printer print head within the line printer buffer. Does not necessarily give the physical position of the print head. X is a dummy argument.

Example: 100 IF LPOS(X)>60 THEN LPRINT CHR\$(13)

3.25 MID\$

Format: MID\$(X\$, I[, J])

Versions: 8K, Extended, Disk

Action: Returns a string of length J characters from X\$ beginning with the Ith character. I and J must be in the range 1 to 255. If J is omitted or if there are fewer than J characters to the right of the Ith character, all rightmost characters beginning with the Ith character are returned. If I>LEN(X\$), MID\$ returns a null string.

Example:

```

LIST
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
Ok
RUN
GOOD EVENING
Ok

```

Also see the LEFT\$ and RIGHT\$ functions.

NOTE: If I=0 is specified, error message "ILLEGAL ARGUMENT IN <line number>" will be returned.

3.26 MKI\$, MKS\$, MKD\$

Format: MKI\$(<integer expression>)
 MKS\$(<single precision expression>)
 MKD\$(<double precision expression>)

Version: Disk

Action: Convert numeric values to string values. Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI\$ converts an integer to a 2-byte string. MKS\$ converts a single precision number to a 4-byte string. MKD\$ converts a double precision number to an 8-byte string.

Example:

```

90 AMT=(K+T)
100 FIELD #1, 8 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
  .
  .

```

See also CVI, CVS, CVD, Section 3.9 and Appendix B.

3.27 OCT\$

Format: OCT\$(X)

Versions: Extended, Disk

Action: Returns a string which represents the octal value of the decimal argument. X is rounded to an integer before OCT\$(X) is evaluated.

Example: PRINT OCT\$(24)
 30
Ok

See the HEX\$ function for hexadecimal conversion.

3.28 PEEK

Format: PEEK(I)

Versions: 8K, Extended, Disk

Action: Returns the byte (decimal integer in the range 0 to 255) read from memory location I. With the 8K version of BASIC-80, I must be less than 32768. To PEEK at a memory location above 32768, subtract 65536 from the desired address. With Extended and Disk BASIC-80, I must be in the range 0 to 65536. PEEK is the complementary function to the POKE statement, Section 2.48.

Example: A=PEEK(&H5A00)

3.29 POS

Format: POS(I)

Versions: 8K, Extended, Disk

Action: Returns the current cursor position. The leftmost position is 1. X is a dummy argument.

Example: IF POS(X)>60 THEN PRINT CHR\$(13)

Also see the LPOS function.

3.30 RIGHT\$

Format: RIGHT\$(X\$, I)

Versions: 8K, Extended, Disk

Action: Returns the rightmost I characters of string X\$. If I=LEN(X\$), returns X\$. If I=0, the null string (length zero) is returned.

Example: 10 A\$="DISK BASIC-80"
20 PRINT RIGHT\$(A\$,8)
RUN
BASIC-80
Ok

Also see the MID\$ and LEFT\$ functions.

3.31 RND

Format: RND[(X)]

Versions: 8K, Extended, Disk

Action: Returns a random number between 0 and 1. The same sequence of random numbers is generated each time the program is RUN unless the random number generator is reseeded (see RANDOMIZE, Section 2.53). However, X<0 always restarts the same sequence for any given X.

X>0 or X omitted generates the next random number in the sequence. X=0 repeats the last number generated.

Example:

```
10 FOR I=1 TO 5
20 PRINT INT(RND*100);
30 NEXT
RUN
 24  30  31  51  5
Ok
```

3.32 SGN

Format: SGN(X)

Versions: 8K, Extended, Disk

Action: If X>0, SGN(X) returns 1.
If X=0, SGN(X) returns 0.
If X<0, SGN(X) returns -1.

Example: ON SGN(X)+2 GOTO 100,200,300 branches to 100 if X is negative, 200 if X is 0 and 300 if X is positive.

3.33 SIN**Format:** SIN(X)**Versions:** 8K, Extended, Disk**Action:** Returns the sine of X in radians. SIN(X) is calculated in single precision.
 $\text{COS}(X) = \text{SIN}(X + 3.14159/2)$.**Example:** PRINT SIN(1.5)

.997495

Ok

3.34 SPACE\$**Format:** SPACE\$(X)**Versions:** Extended, Disk**Action:** Returns a string of spaces of length X. The expression X is rounded to an integer and must be in the range 0 to 255.**Example:** 10 FOR I = 1 TO 5

20 X\$ = SPACE\$(I)

30 PRINT X\$;I

40 NEXT I

RUN

1

2

3

4

5

Ok

Also see the SPC function.

3.35 SPC

Format: SPC(I)

Versions: 8K, Extended, Disk

Action: Prints I blanks on the terminal. SPC may only be used with PRINT and LPRINT statements. I must be in the range 0 to 255. A';' is assumed to follow the SPC(I) command.

Example: PRINT "OVER" SPC(15) "THERE"
OVER THERE
Ok

Also see the SPACES\$ function.

3.36 SQR

Format: SQR(X)

Versions: 8K, Extended, Disk

Action: Returns the square root of X. X must be ≥ 0 .

Example: 10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
10 3.16228
15 3.87298
20 4.47214
25 5
Ok

3.37 STR\$

Format: STR\$(X)

Versions: 8K, Extended, Disk

Action: Returns a string representation of the value of X.

Example:

```
5 REM ARITHMETIC FOR KIDS
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N)) GOSUB 30,100,200,300,400,500
.
.
.
```

Also see the VAL function.

3.38 STRING\$

Formats: STRING\$(I,J)
STRING\$(I,X\$)

Versions: Extended, Disk

Action: Returns a string of length I whose characters all have ASCII code J or the first character of X\$.

Example:

```
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
-----MONTHLY REPORT-----
Ok
```

3.39 TAB

Format: TAB(I)

Versions: 8K, Extended, Disk

Action: Spaces to position I on the terminal. If the current print position is already beyond space I, TAB goes to that position on the next line. Space 1 is the leftmost position, and the rightmost position is the width minus one. I must be in the range 1 to 255. TAB may only be used in PRINT and LPRINT statements.

Example:

```
10 PRINT "NAME" TAB(25) "AMOUNT" : PRINT
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "G. T. JONES","$25.00"
RUN
NAME                AMOUNT
G. T. JONES          $25.00
Ok
```

3.40 TAN

Format: TAN(X)

Versions: 8K, Extended, Disk

Action: Returns the tangent of X in radians. TAN(X) is calculated in single precision. If TAN overflows, the "Overflow" error message is displayed, machine infinity with the appropriate sign is supplied as the result, and execution continues.

Example: 10 Y = Q*TAN(X)/2

3.41 USR

Format : USR[<digit>] (X)

Versions: 8K, Extended, Disk

Action: Calls the user's assembly language subroutine with the argument X. <digit> is allowed in the Extended and Disk versions only. <digit> is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. See Appendix x.

Example:

```
40 B = T*SIN(Y)
50 C = USR(B/2)
60 D = USR(B/3)
```

•
•
•

3.42 VAL

Format: VAL(X\$)

Versions: 8K, Extended, Disk

Action: Returns the numerical value of string X\$. The VAL function also strips leading blanks, tabs, and linefeeds from the argument string. For example,

```
VAL("-3")
returns -3.
```

Example:

```
10 READ NAME$,CITY$,STATE$,ZIP$
20 IF VAL(ZIP$)<90000 OR VAL(ZIP$)>96699 THEN
PRINT NAME$ TAB(25) "OUT OF STATE"
30 IF VAL(ZIP$)>=90801 AND VAL(ZIP$)<=90815 THEN
PRINT NAME$ TAB(25) "LONG BEACH"
•
•
•
```

See the STR\$ function for numeric to string conversion.

3.43 VARPTR

Format 1: VARPTR(<variable name>)

Versions: Extended, Disk

Format 2: VARPTR(#<file number>)

Version: Disk

Action: Format 1: Returns the address of the first byte of data identified with <variable name>. A value must be assigned to <variable name> prior to execution of VARPTR. Otherwise an "Illegal function call" error results. Any type variable name may be used (numeric, string, array), and the address returned will be an integer in the range 32767 to -32768. If a negative address is returned, add it to 65536 to obtain the actual address.

VARPTR is usually used to obtain the address of a variable or array so it may be passed to an assembly language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

NOTE: All simple variables should be assigned before calling VARPTR for an array, because the addresses of the arrays change whenever a new simple variable is assigned.

Format 2: For sequential files, returns the starting address of the disk I/O buffer assigned to <file number>. For random files, returns the address of the FIELD buffer assigned to <file number>.

In Standalone Disk BASIC, VARPTR(#<file number>) returns the first byte of the file block. See Appendix H.

Example: 100 X=USR(VARPTR(Y))

APPENDIX A

New Features in BASIC-80, Release 5.0

The execution of BASIC programs written under Microsoft BASIC, release 4.51 and earlier may be affected by some of the new features in release 5.0. Before attempting to run such programs, check for the following:

1. New reserved words: CALL, CHAIN, COMMON, WHILE, WEND, WRITE, OPTION BASE, RANDOMIZE.
2. Conversion from floating point to integer values results in rounding, as opposed to truncation. This affects not only assignment statements (e.g., I%=2.5 results in I%=3), but also affects function and statement evaluations (e.g., TAB(4.5) goes to the 5th position, A(1.5) yeilds A(2), and X=11.5 MOD 4 yields 0 for X).
3. The body of a FOR...NEXT loop is skipped if the initial value of the loop times the sign of the step exceeds the final value times the sign of the step. See Section 2.22.
4. Division by zero and overflow no longer produce fatal errors. See Section 1.8.1.2.
5. The RND function has been changed so that RND with no argument is the same as RND with a positive argument. The RND function generates the same sequence of random numbers with each RUN, unless RANDOMIZE is used. See Sections 2.53 and 3.30.
6. The rules for PRINTing single precision and double precision numbers have been changed. See Section 2.49.
7. String space is allocated dynamically, and the first argument in a two-argument CLEAR statement sets the end of memory. The second argument sets the amount of stack space. See Section 2.4.

8. Responding to INPUT with too many or too few items, or with non-numeric characters instead of digits, causes the message "?Redo from start" to be printed. If a single variable is requested, a carriage return may be entered to indicate the default values of 0 for numeric input or null for string input. However, if more than one variable is requested, entering a carriage return will cause the "?Redo from start" message to be printed because too few items were entered. No assignment of input values is made until an acceptable response is given.
9. There are two new field formatting characters for use with PRINT USING. An ampersand is used for variable length string fields, and an underscore signifies a literal character in a format string.
10. If the expression supplied with the WIDTH statement is 255, BASIC uses an "infinite" line width, that is, it does not insert carriage returns. WIDTH LPRINT may be used to set the line width at the line printer. See Section 2.66.
11. The at-sign and underscore are no longer used as editing characters.
12. Variable names are significant up to 40 characters and can contain embedded reserved words. However, reserved words must now be delimited by spaces. To maintain compatibility with earlier versions of BASIC, spaces will be automatically inserted between adjoining reserved words and variable names. WARNING: This insertion of spaces may cause the end of a line to be truncated if the line length is close to 255 characters.
13. BASIC programs may be saved in a protected binary format. See SAVE, Section 2.60.

APPENDIX B

BASIC-80 Disk I/O

Disk I/O procedures for the beginning BASIC-80 user are examined in this appendix. If you are new to BASIC-80 or if you're getting disk related errors, read through these procedures and program examples to make sure you're using all the disk statements correctly.

Wherever a filename is required in a disk command or statement, use a name that conforms to your operating system's requirements for filenames. The CP/M operating system will append a default extension .BAS to the filename given in a SAVE, RUN, MERGE or LOAD command.

B.1 PROGRAM FILE COMMANDS

Here is a review of the commands and statements used in program file manipulation.

SAVE <filename>[,A] Writes to disk the program that is currently residing in memory. Optional A writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

LOAD <filename>[,R] Loads the program from disk into memory. Optional R runs the program immediately. LOAD always deletes the current contents of memory and closes all files before LOADING. If R is included, however, open data files are kept open. Thus programs can be chained or loaded in sections and access the same data files.

RUN <filename>[,R]	RUN <filename> loads the program from disk into memory and runs it. RUN deletes the current contents of memory and closes all files before loading the program. If the R option is included, however, all open data files are kept open.
MERGE <filename>	Loads the program from disk into memory but does not delete the current contents of memory. The program line numbers on disk are merged with the line numbers in memory. If two lines have the same number, only the line from the disk program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.
KILL <filename>	Deletes the file from the disk. <filename> may be a program file, or a sequential or random access data file.
NAME <old filename> AS<new filename>	To change the name of a disk file, execute the NAME statement, NAME <oldfile> AS <newfile>. NAME may be used with program files, random files, or sequential files.

B.2 PROTECTED FILES

If you wish to save a program in an encoded binary format, use the "Protect" option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed or edited. You may also want to save an unprotected copy of the program for listing and editing purposes.

B.3 DISK DATA FILES - SEQUENTIAL AND RANDOM I/O

There are two types of disk data files that may be created and accessed by a BASIC-80 program: sequential files and random access files.

B.3.1 Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored, one item after another (sequentially), in the order it is sent and is read back in the same way.

The statements and functions that are used with sequential files are:

OPEN	PRINT#	INPUT#	WRITE#
	PRINT# USING	LINE INPUT#	
CLOSE	EOF	LOC	

The following program steps are required to create a sequential file and access the data in the file:

- | | | |
|----|--|--------------------------------|
| 1. | OPEN the file in "O" mode. | OPEN "O",#1,"DATA" |
| 2. | Write data to the file
using the PRINT# statement.
(WRITE# may be used instead.) | PRINT#1,A\$;B\$;C\$ |
| 3. | To access the data in the
file, you must CLOSE the file
and reOPEN it in "I" mode. | CLOSE #1
OPEN "I",#1,"DATA" |
| 4. | Use the INPUT# statement to
read data from the sequential
file into the program. | INPUT#1,X\$,Y\$,Z\$ |

Program B-1 is a short program that creates a sequential file, "DATA", from information you input at the terminal.

```
10 OPEN "O",#1,"DATA"
20 INPUT "NAME";N$
25 IF N$="DONE" THEN END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 PRINT#1,N$;",";D$;",";H$
60 PRINT:GOTO 20
RUN
NAME? MICKEY MOUSE
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? etc.
```

PROGRAM B-1 - CREATE A SEQUENTIAL DATA FILE

Now look at Program B-2. It accesses the file "DATA" that was created in Program B-1 and displays the name of everyone hired in 1978.

```

10 OPEN "I",#1,"DATA"
20 INPUT#1,N$,D$,H$
30 IF RIGHTS$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEZER SCROOGE
SUPER MANN
Input past end in 20
Ok

```

PROGRAM B-2 - ACCESSING A SEQUENTIAL FILE

Program B-2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an "Input past end" error. To avoid getting this error, insert line 15 which uses the EOF function to test for end-of-file:

```
15 IF EOF(1) THEN END
```

and change line 40 to GOTO 15.

A program that creates a sequential file can also write formatted data to the disk with the PRINT# USING statement. For example, the statement

```
PRINT#1,USING"####.##,";A,B,C,D
```

could be used to write numeric data to disk without explicit delimiters. The comma at the end of the format string serves to separate the items in the disk file.

The LOC function, when used with a sequential file, returns the number of sectors that have been written to or read from the file since it was OPENed. A sector is a 128-byte block of data.

B.3.1.1 Adding Data To A Sequential File -

If you have a sequential file residing on disk and later want to add more data to the end of it, you cannot simply open the file in "O" mode and start writing data. As soon as you open a sequential file in "O" mode, you destroy its current contents. The following procedure can be used to add data to an existing file called "NAMES".

1. OPEN "NAMES" in "I" mode.
2. OPEN a second file called "COPY" in "O" mode.
3. Read in the data in "NAMES" and write it to "COPY".
4. CLOSE "NAMES" and KILL it.
5. Write the new information to "COPY".
6. Rename "COPY" as "NAMES" and CLOSE.
7. Now there is a file on disk called "NAMES" that includes all the previous data plus the new data you just added.

Program B-3 illustrates this technique. It can be used to create or add onto a file called NAMES. This program also illustrates the use of LINE INPUT# to read strings with embedded commas from the disk file. Remember, LINE INPUT# will read in characters from the disk until it sees a carriage return (it does not stop at quotes or commas) or until it has read 255 characters.

```
10 ON ERROR GOTO 2000
20 OPEN "I",#1,"NAMES"
30 REM IF FILE EXISTS, WRITE IT TO "COPY"
40 OPEN "O",#2,"COPY"
50 IF EOF(1) THEN 90
60 LINE INPUT#1,A$
70 PRINT#2,A$
80 GOTO 50
90 CLOSE #1
100 KILL "NAMES"
110 REM ADD NEW ENTRIES TO FILE
120 INPUT "NAME";N$
130 IF N$="" THEN 200 'CARRIAGE RETURN EXITS INPUT LOOP
140 LINE INPUT "ADDRESS? ";A$
150 LINE INPUT "BIRTHDAY? ";B$
160 PRINT#2,N$
170 PRINT#2,A$
180 PRINT#2,B$
190 PRINT:GOTO 120
200 CLOSE
205 REM CHANGE FILENAME BACK TO "NAMES"
210 NAME "COPY" AS "NAMES"
2000 IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"COPY":RESUME 120
2010 ON ERROR GOTO 0
```

PROGRAM B-3 - ADDING DATA TO A SEQUENTIAL FILE

The error trapping routine in line 2000 traps a "File does not exist" error in line 20. If this happens, the statements that copy the file are skipped, and "COPY" is created as if it were a new file.

B.3.2 Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. One advantage is that random files require less room on the disk, because BASIC stores them in a packed binary format. (A sequential file is stored as a series of ASCII characters.)

The biggest advantage to random files is that data can be accessed randomly, i.e., anywhere on the disk -- it is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records and each record is numbered.

The statements and functions that are used with random files are:

OPEN	FIELD	LSET/RSET	GET
PUT	CLOSE	LOC	
MKI\$	CVI		
MKSS\$	CVS		
MKD\$	CVD		

B.3.2.1 Creating A Random File -

The following program steps are required to create a random file.

1. OPEN the file for random access ("R" mode). This example specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

OPEN "R",#1,"FILE",32

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

FIELD #1 20 AS N\$,
4 AS A\$, 8 AS P\$
3. Use LSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI\$ to make an integer value into a string, MKSS\$ for a single precision value, and MKD\$ for a double precision value.

LSET N\$=X\$
LSET A\$=MKSS\$(AMT)
LSET P\$=TEL\$
4. Write the data from the buffer to the disk using the PUT statement.

PUT #1,CODE%

Look at Program B-4. It takes information that is input at the terminal and writes it to a random file. Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

NOTE

Do not use a FIELDED string variable in an INPUT or LET statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$:PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

PROGRAM B-4 - CREATE A RANDOM FILE

B.3.2.2 Access A Random File -

The following program steps are required to access a random file:

1. OPEN the file in "R" mode. OPEN "R",#1,"FILE",32
2. Use the FIELD statement to FIELD #1 20 AS N\$,
 allocate space in the random 4 AS A\$, 8 AS P\$
 buffer for the variables that
 will be read from the file.

NOTE:

In a program that performs both input and output on the same random file, you can often use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer. GET #1, CODE\$
 4. The data in the buffer may now be accessed by the program. Numeric values must be converted back to numbers using the "convert" functions: CVI for integers, CVS for single precision values, and CVD for double precision values. PRINT NS
PRINT CVS(A\$)

Program B-5 accesses the random file "FILE" that was created in Program B-4. By inputting the three-digit code at the terminal, the information associated with that code is read from the file and displayed.

```
10 OPEN "R",#1,"FILE",32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$##.##";CVS(A$)
70 PRINT P$:PRINT
80 GOTO 30
```

PROGRAM B-5 - ACCESS A RANDOM FILE

The LOC function, with random files, returns the "current record number." The current record number is one plus the last record number that was used in a GET or PUT statement. For example, the statement

IF LOC(1)>50 THEN END

ends program execution if the current record number in file#1 is higher than 50.

Program B-6 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 900-960 initialize the data file by writing CHR\$(255) as the first character of each record. This is used later (line 270 and line 500) to determine whether an entry already exists for that part number.

Lines 130-220 display the different inventory functions that the program performs. When you type in the desired function number, line 230 branches to the appropriate subroutine.

```
120 OPEN "R", #1, "INVEN.DAT", 39
125 FIELD#1,1 AS F$, 30 AS D$, 2 AS Q$, 2 AS R$, 4 AS P$
130 PRINT:PRINT "FUNCTIONS"::PRINT
135 PRINT 1, "INITIALIZE FILE"
140 PRINT 2, "CREATE A NEW ENTRY"
150 PRINT 3, "DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4, "ADD TO STOCK"
170 PRINT 5, "SUBTRACT FROM STOCK"
180 PRINT 6, "DISPLAY ALL ITEMS BELOW REORDER LEVEL"
220 PRINT:PRINT:INPUT "FUNCTION"; FUNCTION
225 IF (FUNCTION<1) OR (FUNCTION>6) THEN PRINT
    "BAD FUNCTION NUMBER":GO TO 130
230 ON FUNCTION GOSUB 900, 250, 390, 480, 560, 680
240 GOTO 220
250 REM BUILD NEW ENTRY
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT "OVERWRITE"; A$:
    IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION"; DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK"; Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL"; R%
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE"; P
360 LSET P$=MKSS$(P)
370 PUT#1, PART%
380 RETURN
390 REM DISPLAY ENTRY
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
420 PRINT USING "PART NUMBER ####"; PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####"; CVI(Q$)
450 PRINT USING "REORDER LEVEL #####"; CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##"; CVS(P$)
470 RETURN
480 REM ADD TO STOCK
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
510 PRINT D$: INPUT "QUANTITY TO ADD "; A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT#1, PART%
550 RETURN
560 REM REMOVE FROM STOCK
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY":RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT"; S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY"; Q%; " IN STOCK":GOTO 600
630 Q%=Q%-S%
```

```
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";Q%;
      " REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT#1,PART%
670 RETURN
680 DISPLAY ITEMS BELOW REORDER LEVEL
690 FOR I=1 TO 100
710 GET#1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;" QUANTITY";
      CVI(Q$) TAB(50) "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF(PART%<1)OR(PART%>100) THEN PRINT "BAD PART NUMBER":
      GOTO 840 ELSE GET#1,PART%:RETURN
890 END
900 REM INITIALIZE FILE
910 INPUT "ARE YOU SURE";B$:IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
940 PUT#1,I
950 NEXT I
960 RETURN
```

PROGRAM B-6 - INVENTORY

APPENDIX C

Assembly Language Subroutines

All versions of BASIC-80 have provisions for interfacing with assembly language subroutines. The USR function allows assembly language subroutines to be called in the same way BASIC's intrinsic functions are called.

NOTE

The addresses of the DEINT, GIVABF, MAKINT and FRCINT routines are stored in locations that must be supplied individually for different implementations of BASIC.

C.1 MEMORY ALLOCATION

Memory space must be set aside for an assembly language subroutine before it can be loaded. During initialization, enter the highest memory location minus the amount of memory needed for the assembly language subroutine(s). BASIC uses all memory available from its starting location up, so only the topmost locations in memory can be set aside for user subroutines.

When an assembly language subroutine is called, the stack pointer is set up for 8 levels (16 bytes) of stack storage. If more stack space is needed, BASIC's stack can be saved and a new stack set up for use by the assembly language subroutine. BASIC's stack must be restored, however, before returning from the subroutine.

The assembly language subroutine may be loaded into memory by means of the system monitor, or the BASIC POKE statement, or (if the user has the MACRO-80 or FORTRAN-80 package) routines may be assembled with MACRO-80 and loaded using LINK-80.

C.2 USR FUNCTION CALLS - 8K BASIC

The starting address of the assembly language subroutine must be stored in USRLOC, a two-byte location in memory that is supplied individually with different implementations of BASIC-80. With 8K BASIC, the starting address may be POKEd into USRLOC. Store the low order byte first, followed by the high order byte.

The function USR will call the routine whose address is in USRLOC. Initially USRLOC contains the address of ILLFUN, the routine that gives the "Illegal function call" error. Therefore, if USR is called without changing the address in USRLOC, an "Illegal function call" error results.

The format of a USR function call is

USR(argument)

where the argument is a numeric expression. To obtain the argument, the assembly language subroutine must call the routine DEINT. DEINT places the argument into the D,E register pair as a 2-byte, 2's complement integer. (If the argument is not in the range -32768 to 32767, an "Illegal function call" error occurs.)

To pass the result back from an assembly language subroutine, load the value in register pair [A,B], and call the routine GIVABF. If GIVABF is not called, USR(X) returns X. To return to BASIC, the assembly language subroutine must execute a RET instruction.

For example, here is an assembly language subroutine that multiplies the argument by 2:

```
USRSUB: CALL DEINT      ;put arg in D,E
        XCHG            ;move arg to H,L
        DAD H             ;H,L=H,L+H,L
        MOV A,H           ;move result to A,B
        MOV B,L
        JMP GIVABF       ;pass result back and RETurn
```

Note that valid results will be obtained from this routine for arguments in the range $-16384 \leq x \leq 16383$. The single instruction JMP GIVABF has the same effect as:

```
CALL GIVABF
RET
```

To return additional values to the program, load them into memory and read them with the PEEK function.

There are several methods by which a program may call more than one USR routine. For example, the starting address of each routine may be POKEd into USRLOC prior to each USR call, or the argument to USR could be an index into a table of USR routines.

C.3 USR FUNCTION CALLS - EXTENDED AND DISK BASIC

In the Extended and Disk versions, the format of the USR function is

USR[<digit>](argument)

where DIGIT> is from 0 to 9 and the argument is any numeric or string expression. <digit> specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If <digit> is omitted, USR0 is assumed. The address given in the DEF USR statement determines the starting address of the subroutine.

When the USR function call is made, register A contains a value that specifies the type of argument that was given. The value in A may be one of the following:

<u>Value in A</u>	<u>Type of Argument</u>
2	Two-byte integer (two's complement)
3	String
4	Single precision floating point number
8	Double precision floating point number

If the argument is a number, the [H,L] register pair points to the Floating Point Accumulator (FAC) where the argument is stored.

If the argument is an integer:

FAC-3 contains the lower 8 bits of the argument and
FAC-2 contains the upper 8 bits of the argument.

If the argument is a single precision floating point number:

FAC-3 contains the lowest 8 bits of mantissa and

FAC-2 contains the middle 8 bits of mantissa and FAC-1 contains the highest 7 bits of mantissa with leading 1 suppressed (implied). Bit 7 is the sign of the number (0=positive, 1=negative). FAC is the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa.

If the argument is a double precision floating point number:

FAC-7 through FAC-4 contain four more bytes of mantissa (FAC-7 contains the lowest 8 bits).

If the argument is a string, the [D,E] register pair points to 3 bytes called the "string descriptor." Byte 0 of the string descriptor contains the length of the string (0 to 255). Bytes 1 and 2, respectively, are the lower and upper 8 bits of the string starting address in string space.

CAUTION: If the argument is a string literal in the program, the string descriptor will point to program text. Be careful not to alter or destroy your program this way. To avoid unpredictable results, add "+" to the string literal in the program. Example:

```
A$ = "BASIC-80""
```

This will copy the string literal into string space and will prevent alteration of program text during a subroutine call.

Usually, the value returned by a USR function is the same type (integer, string, single precision or double precision) as the argument that was passed to it. However, calling the MAKINT routine returns the integer in [H,L] as the value of the function, forcing the value returned by the function to be integer. To execute MAKINT, use the following sequence to return from the subroutine:

```
PUSH    H      ;save value to be returned
LHLD    xxx    ;get address of MAKINT routine
XTHL          ;save return on stack and
              ;get back [H,L]
RET           ;return
```

Also, the argument of the function, regardless of its type, may be forced to an integer by calling the FRCINT routine to get the integer value of the argument in [H,L]. Execute the following routine:

```
LXI    H      ;get address of subroutine
              ;continuation
PUSH    H      ;place on stack
LHLD    xxx    ;get address of FRCINT
PCHL
SUB1: . . . . .
```

C.4 CALL STATEMENT

Extended and Disk BASIC-80 user function calls may also be made with the CALL statement. The calling sequence used is the same as that in Microsoft's FORTRAN, COBOL and BASIC compilers.

A CALL statement with no arguments generates a simple "CALL" instruction. The corresponding subroutine should return via a simple "RET." (CALL and RET are 8080 opcodes - see an 8080 reference manual for details.)

A subroutine CALL with arguments results in a somewhat more complex calling sequence. For each argument in the CALL argument list, a parameter is passed to the subroutine. That parameter is the address of the low byte of the argument. Therefore, parameters always occupy two bytes each, regardless of type.

The method of passing the parameters depends upon the number of parameters to pass:

1. If the number of parameters is less than or equal to 3, they are passed in the registers. Parameter 1 will be in HL, 2 in DE (if present), and 3 in BC (if present).
2. If the number of parameters is greater than 3, they are passed as follows:
 1. Parameter 1 in HL.
 2. Parameter 2 in DE.
 3. Parameters 3 through n in a contiguous data block. BC will point to the low byte of this data block (i.e., to the low byte of parameter 3).

Note that, with this scheme, the subroutine must know how many parameters to expect in order to find them. Conversely, the calling program is responsible for passing the correct number of parameters. There are no checks for the correct number or type of parameters.

If the subroutine expects more than 3 parameters, and needs to transfer them to a local data area, there is a system subroutine which will perform this transfer. This argument transfer routine is named \$AT (located in the FORTRAN library, FORLIB.REL), and is called with HL pointing to the local data area, BC pointing to the third parameter, and A containing the number of arguments to transfer (i.e., the total number of arguments minus 2). The subroutine is

responsible for saving the first two parameters before calling \$AT. For example, if a subroutine expects 5 parameters, it should look like:

```

SUBR: SHLD    P1      ;SAVE PARAMETER 1
      XCHG
      SHLD    P2      ;SAVE PARAMETER 2
      MVI     A,3      ;NO. OF PARAMETERS LEFT
      LXI     H,P3      ;pointer to LOCAL AREA
      CALL    $AT      ;TRANSFER THE OTHER 3 PARAMETERS
      .
      .
      .
      [Body of subroutine]
      .
      .
      .
      RET      ;RETURN TO CALLER
P1:   DS      2      ;SPACE FOR PARAMETER 1
P2:   DS      2      ;SPACE FOR PARAMETER 2
P3:   DS      6      ;SPACE FOR PARAMETERS 3-5

```

A listing of the argument transfer routine \$AT follows.

```

00100  ;      ARGUMENT TRANSFER
00200  ;[B,C]  POINTS TO 3RD PARAM.
00300  ;[H,L]  POINTS TO LOCAL STORAGE FOR PARAM 3
00400  ;[A]    CONTAINS THE # OF PARAMS TO XFER (TOTAL-2)
00500
00600
00700  ENTRY    $AT
00800  $AT:    XCHG      ;SAVE [H,L] IN [D,E]
00900  MOV       H,B
01000  MOV       L,C      ;[H,L] = PTR TO PARAMS
01100  AT1:    MOV       C,M
01200  INX       H
01300  MOV       B,M
01400  INX       H      ;[B,C] = PARAM ADR
01500  XCHG      ;[H,L] POINTS TO LOCAL STORAGE
01600  MOV       M,C
01700  INX       H
01800  MOV       M,B
01900  INX       H      ;STORE PARAM IN LOCAL AREA
02000  XCHG      ;SINCE GOING BACK TO AT1
02100  DCR       A      ;TRANSFERRED ALL PARAMS?
02200  JNZ       AT1      ;NO, COPY MORE
02300  RET      ;YES, RETURN

```

When accessing parameters in a subroutine, don't forget that they are pointers to the actual arguments passed.

NOTE

It is entirely up to the programmer to see to it that the arguments in the calling program match in number, type, and length with the parameters expected by the subroutine. This applies to BASIC subroutines, as well as those written in assembly language.

C.5 INTERRUPTS

Assembly language subroutines can be written to handle interrupts. All interrupt handling routines should save the stack, register A-L and the PSW. Interrupts should always be re-enabled before returning from the subroutine., since an interrupt automatically disables all further interrupts once it is received. The user should be aware of which interrupt vectors are free in the particular version of BASIC that has been supplied. (Note to CP/M users: In CP/M BASIC, all interrupt vectors are free.)

APPENDIX D

BASIC-80 with the CP/M Operating System

The CP/M version of BASIC-80 (MBASIC) is supplied on a standard size 3740 single density diskette. The name of the file is MBASIC.COM. (A 28K or larger CP/M system is recommended.)

To run MBASIC, bring up CP/M and type the following:

```
A>MBASIC <carriage return>
```

The system will reply:

```
xxxx Bytes Free  
BASIC-80 Version 5.0  
(CP/M Version)  
Copyright 1978 (C) by Microsoft  
Created: dd-mmm-yy  
Ok
```

MBASIC is the same as Disk BASIC-80 as described in this manual, with the following exceptions:

D.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the MBASIC command to CP/M. The format of the command line is:

```
A>MBASIC [<filename>] [/F:<number of files>] [/M:<highest memory location>]  
[ /S:<maximum record size> ]
```

If <filename> is present, MBASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied and the filename is less than 9 characters long. This allows BASIC programs to be executed in batch mode using the SUBMIT facility of CP/M. Such programs should include a SYSTEM statement (see below) to return to CP/M when they have finished, allowing the next program in the

batch stream to execute.

If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. Each file data block allocated in this fashion requires 166 bytes of memory. If the /F option is omitted, the number of files defaults to 3.

The /M:<highest memory location> option sets the highest memory location that will be used by MBASIC. In some cases it is desirable to set the amount of memory well below the CP/M's FDOS to reserve space for assembly language subroutines. In all cases, <highest memory location> should be below the start of FDOS (whose address is contained in locations 6 and 7). If the /M option is omitted, all memory up to the start of FDOS is used.

/S:<maximum record size> may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

NOTE

<number of files>, <highest memory location>, and <maximum record size> are numbers that may be either decimal, octal (preceded by &O) or hexadecimal (preceded by &H).

Examples:

A>MBASIC PAYROLL.BAS Use all memory and 3 files,
load and execute PAYROLL.BAS.

A>MBASIC INVENT/F:6 Use all memory and 6 files,
load and execute INVENT.BAS.

A>MBASIC /M:32768 Use first 32K of memory and
3 files.

A>MBASIC DATAACK/F:2/M:&H9000
 Use first 36K of memory, 2
files, and execute DATAACK.BAS.

D.2 DISK FILES

Disk filenames follow the normal CP/M naming conventions. All filenames may include A: or B: as the first two characters to specify a disk drive, otherwise the currently selected drive is assumed. A default extension of .BAS is

used on LOAD, SAVE, MERGE and RUN <filename> commands if no "." appears in the filename and the filename is less than 9 characters long.

For systems with CP/M 2.x, large random files are supported. The maximum logical record number is 32767. If a record size of 256 is specified, then files up to 8 megabytes can be accessed.

D.3 FILES COMMAND

Format: FILES[<filename>]

Purpose: To print the names of files residing on the current disk.

Remarks: If <filename> is omitted, all the files on the currently selected drive will be listed. <filename> is a string formula which may contain question marks (?) to match any character in the filename or extension. An asterisk (*) as the first character of the filename or extension will match any file or any extension.

Examples:
FILES
FILES "*.BAS"
FILES "B:*.*"
FILES "TEST?.BAS"

D.4 RESET COMMAND

Format: RESET

Purpose: To close all disk files and write the directory information to a diskette before it is removed from a disk drive.

Remarks: Always execute a RESET command before removing a diskette from a disk drive. Otherwise, when the diskette is used again, it will not have the current directory information written on the directory track.

RESET closes all open files on all drives and writes the directory track to every diskette with open files.

D.5 LOF FUNCTION

Format: LOF(<file number>)

Action: Returns the number of records present in the last extent read or written. If the file does not exceed one extent (128 records), then LOF returns the true length of the file.

Example: 110 IF NUM%>LOF(1) THEN PRINT "INVALID ENTRY"

D.6 EOF

With CP/M, the EOF function may be used with random files. If a GET is done past the end of file, EOF will return -1. This may be used to find the size of a file using a binary search or other algorithm.

D.7 MISCELLANEOUS

1. CSAVE and CLOAD are not implemented.
2. To return to CP/M, use the SYSTEM command or statement. SYSTEM closes all files and then performs a CP/M warm start. Control-C always returns to MBASIC, not to CP/M.
3. FRCINT is at 103 hex and MAKINT is at 105 hex. (Add 1000 hex for ADDS versions, 4000 for SBC CP/M versions.)

APPENDIX E

BASIC-80 with the ISIS-II Operating System

With ISIS-II, BASIC-80 is the same as described in this manual, with the following exceptions:

E.1 INITIALIZATION

The initialization dialog has been replaced by a set of options which are placed after the MBASIC command to ISIS-II. The format of the command line is:

```
-MBASIC [<filename>] [/F:<number of files>] [/M:<highest memory location>]  
[ /S:<maximum record size>]
```

If <filename> is present, BASIC proceeds as if a RUN <filename> command were typed after initialization is complete. A default extension of .BAS is used if none is supplied.

If /F:<number of files> is present, it sets the number of disk data files that may be open at any one time during the execution of a BASIC program. The maximum is six and the default is three. The /M:<highest memory location> option sets the highest memory location that will be used by BASIC. Use this option to reserve memory locations above BASIC for assembly language subroutines. /S:<maximum record size> may be added at the end of the command line to set the maximum record size for use with random files. The default record size is 128 bytes.

At initialization, the system will reply:

```
xxxx Bytes Free  
BASIC-80 Version x.x  
(ISIS-II Version)  
Copyright 1978 (C) by Microsoft
```

E.2 LINE PRINTER I/O

To send output to the printer during execution of a BASIC program, open the line printer as if it were a disk file:

```
50 N=4
100 OPEN "O",N,:LP:
.
.
.
120 PRINT #N,A,B,C
```

Since BASIC buffers disk I/O, you may want to force buffers out by CLOSEing the printer channel.

To LIST a program on the line printer, use:

```
SAVE":LP:",A
```

E.3 ATTRIB STATEMENT

In ISIS-II BASIC-80, the ATTRIB statement sets file attributes. The format of the statement is:

```
ATTRIB <filename string>,<attribute string>
```

The attribute string consists of F, W, S or I for the attribute, followed by a 1 to set the attribute or a 0 to reset.

Examples:

```
ATTRIB "INFO.DAT","W1"
ATTRIB "GHOST.BAS","I1"
ATTRIB ":F1:SYSFIL","W1F1S1I1"
ATTRIB A$,B$
```

E.4 MISCELLANEOUS

Note these other differences for ISIS-II BASIC:

1. MAKINT is located at 3903 hex, and GIVINT is located at 3905 hex.
2. There is no FILES command in ISIS-II BASIC. Filenames do not default to .BAS on SAVEs, LOADs, and MERGEs.

APPENDIX F

BASIC-80 with the TEKDOS Operating System

The operation of BASIC-80 with the TEKDOS operating system is the same as described in this manual with the following exceptions:

1. At initialization, BASIC asks MEMORY SIZE? If you respond with a carriage return, BASIC will use all available memory. If you respond with a memory location (in decimal), BASIC will use memory only up to that location. This lets you reserve space at the top of memory for assembly language subroutines.
2. The number of disk files that may be open at one time defaults to 5.
3. LPRINT and LLIST are not implemented. Instead, open a file to the printer.
4. TEKDOS does not support random disk I/O. The corresponding BASIC-80 statements (PUT, GET, OPEN"R", etc.) are inoperable under TEKDOS.
5. Control-C works only once due to a bug in TEKDOS. If you interrupt a running program or a LIST command with Control-C, BASIC appears to be in "single statement" mode. To clear this condition, exit BASIC with a SYSTEM command and re-enter BASIC with an XEQ BASIC. Avoid using the AUTO command, since it requires a Control-C to return to BASIC command level.

APPENDIX G

BASIC-80 with the INTEL SBC and MDS Systems

G.1 INITIALIZATION

The paper tape of BASIC-80 supplied for SBC and MDS systems is in Intel-compatible hex format. Use the monitor's R command to load the tape, then execute the G command to start BASIC-80. The command is:

.G4000

BASIC will respond:

Memory size?

If you want BASIC to use all available RAM, just type a carriage return. If you want to reserve space at the top of memory for machine language subroutines, enter the highest memory address (in decimal) that BASIC may use.

Terminal Width?

(8K versions only) Respond with the number of characters for the output line width in PRINT statements. The default is 72 characters. (Extended versions use WIDTH command.)

Want SIN-COS-TAN-ATN?

Type Y to retain these functions, type N to delete them, or type A to delete ATN only.

G.2 SUBROUTINE ADDRESSES

In the 8K version of SBC and MDS BASIC-80, DEINT is located at 0043 hex and GIVABF is located at 0045 hex. USRLOC is at xxxx hex. In the Extended version, FRCINT is located at xxxx hex, and MAKINT is located at xxxx hex.

G.3 LLIST AND LLPRINT

LLIST and LPRINT are not implemented.

APPENDIX H

Standalone Disk BASIC

Standalone Disk BASIC is an easily implemented, self-contained version of BASIC-80 that runs on almost any 8080 or Z80 based disk hardware without an operating system. Standalone Disk BASIC incorporates several unique disk I/O methods that make faster and more efficient use of disk access and storage.

Random access with Standalone BASIC is faster than other disk operating systems because the file allocation table is kept in memory and updated periodically on the diskette. Therefore, there is no need for index blocks for random files, and there is no need to distinguish between random and sequential files. Because there are no index blocks, there is no large per-file-overhead either in memory or on disk. Binary SAVES and LOADS are also faster because they are optimized by cluster, i.e., an entire cluster is read or written at one time, instead of a single sector.

To initialize Standalone Disk BASIC, insert the BASIC diskette and power up the system. In one- or two-drive systems, BASIC asks if there are two drives. In systems with more than two drives, BASIC asks for the number of drives. BASIC then asks how many files, i.e., how many disk files may be open at one time. Answer with a number from 0 to 15, or, for a default of 1 file per drive, just enter a carriage return.

The operation of Standalone Disk BASIC is the same as Disk BASIC-80 as described in this manual, with the following exceptions:

H.1 FILENAMES

The format for disk filenames is:

[drive#:]filename[.extension]

The first drive is 1.

Disk filenames are six characters with an optional three-character extension that is preceded by a decimal point. If a decimal point appears in a filename after fewer than six characters, the name is blank-filled to six characters and the next three characters are the extension. If the filename is six or fewer characters with no decimal point, there is no extension. If the filename is more than six characters, BASIC inserts a decimal point after the sixth character and uses the next three characters as an extension. (Any additional characters are ignored.)

H.2 DISK FILES

The FILES command prints the names of the files residing on a disk. The format is: [L]FILES[<drive number>] LFILES outputs to the line printer. In addition to the filename, the size of each file, in clusters, is output. A cluster is the minimum unit of allocation for a file -- it is one-half of a track. Filenames of files created with OPEN or ASCII SAVE are listed with a space between the name and extension. Filenames of binary files created with binary SAVE are listed with a decimal point between the name and extension. The protected file option with SAVE is not supported in Standalone Disk BASIC.

H.3 FPOS

The FPOS function:

`FPOS(<file number>)`

is the same as BASIC-80's LOC function except it returns the number of the physical sector where <filenumber> is located. (BASIC-80's LOC function and CP/M BASIC-80's LOF function are also implemented.)

H.4 DSKI\$/DSKO\$

The DSKO\$ statement:

`DSKO$<drive>,<track>,<sector>,<string expression>`

writes the string on the specified sector. The maximum length for the string is 128 characters. A string of fewer than 128 characters is zero-filled at the end to 128 characters.

DSKI\$ is the complementary function to the DSKO\$ statement. DSKI\$ returns the contents of a sector to a string variable name. The format is:

```
DSKI$(<drive>,<track>,<sector>)
```

Example: A\$=DSKI\$(0,I,J)

H.5 MOUNT COMMAND

Before a diskette can be used for file operations (i.e., any disk I/O besides DSKI\$, DSKO\$, or IBM or USR modes), it must be MOUNTed. The format of the command is:

```
MOUNT[<drive>[,<drive>...]]
```

MOUNT with no arguments mounts all drives. When a diskette is mounted, BASIC reads the File Allocation Table (see Section H.11.2) from the diskette into memory and checks it for errors. If there are no errors, the disk is mounted. If an error is found, BASIC reads one or both of the back-up allocation tables from the diskette in an attempt to mount the disk; and a warning message, "x copies of allocation bad on drive y", is issued. x is 1 or 2 and y is the drive number. When a warning occurs, it is a good idea to make a new copy of the diskette. If all copies of the allocation table are bad or if a free entry is encountered in the file chain, a fatal error--"bad allocation table"--is given and the diskette will not be mounted.

While a disk is mounted, BASIC occasionally writes the allocation table to the directory track, but it does not check for errors unless the read after write attribute is set for that drive (see SET statement).

H.6 REMOVE COMMAND

REMOVE is the complement of MOUNT. Before a diskette can be taken out of the drive, a REMOVE command must be executed. The format of the command is:

```
REMOVE[<drive>[,<drive>...]]
```

REMOVE writes three copies of the current allocation table to disk and follows the same error-check procedure as MOUNT. MOUNT and REMOVE replace the RESET command that is in BASIC-80.

NOTE

ALWAYS do a REMOVE before taking a diskette out of a drive. If you do not, the diskette you took out will not have an updated and checked allocation table, and the data on the next diskette inserted will be destroyed when the wrong allocation table is written to the directory track.

H.7 SET STATEMENT

The SET statement determines the attributes of the currently mounted disk drive, a currently open file, or a file that need not be open. The format of the SET statement is:

SET<drive> | #<file> | <filename>,<attribute string>

<attribute string> is a string of characters that determines what attributes are set. Any characters other than the following are ignored:

R Read after write
P Write protect
E EBCDIC conversion (if available)

Attributes are assigned in the following order:

1. MOUNT command

When a MOUNT is done for a particular drive, the first byte of the information sector on the diskette (track 35, sector 20 for floppy; track 18, sector 13 for minifloppy) contains the attributes for the disk. (octal values: R=100, P=20, E=40)

2. SET<drive>,<attribute string> Statement

This statement sets the current attributes for the disk, in memory, while it is mounted. The attributes are not permanently recorded and apply only while the disk is mounted.

3. When a file is created, the permanent file attributes recorded on the disk will be the same as the current drive attributes.

4. SET<filename>,<attribute string> Statement
This statement changes the permanent file attributes that are stored in the directory entry for that file. It does not affect the drive attributes.
5. When an existing file is OPENed, the attributes of the file number are those of the directory entry.
6. SET#<file number>,<attribute string> Statement
This statement changes the attributes for that file number but does not change the directory entry.

Examples:

SET 1,"R"	Force read after write checking on all output to drive 1
SET #1,"R"	Force read after write for all output to file 1 while it is open
SET #1,"P"	Give write protect error if any output is attempted to file 1
SET "TEST","P"	Protect TEST from deletion and modification
SET 1,""	Turn off all attributes for drive 1

H.8 ATTR\$ FUNCTION

ATTR\$ returns a string of the current attributes for a drive, currently open file, or file that need not be open. The format of ATTR\$ is:

```
ATTR$(<drive> | #<file number> | <filename>)
```

For example:

```
SET 1,"R":A$=ATTR$(1):PRINT A$  
R  
Ok
```

H.9 OPEN STATEMENT

The format for the OPEN statement in Standalone BASIC is:

```
OPEN <filename> [FOR <mode>] AS [#]<file number>
```

where <mode> is one of the following:

```
INPUT  
OUTPUT  
APPEND  
IBM  
USR
```

The mode determines only the initial positioning within the file and the actions to be taken if the file does not exist. The action taken in each mode is:

INPUT	The initial position is at the start of the file. An error is returned if the file is not found.
OUTPUT	The initial position is at the start of the file. A new file is always created.
APPEND	The initial position is at the end of the file. An error is returned if the file is not found.
IBM	The initial position is after the last DSKI\$ or DSKO\$. The file is then set up to write contiguous. No file search is done. (The same effect may be achieved in many cases by altering the FORMAT program. See Section H.11.2.1.)
USR	Same as IBM mode except, instead of write contiguous, USR0 is called and returns the next track/sector number. The USR0 routine should read the current track/sector from B,C and return the next location in B,C. When USR0 is first called, B,C contains the track and sector number of the previous DSKI\$ or DSKO\$.

If the FOR <mode> clause is omitted, the initial position is at the start of the file. If the file is not found, it is created.

Note that variable length records are not supported in Standalone Disk BASIC. All records are 128 bytes in length.

USR mode is especially useful for creating diskettes that require sector mapping. This is the case if the diskette is intended for use on another system, for example, a CP/M system. Instead of opening the file for write contiguous (IBM mode), the USR0 routine may be used to map the sectors logically, as required by the other system.

When a file is OPENed FOR APPEND, the file mode is set to APPEND and the record number is set to the last record of the file. The program may subsequently execute disk I/O statements that move the pointer elsewhere in the file. When the last record is read, the file mode is reset to FILE and the pointer is left at the end of the file. Then, if you wish to append another record, execute:

```
GET#n,LOF(n)
```

This positions the pointer at the end of the file in preparation for appending.

At any one time, it is possible to have a particular

filename OPEN under more than one file number. This allows different attributes to be used for different purposes. Or, for program clarity, you may wish to use different file numbers for different methods of access. Each file number has a different buffer, so changes made under one file are not accessible to (or affected by) the other numbers until that record is written (e.g., GET#n,LOC(n)).

H.10 DISK I/O

A GET or PUT (i.e., random access) cannot be done on a file that is OPEN FOR IBM or OPEN FOR USR. Otherwise, GET/PUT may be executed along with PRINT#/INPUT# on the same file, which makes midfile updating possible. The statement formats for GET, PUT, PRINT#, and INPUT# are the same as those in BASIC-80. The action of each statement in Standalone BASIC is as follows:

- | | |
|--------|--|
| GET | If the "buffer changed" flag is set, write the buffer to disk. Then execute the GET (read the record into the buffer), and reset the position for sequential I/O to the beginning of the buffer. |
| PUT | Execute the PUT (write the buffer to the specified record number), and set the "sequential I/O is illegal" flag until a GET is done. |
| INPUT# | If the buffer is empty, write it if the "buffer changed" flag is set, then read the next buffer. |
| PRINT# | Set the "buffer changed" flag. If the buffer is full, write it to disk. Then, if end of file has not been reached, read the next buffer. |

H.10.1 File Format

For a single density floppy, each file requires 137 bytes: 9 bytes plus the 128-byte buffer. Because the File Allocation Table keeps random access information for all files, random and sequential files are identical on the disk. The only distinction is that sequential files have a Control-Z (32 octal) as the last character of the last sector. When this sector is read, it is scanned from the end for a non-zero byte. If this byte is Control-Z, the size of the buffer is set so that a PRINT overwrites this byte. If the byte is not Control-Z, the size is set so the last null seen is overwritten.

Any sequential file can be copied in random mode and remain identical. If a file is written to disk in random mode

(i.e., with PUT instead of PRINT) and then read in sequential mode, it will still have proper end of file detection.

H.11 DISK ALLOCATION INFORMATION

With Standalone Disk BASIC, storage space on the diskette is allocated beginning with the cluster closest to the current position of the head. (This method is optimized for writing. Custom versions can be optimized for reading.) Disk allocation information is placed in memory when the disk is mounted and is periodically written back to the disk. Because this allocation information is kept in memory, there is no need for index blocks for random files, and there is no need to distinguish between random and sequential files.

H.11.1 Directory Format

On the diskette, each sector of the directory track contains eight file entries. Each file entry is 16 bytes long and formatted as follows:

<u>Bytes</u>	<u>Usage</u>
0-8	Filename, 1 to 9 characters. The first character may not be 0 or 255.
9	Attribute: Octal 200 Binary file 100 Force read after write check 40 EBCDIC file 20 Write protected file Excluding 200, these bits are the same for the disk attribute byte which is the first byte of the information sector.
10	Pointer into File Allocation Table to the first cluster of the file's cluster chain.
11-15	Reserved for future expansion.

If the first byte of a filename is zero, that file entry slot is free. If the first byte is 255, that slot is the last occupied slot in the directory, i.e., this flags the end of the directory.

H.11.2 Drive Information

For each disk drive that is MOUNTed, the following information is kept in memory:

1. Attributes

Drive attributes are read from the information sector when the drive is mounted and may be changed with the SET statement. Current attributes may be examined with the ATTR\$ function.

2. Track Number

This is the current track while the disk is mounted. Otherwise, track number contains 255 as a flag that the disk is not mounted.

3. Modification Counter

This counter is incremented whenever an entry in the File Allocation Table is changed. After a given number of changes has been made, the File Allocation Table is written to disk.

4. Number of Free Clusters

This is calculated when the drive is mounted, and updated whenever a file is deleted or a cluster is allocated.

5. File Allocation Table

The File Allocation Table has a one-byte entry for every cluster allocated on the disk. If the cluster is free, this entry is 255. If the cluster is reserved, this entry is 254. If the cluster is the last cluster of the file, this entry is 300 (octal) plus the number of sectors from this cluster that were used. Otherwise, the entry is a pointer to the next cluster of the file. The File Allocation Table is read into memory when the drive is mounted, and updated:

1. When a file is deleted

2. When a file is closed

3. When modifications to the table total twice the number of sectors in a cluster (this can be changed in custom versions)

4. When modifications to the table have been made and the disk head is on (or passes) the directory track.

H.11.2.1 FORMAT Program - Before mounting a drive with a new diskette, run BASIC's FORMAT program to initialize the directory (set all bytes to 255), set the information sector to 0, and set all the File Allocation Table entries (except the directory track entry (254)) to "free" (255).

The FORMAT program is:

```
10 CLEAR 1500
20 A$=STRING$(128,255)
30 B$=STRING$(35*2,255)+STRING$(2,254)+STRING$(56,255)
40 FOR S=1 TO 19:DSKO$ 1,35,S,A$:NEXT
50 FOR S=21 TO 25 STEP 2:DSKO$ 1,35,S,B$
60 DSKO$ 1,35,S+1,A$:NEXT
70 DSKO$ 1,35,20,CHR$(0)
```

After running FORMAT and MOUNTing the drive, files will be allocated as usual, i.e., on either side of the directory track.

The FORMAT program may be altered to pre-allocate selected files. For instance, you may wish to use the FORMAT program to pre-allocate files contiguously (as they would be allocated in IBM mode). Then IBM and BASIC files may both exist on the diskette. The altered FORMAT program must also write the name of the file(s) to the directory track (i.e., files 1-8 in sector 1, files 9-16 in sector 2, etc.), so BASIC knows where the files start.

H.11.3 File Block

Each file on the disk has a file block that contains the following information:

1. File Mode (byte 0)

This is the first byte (byte 0) of the file block, and its location may be read with VARPTR(#filenumber). The location of any other byte in the file block is relative to the file mode byte. The file mode byte is one of the following:

(octal)

1	Input only
2	Output only
4	File mode
10	Append mode
20	Delete file
40	IBM mode
100	Special format (USR)
200	Binary save

NOTE

It is not recommended that the user attempt to modify the next four bytes of the File Allocation Table. Many unforeseen complications may result.

2. Pointer to the File Allocation Table entry for the first cluster allocated to the file (+1)
3. Pointer to the File Allocation Table entry for the last cluster accessed (+2)
4. Last sector accessed (+3)
5. Disk number of file (+4)
6. The size of the last buffer read (+5). This is 128 unless the last sector of the file is not full (i.e., Control-Z).
7. The current position in the buffer (+6). This is the offset within the buffer for the next print or input.
8. File flag (+7), is one of the following:
Octal
 - 100 Read after write check
 - 40 Read/Write EBCDIC, not ASCII
(Not available in all versions.)
 - 20 File write protected
 - 10 Buffer changed by PRINT
 - 4 PUT has been done. PRINT/INPUT are errors until a GET is done.
(See Section H.10.)
 - 2 Flags buffer is empty
9. Terminal position for TAB function and comma in PRINT statements (+8)
10. Beginning of sector buffer (+9), 128 bytes in length

H.12 ADVANCED USES OF FILE BUFFERS

1. Information may be passed from one program to another by FIELDing it to an unopened file number (not #0). The FIELD buffer is not cleared as long as the file is not OPENed.

2. The FIELDed buffer for an unopened file can also be used to format strings. For example, an 80-character string could be placed into a FIELDed buffer with LSET. The strings could then be accessed as four 20-character strings using their FIELDed variable names. For example:

```
100 FIELD#1, 80 AS A$  
200 FIELD#1, 20 AS A1$, 20 AS A2$, 20 AS A3$, 20 AS A4$  
300 LINE INPUT "CUSTOMER INFORMATION: ";B$  
400 LSET A$=B$  
500 PRINT "NAME ";A1$;"SSN: ";A2$
```

3. FIELD#0 may be used as a temporary buffer, but note that this buffer is cleared after each of the following commands: FILES, LOAD, SAVE, MERGE, RUN, DSKO\$, MOUNT, OPEN.
4. The effect of PRINT[USING]# into a string may be achieved by printing to a FIELDed buffer and then accessing it without reopening the file. To assure that this temporary buffer is not written to the disk, return the pointer to the beginning of the buffer and reset the "buffer changed" flag as follows:

```
10 OPEN "D" FOR IBM AS 1:REM THIS DOESN'T USE SPACE  
20 PRINT USING#1 ...  
30 P=PEEK(6+VARPTR(#1)):REM OPTIONAL, TO GET LENGTH OF PRINT  
USING  
40 FIELD#1 ... AS ...  
50 Y=7+VARPTR(#1)  
60 POKE Y,PEEK(Y AND &360):REM RESET BUFFER CHANGED FLAG  
70 POKE 6+VARPTR,0:REM CLEAR POSITION IN BUFFER
```

H.13 STANDALONE BASIC DISK ERRORS

```

50   FIELD overflow
51   Internal error
52   Bad file number
53   File not found
54   File already open
55   Disk not mounted
56   Disk I/O error
57   File already exists
59   Disk already mounted
61   Input past end
62   Bad file name
63   Direct statement in file
64   Bad allocation table
65   Bad drive number
66   Bad track/sector
67   File write protected
68   Disk offline
69   Deleted record
70   Rename across disks
71   Sequential after PUT
72   Sequential I/O only
73   File not OPEN

```

H.14 DOUBLE DENSITY, DOUBLE SIDED DISKETTES

For diskettes with 256-byte sectors, DSKI\$ and DSKO\$ are modified.

The DSKI\$ function returns as its value the first 255 bytes of the sector read.

The DSKO\$ statement does not use the <string expression> field. The format is:

```
DSKO$ <drive>,<track>,<sector>
```

In order to specify the data to write with DSKO\$ and to retrieve all 256 bytes of the data read by DSKI\$, the user must FIELD two or more variables (for a total of 256 bytes) to the file#0 buffer. The FIELDED variables will be identical to the data read with DSKI\$ and written with DSKO\$. For example:

```
FIELD#0,128 AS A$,128 AS B$
```

For double-sided diskettes, the formats of DSKI\$ and DSKO\$ must also include the surface number:

```
DSKI$(<drive>,<surface>,<track>,<sector>)
```

```
DKSO$ <drive>,<surface>,<track>,<sector>
      or
```

```
DKSO$ <drive>,<surface>,<track>,<sector>,<string exp>
```


APPENDIX I

Converting Programs to BASIC-80

If you have programs written in a BASIC other than BASIC-80, some minor adjustments may be necessary before running them with BASIC-80. Here are some specific things to look for when converting BASIC programs.

I.1 STRING DIMENSIONS

Delete all statements that are used to declare the length of strings. A statement such as `DIM A$(I,J)`, which dimensions a string array for J elements of length I, should be converted to the BASIC-80 statement `DIM A$(J)`.

Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for BASIC-80 string concatenation.

In BASIC-80, the `MID$`, `RIGHT$`, and `LEFT$` functions are used to take substrings of strings. Forms such as `A$(I)` to access the Ith character in `A$`, or `A$(I,J)` to take a substring of `A$` from position I to position J, must be changed as follows:

Other BASIC

`X$=A$(I)`
`X$=A$(I,J)`

BASIC-80

`X$=MID$(A$,I,1)`
`X$=MID$(A$,I,J-I+1)`

If the substring reference is on the left side of an assignment and `X$` is used to replace characters in `A$`, convert as follows:

Other BASIC

`A$(I)=X$`
`A$(I,J)=X$`

8K BASIC-80

`A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)`
`A$=LEFT$(A$,I-1);X$;MID$(A$,J+1)`

Ext. and Disk BASIC-80

`A$(I)=X$`
`A$(I,J)=X$`

`MID$(A$,1,1)=X$`
`MID$(A$,I,J-I+1)=X$`

I.2 MULTIPLE ASSIGNMENTS

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. BASIC-80 would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

I.3 MULTIPLE STATEMENTS

Some BASICs use a backslash (\) to separate multiple statements on a line. With BASIC-80, be sure all statements on a line are separated by a colon (:).

I.4 MAT FUNCTIONS

Programs using the MAT functions available in some BASICs must be rewritten using FOR...NEXT loops to execute properly.

APPENDIX J

Summary of Error Codes and Error Messages

<u>Code</u>	<u>Number</u>	<u>Message</u>
NF	1	NEXT without FOR A variable in a NEXT statement does not correspond to any previously executed, unmatched FOR statement variable.
SN	2	Syntax error A line is encountered that contains some incorrect sequence of characters (such as unmatched parenthesis, misspelled command or statement, incorrect punctuation, etc.).
RG	3	Return without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
OD	4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
FC	5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: <ol style="list-style-type: none">1. a negative or unreasonably large subscript2. a negative or zero argument with LOG3. a negative argument to SQR4. a negative mantissa with a non-integer exponent

5. a call to a USR function for which the starting address has not yet been given
 6. an improper argument to MID\$, LEFT\$, RIGHT\$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRINGS\$, SPACES\$, INSTR, or ON...GOTO.
- OV 6 **Overflow**
The result of a calculation is too large to be represented in BASIC-80's number format. If underflow occurs, the result is zero and execution continues without an error.
- OM 7 **Out of memory**
A program is too large, has too many FOR loops or GOSUBs, too many variables, or expressions that are too complicated.
- UL 8 **Undefined line**
A line reference in a GOTO, GOSUB, IF...THEN...ELSE or DELETE is to a nonexistent line.
- BS 9 **Subscript out of range**
An array element is referenced either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.
- DD 10 **Redimensioned array**
Two DIM statements are given for the same array, or a DIM statement is given for an array after the default dimension of 10 has been established for that array.
- /0 11 **Division by zero**
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.
- ID 12 **Illegal direct**
A statement that is illegal in direct mode is entered as a direct mode command.
- TM 13 **Type mismatch**
A string variable name is assigned a numeric value or vice versa; a function that expects a numeric argument is given a string argument or vice versa.

OS	14	Out of string space String variables have caused BASIC to exceed the amount of free memory remaining. BASIC will allocate string space dynamically, until it runs out of memory.
LS	15	String too long An attempt is made to create a string more than 255 characters long.
ST	16	String formula too complex A string expression is too long or too complex. The expression should be broken into smaller expressions.
CN	17	Can't continue An attempt is made to continue a program that: <ol style="list-style-type: none">1. has halted due to an error,2. has been modified during a break in execution, or3. does not exist.
UF	18	Undefined user function A USR function is called before the function definition (DEF statement) is given.

Extended and Disk Versions Only

- | | |
|----|---|
| 19 | No RESUME
An error trapping routine is entered but contains no RESUME statement. |
| 20 | RESUME without error
A RESUME statement is encountered before an error trapping routine is entered. |
| 21 | Unprintable error
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code. |
| 22 | Missing operand
An expression contains an operator with no operand following it. |
| 23 | Line buffer overflow
An attempt is made to input a line that has too many characters. |

- 26 FOR without NEXT
A FOR was encountered without a matching NEXT.
- 29 WHILE without WEND
A WHILE statement does not have a matching WEND.
- 30 WEND without WHILE
A WEND was encountered without a matching WHILE.

Disk Errors

- 50 Field overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.
- 51 Internal error
An internal malfunction has occurred in Disk BASIC-80. Report to Microsoft the conditions under which the message appeared.
- 52 Bad file number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.
- 53 File not found
A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.
- 54 Bad file mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.
- 55 File already open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.
- 57 Disk I/O error
An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

- 58 File already exists
The filename specified in a NAME statement is identical to a filename already in use on the disk.
- 61 Disk full
All disk storage space is in use.
- 62 Input past end
An INPUT statement is executed after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.
- 63 Bad record number
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.
- 64 Bad file name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).
- 66 Direct statement in file
A direct statement is encountered while LOADING an ASCII-format file. The LOAD is terminated.
- 67 Too many files
An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

APPENDIX K

Mathematical Functions

Derived Functions

Functions that are not intrinsic to BASIC-80 may be calculated as follows.

<u>Function</u>	<u>BASIC-80 Equivalent</u>
SECANT	$\text{SEC}(X) = 1/\text{COS}(X)$
COSECANT	$\text{CSC}(X) = 1/\text{SIN}(X)$
COTANGENT	$\text{COT}(X) = 1/\text{TAN}(X)$
INVERSE SINE	$\text{ARCSIN}(X) = \text{ATN}(X/\text{SQR}(-X*X+1))$
INVERSE COSINE	$\text{ARCCOS}(X) = -\text{ATN}(X/\text{SQR}(-X*X+1)) + 1.5708$
INVERSE SECANT	$\text{ARCSEC}(X) = \text{ATN}(X/\text{SQR}(X*X-1)) + \text{SGN}(\text{SGN}(X)-1)*1.5708$
INVERSE COSECANT	$\text{ARCCSC}(X) = \text{ATN}(X/\text{SQR}(X*X-1)) + (\text{SGN}(X)-1)*1.5708$
INVERSE COTANGENT	$\text{ARCCOT}(X) = \text{ATN}(X) + 1.5708$
HYPERBOLIC SINE	$\text{SINH}(X) = (\text{EXP}(X) - \text{EXP}(-X))/2$
HYPERBOLIC COSINE	$\text{COSH}(X) = (\text{EXP}(X) + \text{EXP}(-X))/2$
HYPERBOLIC TANGENT	$\text{TANH}(X) = \text{EXP}(-X)/(\text{EXP}(X) + \text{EXP}(-X))*2+1$
HYPERBOLIC SECANT	$\text{SECH}(X) = 2/(\text{EXP}(X) + \text{EXP}(-X))$
HYPERBOLIC COSECANT	$\text{CSCH}(X) = 2/(\text{EXP}(X) - \text{EXP}(-X))$
HYPERBOLIC COTANGENT	$\text{COTH}(X) = \text{EXP}(-X)/(\text{EXP}(X) - \text{EXP}(-X))*2+1$
INVERSE HYPERBOLIC SINE	$\text{ARCSINH}(X) = \text{LOG}(X + \text{SQR}(X*X+1))$
INVERSE HYPERBOLIC COSINE	$\text{ARCCOSH}(X) = \text{LOG}(X + \text{SQR}(X*X-1))$
INVERSE HYPERBOLIC TANGENT	$\text{ARCTANH}(X) = \text{LOG}((1+X)/(1-X))/2$
INVERSE HYPERBOLIC SECANT	$\text{ARCSECH}(X) = \text{LOG}((\text{SQR}(-X*X+1)+1)/X)$
INVERSE HYPERBOLIC COSECANT	$\text{ARCCSCH}(X) = \text{LOG}((\text{SGN}(X)*\text{SQR}(X*X+1)+1)/X)$
INVERSE HYPERBOLIC COTANGENT	$\text{ARCCOTH}(X) = \text{LOG}((X+1)/(X-1))/2$

APPENDIX L

Microsoft BASIC Compiler

The Microsoft BASIC Compiler package contains the following software: BASIC Compiler, MACRO-80 assembler, and LINK-80 loader. The following manuals are also supplied: BASIC-80 Reference Manual, BASIC Compiler User's Manual, Utility Software Manual. The Utility Software Manual is the reference manual for MACRO-80 and LINK-80. The BASIC Compiler User's Manual describes the use of the compiler, its command format, compilation switches and error messages. The BASIC language that is used with the Microsoft BASIC Compiler is the same as described in this manual for Disk BASIC-80 with the following exceptions:

L.1 OPERATIONAL DIFFERENCES

The Compiler interacts with the console only to read compiler commands. These specify what files are to be compiled. There is no "direct mode," as with the BASIC-80 interpreter. Commands that are usually issued in the direct mode with the BASIC-80 interpreter are not implemented on the Compiler.

The following statements and commands are not implemented and will generate an error message:

AUTO	CLOAD	CSAVE	CONT	DELETE
EDIT	ERASE	LIST	LLIST	LOAD
MERGE	NEW	RENUM	SAVE	

Because there is no direct mode for typing in programs or edit mode for editing programs, use Microsoft's EDIT-80 Text Editor or BASIC-80 interpreter for creating and editing programs. If you use the interpreter, be sure to SAVE the file with the A (ASCII format) option.

The compiler cannot accept a physical line that is more than 253 characters in length. A logical statement, however, may contain as many physical lines as desired. Use line feed to start a new physical line within a logical statement.

To reduce the size of the compiled program, there are no program line numbers included in the object code generated by the compiler unless the /D, /X, or /E switch is set in the compiler command. Error messages, therefore, contain the address where the error occurred, instead of a line number. The compiler listing and the map generated by LINK-80 are used to identify the line that has the error. It is always a good idea to debug programs using the BASIC-80 interpreter before attempting to compile them. See the BASIC Compiler User's Manual for more information.

L.2 LANGUAGE DIFFERENCES

Most programs that run on the Microsoft BASIC-80 interpreter will run on the BASIC Compiler with little or no change. However, it is necessary to note differences in the use of the following program statements:

1. CALL
The <variable name> field in the CALL statement must contain an External symbol, i.e., one that is recognized by LINK-80 as a global symbol. This routine must be supplied by the user as an assembly language subroutine or a routine from the FORTRAN-80 library.
2. CHAIN and RUN
The CHAIN statement is used to chain to a new program overlay using the runtime module. The RUN statement is to be used to execute any executable file. (Under CP/M, any .COM file may be RUN.)
3. CLEAR
The CLEAR statement is only supported in compiled programs using the runtime module.
4. COMMON
The COMMON statement must appear before any executable statements. See section 2.7 for further details.
5. DEFINT/SNG/DBL/STR
The compiler does not "execute" DEFxxx statements; it reacts to the static occurrence of these statements, regardless of the order in which program lines are executed. A DEFxxx statement takes effect as soon as its line is encountered. Once the type has been defined for a given variable, it remains in effect until the end of the program or until a different DEFxxx statement with that variable takes effect.

6. DIM and ERASE

The DIM statement is similar to the DEFxxx statement in that it is scanned rather than executed. That is, DIM takes effect when its line is encountered. If the default dimension (10) has already been established for an array variable and that variable is later encountered in a DIM statement, a "Redimensioned array" error results.

There is no ERASE statement in the compiler, so arrays cannot be erased and redimensioned. An ERASE statement will produce a fatal error.

Also note that the values of the subscripts in a DIM statement must be integer constants; they may not be variables, arithmetic expressions, or floating point values. For example,

```
DIM A1(I)  
DIM A1(3+4)
```

are both illegal.

7. END

During execution of a compiled program, an END statement closes files and returns control to the operating system. The compiler assumes an END statement at the end of the program, so "running off the end" produces proper program termination.

8. FOR/NEXT and WHILE/WEND

FOR/NEXT and WHILE/WEND loops must be statically nested.

9. ON ERROR GOTO/RESUME <line number>

If a program contains ON ERROR GOTO and RESUME <line number> statements, the /E compilation switch must be used. If the RESUME NEXT, RESUME, or RESUME 0 form is used, the /X switch must also be included. See the BASIC Compiler User's Manual for an explanation of these switches.

10. REM

REM statements or remarks starting with a single quotation mark do not take up time or space during execution, and so may be used as freely as desired.

11. STOP

The STOP statement is identical to the END statement. Open files are closed and control returns to the operating system.

12. TRON/TROFF

In order to use TRON/TROFF, the /D compilation switch must be used. Otherwise, TRON and TROFF are ignored and a warning message is generated.

13. USRn Functions

USRn Functions are significantly different from the interpreter versions. The argument to the USRn function is ignored and an integer result is returned in the HL registers. It is recommended that USRn functions be replaced by the CALL statement.

14. %INCLUDE

The %INCLUDE <filename> statement allows the compiler to include source from an alternate file. The %INCLUDE statement must be the last statement on a line. The format of the %INCLUDE statement is:

<line number> %INCLUDE <filename>

For example,

999 %INCLUDE SUB1000.BAS

15. Double Precision Transcendental Functions

SIN, COS, TAN, SQR, LOG, and EXP return double precision results if given a double precision argument. Exponentiation with double precision operands will return a double precision result.

16. String Variables

The string space is maintained differently with the BASIC Compiler than with the interpreter. Using PEEK, POKE, VARPTR, or assembly language routines to change string descriptors will result in a String Space Corrupt error.

L.3 EXPRESSION EVALUATION

During expression evaluation, the operands of each operator are converted to the same type, that of the most precise operand. For example,

QR=J%+A!+Q#

causes J% to be converted to single precision and added to A!. This result is converted to double precision and added to Q#.

The Compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter the following program

```
I%=20000
J%=20000
K%=-30000
M%=I%+J%-K%
```

yields 10000 for M%. That is, it adds I% to J% and, because the number is too large, it converts the result into a floating point number. K% is then converted to floating point and subtracted. The result of 10000 is found, and is converted back to integer and saved as M%.

The compiler, however, must make type conversion decisions during compilation. It cannot defer until the actual values are known. Thus, the compiler would generate code to perform the entire operation in integer mode. If the /D switch were set, the error would be detected. Otherwise, an incorrect answer would be produced.

In order to produce optimum efficiency in the compiled program, the compiler may perform any number of valid algebraic transformations before generating the code. For example, the program

```
I%=20000
J%=-18000
K%=20000
M%=I%+J%+K%
```

could produce an incorrect result when run. If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I%+K% first and then adds J%, an overflow will occur. The compiler follows the rules for operator precedence and parenthetical modification of such precedence, but no other guarantee of evaluation order can be made.

L.4 INTEGER VARIABLES

In order to produce the fastest and most compact object code possible, make maximum use of integer variables. For example, this program

```
FOR I=1 TO 10
A(I)=0
NEXT I
```

can execute approximately 30 times faster by simply substituting "I%" for "I". It is especially advantageous to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

APPENDIX M

ASCII Character Codes

ASCII Code	Character	ASCII Code	Character	ASCII Code	Character
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093]
008	BS	051	3	094	^
009	HT	052	4	095	<
010	LF	053	5	096	·
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	\$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	}
039	'	082	R	125	~
040	(083	S	126	
041)	084	T	127	DEL
042	*	085	U		

ASCII codes are in decimal.

LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

INDEX

%INCLUDE	L-4
ABS	3-2
Addition	1-10
ALL	2-4, 2-9
Arctangent	3-3
Array variables	1-7, 2-9, 2-19, L-5
Arrays	1-7, 2-7, 2-12, 2-25
ASC	3-2
ASCII codes	3-2, 3-4
ASCII format	2-4, 2-50, 2-78, L-1
Assembly language subroutines	2-3, 2-17, 2-60, 3-23 to 3-24, C-1, L-2
ATN	3-3, L-4
ATTR\$	H-5
ATTRIB	E-2
AUTO	1-2, 2-2
Boolean operators	1-12
CALL	2-3, C-5, L-2
Carriage return	1-3, 2-37, 2-42 to 2-43, 2-84 to 2-86
Cassette tape	2-7, 2-12
CDBL	3-3
CHAIN	2-4, 2-9, L-2
Character set	1-3
CHR\$	3-4
CINT	3-4
CLEAR	2-6, A-1, L-2
CLOAD	2-7
CLOAD*	2-7
CLOAD?	2-7
CLOSE	2-8, B-3, B-8
Command level	1-1
COMMON	2-4, 2-9, L-2
Concatenation	1-15
Constants	1-4
CONT	2-11, 2-42
Control characters	1-4
Control-A	2-23
COS	3-5, L-4
CP/M	2-47, 2-50, 2-77 to 2-78, B-1, D-1
CSAVE	2-12
CSAVE*	2-12
CSNG	3-5
CVD	3-6, B-8
CVI	3-6, B-8
CVS	3-6, B-8

DATA	2-13, 2-75
DEF FN	2-14
DEF USR	2-17, 3-23
DEFDBL	1-7, 2-16, L-2
DEFINT	1-7, 2-16, L-2
DEFSNG	1-7, 2-16, L-2
DEFSTR	1-7, 2-16, L-2
DEINT	C-1, G-1
DELETE	1-2, 2-4, 2-18
DIM	2-19, L-3
Direct mode	1-1, 2-35, 2-55, L-1
Division	1-10
Double precision	1-5, 2-16, 2-61, 3-3, A-1, L-4
DSKI\$	H-2, H-13
DSKO\$	H-2, H-13
EDIT	1-2, 2-20
Edit mode	1-4, 2-20, L-1
END	2-8, 2-11, 2-24, 2-33, L-3
EOF	3-6, B-3, B-5, D-4
ERASE	2-25, L-3
ERL	2-26
ERR	2-26
ERROR	2-27
Error codes	1-16, 2-26 to 2-27, J-1
Error messages	1-16, J-1, L-2
Error trapping	2-26 to 2-27, 2-55, 2-76, B-7, L-3
Escape	1-3, 2-20
EXP	3-7, L-4
Exponentiation	1-10 to 1-11, L-4
Expressions	1-9
FIELD	2-29, B-8, H-11
FILES	D-3, H-2
FIX	3-7
FOR...NEXT	2-30, A-1, L-3
FORMAT program	H-10
FPOS	H-2
FRCINT	C-1, C-4, D-4, G-1
FRE	3-8
Functions	1-14, 2-14, 3-1, K-1
GET	2-29, 2-32, B-8, D-4, H-7
GIVABF	C-1 to C-2, G-1
GIVINT	E-2
GOSUB	2-33
GOTO	2-33 to 2-34
HEX\$	3-8
Hexadecimal	1-5, 3-8
IF...GOTO	2-35
IF...THEN	2-26, 2-35
IF...THEN...ELSE	2-35

Indirect mode	1-1
INKEY\$	3-9
INP	3-9
INPUT	2-11, 2-29, 2-37, A-2, B-9
INPUT\$	3-10
INPUT#	H-7
INPUT#	B-3
INPUT#	2-39
INSTR	3-11
INT	3-7, 3-12
Integer	3-4, 3-7, 3-12
Integer division	1-11
INTEL	G-1
Interrupts	C-7
ISIS-II	2-77, E-1
 KILL	2-40, B-2
 LEFT\$	3-12
LEN	3-13
LET	2-29, 2-41, B-9
LFILES	H-2
Line feed	1-2, 2-37, 2-42 to 2-43, 2-85 to 2-86, L-1
LINE INPUT	2-42
LINE INPUT#	B-3
LINE INPUT#	2-43
Line numbers	1-1 to 1-2, 2-2, 2-74, L-2
Line printer	2-46, 2-48, 2-84, 3-14, A-2, E-2
Lines	1-1, L-1
LIST	1-2, 2-44
LLIST	2-46, F-1, G-2
LOAD	2-47, 2-78, B-1
LOC	3-13, B-3, B-5, B-8, H-2
LOF	D-4, H-2
LOG	3-14, L-4
Logical operators	1-12
Loops	2-30, 2-83
LPOS	2-84, 3-14
LPRINT	2-48, 2-84, F-1, G-2
LPRINT USING	2-48
LSET	2-49, B-8
 MAKINT	C-1, C-4, D-4, E-2, G-1
MBASIC	D-1
MDS	G-1
MERGE	2-4, 2-50, B-2
MID\$	2-51, 3-15, I-1
MKD\$	3-15, B-8
MKI\$	3-15, B-8
MKS\$	3-15, B-8
MOD operator	1-11
Modulus arithmetic	1-11
MOUNT	H-3
Multiplication	1-10

NAME	2-52
Negation	1-10
NEW	2-8, 2-53
NULL	2-54
Numeric constants	1-4
Numeric variables	1-7
 OCT\$	3-16
Octal	1-5, 3-16
ON ERROR GOTO	2-55, L-3
ON...GOSUB	2-56
ON...GOTO	2-56
OPEN	2-8, 2-29, 2-57, B-3, B-8, H-5 to H-6
Operators	1-9, 1-11 to 1-13, 1-15, L-4
OPTION BASE	2-58
OUT	2-59
Overflow	1-11, 3-7, 3-22, A-1, L-4
Overlay	2-4
 Paper tape	2-54
PEEK	2-60, 3-16
POKE	2-60, 3-16
POS	2-84, 3-17
PRINT	2-61, A-1
PRINT USING	2-63, A-2
PRINT#	H-7
PRINT# USING	B-5
PRINT# USING	B-3
PRINT#	B-3
PRINT# USING	2-67
PRINT#	2-67
Protected files	2-78, A-2, B-2
PUT	2-29, 2-69, B-8, H-7
 Random files	2-29, 2-32, 2-40, 2-49, 2-57, 2-69, 3-13, 3-15, B-7, D-4
Random numbers	2-70, 3-18
RANDOMIZE	2-70, 3-18, A-1
READ	2-71, 2-75
Relational operators	1-11
REM	2-73, L-3
REMOVE	H-3
RENUM	2-4, 2-26, 2-74
RESET	D-3
RESTORE	2-75
RESUME	2-76, L-3
RETURN	2-33
RIGHT\$	3-17
RND	2-70, 3-18, A-1
RSET	2-49, B-8
Rubout	1-3, 1-15, 2-21
RUN	2-77 to 2-78, B-2, L-2
 SAVE	2-47, 2-77 to 2-78, B-1

SBC	G-1
Sequential files	2-39 to 2-40, 2-43, 2-57, 2-67, 2-86, 3-6, 3-13, B-3
SET	H-4
SGN	3-18
SIN	3-19, L-4
Single precision	1-5, 2-16, 2-61, 3-5, A-1
Space Requirements for variables	1-8
SPACES\$	3-19
SPC	3-20
SQR	3-20, L-4
Standalone Disk BASIC	H-1
STOP	2-11, 2-24, 2-33, 2-79, L-3
STR\$	3-21
String constants	1-4
String functions	3-6, 3-11 to 3-13, 3-15, 3-17, 3-21, 3-23, I-1
String operators	1-15
String space	2-6, 3-8, A-1, B-9
String Variables	L-4
String variables	1-7, 2-16, 2-42 to 2-43
STRINGS\$	3-21
Subroutines	2-3, 2-33, 2-56, C-1
Subscripts	1-7, 2-19, 2-58, L-3
Subtraction	1-10
SWAP	2-80
SYSTEM	D-4, F-1
TAB	3-22
Tab	1-3 to 1-4
TAN	3-22, L-4
TEKDOS	F-1
TROFF	2-81, L-3
TRON	2-81, L-3
USR	2-17, 3-23, C-1
USRLOC	C-2, G-1
VAL	3-23
Variables	1-6, L-5
VARPTR	3-24, H-10
WAIT	2-82
WEND	2-83, L-3
WHILE	2-83, L-3
WIDTH	2-84, A-2
WIDTH LPRINT	2-84, A-2
WRITE	2-85
WRITE#	B-3
WRITE#	2-86

utility software manual

MACRO-80 Assembler

LINK-80 Loader

CREF-80 Cross Reference Facility

LIB-80 Library Manager (CP/M Versions)

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft. The software described in this document is furnished under a license agreement or non-disclosure agreement. The software may be used or copied only in accordance with the terms of the agreement.

(C) Microsoft, 1979

CP/M is a registered trade mark of Digital Research

Microsoft

CONTENTS

CHAPTER 1 Introduction

CHAPTER 2 MACRO-80 Assembler

- 2.1 Running MACRO-80
- 2.2 Command Format
 - 2.2.1 Devices
 - 2.2.2 Switches
- 2.3 Format of MACRO-80 Source Files
 - 2.3.1 Statements
 - 2.3.2 Symbols
 - 2.3.3 Numeric Constants
 - 2.3.4 Strings
- 2.4 Expression Evaluation
 - 2.4.1 Arithmetic and Logical Operators
 - 2.4.2 Modes
 - 2.4.3 Externals
- 2.5 Opcodes as Operands
- 2.6 Pseudo Operations
 - 2.6.1 ASEG
 - 2.6.2 COMMON
 - 2.6.3 CSEG
 - 2.6.4 DB - Define Byte
 - 2.6.5 DC - Define Character
 - 2.6.6 DS - Define Space
 - 2.6.7 DSEG
 - 2.6.8 DW - Define Word
 - 2.6.9 END
 - 2.6.10 ENTRY/PUBLIC
 - 2.6.11 EQU
 - 2.6.12 EXT/EXTRN
 - 2.6.13 INCLUDE
 - 2.6.14 NAME
 - 2.6.15 ORG - Define Origin
 - 2.6.16 PAGE
 - 2.6.17 SET
 - 2.6.18 SUBTTL
 - 2.6.19 TITLE
 - 2.6.20 .COMMENT
 - 2.6.21 .PRINTX
 - 2.6.22 .RADIX
 - 2.6.23 .Z80
 - 2.6.24 .8080
 - 2.6.25 .REQUEST
 - 2.6.26 Conditional Pseudo Operations
 - 2.6.26.1 ELSE
 - 2.6.26.2 ENDIF
 - 2.6.27 Listing Control Pseudo Operations

2.6.28 Relocation Pseudo Operations
2.6.28.1 ORG Pseudo-op
2.6.28.2 LINK-80
2.6.29 Relocation Before Loading
2.7 Macros and Block Pseudo Operations
2.7.1 Terms
2.7.2 REPT-ENDM
2.7.3 IRP-ENDM
2.7.4 IRPC-ENDM
2.7.5 MACRO
2.7.6 ENDM
2.7.7 EXITM
2.7.8 LOCAL
2.7.9 Special Macro Operators and Forms
2.8 Using Z80 Pseudo-ops
2.9 Sample Assembly
2.10 MACRO-80 Errors
2.11 Compatibility with Other Assemblers
2.12 Format of Listings
2.12.1 Symbol Table Listing

CHAPTER 3 CREF-80 Cross Reference Facility

CHAPTER 4 LINK-80 Linking Loader

4.1 Running LINK-80
4.2 Command Format
4.2.1 LINK-80 Switches
4.2.2 Sample Link
4.3 Format of LINK Compatible Object Files
4.4 LINK-80 Error Messages
4.5 Program Break Information

CHAPTER 5 LIB-80 Library Manager

5.1 LIB-80 Commands
5.1.1 Modules
5.2 LIB-80 Switches
5.3 LIB-80 Listings
5.4 Sample LIB Session
5.5 Summary of Switches and Syntax

APPENDIX A TEKDOS Operating System

A.1 TEKDOS Command Files
A.2 MACRO-80
A.3 CREF-80
A.4 LINK-80

CHAPTER 1

INTRODUCTION

MACRO-80 is a relocatable macro assembler for 8080 and Z80 microcomputer systems. It assembles 8080 or Z80 code on any 8080 or Z80 development system running the CP/M, ISIS-II, TRSDOS or TEKDOS operating system. The MACRO-80 package includes the MACRO-80 assembler, the LINK-80 linking loader, and the CREF-80 cross reference facility. CP/M versions also include the LIB-80 Library Manager. MACRO-80 resides in approximately 14K of memory and has an assembly rate of over 1000 lines per minute.

MACRO-80 incorporates almost all "big computer" assembler features without sacrificing speed or memory space. The assembler supports a complete, Intel standard macro facility, including IRP, IRPC, REPEAT, local variables and EXITM. Nesting of macros is limited only by memory. Code is assembled in relocatable modules that are manipulated with the flexible linking loader. Conditional assembly capability is enhanced by an expanded set of conditional pseudo operations that include testing of assembly pass, symbol definition, and parameters to macros. Conditionals may be nested up to 255 levels.

MACRO-80's linking loader provides a versatile array of loader capabilities, which are executed by means of easy command lines and switches. Any number of programs may be loaded with one command, relocatable modules may be loaded in user-specified locations, and external references between modules are resolved automatically by the loader. The loader also performs library searches for system subroutines and generates a load map of memory showing the locations of the main program and subroutines. The cross reference facility that is included in this package supplies a convenient alphabetic list of all program variable names, along with the line numbers where they are referenced and defined.

This manual is designed to serve as a reference guide to the MACRO-80 package. It defines, explains and gives examples of all the features in MACRO-80 in terms that should be understandable to anyone familiar with assembly language programming. It is not intended, however, to serve as instructional material and presumes the user has substantial knowledge of assembly language programming. The user should refer to instructional material available from a variety of sources for additional tutorial information.

CHAPTER 2

MACRO-80 ASSEMBLER

2.1 RUNNING MACRO-80

The command to run MACRO-80 is

M80

MACRO-80 returns the prompt "*", indicating it is ready to accept commands.

NOTE

If you are using the TEKDOS operating system, see Appendix A for proper command formats.

2.2 COMMAND FORMAT

A command to MACRO-80 consists of a string of filenames with optional switches. All filenames should follow the operating system's conventions for filenames and extensions. The default extensions supplied by Microsoft software are as follows:

<u>File</u>	<u>CP/M</u>	<u>ISIS-II</u>
Relocatable object file	REL	REL
Listing file	PRN	LST
MACRO-80 source file	MAC	MAC
FORTRAN source file	FOR	FOR
COBOL source	COB	COB
Absolute file	COM	

A command to MACRO-80 conveys the name of the source file to be assembled, the names of the file(s) to be created, and which assembly options are desired. The format of a MACRO-80 command is:

```
objfile, lstfile=source file
```

Only the equal sign and the source file field are required to create a relocatable object file with the default (source) filename and the default extension REL.

Otherwise, an object file is created only if the objfile field is filled, and a listing file is created only if the lstfile field is filled.

To assemble the source file without producing an object file or listing file, place only a comma to the left of the equal sign. This is a handy procedure that lets you check for syntax errors before assembling to an object file.

Examples:

*=TEST	Assemble the source file TEST.MAC and place the object file in TEST.REL.
* ,=TEST	Assemble the source file TEST.MAC without creating an object or listing file. Useful for error checking.
TEST,TEST=TEST	Assemble the source file TEST.MAC, placing the object file in TEST.REL and the listing file in TEST.PRN. (With ISIS-II, the listing file is TEST.LST.)
*OBJECT=TEST	Assemble the source file TEST.MAC and place the object file in OBJECT.REL.
OBJECT,LIST=TEST	Assemble the source file TEST.MAC, placing the object file in OBJECT.REL and the listing file in LIST.PRN. (With ISIS-II, the listing file is LIST.LST.)

MACRO-80 also supports command lines; that is, the invocation and command may be typed on the same line. For example:

```
M80 ,=TEST
```

2.2.1 Devices

Any field in the MACRO-80 command string can also specify a device name. The default device name with the CP/M operating system is the currently logged disk. The default device name with the ISIS-II operating system is disk drive 0. The command format is:

```
dev:objfile,dev:lstfile=dev:source file
```

The device names are as follows:

<u>Device</u>	<u>CP/M</u>	<u>ISIS-II</u>
Disk drives	A:, B:, C:,...	:F0:, :F1:, :F2:, ...
Line printer	LST:	LST:
Teletype or CRT	TTY:	TTY:
High speed reader	HSR	

Examples:

* ,TTY:=TEST Assemble the source file TEST.MAC and list the program on the console. No object code is generated. Useful for error check.

*SMALL,TTY:=B:TEST Assemble TEST.MAC (found on disk drive B), place the object file in SMALL.REL, and list the program on the console.

2.2.2 Switches

A switch is a letter that is appended to the command string, preceded by a slash. It specifies an optional task to be performed during assembly. More than one switch can be used, but each must be preceded by a slash. (With the TEKDOS operating system, switches are preceded by commas or spaces. See Appendix A.) All switches are optional. The available switches are:

<u>Switch</u>	<u>Action</u>
O	Octal listing
H	Hexadecimal listing (default)
R	Force generation of an object file
L	Force generation of a listing file
C	Force generation of a cross reference file

- Z Assemble Z80 opcodes (default for Z80 operating systems)
- I Assemble 8080 opcodes (default for 8080 operating systems)
- P Each /P allocates an extra 256 bytes of stack space for use during assembly. Use /P if stack overflow errors occur during assembly. Otherwise, not needed.
- M Initialize Block Data Areas. If the programmer wants the area that is defined by the DS (Define Space) pseudo-op initialized to zeros, then the programmer should use the /M switch in the command line. Otherwise, the space is not guaranteed to contain zeros. That is, DS does not automatically initialize the space to zeros.
- X Usually used to suppress the listing of false conditionals. The following paragraph describes the /X switch more completely but in very technical terms.

The presence or absence of /X in the command line sets the initial current mode and the initial value of the default for listing or suppressing lines in false conditional blocks. /X sets the current mode and initial value of default to not-to-list. No /X sets current mode and initial value of default to list. Current mode determines whether false conditionals will be listed or suppressed. The initial value of the default is used with the .TFCOND pseudo-op so that .TFCOND is independent of .SFCOND and .LFCOND. If the program contains .SFCOND or .LFCOND, /X has no effect after .SFCOND or .LFCOND is encountered until a .TFCOND is encountered in the file. So /X has an effect only when used with a file that contains no conditional listing pseudo-ops or when used with .TFCOND.

Examples:

*=TEST/L Assemble TEST.MAC, place the object file in TEST.REL and a listing file in TEST.PRN. (With ISIS-II, the listing file is TEST.LST.)

*=TEST/L/O Same as above, but listing file addresses will be in octal.

*LAST=TEST/C Assemble TEST.MAC, place the object file in LAST.REL and cross reference file in TEST.CRF. (See Chapter 3.)

2.3 FORMAT OF MACRO-80 SOURCE FILES

Input source lines of up to 132 characters in length are acceptable.

MACRO-80 preserves lower case letters in quoted strings and comments. All symbols, opcodes and pseudo-opcodes typed in lower case will be converted to upper case.

If the source file includes line numbers from an editor, each byte of the line number must have the high bit on. Line numbers from Microsoft's EDIT-80 Editor are acceptable.

2.3.1 Statements

Source files input to MACRO-80 consist of statements of the form:

[label::[:]] [operator] [arguments] [;comment]

With the exception of the ISIS assembler \$ controls (see Section 2.11), it is not necessary that statements begin in column 1. Multiple blanks or tabs may be used to improve readability.

If a label is present, it is the first item in the statement and is immediately followed by a colon. If it is followed by two colons, it is declared as PUBLIC (see ENTRY/PUBLIC, Section 2.6.10). For example:

FOO:: RET

is equivalent to

PUBLIC FOO
FOO: RET

The next item after the label, or the first item on the line if no label is present, is an operator. An operator may be an 8086 mnemonic, pseudo-op, macro call or expression. The evaluation order is as follows:

1. Macro call
2. Mnemonic/Pseudo operation
3. Expression

Instead of flagging an expression as an error, the assembler treats it as if it were a DB statement (see Section 2.6.4).

The arguments following the operator will, of course, vary in form according to the operator.

A comment always begins with a semicolon and ends with a carriage return. A comment may be a line by itself or it may be appended to a line that contains a statement. Extended comments can be entered using the .COMMENT pseudo operation (see Section 2.6.20).

2.3.2 Symbols

MACRO-80 symbols may be of any length, however, only the first six characters are significant. The following characters are legal in a symbol:

A-Z 0-9 \$. ? @

With Microsoft's 8080/Z80/8086 assemblers, the underline character is also legal in a symbol. A symbol may not start with a digit. When a symbol is read, lower case is translated into upper case. If a symbol reference is followed by ## it is declared external (see also the EXT/EXTRN pseudo-op, Section 2.6.12).

2.3.3 Numeric Constants

The default base for numeric constants is decimal. This may be changed by the .RADIX pseudo-op (see Section 2.6.22). Any base from 2 (binary) to 16 (hexadecimal) may be selected. When the base is greater than 10, A-F are the digits following 9. If the first digit of the number is not numeric the number must be preceded by a zero.

Numbers are 16-bit unsigned quantities. A number is always evaluated in the current radix unless one of the following special notations is used:

nnnnB	Binary
nnnnD	Decimal
nnnnO	Octal
nnnnQ	Octal
nnnnH	Hexadecimal
X'nnnn'	Hexadecimal

Overflow of a number beyond two bytes is ignored and the result is the low order 16-bits.

A character constant is a string comprised of zero, one or two ASCII characters, delimited by quotation marks, and used in a non-simple expression. For example, in the statement

```
DB      'A' + 1
```

'A' is a character constant. But the statement

```
DB      'A'
```

uses 'A' as a string because it is in a simple expression. The rules for character constant delimiters are the same as for strings.

A character constant comprised of one character has as its value the ASCII value of that character. That is, the high order byte of the value is zero, and the low order byte is the ASCII value of the character. For example, the value of the constant 'A' is 41H.

A character constant comprised of two characters has as its value the ASCII value of the first character in the high order byte and the ASCII value of the second character in the low order byte. For example, the value of the character constant "AB" is 41H*256+42H.

2.3.4 Strings

A string is comprised of zero or more characters delimited by quotation marks. Either single or double quotes may be used as string delimiters. The delimiter quotes may be used as characters if they appear twice for every character occurrence desired. For example, the statement

```
DB      "I am ""great"" today"
```

stores the string

I am "great" today

If there are zero characters between the delimiters, the string is a null string.

2.4 EXPRESSION EVALUATION

2.4.1 Arithmetic And Logical Operators

The following operators are allowed in expressions. The operators are listed in order of precedence.

NUL

LOW, HIGH

*, /, MOD, SHR, SHL

Unary Minus

+, -

EQ, NE, LT, LE, GT, GE

NOT

AND

OR, XOR

Parentheses are used to change the order of precedence. During evaluation of an expression, as soon as a new operator is encountered that has precedence less than or equal to the last operator encountered, all operations up to the new operator are performed. That is, subexpressions involving operators of higher precedence are computed first.

All operators except +, -, *, / must be separated from their operands by at least one space.

The byte isolation operators (HIGH, LOW) isolate the high or low order 8 bits of an Absolute 16-bit value. If a relocatable value is supplied as an operand, HIGH and LOW will treat it as if it were relative to location zero.

2.4.2 Modes

All symbols used as operands in expressions are in one of the following modes: Absolute, Data Relative, Program (Code) Relative or COMMON. (See Section 2.6 for the ASEG, CSEG, DSEG and COMMON pseudo-ops.) Symbols assembled under the ASEG, CSEG (default), or DSEG pseudo-ops are in Absolute, Code Relative or Data Relative mode respectively.

The number of COMMON modes in a program is determined by the number of COMMON blocks that have been named with the COMMON pseudo-op. Two COMMON symbols are not in the same mode unless they are in the same COMMON block. In any operation other than addition or subtraction, the mode of both operands must be Absolute.

If the operation is addition, the following rules apply:

1. At least one of the operands must be Absolute.
2. Absolute + <mode> = <mode>

If the operation is subtraction, the following rules apply:

1. <mode> - Absolute = <mode>
2. <mode> - <mode> = Absolute
where the two <mode>s are the same.

Each intermediate step in the evaluation of an expression must conform to the above rules for modes, or an error will be generated. For example, if FOO, BAZ and ZAZ are three Program Relative symbols, the expression

FOO + BAZ - ZAZ

will generate an R error because the first step (FOO + BAZ) adds two relocatable values. (One of the values must be Absolute.) This problem can always be fixed by inserting parentheses. So that

FOO + (BAZ - ZAZ)

is legal because the first step (BAZ - ZAZ) generates an Absolute value that is then added to the Program Relative value, FOO.

2.4.3 Externals

Aside from its classification by mode, a symbol is either External or not External. (See EXT/EXTRN, Section 2.6.12.) An External value must be assembled into a two-byte field. (Single-byte Externals are not supported.) The following rules apply to the use of Externals in expressions:

1. Externals are legal only in addition and subtraction.
2. If an External symbol is used in an expression, the result of the expression is always External.
3. When the operation is addition, either operand (but not both) may be External.

4. When the operation is subtraction, only the first operand may be External.

2.5 OPCODES AS OPERANDS

8080 opcodes are valid one-byte operands. Note that only the first byte is a valid operand. For example:

```
MVI    A, (JMP)
ADI    (CPI)
MVI    B, (RNZ)
CPI    (INX H)
ACI    (LXI B)
MVI    C,MOV A,B
```

Errors will be generated if more than one byte is included in the operand -- such as (CPI 5), LXI B,LABEL1) or (JMP LABEL2).

Opcodes used as one-byte operands need not be enclosed in parentheses.

NOTE

Opcodes are not valid operands
in Z80 mode.

2.6 PSEUDO OPERATIONS

2.6.1 ASEG

ASEG

ASEG sets the location counter to an absolute segment of memory. The location of the absolute counter will be that of the last ASEG (default is 0), unless an ORG is done after the ASEG to change the location. The effect of ASEG is also achieved by using the code segment (CSEG) pseudo operation and the /P switch in LINK-80. See also Section 2.6.28

2.6.2 COMMON

COMMON <block name>/

COMMON sets the location counter to the selected common block in memory. The location is always the beginning of the area so that compatibility with the FORTRAN COMMON statement is maintained. If <block name> is omitted or consists of spaces, it is considered to be blank common. See also Section 2.6.28.

2.6.3 CSEG

CSEG

CSEG sets the location counter to the code relative segment of memory. The location will be that of the last CSEG (default is 0), unless an ORG is done after the CSEG to change the location. CSEG is the default condition of the assembler (the INTEL assembler defaults to ASEG). See also Section 2.6.28.

2.6.4 DB - Define Byte

DB <exp>[,<exp>...]

DB <string>[<string>...]

The arguments to DB are either expressions or strings. DB stores the values of the expressions or the characters of the strings in successive memory locations beginning with the current location counter.

Expressions must evaluate to one byte. (If the high byte of the result is 0 or 255, no error is given; otherwise, an A error results.)

Strings of three or more characters may not be used in expressions (i.e., they must be immediately followed by a comma or the end of the line). The characters in a string are stored in the order of appearance, each as a one-byte value with the high order bit set to zero.

Example:

0000'	41 42	DB	'AB'
0002'	42	DB	'AB' AND OFFH
0003'	41 42 43	DB	'ABC'

2.6.5 DC - Define Character

DC <string>

DC stores the characters in <string> in successive memory locations beginning with the current location counter. As with DB, characters are stored in order of appearance, each as a one-byte value with the high order bit set to zero. However, DC stores the last character of the string with the high order bit set to one. An error will result if the argument to DC is a null string.

2.6.6 DS - Define Space

DS <exp>

DS reserves an area of memory. The value of <exp> gives the number of bytes to be allocated. All names used in <exp> must be previously defined (i.e., all names known at that point on pass 1). Otherwise, a V error is generated during pass 1 and a U error may be generated during pass 2. If a U error is not generated during pass 2, a phase error will probably be generated because the DS generated no code on pass 1.

2.6.7 DSEG

DSEG

DSEG sets the location counter to the Data Relative segment of memory. The location of the data relative counter will be that of the last DSEG (default is 0), unless an ORG is

done after the DSEG to change the location. See also Section 2.6.28.

2.6.8 DW - Define Word

DW <exp>[,<exp>...]

DW stores the values of the expressions in successive memory locations beginning with the current location counter. Expressions are evaluated as 2-byte (word) values.

2.6.9 END

END [<exp>]

The END statement specifies the end of the program. If <exp> is present, it is the start address of the program. If <exp> is not present, then no start address is passed to LINK-80 for that program.

NOTE

If an assembly language program is the main program, a start address (label) must be specified. If not, LINK-80 will issue a "no start address" error. If the program is a subroutine to a FORTRAN program (for example), the start address is not required because FORTRAN has supplied one.

2.6.10 ENTRY/PUBLIC

ENTRY <name>[,<name>...]
or
PUBLIC <name>[,<name>...]

ENTRY or PUBLIC declares each name in the list as internal and therefore available for use by this program and other programs to be loaded concurrently. All of the names in the list must be defined in the current program or a U error results. An M error is generated if the name is an external name or common-blockname.

2.6.11 EQU

```
<name> EQU <exp>
```

EQU assigns the value of <exp> to <name>. If <exp> is external, an error is generated. If <name> already has a value other than <exp>, an M error is generated.

2.6.12 EXT/EXTRN

```
EXT    <name>[,<name>...]  
or  
EXTRN <name>[,<name>...]
```

EXT or EXTRN declares that the name(s) in the list are external (i.e., defined in a different program). If any item in the list references a name that is defined in the current program, an M error results. A reference to a name where the name is followed immediately by two pound signs (e.g., NAME##) also declares the name as external.

2.6.13 INCLUDE

```
INCLUDE <filename>
```

The INCLUDE pseudo-op applies only to CP/M versions of MACRO-80. The pseudo-ops INCLUDE, \$INCLUDE and MACLIB are synonymous.

The INCLUDE pseudo-op assembles source statements from an alternate source file into the current source file. Use of INCLUDE eliminates the need to repeat an often-used sequence of statements in the current source file.

<filename> is any valid specification, as determined by the operating system. Defaults for filename extensions and device names are the same as those in a MACRO-80 command line.

The INCLUDE file is opened and assembled into the current source file immediately following the INCLUDE statement. When end-of-file is reached, assembly resumes with the statement following INCLUDE.

On a MACRO-80 listing, a plus sign is printed between the assembled code and the source line on each line assembled from an INCLUDE file. (See Section 2.12.)

Nested INCLUDEs are not allowed. If encountered, they will result in an objectionable syntax error 'O'.

The file specified in the operand field must exist. If the file is not found, the error 'V' (value error) is given, and the INCLUDE is ignored.

2.6.14 NAME

```
NAME      ('modname')
```

NAME defines a name for the module. Only the first six characters are significant in a module name. A module name may also be defined with the TITLE pseudo-op. In the absence of both the NAME and TITLE pseudo-ops, the module name is created from the source file name.

2.6.15 ORG - Define Origin

```
ORG      <exp>
```

The location counter is set to the value of <exp> and the assembler assigns generated code starting with that value. All names used in <exp> must be known on pass 1, and the value must either be absolute or in the same area as the location counter.

2.6.16 PAGE

```
PAGE      [<exp>]
```

PAGE causes the assembler to start a new output page. The value of <exp>, if included, becomes the new page size (measured in lines per page) and must be in the range 10 to 255. The default page size is 50 lines per page. The assembler puts a form feed character in the listing file at the end of a page.

2.6.17 SET

```
<name> SET <exp>
```

SET is the same as EQU, except no error is generated if <name> is already defined.

2.6.18 SUBTTL

```
SUBTTL <text>
```

SUBTTL specifies a subtitle to be listed on the line after the title (see TITLE, Section 2.6.19) on each page heading. <text> is truncated after 60 characters. Any number of SUBTTLs may be given in a program.

2.6.19 TITLE

```
TITLE <text>
```

TITLE specifies a title to be listed on the first line of each page. If more than one TITLE is given, a Q error results. The first six characters of the title are used as the module name unless a NAME pseudo operation is used. If neither a NAME or TITLE pseudo-op is used, the module name is created from the source filename.

2.6.20 .COMMENT

```
.COMMENT <delim><text><delim>
```

The first non-blank character encountered after .COMMENT is the delimiter. The following <text> comprises a comment block which continues until the next occurrence of <delimiter> is encountered. For example, using an asterisk as the delimiter, the format of the comment block would be:

```
.COMMENT *
any amount of text entered
here as the comment block
*
*
*
;return to normal mode
```

2.6.21 .PRINTX

```
.PRINTX <delim><text><delim>
```

The first non-blank character encountered after .PRINTX is the delimiter. The following text is listed on the terminal during assembly until another occurrence of the delimiter is encountered. .PRINTX is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches. For example:

```
IF      CPM
.PRINTX /CPM version/
ENDIF
```

NOTE

.PRINTX will output on both passes. If only one printout is desired, use the IF1 or IF2 pseudo-op. For example:

```
IF2
IF CPM
.PRINTX /CPM version/
ENDIF
ENDIF
```

will only print if CPM is true and M80 is in pass 2.

2.6.22 .RADIX

```
.RADIX <exp>
```

The default base (or radix) for all constants is decimal. The .RADIX statement allows the default radix to be changed to any base in the range 2 to 16. For example:

```
MOVI  BX,0FFH
.RADIX 16
MOVI  BX,0FF
```

The two MOVIs in the example are identical. The <exp> in a .RADIX statement is always in decimal radix, regardless of the current radix.

2.6.23 .Z80

.Z80 enables the assembler to accept Z80 opcodes. This is the default condition when the assembler is running on a Z80 operating system. Z80 mode may also be set by appending the Z switch to the MACRO-80 command string -- see Section 2.2.2.

2.6.24 .8080

.8080 enables the assembler to accept 8080 opcodes. This is the default condition when the assembler is running on an 8080 operating system. 8080 mode may also be set by appending the I switch to the MACRO-80 command string -- see Section 2.2.2.

2.6.25 .REQUEST

.REQUEST <filename>[,<filename>...]

.REQUEST sends a request to the LINK-80 loader to search the filenames in the list for undefined globals. The filenames in the list should be in the form of legal symbols. They should not include filename extensions or disk specifications. LINK-80 supplies a default extension and assumes the default disk drive.

2.6.26 Conditional Pseudo Operations

The conditional pseudo operations are:

IF/IFT <exp>	True if <exp> is not 0.
IFE/IFF <exp>	True if <exp> is 0.
IF1	True if pass 1.
IF2	True if pass 2.
IFDEF <symbol>	True if <symbol> is defined or has been declared External.
IFNDEF <symbol>	True if <symbol> is undefined or not declared External.
IFB <arg>	True if <arg> is blank. The angle brackets around <arg> are required.
IFNB <arg>	True if <arg> is not blank. Used for testing when dummy parameters are supplied. The angle brackets around <arg> are required.
IFIDN <arg1>,<arg2>	True if the string <arg1> is IDeNtical to the string <arg2>. The angle brackets around <arg1> and <arg2> are required.
IFDIF <arg1>,<arg2>	True if the string <arg1> is DIFFerent from the string <arg2>. The angle brackets around <arg1> and <arg2> are required.

All conditionals use the following format:

```
IFxx [argument]
.
.
.
[ELSE
.
.
.
]
ENDIF
```

Conditionals may be nested to any level. Any argument to a conditional must be known on pass 1 to avoid V errors and incorrect evaluation. For IF, IFT, IFF, and IFE the expression must involve values which were previously defined and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

2.6.26.1 ELSE - Each conditional pseudo operation may optionally be used with the ELSE pseudo operation which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF, and an ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a C error.

2.6.26.2 ENDIF - Each IF must have a matching ENDIF to terminate the conditional. Otherwise, an 'Unterminated conditional' message is generated at the end of each pass. An ENDIF without a matching IF causes a C error.

2.6.27 Listing Control Pseudo Operations

Output to the listing file can be controlled by two pseudo-ops:

.LIST and .XLIST

If a listing is not being made, these pseudo-ops have no effect. .LIST is the default condition. When a .XLIST is encountered, source and object code will not be listed until a .LIST is encountered.

The output of false conditional blocks is controlled by three pseudo-ops: .SFCOND, .LFCOND, and .TFCOND.

These pseudo-ops give the programmer control over four cases.

1. Normally list false conditionals

For this case, the programmer simply allows the default mode to control the listing. The default mode is list false conditionals. If the programmer decides to suppress false conditionals, the /X switch can be issued in the command line instead of editing the source file.

2. Normally suppress false conditionals
For this case, the programmer issues the .TFCOND pseudo-op in the program file. .TFCOND reverses (toggles) the default, causing false conditionals to be suppressed. If the programmer decides to list false conditionals, the /X switch can be issued in the command line instead of editing the source file.
3. Always suppress/list false conditionals
For these cases, the programmer issues either the .SFCOND pseudo-op to always suppress false conditionals, or the .LFCOND pseudo-op to always list all false conditionals.
4. Suppress/list some false conditionals
For this case, the programmer has decided for most false conditionals whether to list or suppress, but for some false conditionals the programmer has not yet decided. For the false conditionals decided about, use .SFCOND or .LFCOND. For those not yet decided, use .TFCOND. .TFCOND sets the current and default settings to the opposite of the default. Initially, the default is set by giving /X or no /X in the command line. Two subcases exist:
 1. The programmer wants some false conditionals not to list unless /X is given. The programmer uses the .SFCOND and .LFCOND pseudo-ops to control which areas always suppress or list false conditionals. To selectively suppress some false conditionals, the programmer issues .TFCOND at the beginning of the conditional block and again at the end of the conditional block. (NOTE: The second .TFCOND should be issued so that the default setting will be the same as the initial setting. Leaving the default equal to the initial setting makes it easier to keep track of the default mode if there are many such areas.) If the conditional block evaluates as false, the lines will be suppressed. In this subcase, issuing the /X switch in the command line causes the conditional block affected by .TFCOND to list even if it evaluates as false.

2. The programmer wants some false conditionals to list unless /X is given. Two consecutive .TFCONDs places the conditional listing setting in initial state which is determined by the presence or absence of the /X switch in the command line (the first .TFCOND sets the default to not initial; the second to initial). The selected conditional block then responds to the /X switch: if a /X switch is issued in the command line, the conditional block is suppressed if false; if no /X switch is issued in the command line, the conditional block is listed even if false.

The programmer then must reissue the .SFCOND or .LFCOND conditional listing pseudo-op to restore the suppress or list mode. Simply issuing another .TFCOND will not restore the prior mode, but will toggle the default setting. Since in this subcase, the next area of code is supposed to list or suppress false conditionals always, the programmer must issue .SFCOND or .LFCOND.

The three conditional listing pseudo-ops are summarized below.

<u>PSEUDO-OP</u>	<u>DEFINITION</u>
.SFCOND	Suppresses the listing of conditional blocks that evaluate as false.
.LFCOND	Restores the listing of conditional blocks that evaluate as false.
.TFCOND	Toggles the current setting which controls the listing false conditionals. .TFCOND sets the current and default setting to not default. If a /X switch is given in the MACRO-80 run command line for a file which contains .TFCOND, /X reverses the effect of .TFCOND.

The following chart illustrates the effects of the three pseudo-ops when encountered under /X and under no /X.

<u>PSEUDO-OP</u>	<u>NO /X</u>	<u>/X</u>
(none)	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.LFCOND	ON	ON
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.	.	.
.	.	.
.	.	.
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.SFCOND	OFF	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON
.TFCOND	ON	OFF
.	.	.
.	.	.
.	.	.
.TFCOND	OFF	ON

The output of cross reference information is controlled by .CREF and .XREF. If the cross reference facility (see Chapter 3) has not been invoked, .CREF and .XREF have no effect. The default condition is .CREF. When a .XREF is encountered, no cross reference information is output until .CREF is encountered.

The output of MACRO/REPT/IRP/IRPC expansions is controlled by three pseudo-ops: .LALL, .SALL, and .XALL. .LALL lists the complete macro text for all expansions. .SALL suppresses listing of all text and object code produced by macros. .XALL is the default condition; a source line is listed only if it generates object code.

2.6.28 Relocation Pseudo Operations

The ability to create relocatable modules is one of the major features of Microsoft assemblers. Relocatable modules offer the advantages of easier coding and faster testing, debugging and modifying. In addition, it is possible to specify segments of assembled code that will later be loaded into RAM (the Data Relative segment) and ROM/PROM (the Code Relative segment). The pseudo operations that select relocatable areas are CSEG and DSEG. The ASEG pseudo-op is used to generate non-relocatable (absolute) code. The COMMON pseudo-op creates a common data area for every COMMON block that is named in the program.

The default mode for the assembler is Code Relative. That is, assembly begins with a CSEG automatically executed and the location counter in the Code Relative mode, pointing to location 0 in the Code Relative segment of memory. All subsequent instructions will be assembled into the Code Relative segment of memory until an ASEG or DSEG or COMMON pseudo-op is executed. For example, the first DSEG encountered sets the location counter to location zero in the Data Relative segment of memory. The following code is assembled in the Data Relative mode, that is, it is assigned to the Data Relative segment of memory. If a subsequent CSEG is encountered, the location counter will return to the next free location in the Code Relative segment and so on.

The ASEG, DSEG, CSEG pseudo-ops never have operands. If you wish to alter the current value of the location counter, use the ORG pseudo-op.

2.6.28.1 ORG Pseudo-op - At any time, the value of the location counter may be changed by use of the the ORG pseudo-op. The form of the ORG statement is:

ORG <exp>

where the value of <exp> will be the new value of the location counter in the current mode. All names used in <exp> must be known on pass 1 and the value of <exp> must be either Absolute or in the current mode of the location counter. For example, the statements

DSEG
ORG 50

set the Data Relative location counter to 50, relative to the start of the Data Relative segment of memory.

2.6.28.2 LINK-80 - The LINK-80 linking loader (see Chapter 4 of this manual) combines the segments and creates each relocatable module in memory when the program is loaded. The origins of the relocatable segments are not fixed until the program is loaded and the origins are assigned by LINK-80. The command to LINK-80 may contain user-specified origins through the use of the /P (for Code Relative) and /D (for Data and COMMON segments) switches.

For example, a program that begins with the statements

```
ASEG  
ORG      800H
```

and is assembled entirely in Absolute mode will always load beginning at 800 unless the ORG statement is changed in the source file. However, the same program, assembled in Code Relative mode with no ORG statement, may be loaded at any specified address by appending the /P:<address> switch to the LINK-80 command string.

2.6.29 Relocation Before Loading

Two pseudo-ops, .PHASE and .DEPHASE, allow code to be located in one area, but executed only at a different, specified area.

For example:

```
0000'          .PHASE 100H  
0100  E8 0003    FOO:    CALL    BAZ  
0103  E9 FF01    FOO:    JMP     ZOO  
0106  C3         BAZ:    RET  
                  .DEPHASE  
0007'  E9 FFFF    ZOO:    JMP     5
```

All labels within a .PHASE block are defined as the absolute value from the origin of the phase area. The code, however, is loaded in the current area (i.e., from 0' in this example). The code within the block can later be moved to 100H and executed.

2.7 MACROS AND BLOCK PSEUDO OPERATIONS

The macro facilities provided by MACRO-80 include three repeat pseudo operations: repeat (REPT), indefinite repeat (IRP), and indefinite repeat character (IRPC). A macro definition operation (MACRO) is also provided. Each of these four macro operations is terminated by the ENDM pseudo operation.

2.7.1 Terms

For the purposes of discussion of macros and block operations, the following terms will be used:

1. <dummy> is used to represent a dummy parameter. All dummy parameters are legal symbols that appear in the body of a macro expansion.
2. <dummylist> is a list of <dummy>s separated by commas.
3. <arglist> is a list of arguments separated by commas. <arglist> must be delimited by angle brackets. Two angle brackets with no intervening characters (<>) or two commas with no intervening characters enter a null argument in the list. Otherwise an argument is a character or series of characters terminated by a comma or >. With angle brackets that are nested inside an <arglist>, one level of brackets is removed each time the bracketed argument is used in an <arglist>. See example, Section 2.7.5.) A quoted string is an acceptable argument and is passed as such. Unless enclosed in brackets or a quoted string, leading and trailing spaces are deleted from arguments.
4. <paramlist> is used to represent a list of actual parameters separated by commas. No delimiters are required (the list is terminated by the end of line or a comment), but the rules for entering null parameters and nesting brackets are the same as described for <arglist>. (See example, Section 2.7.5)

2.7.2 REPT-ENDM

```
REPT <exp>
.
.
.
ENDM
```

The block of statements between REPT and ENDM is repeated <exp> times. <exp> is evaluated as a 16-bit unsigned number. If <exp> contains any external or undefined terms, an error is generated. Example:

```
SET 0
REPT 10      ;generates DB 1 - DB 10
SET X+1
DB X
ENDM
```

2.7.3 IRP-ENDM

```
IRP <dummy>,<arglist>
  .
  .
  .
ENDM
```

The <arglist> must be enclosed in angle brackets. The number of arguments in the <arglist> determines the number of times the block of statements is repeated. Each repetition substitutes the next item in the <arglist> for every occurrence of <dummy> in the block. If the <arglist> is null (i.e., <>), the block is processed once with each occurrence of <dummy> removed. For example:

```
IRP X,<1,2,3,4,5,6,7,8,9,10>
DB X
ENDM
```

generates the same bytes as the REPT example.

2.7.4 IRPC-ENDM

```
IRPC <dummy>,string (or <string>)
  .
  .
  .
ENDM
```

IRPC is similar to IRP but the arglist is replaced by a string of text and the angle brackets around the string are optional. The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <dummy> in the block. For example:

```
IRPC X,0123456789
DB X+1
ENDM
```

generates the same code as the two previous examples.

2.7.5 MACRO

Often it is convenient to be able to generate a given sequence of statements from various places in a program, even though different parameters may be required each time the sequence is used. This capability is provided by the MACRO statement.

The form is

```
<name> MACRO <dummylist>
.
.
.
ENDM
```

where <name> conforms to the rules for forming symbols. <name> is the name that will be used to invoke the macro. The <dummy>s in <dummylist> are the parameters that will be changed (replaced) each time the MACRO is invoked. The statements before the ENDM comprise the body of the macro. During assembly, the macro is expanded every time it is invoked but, unlike REPT/IRP/IRPC, the macro is not expanded when it is encountered.

The form of a macro call is

```
<name> <paramlist>
```

where <name> is the name supplied in the MACRO definition, and the parameters in <paramlist> will replace the <dummy>s in the MACRO <dummylist> on a one-to-one basis. The number of items in <dummylist> and <paramlist> is limited only by the length of a line. The number of parameters used when the macro is called need not be the same as the number of <dummy>s in <dummylist>. If there are more parameters than <dummy>s, the extras are ignored. If there are fewer, the extra <dummy>s will be made null. The assembled code will contain the macro expansion code after each macro call.

NOTE

A dummy parameter in a MACRO/REPT/IRP/IRPC is always recognized exclusively as a dummy parameter. Register names such as A and B will be changed in the expansion if they were used as dummy parameters.

Here is an example of a MACRO definition that defines a macro called FOO:

```
FOO    MACRO      X
      SET      0
      REPT     X
      Y      SET      Y+1
      DB      Y
      ENDM
      ENDM
```

This macro generates the same code as the previous three examples when the call

```
FOO      10
```

is executed.

Another example, which generates the same code, illustrates the removal of one level of brackets when an argument is used as an arglist:

```
FOO    MACRO      X
      IRP      Y,<X>
      DB      Y
      ENDM
      ENDM
```

When the call

```
FOO      <1,2,3,4,5,6,7,8,9,10>
```

is made, the macro expansion looks like this:

```
IRP      Y,<1,2,3,4,5,6,7,8,9,10>
DB      Y
ENDM
```

2.7.6 ENDM

Every REPT, IRP, IRPC and MACRO pseudo-op must be terminated with the ENDM pseudo-op. Otherwise, the 'Unterminated REPT/IRP/IRPC/MACRO' message is generated at the end of each pass. An unmatched ENDM causes an O error.

2.7.7 EXITM

The EXITM pseudo-op is used to terminate a REPT/IRP/IRPC or MACRO call. When an EXITM is executed, the expansion is exited immediately and any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

2.7.8 LOCAL

```
LOCAL <dummylist>
```

The LOCAL pseudo-op is allowed only inside a MACRO definition. When LOCAL is executed, the assembler creates a unique symbol for each <dummy> in <dummylist> and substitutes that symbol for each occurrence of the <dummy> in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiply-defined labels on successive expansions of the macro. The symbols created by the assembler range from ..0001 to ..FFFF. Users will therefore want to avoid the form ..nnnn for their own symbols. If LOCAL statements are used, they must be the first statements in the macro definition.

2.7.9 Special Macro Operators And Forms

- & The ampersand is used in a macro expansion to concatenate text or symbols. A dummy parameter that is in a quoted string will not be substituted in the expansion unless it is immediately preceded by &. To form a symbol from text and a dummy, put & between them. For example:

```
ERRGEN MACRO X
ERROR&X:PUSH BX
      MOVI BX,'&X'
      JMP  ERROR
ENDM
```

In this example, the call ERRGEN A will generate:

```
ERRORA: PUSH B
      MOVI BX,'A'
      JMP  ERROR
```

- ; In a block operation, a comment preceded by two semicolons is not saved as part of the expansion (i.e., it will not appear on the listing even under .LALL). A comment preceded by one semicolon, however, will be preserved and appear in the expansion.

- ! When an exclamation point is used in an argument, the next character is entered literally (i.e., !; and <;> are equivalent).

NUL NUL is an operator that returns true if its argument (a parameter) is null. The remainder of a line after NUL is considered to be the argument to NUL. The conditional

IF NUL argument

is false if, during the expansion, the first character of the argument is anything other than a semicolon or carriage return. It is recommended that testing for null parameters be done using the IFB and IFNB conditionals.

% The percent sign is used only in a macro argument. % converts the expression that follows it (usually a symbol) to a number in the current radix. During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must conform to the same rules as the DS (Define Space) pseudo-op. A valid expression returning a non-relocatable constant is required.

EXAMPLE: Normally, LB, the argument to MAKLAB, would be substituted for Y, the argument to MACRO, as a string. The % causes LB to be converted to a non-relocatable constant which is then substituted for Y. Without the % special operator, the result of assembly would be 'Error LB' rather than 'Error l', etc.

MAKLAB	MACRO	Y
ERR&Y:	DB	'Error &Y',0
	ENDM	
MAKERR	MACRO	X
LB	SET	0
	REPT	X
LB	SET	LB+1
	MAKLAB	%LB
	ENDM	
	ENDM	

When called by MAKERR 3, the assembler will generate:

```
ERR1: DB      'Error 1',0
ERR2: DB      'Error 2',0
ERR3: DB      'Error 3',0
```

TYPE The TYPE operator returns a byte that describes two characteristics of its argument: 1) the mode, and 2) whether it is External or not. The argument to TYPE may be any expression (string, numeric, logical). If the expression is invalid, TYPE returns zero.

The byte that is returned is configured as follows:

The lower two bits are the mode. If the lower two bits are:

- 0 the mode is Absolute
- 1 the mode is Program Relative
- 2 the mode is Data Relative
- 3 the mode is Common Relative

The high bit (80H) is the External bit. If the high bit is on, the expression contains an External. If the high bit is off, the expression is local (not External).

The Defined bit is 20H. This bit is on if the expression is locally defined, and it is off if the expression is undefined or external. If neither bit is on, the expression is invalid.

TYPE is usually used inside macros, where an argument type may need to be tested to make a decision regarding program flow. For example:

```
FOO      MACRO      X
          LOCAL      Z
Z          SET TYPE X
IF        Z...
```

2.8 USING Z80 PSEUDO-OPS

When using the MACRO-80 assembler, the following Z80 pseudo-ops are valid. The function of each pseudo-op is equivalent to that of its counterpart.

<u>Z80 pseudo-op</u>	<u>Equivalent pseudo-op</u>
COND	IFT
ENDC	ENDIF
*EJECT	PAGE
DEFB	DB
DEFS	DS
DEFW	DW
DEFM	DB
DEFL	SET
GLOBAL	PUBLIC
EXTERNAL	EXTRN

The formats, where different, conform to the previous format. That is, DEFB and DEFW are permitted a list of arguments (as are DB and DW), and DEFM is permitted a string or numeric argument (as is DB).

2.9 SAMPLE ASSEMBLY

A>M80

*EXMP11,TTY:=EXMP11

MAC80 3.2 PAGE 1

		00100	;CSL3(P1,P2)
		00200	;SHIFT P1 LEFT CIRCULARLY 3 BITS
		00300	;RETURN RESULT IN P2
		00400	ENTRY CSL3
		00450	;GET VALUE OF FIRST PARAMETER
		00500	CSL3:
0000'	7E	00600	MOV A,M
0001'	23	00700	INX H
0002'	66	00800	MOV H,M
0003'	6F	00900	MOV L,A
		01000	;SHIFT COUNT
0004'	06 03	01100	MVI B,3
0006'	AF	01200	LOOP: XRA A
		01300	;SHIFT LEFT
0007'	29	01400	DAD H
		01500	;ROTATE IN CY BIT
0008'	17	01600	RAL
0009'	85	01700	ADD L
000A'	6F	01800	MOV L,A
		01900	;DECREMENT COUNT
000B'	05	02000	DCR B
		02100	;ONE MORE TIME
000C'	C2 0006'	02200	JNZ LOOP
000F'	EB	02300	XCHG
		02400	;SAVE RESULT IN SECOND PARAMETER
0010'	73	02500	MOV M,E
0011'	23	02600	INX H
0012'	72	02700	MOV M,D
0013'	C9	02800	RET
		02900	END

MAC80 3.2 PAGE S

CSL3 0000I' LOOP 0006'

No Fatal error(s)

2.10 MACRO-80 ERRORS

MACRO-80 errors are indicated by a one-character flag in column one of the listing file. If a listing file is not being printed on the terminal, each erroneous line is also printed or displayed on the terminal. Below is a list of the MACRO-80 Error Codes:

- A Argument error
Argument to pseudo-op is not in correct format or is out of range (.PAGE 1; .RADIX 1; PUBLIC 1; JMPS TOOFAR).
- C Conditional nesting error
ELSE without IF, ENDIF without IF, two ELSEs on one IF.
- D Double Defined symbol
Reference to a symbol which is multiply defined.
- E External error
Use of an external illegal in context (e.g., FOO SET NAME##; MOVI AX,2-NAME##).
- M Multiply Defined symbol
Definition of a symbol which is multiply defined.
- N Number error
Error in a number, usually a bad digit (e.g., 8Q).
- O Bad opcode or objectionable syntax
ENDM, LOCAL outside a block; SET, EQU or MACRO without a name; bad syntax in an opcode; or bad syntax in an expression (mismatched parenthesis, quotes, consecutive operators, etc.).
- P Phase error
Value of a label or EQU name is different on pass 2.
- Q Questionable
Usually means a line is not terminated properly.
This is a warning error (e.g. MOV AX,BX,).
- R Relocation
Illegal use of relocation in expression, such as abs-rel. Data, code and COMMON areas are relocatable.
- U Undefined symbol
A symbol referenced in an expression is not defined. (For certain pseudo-ops, a V error is printed on pass 1 and a U on pass 2.)

V Value error
On pass 1 a pseudo-op which must have its value known on pass 1 (e.g., .RADIX, .PAGE, DS, IF, IFE, etc.), has a value which is undefined. If the symbol is defined later in the program, a U error will not appear on the pass 2 listing.

Error Messages:

'No end statement encountered on input file'
No END statement: either it is missing or it is not parsed due to being in a false conditional, unterminated IRP/IRPC/REPT block or terminated macro.

'Unterminated conditional'
At least one conditional is unterminated at the end of the file.

'Unterminated REPT/IRP/IRPC/MACRO'
At least one block is unterminated.

[xx] [No] Fatal error(s) [,xx warnings]
The number of fatal errors and warnings. The message is listed on the CRT and in the list file.

2.11 COMPATIBILITY WITH OTHER ASSEMBLERS

The \$EJECT and \$TITLE controls are provided for compatibility with INTEL's ISIS assembler. The dollar sign must appear in column 1 only if spaces or tabs separate the dollar sign from the control word. The control

\$EJECT

is the same as the MACRO-80 PAGE pseudo-op.
The control

\$TITLE('text')

is the same as the MACRO-80 SUBTTL <text> pseudo-op.

The INTEL operands PAGE and INPAGE generate Q errors when used with the MACRO-80 CSEG or DSEG pseudo-ops. These errors are warnings; the assembler ignores the operands.

When MACRO-80 is entered, the default for the origin is Code Relative 0.

With the INTEL ISIS assembler, the default is Absolute 0.

With MACRO-80, the dollar sign (\$) is a defined constant that indicates the value of the location counter at the start of the statement. Other assemblers may use a decimal point or an asterisk. Other constants are defined by MACRO-80 to have the following values:

A=7	B=0	C=1	D=2	E=3
H=4	L=5	M=6	SP=6	PSW=6

2.12 FORMAT OF LISTINGS

On each page of a MACRO-80 listing, the first two lines have the form:

```
[TITLE text]      M80 3.3      PAGE x[-y]  
[SUBTTL text]
```

where:

1. TITLE text is the text supplied with the TITLE pseudo-op, if one was given in the source program.
2. x is the major page number, which is incremented only when a form feed is encountered in the source file. (When using Microsoft's EDIT-80 text editor, a form feed is inserted whenever a page mark is done.) When the symbol table is being printed, x = S.
3. y is the minor page number, which is incremented whenever the .PAGE pseudo-op is encountered in the source file, or whenever the current page size has been filled.
4. SUBTTL text is the text supplied with the SUBTTL pseudo-op, if one was given in the source program.

Next, a blank line is printed, followed by the first line of output.

A line of output on a MACRO-80 listing has the following form:

```
[crf#] [error] loc#m |xx | xxxx|... source
```

If cross reference information is being output, the first item on the line is the cross reference number, followed by a tab.

A one-letter error code followed by a space appears next on the line, if the line contains an error. If there is no error, a space is printed. If there is no cross reference number, the error code column is the first column on the listing.

The value of the location counter appears next on the line. It is a 4-digit hexadecimal number or 6-digit octal number, depending on whether the /O or /H switch was given in the MACRO-80 command string.

The character at the end of the location counter value is the mode indicator. It will be one of the following symbols:

'	Code Relative
"	Data Relative
!	COMMON Relative
<space>	Absolute
*	External

Next, three spaces are printed followed by the assembled code. One-byte values are followed by a space. Two-byte values are followed by a mode indicator. Two-byte values are printed in the opposite order they are stored in, i.e., the high order byte is printed first. Externals are either the offset or the value of the pointer to the next External in the chain.

If a line of output on a MACRO-80 listing is from an INCLUDE file, the character 'C' is printed after the assembled code on that line. If a line of output is part of a text expansion (MACRO, REPT, IRP, IRPC) a plus sign '+' is printed after the assembled code on that line.

The remainder of the line contains the line of source code, as it was input.

Example:

0C49 3A A91Z' C+ LDA LCOUNT

'C+' indicates this line is from an INCLUDE file and part of a macro expansion.

2.12.1 Symbol Table Listing

In the symbol table listing, all the macro names in the program are listed alphabetically, followed by all the symbols in the program, listed alphabetically. After each symbol, a tab is printed, followed by the value of the symbol. If the symbol is Public, an I is printed immediately after the value. The next character printed will be one of the following:

- U Undefined symbol.
- C COMMON block name. (The "value" of the COMMON block is its length (number of bytes) in hexadecimal or octal.)
- * External symbol.
- <space> Absolute value.
- ' Program Relative value.
- " Data Relative value.
- ! COMMON Relative value.

CHAPTER 3

CREF-80 CROSS REFERENCE FACILITY

NOTE

If you are using the TEKDOS operating system, see Appendix A for proper command formats.

In order to generate a cross reference listing, the assembler must output a special listing file with embedded control characters. The MACRO-80 command string tells the assembler to output this special listing file. /C is the cross reference switch. When the /C switch is encountered in a MACRO-80 command string, the assembler opens a .CRF file instead of a .LST file. (See Section 2.6.27 for the .CREF and .XREF pseudo-ops.)

Examples:

*=TEST/C	Assemble file TEST.MAC and create object file TEST.REL and cross reference file TEST.CRF.
*T,U=TEST/C	Assemble file TEST.MAC and create object file T.REL and cross reference file U.CRF.

When the assembler is finished, run the cross reference facility by typing CREF80. CREF80 prompts the user with an asterisk. CREF80 generates a cross reference listing from the .CRF file that was created during assembly. The CREF80 command format is:

*listing file=source file

The default extension for the source file is .CRF. There are no switches in CREF80 commands.

Examples of CREF-80 command strings:

*=TEST Examine file TEST.CRF and
 generate a cross reference
 listing file TEST.LST.

*T=TEST Examine file TEST.CRF and
 generate a cross reference
 listing file T.LST.

Cross reference listing files differ from ordinary listing files in that:

1. Each source statement is numbered with a cross reference number.
2. At the end of the listing, variable names appear in alphabetic order along with the numbers of the lines on which they are referenced or defined. Line numbers on which the symbol is defined are flagged with '#'.

CHAPTER 4

LINK-80 LINKING LOADER

NOTE

If you are using the TEKDOS operating system, see Appendix A for proper command formats.

4.1 RUNNING LINK-80

The command to run LINK-80 is

L80

LINK-80 returns the prompt "*", indicating it is ready to accept commands.

4.2 COMMAND FORMAT

Each command to LINK-80 consists of a string of object filenames separated by commas. These are the files to be loaded by LINK-80. The command format is:

objfile1,objfile2,...objfilen

The default extension for all filenames is REL. Command lines are supported, that is, the invocation and command may be typed on the same line.

Example:

L80 MYPROG,YRPROG

Any filename in the LINK-80 command string can also specify a device name. The default device name with the CP/M operating system is the currently logged disk. The default device with the ISIS-II operating system is disk drive 0. The format is:

```
dev1:objfile1,dev2:objfile2,...devn:objfilen
```

The device names are as listed in Section 2.2.1.

Example:

```
L80 MYPROG,A:YRPROG
```

After each line is typed, LINK-80 will load the specified files. After LINK finishes this process, it will list all symbols that remained undefined followed by an asterisk.

Example:

```
*MAIN
```

```
DATA 0100 0200
```

```
SUBR1* (SUBR1 is undefined)
```

```
*SUBR1
```

```
DATA 0100 0300
```

```
*
```

Typically, to execute a MACRO-80 program and subroutines, the user types the list of filenames followed by /G (begin execution). To resolve any external, undefined symbols, you can first search your library routines (See Chapter 5, LIB-80) by appending the filenames, followed by /S, to the loader command string.

***MYLIB/S** Searches MYLIB.REL for unresolved global symbols

***/G** Starts execution

4.2.1 LINK-80 Switches

A number of switches may be given in the LINK-80 command string to specify actions affecting the loading or execution of the program(s). Each switch must be preceded by a slash (/). (With the TEKDOS operating system, switches are preceded by hyphens . See Appendix A.)

Switches may be placed wherever applicable in the command string:

1. At command level. It is possible for a switch to be the entire LINK-80 command, or to appear first in the command string. For example:

*/G Tells LINK-80 to begin execution of program(s) already loaded

*/M List all global references from program(s) already loaded

*/P:200,FOO Load FOO, with program area beginning at address 200

2. Immediately after a filename. An S or N switch may refer to only one filename in the command string. Therefore, when the S or N switch is required, it is placed immediately after that filename, regardless of where the filename appears in the command string. For example:

*MYLIB/S,MYPROG Search MYLIB.REL and load necessary library modules, then load MYPROG.REL.

*MYPROG/N,MYPROG/E Load MYPROG.REL, save MYPROG.COM on disk and exit LINK-80.

3. At the end of the command string. Any required switches that affect the entire load process may be appended at the end of the command string. For example:

*MYPROG/N,MYPROG/M/E Open a CP/M COM file called MYPROG.COM, load MYPROG.REL and list all global references. Exit LINK-80 and save the COM file.

MYLIB/S,MYSUB,MYPROG/N,MYPROG/M/G Search MYLIB.REL, load and link MYSUB.REL and MYPROG.REL, open a CP/M COM file called MYPROG.COM, list all global references, save the COM file, and execute MYPROG.

The available switches are:

<u>Switch</u>	<u>Action</u>
R	Reset. Put loader back in its initial state. Use /R if you loaded the wrong file by mistake and want to restart. /R takes effect as soon as it is encountered in a command string.
E or E:Name	Exit LINK-80 and return to the operating system. The system library will be searched on the current disk to satisfy any existing undefined globals. Before exiting, LINK-80 prints three numbers: the start address, the address of the next available byte, and the number of 256-byte pages used. The optional form E:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program. Use /E to load a program and exit back to the monitor.
G or G:Name	Start execution of the program as soon as the current command line has been interpreted. The system library will be searched on the current disk to satisfy any existing undefined globals if they exist. Before execution actually begins, LINK-80 prints three numbers and a BEGIN EXECUTION message. The three numbers are the start address, the address of the next available byte, and the number of 256-byte pages used. The optional form G:Name (where Name is a global symbol previously defined in one of the modules) uses Name for the start address of the program.
N	If a <filename>/N is specified, the program will be saved on disk under the selected name (with a default extension of .COM for CP/M) when a /E or /G is done. A jump to the start of the program is inserted if needed so the program can run properly (at 100H for CP/M).

P and D

/P and /D allow the origin(s) to be set for the next program loaded. /P and /D take effect when seen (not deferred), and they have no effect on programs already loaded. The form is /P:<address> or /D:<address>, where <address> is the desired origin in the current typeout radix. (Default radix is hex. /O sets radix to octal; /H to hex.) LINK-80 does a default /P:<link origin>+3 (i.e., 103H for CP/M and 4003H for ISIS) to leave room for the jump to the start address. NOTE: Do not use /P or /D to load programs or data into the locations of the loader's jump to the start address (100H to 102H for CP/M) unless it is to load the start of the program there. If programs or data are loaded into these locations, the jump will not be generated.

If no /D is given, data areas are loaded before program areas for each module. If a /D is given, all Data and Common areas are loaded starting at the data origin and the program area at the program origin. Example:

```
* /P:200,FOO  
Data    200    300  
*/R  
*/P:200 /D:400,FOO  
Data    400    480  
Program 200    280
```

U

List the origin and end of the program and data area and all undefined globals as soon as the current command line has been interpreted. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

M

List the origin and end of the program and data area, all defined globals and their values, and all undefined globals followed by an asterisk. The program information is only printed if a /D has been done. Otherwise, the program is stored in the data area.

S

Search the filename immediately preceding the /S in the command string to satisfy any undefined globals.

4.2.2 CP/M LINK-80 Switches

The following switches apply to CP/M versions only.

X If a filename/N was specified, /X will cause the file to be saved in Intel ASCII HEX format with an extension of HEX.

Example: FOO/N/X/E will create an Intel ASCII HEX formatted load module named FOO.HEX.

Y If a filename/N was specified, /Y will create a filename.SYM file when /E is entered. This file contains the names and addresses of all Globals for use with Digital Research's Symbolic Debugger, SID and ZSID.

Example: FOO/N/Y/E creates FOO.COM and FOO.SYM. MYPROG/N/X/Y/E creates MYPROG.HEX and MYPROG.SYM.

4.2.3 Sample Links

LINK AND GO

```
A>L80
*EXAMPL,EXMPL1/G
DATA 3000 30AC
[304F 30AC 49]
[BEGIN EXECUTION]

1792      14336
14336     -16383
-16383      14
        14      112
       112      896
A>
```

LINK AND SAVE

```
A>L80
*EXAMPL,EXMPL1,EXAM/N/E
DATA 3000 30AC
[304F 30AC 49]
A>
```

Loads and links EXAMPL.REL, EXMPL1.REL and creates EXAM.COM.

4.3 FORMAT OF LINK COMPATIBLE OBJECT FILES

NOTE

Section 4.3 is reference material for users who wish to know the load format of LINK-80 relocatable object files. Most users will want to skip this section, as it does not contain material necessary to the operation of the package.

LINK-compatible object files consist of a bit stream. Individual fields within the bit stream are not aligned on byte boundaries, except as noted below. Use of a bit stream for relocatable object files keeps the size of object files to a minimum, thereby decreasing the number of disk reads/writes.

There are two basic types of load items: Absolute and Relocatable. The first bit of an item indicates one of these two types. If the first bit is a 0, the following 8 bits are loaded as an absolute byte. If the first bit is a 1, the next 2 bits are used to indicate one of four types of relocatable items:

- 00 Special LINK item (see below).
- 01 Program Relative. Load the following 16 bits after adding the current Program base.
- 10 Data Relative. Load the following 16 bits after adding the current Data base.
- 11 Common Relative. Load the following 16 bits after adding the current Common base.

Special LINK items consist of the bit stream 100 followed by:

a four-bit control field

an optional A field consisting of a two-bit address type that is the same as the two-bit field above except 00 specifies absolute address

an optional B field consisting of 3 bits that give a symbol length and up to 8 bits for each character of the symbol

A general representation of a special LINK item is:

1 00 xxxx	yy nn	<u>zzz + characters of symbol name</u>
	A field	B field
xxxx		Four-bit control field (0-15 below)
yy		Two-bit address type field
nn		Sixteen-bit value
zzz		Three-bit symbol length field

The following special types have a B-field only:

0	Entry symbol (name for search)
1	Select COMMON block
2	Program name
3	Request library search
4	Extension LINK items (see below)

The following special LINK items have both an A field and a B field:

5	Define COMMON size
6	Chain external (A is head of address chain, B is name of external symbol)
7	Define entry point (A is address, B is name)

The following special LINK items have an A field only:

8	External - offset. Used for JMP and CALL to externals
9	External + offset. The A value will be added to the two bytes starting at the current location counter immediately before execution.
10	Define size of Data area (A is size)
11	Set loading location counter to A
12	Chain address. A is head of chain, replace all entries in chain with current location counter. The last entry in the chain has an address field of absolute zero.
13	Define program size (A is size)
14	End program (forces to byte boundary)

The following special Link item has neither an A nor a B field:

15 End file

An Extension LINK item follows the general format of a B-field-only special LINK item, but contents of the B-field are not a symbol name. Instead, the symbol area contains one character to identify the type of Extension LINK item, followed by from 1 to 7 characters of additional information.

Thus, every Extension LINK item has the format:

1 00 0100 zzz i jjjjjjj

where

zzz may be any three bit integer (with 000 representing 8),

i is an eight bit Extension LINK item type identifier, and

j jjjjjjj are zzz-1 eight bit characters of information whose significance depends on i

At present, there is only one Extension LINK item:

i = X'35' COBOL overlay segment sentinel

zzz = 010 (binary)

j = COBOL segment number -49 (decimal)

When the overlay segment sentinel is encountered by the linker, the current overlay segment number is set to the value of j+49. If the previously existing segment number was non-zero and a /N switch is in effect, the data area is written to disk in a file whose name is the current program name and whose extension is Vnn, where nn are the two hexadecimal digits representing the number j+49 (decimal).

4.4 LINK-80 ERROR MESSAGES

LINK-80 has the following error messages:

- ?No Start Address A /G switch was issued, but no main program had been loaded.
- ?Loading Error The last file given for input was not a properly formatted LINK-80 object file.
- ?Out of Memory Not enough memory to load program.
- ?Command Error Unrecognizable LINK-80 command.
- ?<file> Not Found <file>, as given in the command string, did not exist.
- %2nd COMMON Larger /XXXXXX/
The first definition of COMMON block /XXXXXX/ was not the largest definition. Reorder module loading sequence or change COMMON block definitions.
- %Mult. Def. Global YYYYYY
More than one definition for the global (internal) symbol YYYYYY was encountered during the loading process.
- %Overlaying { Program } Area ,Start = xxxx
 { Data } ,Public = <symbol name>(xxxx)
 ,External = <symbol name>(xxxx)
A /D or /P will cause already loaded data to be destroyed.
- ?Intersecting { Program } Area
 { Data }
The program and data area intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.
- ?Start Symbol - <name> - Undefined
After a /E: or /G: is given, the symbol specified was not defined.

Origin { Above } Loader Memory, Move Anyway (Y or N)?
Below }

After a /E or /G was given, either the data or program area has an origin or top which lies outside loader memory (i.e., loader origin to top of memory). If a Y <cr> is given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit. In either case, if a /N was given, the image will already have been saved.

?Can't Save Object File

A disk error occurred when the file was being saved.

4.5 PROGRAM BREAK INFORMATION

LINK-80 stores the address of the first free location in a global symbol called \$MEMORY if that symbol has been defined by a program loaded. \$MEMORY is set to the top of the data area +1.

NOTE

If /D is given and the data origin is less than the program area, the user must be sure there is enough room to keep the program from being destroyed. This is particularly true with the disk driver for FORTRAN-80 which uses \$MEMORY to allocate disk buffers and FCB's.

CHAPTER 5
LIB-80 LIBRARY MANAGER

(CP/M Versions Only)

LIB-80 is the object time library manager for CP/M versions of FORTRAN-80 and COBOL-80. LIB-80 will be interfaced to other operating systems in future releases of FORTRAN-80 and COBOL-80.

WARNING

Read this chapter carefully and make a back-up copy of your libraries before using LIB. It is not difficult to destroy a library with LIB-80.

5.1 LIB-80 COMMANDS

To run LIB-80, type LIB followed by a carriage return. LIB-80 will return the prompt "*" indicating it is ready to accept commands. Each command in LIB-80 either lists information about a library or adds new modules to the library under construction.

Commands to LIB-80 consist of an optional destination filename which sets the name of the library being created, followed by an equal sign, followed by module names separated by commas. The default destination filename is FORLIB.LIB. Examples:

```
*NEWLIB=FILE1 <MOD2>, FILE3,TEST  
*SIN,COS,TAN,ATAN
```

Any command specifying a set of modules concatenates the modules selected onto the end of the last destination filename given. Therefore,

```
*FILE1,FILE2 <BIGSUB>, TEST
```

is equivalent to

```
*FILE1  
*FILE2 <BIGSUB>  
*TEST
```

5.1.1 Modules

A module is typically a FORTRAN or COBOL subprogram, main program or a MACRO-80 assembly that contains ENTRY statements.

The primary function of LIB-80 is to concatenate modules in .REL files to form a new library. In order to extract modules from previous libraries or .REL files, a powerful syntax has been devised to specify ranges of modules within a .REL file.

The simplest way to specify a module within a file is simply to use the name of the module. For example:

SIN

But a relative quantity plus or minus 255 may also be used. For example:

SIN+1

specifies the module after SIN and

SIN-1

specifies the one before it.

Ranges of modules may also be specified by using two dots:

..SIN means all modules up to and including SIN.

SIN.. means all modules from SIN to the end of the file.

SIN..COS means SIN and COS and all the modules in between.

Ranges of modules and relative offsets may also be used in combination:

SIN+1..COS-1

To select a given module from a file, use the name of the file followed by the module(s) specified enclosed in angle brackets and separated by commas:

FORLIB <SIN..COS>

or

MYLIB.REL <TEST>

or

BIGLIB.REL <FIRST,MIDDLE,LAST>

etc.

If no modules are selected from a file, then all the modules in the file are selected:

TESTLIB.REL

5.2 LIB-80 SWITCHES

NOTE

/E will destroy your current library if there is no new library under construction. Exit LIB-80 using Control-C if you are not revising the library.

A number of switches are used to control LIB-80 operation. These switches are always preceded by a slash:

/O Octal - set Octal typeout mode for /L command.

/H Hex - set Hex typeout mode for /L command (default).

/U List the symbols which would remain undefined on a search through the file specified.

- /L List the modules in the files specified and symbol definitions they contain.
- /C (Create) Throw away the library under construction and start over.
- /E Exit to CP/M. The library under construction (.LIB) is revised to .REL and any previous copy is deleted.

NOTE

/E will destroy your current library if there is no new library under construction.
Exit LIB-80 using Control-C if you are not revising the library.

- /R Rename - same as /E but does not exit to CP/M on completion.

5.3 LIB-80 LISTINGS

To list the contents of a file in cross reference format, use /L:

*FORLIB/L

When building libraries, it is important to order the modules such that any intermodule references are "forward." That is, the module containing the global reference should physically appear ahead of the module containing the entry point. Otherwise, LINK-80 may not satisfy all global references on a single pass through the library.

Use /U to list the symbols which could be undefined in a single pass through a library. If a module in the library makes a backward reference to a symbol in another module, /U will list that symbol. Example:

*SYSLIB/U

NOTE: Since certain modules in the standard FORTRAN and COBOL systems are always force-loaded, they will be listed as undefined by /U but will not cause a problem when loading FORTRAN or COBOL programs.

Listings are currently always sent to the terminal; use control-P to send the listing to the printer.

5.4 SAMPLE LIB SESSION

BUILDING A LIBRARY:

```
A>LIB  
*TRANLIB=SIN,COS,TAN,ATAN,ACOG  
*EXP  
*/E  
A>
```

LISTING A LIBRARY:

```
A>LIB *TRANLIB.LIB/U  
*TRANLIB.LIB/L  
.  
. .  
. .  
(List of symbols in TRANLIB.LIB)  
. .  
. .  
*Control-C  
A>
```

5.5 SUMMARY OF SWITCHES AND SYNTAX

/O Octal - set listing radix
/H Hex - set listing radix
/U List undefineds
/L List cross reference
/C Create - start LIB over
/E Exit - Rename .LIB to .REL and exit
/R Rename - Rename .LIB to .REL

```
module ::= module name {+ or - number}  
  
module sequence ::=  
  
module | ..module | module.. | module1..module2  
  
file specification ::= filename {<module sequence>{,<module sequence>}}  
  
command ::= {library filename=} {list of file specifications}  
          {list of switches}
```


APPENDIX A

TEKDOS Operating System

The command formats for MACRO-80, LINK-80 and CREF-80 differ slightly under the TEKDOS operating system.

A.1 TEKDOS COMMAND FILES

The files F80, M80, and C80 are actually TEKDOS command files for the compiler, assembler, loader, and cross reference programs, respectively. These command files set the emulation mode to 0 and select the Z-80 assembler processor (see TEKDOS documentation), then execute the appropriate program file. You will note that all of these command files are set up to execute the Microsoft programs from drive 1. LINK-80 will also look for the library (FORLIB) on drive 1. If you wish to execute any of this software from drive 0, the command file must be edited and LINK-80 should be given an explicit library search directive "FORLIB-S". (See Section 4.2.1 of this manual.)

A.2 MACRO-80

The M80 assembler accepts command lines only. A prompt is not displayed and interactive commands are not accepted. Commands have the same format as TEKDOS assembler commands; i.e., three filename or device name parameters plus optional switches.

```
M80 [objfile] [lstfile] sourcefile [sw1] [sw2...]
```

The object and listing file parameters are optional. These files will not be created if the parameters are omitted, however any error messages will still be displayed on the console. The available switches are as described in Chapter 2 of this manual. except that the switches are delimited by commas or spaces instead of slashes.

A.3 CREF-80

The form of commands to CREF80 is:

```
C80 lstfile sourcefile
```

Both filename parameters are required. The sourcefile parameter is always the name of a CREF80 file created during assembly, by use of the C switch.

Example:

Create a CREF80 file using MACRO-80:

```
M80 ,, TSTCRF TSTMAC C
```

Create a cross reference listing from the CREF80 file:

```
C80 TSTLST TSTCRF
```

A.4 LINK-80

With TEKDOS, the LINK-80 loader accepts interactive commands only. Command lines are not supported.

When LINK-80 is invoked, and whenever it is waiting for input, it will prompt with an asterisk. Commands are lists of filenames and/or devices separated by commas or spaces and optionally interspersed with switches. The input to LINK-80 must be Microsoft relocatable object code (not the same as TEKDOS loader format).

Switches to LINK-80 are delimited by hyphens under TEKDOS, instead of slashes. All LINK-80 switches (as documented in Chapter 3) are supported, except "G" and "N", which are not implemented at this time.

Examples:

1. Assemble a MACRO-80 program named XTEST, creating an object file called XREL and a listing file called XLST:

```
>M80 XREL XLST XTEST
```

2. Load XTEST and save the loaded module:

```
>L80  
*XREL-E  
[04AD 22B8]  
*DOS*ERROR 46  
L80 TERMINATED  
>M XMOD 400 22B8 04AD
```

Note that "-E" exits via an error message due to execution of a Halt instruction. The memory image is intact, however, and the "Module" command may be used to save it. Once a program is saved in module format, it may then be executed directly without going through LINK-80 again.

The bracketed numbers printed by LINK-80 before exiting are the entry point address and the highest address loaded, respectively. The loader default is to begin loading at 400H. However, the loader also places a jump to the start address in location 0, thereby allowing execution to begin at 0. The memory locations between 0003 and 0400H are reserved for SRB's and I/O buffers at runtime.

INDEX

\$INCLUDE	2-14
\$MEMRY	4-11
.COMMENT	2-16
.CREF	2-23
.DEPHASE	2-25
.LALL	2-23
.LFCOND	2-20
.LIST	2-20
.PAGE	2-37
.PHASE	2-25
.PRINTX	2-17
.RADIX	2-6, 2-17
.REQUEST	2-18
.SALL	2-23
.SFCOND	2-20
.TFCOND	2-20
.XALL	2-23
.XCREF	2-23
.XLIST	2-20
Absolute memory	2-8, 2-11, 2-38
Arithmetic operators	2-8
ASEG	2-8, 2-11, 2-24
Block pseudo ops	2-25
Character constants	2-7
Code Relative	2-11, 2-24 to 2-25, 2-38
Command format	2-1, 3-1, 4-1, 5-1
Comments	2-6, 2-16
COMMON	2-8, 2-11, 2-24 to 2-25, 2-38 to 2-39
Conditionals	2-19
Constants	2-6
CP/M	2-2 to 2-3, 4-4 to 4-6, 5-1, 5-4
Cross reference facility . . .	2-4, 2-23, 2-37, 3-1
CSEG	2-8, 2-11, 2-24, 2-36
Data Relative	2-8, 2-12, 2-24 to 2-25, 2-38
DB	2-6, 2-11
DC	2-12
Define Byte	2-6, 2-11
Define Character	2-12
Define Origin	2-15
Define Space	2-12
Define Word	2-13
DS	2-12
DSEG	2-8, 2-12, 2-24, 2-36
DW	2-13

EDIT-80	2-5, 2-37
ELSE	2-20
END	2-13
ENDIF	2-20
ENDM	2-25, 2-29
ENTRY	2-13, 5-2
EQU	2-14 to 2-15
Error codes	2-35, 2-37
Error messages	2-36, 4-10
EXITM	2-29
EXT	2-14
Externals	2-9, 2-14, 2-35, 2-38
EXTRN	2-14
IF	2-19
IF1	2-19
IF2	2-19
IFB	2-19
IFDEF	2-19
IFDIF	2-19
IFE	2-19
IFF	2-19
IFIDN	2-19
IFNB	2-19
IFT	2-19
INCLUDE	2-14
INTEL	2-36
IRP	2-23, 2-25, 2-27
IRPC	2-23, 2-25, 2-27
ISIS-II	2-2 to 2-3, 2-5, 4-5
LIB-80	5-1
Library manager	5-1
LINK-80	2-11, 2-13, 2-18, 2-25, 4-1, 5-4
Listings	2-14, 2-20, 2-37 to 2-38, 3-2, 5-4
LOCAL	2-30
Logical operators	2-8
MACLIB	2-14
MACRO	2-23, 2-25 to 2-26, 2-28 to 2-29
Macro operators	2-30
Modes	2-8
Modules	5-2
NAME	2-15
Operators	2-8
ORG	2-11, 2-13, 2-15, 2-24
PAGE	2-15, 2-36
Program Relative	2-8
PUBLIC	2-5, 2-13, 2-39
REPT	2-23, 2-25 to 2-26
SET	2-15

Microsoft Software Problem Report

Use this form to report errors or problems in: Microsoft BASIC-80

Microsoft BASIC-86

Microsoft BASIC
Compiler

Date _____

Report only one problem per form.

Describe your hardware and operating system: _____

BASIC Release number: _____

Please supply a concise description of the problem and the circumstances surrounding its occurrence. If possible, reduce the problem to a simple test case. Otherwise, include all programs and data in machine readable form (preferably on a diskette). If a patch or interim solution is being used, please describe it.

This form may also be used to describe suggested enhancements to Microsoft BASIC.

Problem Description:

Did you find errors in the documentation supplied with the software? If so, please include page numbers and describe:

Fill in the following information before returning this form:

Name _____ Phone _____

Organization _____

Address _____ City _____ State _____ Zip _____

Return form to:

Microsoft
10800 NE Eighth, Suite 819
Bellevue, WA 98004